Trabalho Prático 1

Grupo 17, constituído por:

-- Joana Castro e Sousa, PG47282

-- Tiago Taveira Gomes, PG47702

-- João Carlos Pereira Rodrigues, PG46534

Pergunta 3:

Use o Sagemath para:

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

i. A implementação deve conter funções para assinar digitalmente e verificar a assinatura.

ii. 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”.

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

Curva de Edwards25519

In [ ]:
```
# Edwards 22519
p = (2^255)-19
K = GF(p)
a = K(-1)
d = -K(121665)/K(121666)

ed25519 = {
'b'  : 256,
'Px' : K(15112221349535400772501151409588531511454012693041857206046113283949847762202),
'Py' : K(46316835694926478169428394003475163141307993866256225615783033603165251855960),
'L'  : ZZ(2^252 + 27742317777372353535851937790883648493), ## ordem do subgrupo primo
'n'  : 254,
'h'  : 8
}
```

In [ ]:
```
def EdDSA():
    # Função que inicializa toda a instância e os valores globais necessários à sua execução
    def __init__(self):
        # EdDSA has 11 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 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.
        # 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 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.
        # 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.
        # 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 }
```

```
        # 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.
        # 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).
        self.p = 2^255 - 19
        self.K = GF(self.p)
        self.Px = self.K(15112221349535400772501151409588531511454012693041857206046113283949847762202)
        self.Py = self.K(46316835694926478169428394003475163141307993866256225615783033603165251855960)
        self.L = ZZ(2^252 + 27742317777372353535851937790883648493)
        self.requested_security_strength = 128
        self.a = self.K(-1)
        self.d = -self.K(121665)/self.K(121666)
        self.n = 254
        self.c = 3
        self.h = 8
        self.b = 256

        self.chave_privada = os.urandom(self.b/8)


    #H é uma função usada durante a derivação
    #SHA512:Ed25519 ou SHAKE256:Ed448
    def H(pk):
        return hashlib.sha512(pk).digest()

    def bit(h,i):
        return ((h[int(i/8)]) >> (i%8)) & 1

    def expmod(b,e,m):
        if e == 0: return 1
        t = expmod(b,e/2,m)**2 % m
        if e & 1: t = (t*b) % m
        return t

    def inv(x):
        return expmod(x,q-2,q)

    def scalarmult(P,e):
        if e == 0: return [0,1]
        Q = scalarmult(P,e/2)
        Q = edwards(Q,Q)
        if e & 1: Q = edwards(Q,P)
        return Q


    def ed2ec(x,y):      ## mapeia Ed --> EC
        if (x,y) == (0,1):
            return EC(0)
        z = (1+y)/(1-y) ; w = z/x
        alfa = constants['alfa']; s = constants['s']
        return EC(z/s + alfa , w/s)

    def encodeKey(self, x):
        return mod(x, 2)

    def compute_point(self,point):
        x = point.xy()[0]
        y = point.xy()[1]
            #bit menos significativa
        leastBit = self.encodeKey(x)
        encoded = bin(y) + chr(leastBit)
            #Fazer o encode do ponto: (h[0] + 28 • h[1] + ... + 2248 * h[31])
        return sum(2^i * bit(self.private_key,i) for i in range(0,len(encoded)))

    #EdDSA Key Pair Generation
    def generate_public_key(self):
            #Para gerar a chave pública é preciso gerar a chave privada
            #1. Gerar uma string de b bits aleatória +
```

```
        #2. Calcular o Hash da string aleatoria para gerar a chave privada
    self.private_key = H(os.urandom(32))
        #3. Calcular e modificar o hdigest1: 3 primeiros bits a 0, ultimo bit a 0
        #e penultimo a 1
        #4. Calcular um inteiro do hdigest1 usando little-endian
    d = 2^(self.b-2) + sum(2^i * bit(self.private_key,i) for i in range(3,self.b-2))
        #5 Calcular o ponto
    point = d * self.P
        #Public key ponto
    self.public_key_point = point
        #Computar o ponto para obter a public key
    d2 = self.compute_point(point)
    self.public_key = d2

def bytes_to_int(bytes):
    result = 0
    for b in bytes:
        result = result * 256 + int(b)
    return result

def convert_to_ZZ(message):
    raw = ascii_to_bin(message)
    return ZZ(int(str(raw),2))

#Calcular o hash em hexadecimal
def HB(m):
    h = hashlib.new('sha512')
    h.update(m)
    return h.hexdigest()

def bit(h,i):
        return ((h[int(i/8)]) >> (i%8)) & 1

# --------------------------------------------- SIGNATURE ---------------------------------------------
# EdDSA signatures are deterministic. The signature is generated using the hash of the private key and the message using the procedure below or an equivalent process.
#   Inputs:
#   1. Bit string M to be signed
#   2. Valid public-private key pair (d, Q) for domain parameters D
#   3. H: SHA-512 for Ed25519 or SHAKE256 for Ed448
#   4. For Ed448, a string context set by the signer and verifier with a maximum length of 255
#   octets; by default, context is the empty string
#
#   Output: The signature R || S, where R is an encoding of a point and S is a little-endian encoded value.

# Process:
#   As specified in IETF RFC 8032, the EdDSA signature of a message M under a private key d is defined as the 2b-bit string R || S. The octet strings R and S are derived as follows:
#   1. Compute the hash of the private key d, H(d) = (h0, h1, ..., h2b-1) using SHA-512 for Ed25519 and SHAKE256 for Ed448 (H(d)= SHAKE256(d, 912)). H(d) may be precomputed.
#   2. Using the second half of the digest hdigest2 = hb || ... || h2b-1, define:
#       2.1 For Ed25519, r = SHA-512(hdigest2 || M); r will be 64-octets.
#       2.2 For Ed448, r = SHAKE256(dom4(0, context) || hdigest2 || M, 912).
#           In IETF RFC 8032, dom4(f, c) is defined to be ("SigEd448" || octet(f) ||
#           octet(octetlength(c)) || c). The string "SigEd448" is in ASCII (8 octets). The value
#           octet(f) is the octet with value f, and octetlength(c) is the number of octets in string
#           c.
#   3. Compute the point [r]G. The octet string R is the encoding of the point [r]G.
#   4. Derive s from H(d) as in the key pair generation algorithm. Use octet strings R, Q, and M to define:
#       4.1 For Ed25519, S = (r + SHA-512(R || Q || M) * s) mod n.
#       4.2 For Ed448, S = (r + SHAKE256(dom4(0, context) || R || Q || M, 912) * s) mod n.
#           The octet string S is the encoding of the resultant integer.
#   5. Form the signature as the concatenation of the octet strings R and S.

def signature_gen(self, M):
    r = H(self.chave_privada)
    #(...)

#EdDSA Signature Generation
def sign(self, msg):
        #1. Computar o hash da chave privada
```

```
        key_hashed=HB(self.private_key)
            #2. Concatenar essa chave com a mensagem
        key_msg=key_hashed.encode('utf-8') + msg
        k = convert_to_ZZ(HB(key_msg))
        r = mod(k ,self.n)
        r_int=ZZ(r)

            #3. Computar o ponto R
        R = r_int * self.P

            #4. Derivar a partir da chave pública
            # Concatenar - R + chavepublica + mensagem
        prov = R + self.public_key_point
        msg_total = str(prov).encode('utf-8')+msg
            #Calcular o hash msg_total
        msg_hashed = HB(msg_total)
        msg_usada=convert_to_ZZ(msg_hashed)
        h= mod(msg_usada,self.n)
            #Calcular o mod da soma de r com o hash anterior com n
        s=mod(r_int+ZZ(h)*bytes_to_int(self.private_key),self.n)
            #5. Concatenar R e s e fazer o return disso
        return R, s

    # Inputs:
    #   1. Message M
    #   2. Signature R || S where R and S are octet strings
    #   3. Purported signature verification key Q that is valid for domain parameters D
    #   4. For Ed448, a string context set by the signer and verifier with a maximum length of 255
    #   octets; by default, context is the empty string
    # Output: Accept or reject the signature over M as originating from the owner of public key Q
    #
    # Process:
    #   1. Decode the first half of the signature as a point R and the second half of the signature as an
    #       integer s. Verify that the integer s is in the range of 0 ≤ s < n. Decode the public key Q into a
    #       point Q’. If any of the decodings fail, output “reject”.
    #   2. Form the bit string HashData as the concatenation of the octet strings R, Q, and M (i.e.,
    #       HashData = R || Q || M).
    #   3. Using the established hash function or XOF,
    #       3.1 For Ed25519, compute digest = SHA-512(HashData).
    #       3.2 For Ed448, compute digest = SHAKE256(dom4(0, context) || HashData, 912)
    #       Interpret digest as a little-endian integer t.
    #   4. Check that the verification equation [2c * S]G = [2c]R + (2c * t)Q. Output “reject” if
    #       verification fails; output “accept” otherwise.
    def signature_ver(M, sig, Q):
        #(...)
        return 0

    #EdDSA Signature Verification
    def verify(self,msg,R,s):
            #1. Obter R e s separadamente
            #2. Formar uma string com R, chave publica e mensagem
        msg_intermedia = R + self.public_key_point
        msg_total = str(msg_intermedia).encode('utf-8')+msg
            #3. Computar o hash da string anterior
        msg_hashed = HB(msg_total)
        msg_usada=convert_to_ZZ(msg_hashed)
        h= mod(msg_usada,self.n)

            #4.Calcular P1 e P2
        P1=ZZ(s)*self.P
        P2=R+ZZ(h)*(self.public_key_point)

            #Comparar P1 e P2
        print(P1==P2)
```

1) Ver: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5-draft.pdf

2) Ver: https://datatracker.ietf.org/doc/html/rfc8032#appendix-A

3) Ver: https://github.com/HarryR/ethsnarks/blob/2020dec635ee606da1f66118a5f7c6283a4cb6a0/src/jubjub/README.md

4) Ver: http://ed25519.cr.yp.to/software.html

5) Ver: http://ed25519.cr.yp.to/python/ed25519.py

6) Ver: http://ed25519.cr.yp.to/python/sign.py

7) Ver: http://ed25519.cr.yp.to/python/checkparams.py