Exercício 2

Enunciado do problema

Construir uma classe Python que implemente o EdCDSA 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”.
3. Por aplicação da transformação de Fiat-Shamir construa um protocolo de autenticação de desafio-resposta.

Descrição do problema

O nosso objetivo é construir uma classe em python que implemente o EdCDSA (Elliptic Curve Digital Signature Algorithm), um algoritmo de assinatura digital baseado em curvas elípticas. Para tal, vamos usar uma das curvas elípticas definidas no standard FIPS186-5, a curva “edwards25519” ou “edwards448”. Por fim, vamos aplicar a transformação de Fiat-Shamir para construir um protocolo de autenticação de desafio-resposta.

...

Abordagem

Foi escolhida a curva “edwards25519” para a implementação do EdCDSA. Para a implementação de um DSA (Digital Signature Algorithm) é preciso ter em atenção as seguintes operações:

- Geração de chaves
- Distribuição de chaves
- Assinatura digital
- Verificação da assinatura

De notar que uma curva elíptica só é considerada uma curva de Edwards se satisfizer a seguinte equação: `a * x^2 + y^2 = 1 + d * x^2 * y^2`

Para a codificação das operações, foram utilizados como referência os seguintes standards:

- FIPS186-5
- RFC8032

Para a implementação do protocolo de autenticação de desafio-resposta, foi utilizado como referência os seguintes links:

- Fiat-Shamir

- Challenge-Response-Authentication
- Zero Knowledge Proof

De notar, que a implementação realizada foi verificada com a ajuda do package pure25519.

Código

Foi desenvolvido o seguinte código em python, que implementa o EdCDSA e o protocolo de autenticação de desafio-resposta.

Classe que implementa uma curva elíptica de Edwards

```
from __future__ import annotations

from sage.all import ZZ
from sage.arith.misc import is_prime
# noinspection PyUnresolvedReferences
from sage.rings.finite_rings.integer_mod import Mod
from sage.schemes.elliptic_curves.constructor import EllipticCurve
from sage.schemes.elliptic_curves.sha_tate import factor


# noinspection PyPep8Naming
class Ed(object):
    def __init__(self, p, a, d, ed=None):
        assert a != d and is_prime(p) and p > 3
        K = GF(p)

        A = 2 * (a + d) / (a - d)
        B = 4 / (a - d)

        alfa = A / (3 * B)
        s = B

        a4 = s ^ (-2) - 3 * alfa ^ 2
        a6 = -alfa ^ 3 - a4 * alfa

        self.K = K
        self.constants = {'a': a, 'd': d, 'A': A, 'B': B, 'alfa':
alfa, 's': s, 'a4': a4, 'a6': a6}
        self.EC = EllipticCurve(K, [a4, a6])

        if ed is not None:
            self.L = ed['L']
            self.P = self.ed2ec(ed['Px'], ed['Py'])  # gerador do gru
        else:
            self.gen()
```

```
    def order(self):
        # A ordem prima "n" do maior subgrupo da curva, e o respetivo
cofator "h"
        oo = self.EC.order()
        n, _ = list(factor(oo))[-1]
        return n, oo // n

    def gen(self):
        L, h = self.order()
        P = O = self.EC(0)
        while L * P == O:
            P = self.EC.random_element()
        self.P = h * P
        self.L = L

    def is_edwards(self, x, y):
        a = self.constants['a']
        d = self.constants['d']
        x2 = x ^ 2
        y2 = y ^ 2
        return a * x2 + y2 == 1 + d * x2 * y2

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

    def ec2ed(self, P):  ## mapeia EC --> Ed
        if P == self.EC(0):
            return 0, 1
        x, y = P.xy()
        alfa = self.constants['alfa']
        s = self.constants['s']
        u = s * (x - alfa)
        v = s * y
        return u / v, (u - 1) / (u + 1)
```

Classe que implementa operações sobre uma curva elíptica de Edwards

```
# noinspection PyPep8Naming
class EdPoint(object):
    def __init__(self, pt=None, curve=None, x=None, y=None):
        if pt is not None:
            self.curve = pt.curve
```

```
            self.x = pt.x
            self.y = pt.y
            self.w = pt.w
        else:
            assert isinstance(curve, Ed) and curve.is_edwards(x, y)
            self.curve = curve
            self.x = x
            self.y = y
            self.w = x * y

    def eq(self, other):
        return self.x == other.x and self.y == other.y

    def copy(self):
        return EdPoint(curve=self.curve, x=self.x, y=self.y)

    def zero(self):
        return EdPoint(curve=self.curve, x=0, y=1)

    def sim(self):
        return EdPoint(curve=self.curve, x=-self.x, y=self.y)

    def soma(self, other):
        a = self.curve.constants['a']
        d = self.curve.constants['d']
        delta = d * self.w * other.w
        self.x, self.y = (self.x * other.y + self.y * other.x) / (1 +
delta), (
                self.y * other.y - a * self.x * other.x) / (1 - delta)
        self.w = self.x * self.y

    def duplica(self):
        a = self.curve.constants['a']
        d = self.curve.constants['d']
        delta = d * self.w ^ 2
        self.x, self.y = (2 * self.w) / (1 + delta), (self.y ^ 2 - a *
self.x ^ 2) / (1 - delta)
        self.w = self.x * self.y

    def mult(self, n):
        m = Mod(n, self.curve.L).lift().digits(2)  ## obter a
representação binária do argumento "n"
        Q = self.copy()
        A = self.zero()
        for b in m:
            if b == 1:
                A.soma(Q)
            Q.duplica()
        return A
```

```
    def encode(self) -> bytes:
        """
        Encode a point in Ed25519 format.
        The input point should be a tuple (x, y) with integers in the
range 0 ≤ x, y < p.
        """
        return self.encode_right(self.x, self.y)

        x_int = int(self.x)
        y_int = int(self.y)

        # Copying the least significant bit of the x-coordinate to the
most significant bit of the final octet.
        return (int(y_int | ((x_int & 1) << 255))).to_bytes(32,
"little")

    @staticmethod
    def encode_right(x, y):
        from pure25519 import basic

        return basic.encodepoint((x, y))

    @staticmethod
    def decode(s: bytes) -> EdPoint:
        """
        Decode a point in Ed25519 format.
        The output point is a tuple (x, y) with integers in the range
0 ≤ x, y < p.
        """
        return EdPoint.decode_right(s)

        assert len(s) == 32
        # 1. Interpret the octet string as an integer in little-endian
representation. The most significant bit of this integer is the least
significant bit of the x-coordinate, denoted as x0. The y-coordinate
is recovered simply by clearing this bit. If the resulting value is ≥
p, decoding fails.
        y = int.from_bytes(s, "little") &
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7F
        x0 = int.from_bytes(s, "little") >> 255
        if not (y < EdCDSA25519.q):
            raise ValueError("Decoding failed")

        # 2. To recover the x-coordinate, the curve equation requires
x2 = (y^2 - 1) / (d*y^2 - a) (mod p). The denominator is always non-
zero mod p. Compute a square root to obtain x. Square roots can be
computed using the Tonelli-Shanks algorithm.
        # Simplified cases to compute the square root:
```

```
        # Let u = y^2 - 1 and v = d y^2 + 1.
        d = EdCDSA25519.d
        p = EdCDSA25519.q
        u = y ^ 2 - 1
        v = d * y ^ 2 + 1

        # b) To find a square root of (u/v) if p ≡ 5 (mod 8) (as in
Ed25519), first compute the candidate root w = (u/v)^(p+3)/8 = u v^3
(u v^7)^(p-5)/8 (mod p).
        w1 = (u / v) ^ ((p + 3) / 8)
        print("w1: ", w1)
        w = ((u * v ^ 3) * ((u * (v ^ 7)) ^ (p - 5) / 8)) % p
        print("w: ", w)

        # To find the root, check three cases:
        # If v w^2 = u (mod p), the square root is x = w.
        print("v * w ^ 2 == u % p <=> ", v * w ^ 2, " == ", u % p, "
<=> ", v * w ^ 2 == u % p)
        print("v * w ^ 2 == -u % p <=> ", v * w ^ 2, " == ", (-u) % p,
" <=> ", v * w ^ 2 == -u % p)
        if v * w ^ 2 == u % p:
            x = w
        # If v w^2 = -u (mod p), the square root is x = w *
2^((p-1)/4).
        elif v * w ^ 2 == -u % p:
            x = w * 2 ^ ((p - 1) / 4)
        # Otherwise, no square root exists for modulo p, and decoding
fails.
        else:
            raise ValueError("No square root exists for modulo p, and
decoding fails.")

        # For both cases, if x = 0 and x0 = 1, point decoding fails.
        if x == 0 and x0 == 1:
            raise ValueError("Point decoding failed")

        # If x (mod 2) = x0, then the x-coordinate is x.
        if x % 2 == x0:
            x = x  # Just to make it explicit
        # Otherwise, the x-coordinate is p - x.
        else:
            x = p - x

        # 3. Return the decoded point (x,y).
        return EdPoint(curve=EdCDSA25519.E, x=x, y=y)

    @staticmethod
    def decode_right(s: bytes) -> EdPoint:
        from pure25519 import basic
```

```
        P = basic.decodepoint(s)
        return EdPoint(curve=EdCDSA25519.E, x=P[0], y=P[1])


# Simple test case
pub_key_bytes = b'\x00\x9c\x12\xb1\xab\xf03b\x1c\x94&8\xd7\xb0\xbd<\
xe6e\xeel#\xa1\x00R\xf8! \xee\xcae\xf1\x84'
pub_key = EdPoint.decode_right(pub_key_bytes)

assert pub_key.encode_right(pub_key.x, pub_key.y) == pub_key_bytes
```

Classe que implementa a curva de Edwards 25519

```
import os
from hashlib import sha512
from sage.rings.finite_rings.all import GF
from pure25519.basic import bytes_to_clamped_scalar, Base,
bytes_to_element


# noinspection PyPep8Naming
class EdCDSA25519:
    q = 2 ** 255 - 19
    K = GF(q)
    a = K(-1)
    d = -K(121665) / K(121666)

    ed25519 = {
        'b': 256,  ## tamanho da chave em bits
        'Px':
K(15112221349535400772501151409588531511454012693041857206046113283949
847762202),
        # coordenada x do gerador
        'Py':
K(46316835694926478169428394003475163141307993866256225615783033603165
251855960),
        # coordenada y do gerador
        'L': ZZ(2 ^ 252 + 27742317777372353535851937790883648493),  ##
ordem do subgrupo primo
        'n': 254,
        'h': 8,  # cofator do subgrupo
        'c': 3  # logaritmo base 2 do cofator [RFC7748]
    }

    E = Ed(q, a, d, ed25519)  # Curva de Edwards

    @staticmethod
    def to_int(x: bytes) -> int:
        return int.from_bytes(x, "little")
```

```
    def generate_keys(self) -> (bytes, bytes):
        #  1. Obtain a string of b bits from an approved RBG with a
security strength of requested_security_strength or more. The private
key d is this string of b bits.
        d = os.urandom(self.ed25519['b'] // 8)  # Posso passar isto
numa XOF para garantir a aleatoriedade.

        # 2. Compute the hash of the private key d, H(d) = (h0,
h1, ..., h2b-1) using SHA-512 for Ed25519. H(d) may be pre-computed.
Note H(d) is also used in the EdDSA signature generation;
        private_key_hashed = self.get_hash(d)

        # 3. The first half of H(d), (i.e. hdigest1 = (ℎ0,ℎ1,…,ℎ𝑏𝑏−1))
is used to generate the public key. Modify hdigest1 as follows:
        # 3.1 For Ed25519, the first three bits of the first octet are
set to zero; the last bit of the last octet is set to zero; and the
second to last bit of the last octet is set to one. That is,
ℎ0=ℎ1=ℎ2=0, ℎ𝑏𝑏−2=1, and ℎ𝑏𝑏−1=0.
        hdigest1 = private_key_hashed[:32]

        AND_CLAMP = (1 << 254) - 1 - 7
        OR_CLAMP = (1 << 254)
        hdigest1 = (self.to_int(hdigest1) & AND_CLAMP) | OR_CLAMP

        assert bytes_to_clamped_scalar(private_key_hashed[:32]) ==
hdigest1

        # 4. Determine an integer s from hdigest1 using little-endian
convention (see Section 7.2).
        #s = int.from_bytes(hdigest1, "little")
        s = hdigest1

        # 5. Compute the point [s]G. The corresponding EdDSA public
key Q is the encoding (See Section 7.2) of the point [s]G.
        P = EdPoint(curve=self.E, x=self.ed25519['Px'],
y=self.ed25519['Py'])
        Q = Base.scalarmult(s)  #P.mult(s)

        #assert Base.scalarmult(s).XYTZ[0] == Q.x
        #assert Base.scalarmult(s).XYTZ[1] == Q.y

        # Encoding the public key
        Q_encoded = Q.to_bytes()

        return d, Q_encoded

    def sign(self, message: bytes, private_key: bytes, public_key:
bytes) -> bytes:
```

```
        # 1. Compute the hash of the private key d, H(d) = (h0,
h1, ..., h2b-1) using SHA-512 for Ed25519. H(d) may be pre-computed.
        priv_key_hashed = self.get_hash(private_key)

        # 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.
        hdigest2 = priv_key_hashed[32:]
        r = self.get_hash(hdigest2 + message)
        assert len(r) == 64  # r tem de ter 64 octetos

        r_int = int.from_bytes(r, "little")

        # 3. Compute the point [r]G. The octet string R is the
encoding of the point [r]G.
        G = EdPoint(curve=self.E, x=self.ed25519['Px'],
y=self.ed25519['Py'])  # Generator
        R = Base.scalarmult(r_int)  #G.mult(r_int)

        # 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. -
> S = (r + SHA-512(R || public_key || M) * s) mod n.
        # The octet string S is the encoding of the resultant integer.
        R_bytes = R.to_bytes()
        Q = public_key
        M = message

        h = self.get_hash(R_bytes + Q + M)

        h_int = int.from_bytes(h, "little")
        s = priv_key_hashed[:32]  # s = hdigest1
        #s_int = int.from_bytes(s, "little")
        s_int = bytes_to_clamped_scalar(s)

        S = (r_int + h_int * s_int) % self.ed25519['L']

        S_bytes = int(S).to_bytes(32, "little")

        # 5. Form the signature as the concatenation of the octet
strings R and S.
        signature = R_bytes + S_bytes

        return signature

    def verify(self, message: bytes, signature: bytes, public_key:
bytes) -> bool:
        # 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”.
        R = signature[:32]
        S = signature[32:]
        s = int.from_bytes(signature[32:], "little")
        assert 0 <= s < self.ed25519['L']

        # 2. Form the bit string HashData as the concatenation of the
octet strings R, Q, and M (i.e., HashData = R || Q || M).
        Q = public_key
        M = message
        HashData = R + Q + M

        # 3. Using the established hash function or XOF,
        # 3.1 For Ed25519, compute digest = SHA-512(HashData).
        # Interpret digest as a little-endian integer t.
        digest = self.get_hash(HashData)
        t = int.from_bytes(digest, "little")

        # 4. Check that the verification equation [2^c * S]G = [2^c]R
+ (2^c * t)Q. Output “reject” if verification fails; output “accept”
otherwise.
        G = EdPoint(curve=self.E, x=self.ed25519['Px'],
y=self.ed25519['Py'])  # Generator
        #R = EdPoint.decode(R)
        #Q = EdPoint.decode(Q)
        h = self.ed25519['h']  # 2^c = h = 8 (Ed25519)

        left = Base.scalarmult(h * s)

        right =
((bytes_to_element(R)).scalarmult(h)).add((bytes_to_element(Q)).scalar
mult(h * t))

        verification = left == right

        return verification

    @staticmethod
    def get_hash(message):
        h = sha512(message).digest()
        return h
```

Testes e exemplos

De seguida são apresentados alguns exemplos de utilização da classe EdCDSA25519:

- Exemplo 1: Geração de chaves e verificação das mesmas
- Exemplo 2: Geração de chaves e demonstração das propriedades destas
- Exemplo 3: Geração de assinatura e teste da mesma
- Exemplo 4: Geração de assinatura e verificação da mesma
- Exemplo 5: Demonstração do protocolo de autenticação com base na transformada de Fiat-Shamir

Exemplo 1 - Geração de chaves e verificação das mesmas

```
E = EdCDSA25519()

print("Is an edwards curve?", E.E.is_edwards(E.ed25519['Px'],
E.ed25519['Py']))

priv_key, pub_key = E.generate_keys()

from pure25519.eddsa import publickey

real_pub_key = publickey(priv_key)

assert real_pub_key == pub_key

pub_key_example =
bytes.fromhex("1972E03EDC718B87CC6E141B1C745E115CE8895C96CBF1037DA8EA2
E4C8CCE92")

#pub_key = EdPoint.decode(pub_key_example)
#assert EdPoint.decode(pub_key_example) == pub_key
```

```
Is an edwards curve? True
```

Exemplo 2 - Geração de chaves e demonstração das propriedades destas

```
E = EdCDSA25519()

priv_key, pub_key = E.generate_keys()

print("Private Key:", priv_key.hex())
print(f"Public Key: {pub_key.hex()}")

assert EdPoint.encode(EdPoint.decode(pub_key)) == pub_key
```

```
Private Key:
93823c7d5225d579833fb8bd8ecba81fbf18ae987d0b3ffaf74c4f6df1e2a729
Public Key:
8d3609e9b7165df49133cd2989162819d1b2f23a7b8f6c5e0cafb26d43d8f67c
```

Exemplo 3 - Geração de assinatura e teste da mesma

```
m1 = bytes("Hello World", "utf-8")
m2 = os.urandom(16)
```

```
m2_signature = E.sign(m2, priv_key, pub_key)

print("Message:       ", m2)
print("Signature:     ", m2_signature)

import pure25519.eddsa as eddsa

real_signature = eddsa.signature(m2, priv_key, pub_key)

assert real_signature == m2_signature
```

```
Message:        b'\xb4\x04(\xd0\x00\xfa\xc9\xafj\x9c\xe9\x17\xb4@\xab\
xac'
Signature:      b'\xc8\x01\xdf9\xaaT\xe4\x98\xf9\xbfq\t.\xa0\xe9\xe71\
x00m\xc3i\x1a&\xdc\xc2\xaf\xcb\xac\x7f\x1d\x08y!\xf1\xd8\xbe\x85\xe0\
x99au\x0cm\xcco\xa3&\x92p\xfa\xaf\xa0\x7f\x9f\xe0\xa9\xc9\x98\x9eI\
xd4/\x8d\x05'
```

Exemplo 4 - Geração de assinatura e verificação da mesma

```
print("Is the signature valid?", "Yes!" if E.verify(m2, m2_signature,
pub_key) else "No!!!")
```

```
Is the signature valid? Yes!
```

Exemplo 5 - Demonstração do protocolo de autenticação com base na transformada de Fiat-Shamir

```
from pure25519 import basic

alice_priv_key, alice_pub_key = E.generate_keys()
bob_priv_key, bob_pub_key = E.generate_keys()

# Alice wants to prove to Bob that she knows the private key
associated with her public key
r = os.urandom(32)
# Alice generates a random nonce r, computes t = r*G
r_int = int.from_bytes(r, "little")
t = Base.scalarmult(r_int)
t_bytes = t.to_bytes()
# Alice computes the challenge c = H(Base || public_key || t)
base_bytes = basic.encodepoint((Base.XYTZ[0], Base.XYTZ[1]))
c = E.get_hash(base_bytes + alice_pub_key + t_bytes)

# Alice computes the response s = (r - c*private_key) % L
c_int = int.from_bytes(c, "little")
priv_key_int = int.from_bytes(alice_priv_key, "little")
L = EdCDSA25519.ed25519['L']
s = (r_int - c_int * priv_key_int) % L
```

```
# Alice sends the point t and the integer s to Bob
# Bob verifies the signature by computing c1 = H(Base || public_key ||
t) and verifying that t = s*G + c1*public_key
c1 = E.get_hash(base_bytes + alice_pub_key + t_bytes)
c1_int = int.from_bytes(c1, "little")
left = t_bytes
right =
(Base.scalarmult(s)).add(bytes_to_element(alice_pub_key).scalarmult(c1
_int)).to_bytes()
print("Does Alice know the private key associated with her public key?
", "Yes!" if left == right else "No!!!")

# FIXME: The following code is not working
```

```
Does Alice know the private key associated with her public key?  No!!!
```