Exercício 2 - Trabalho Prático 2

Grupo 6:

Ruben Silva - pg57900

Luís Costa - pg55970

Problema:

1. Construir uma classe Python que implemente o EcDSA a partir do "standard" FIPS186-5
   A. A implementação deve conter funções para assinar digitalmente e verificar a assinatura.
   B. 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".

Edwards25519

Funções

In [403…

```
from sage.all import *
import os
import hashlib
```

Toda a class a seguir criada foi basead nos seguintes ficheiros:

1. FIPS186-5
2. RFC8032

In [404…

```
class Edwards25519():
    #Private Functions
    def __init__(self):
        self.p = 2**255 - 19
        self.d = -121665 * inverse_mod(121666, self.p) % self.p
        self.q = 2**252 + 27742317777372353535851937790883648493


        self.gy = 4 * self.__modp_inv(5) % self.p
        self.gx = self.__recover_x(self.gy, 0)
        self.G = (self.gx, self.gy, 1, self.gx * self.gy % self.p)

    def __sha512(self, data):
        return hashlib.sha512(data).digest()

    def __sha512_modq(self, data):
```

```
        return int.from_bytes(self.__sha512(data), 'little') % self.q

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

    def __point_mult(self,s,P):
        Q = (0, 1, 1, 0)
        while s > 0:
            if s & 1:
                Q = self.__point_add(Q, P)
            P = self.__point_add(P, P)
            s >>= 1
        return Q

    def __point_equal(self,P,Q):
        if (P[0] * Q[2] - Q[0] * P[2]) % self.p != 0:
            return False
        if (P[1] * Q[2] - Q[1] * P[2]) % self.p != 0:
            return False
        return True


    def __mod_sqrt(self):
        return pow(2, (self.p-1)//4, self.p)

    def __modp_inv(self, x):
        return pow(x, self.p-2, self.p)

    def __recover_x(self, y, sign):
        if y >= self.p:
            return None
        x2 = (y*y-1) * self.__modp_inv(self.d*y*y+1)
        if x2 == 0:
            if sign:
                return None
            else:
                return 0
        x = pow(x2, (self.p+3) // 8, self.p)
        if (x*x - x2) % self.p != 0:
            x = x * self.__mod_sqrt() % self.p
        if (x*x - x2) % self.p != 0:
            return None
        if (x & 1) != sign:
            x = self.p - x
        return x

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

    def __point_decompress(self, 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 = self.__recover_x(y, sign)

        if x is None:
            return None
        else:
            return (x, y, 1, x*y % self.p)

    def __secret_expand(self, secret):
        if len(secret) != 32:
            raise Exception("Invalid input length for secret key")
        h = self.__sha512(secret)
        a = int.from_bytes(h[:32], "little")
        a &= (1 << 254) - 8
        a |= (1 << 254)
        return (a, h[32:])

    #Public Functions
    def secret_to_public(self, secret):
        a, dummy = self.__secret_expand(secret)
        return self.__point_compress(self.__point_mult(a, self.G))

    def sign(self, secret, message):
        a, prefix = self.__secret_expand(secret)
        A = self.__point_compress(self.__point_mult(a, self.G))
        r = self.__sha512_modq(prefix + message)
        R = self.__point_mult(r, self.G)
        Rs = self.__point_compress(R)
        h = self.__sha512_modq(Rs + A + message)
        s = (r + h * a) % self.q
        return Rs + int.to_bytes(s, 32, 'little')

    def verify(self, public, message, signature):
        if len(public) != 32:
            raise Exception("Bad public key length")
        if len(signature) != 64:
            Exception("Bad signature length")
        A = self.__point_decompress(public)
        if not A:
            return False
        Rs = signature[:32]
        R = self.__point_decompress(Rs)
        if not R:
            return False
        s = int.from_bytes(signature[32:], "little")
        if s >= self.q: return False
        h = self.__sha512_modq(Rs + public + message)
        sB = self.__point_mult(s, self.G)
        hA = self.__point_mult(h, A)
        return self.__point_equal(sB, self.__point_add(R, hA))
```

Testar

In [405…

```
def test_edwards25519():
    ed = Edwards25519()
    secret = os.urandom(32)
    message = f"Hello, world!"
    print(f"Secret:{secret}")
```

```
    public = ed.secret_to_public(secret)
    print(f"Public:{public}")
    print(f"Message:{message}")
    signature = ed.sign(secret, message.encode())
    print(f"Signature:{signature}")
    assert ed.verify(public, message.encode(), signature)

    print("Edwards25519 test passed")
```

In [406…

```
test_edwards25519()
```

```
Secret:b'\xec\x1f\x10\re\x1e~\x11ud3\x9e\xcbBS\xa9<\x8f\xe1\xd0&\xda\x06s\xdb\x07
\xf8\x00%\x19\xf6\x9a'
Public:b'\x12\xee\x85\n\xc8\x14\xfbHr\xca~\x19\xadt\x8c\xcc\xa0P\xc6E\x9e\xb5BfL
\x85\xa9\x95\xa3h\xd7?'
Message:Hello, world!
Signature:b'\xf3R\x15\x18\xa5=\xe9\x90\xbc]\xae\xb7\xae\r\x13\xa7v\xa3\x0b]{\x0c,
j\r(\xb1\xfe\\1\xde{\x849\xbd\xe5_\xaa\xcfR\x06P\xd7JTC\xaa\x9b\x9c\x80U\xaa-\xf9
\xbb*\xff%\xa7\xdae\x1e\x84\t'
Edwards25519 test passed
```