Estruturas Criptográficas 2022/23

TP2. Problema 2

Grupo 7. Leonardo Berteotti e Paulo R. Pereira

Pretende-se a construção de uma classe Python que implemente o EdCDSA a partir do “standard” FIPS186-5. A implementação deve conter funções para assinar digitalmente e verificar a assinatura e deve usar uma das “Twisted Edwards Curves” definidas no standard e escolhida na iniciação da classe: a curva “edwards25519” ou “edwards448”. Por fim e por aplicação da transformação de Fiat-Shamir, deve ser construido um protocolo de autenticação de desafio-resposta.

In [54]:

```
import hashlib, os
from pickle import dumps
from sage.all import *
```

Classe que implementa a curva de Edwards: A classe Ed, fornecida pelo o docente, tem como função transformar curvas ED2556 e ED448 na curva elíptica de edwards bi-racional Curve25519 e Curve448, respetivamente. Desta forma é possivel que apartir dos parâmetros da curva ED2556 ou ED448 criar uma curva eliptica isomórfica.

In [55]:

```
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
        self.EC = EllipticCurve(K,[a4,a6])

        if ed != 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 cofato
        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 de implementação dos métodos dos pontos de edwards: A classe ed, também fornecida pelo docente, tem como função implementar funções de multiplicação entre inteiros e pontos (mult), soma entre dois pontos (soma) e igualdade entre dois pontos (eq).

In [56]:

```
class ed(object):
    def __init__(self,pt=None,curve=None,x=None,y=None):
        if pt != 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 ed(curve=self.curve, x=self.x, y=self.y)

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

    def sim(self):
        return ed(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.
        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
        self.w = self.x*self.y

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

Classe que implementa as assinaturas EdDSA: São implementadas os 3 métodos dos algoritmos de assinatura: a geração de chaves (keyGen), assinatura (signature) e verificação da assinatura (verify). Na inicilização da classe é definido qual o algoritmo a ser usado entre Ed25519 ou Ed448.

Encoding e Decoding: Os pontos gerados pelas operações vão ser armazenados num array *storage* para posteriormente serem utilizados. Este armazenamento funciona como "compressão" de um ponto e utiliza a função *encoding* que armazena um ponto P no indíce i do array. Para decompressão do ponto utilizamos o decoding onde através de um índice retornamos o ponto armazenado nessa posição.

Geração de chaves: Começa-se por gerar a chave privada a partir de uma função que gera 32 bytes ou 57 bytes, dependendo da curva, de forma pseudo-aleatória. É calculado o *hash* desta chave nas curvas Ed25519 com o SHA-512 ou nas curvas Ed448 com o SHAKE256. O valor do *hash* é depois processadp a partir da função s_value, originando o valor de s a ser múltiplicado pelo ponto G (parâmetro da curva). A chave pública será o valor x do ponto gerado.

Assinatura: Começa-se por determinar o *hash* da chave privada que será usada juntamente com a mensagem para determinar o valor r. Este valor vai ser multiplicado pelo ponto G sendo o resultado igual ao ponto R. De forma a determinar o valor S, utiliza-se a expressão *S = (r + SHA-512(R || Q || M) * s) mod n* no caso da curva Ed25519 ou a expressão *S = (r + SHAKE256(dom4(0, context) || R || Q || M, 912) * s) mod n* no caso da curva de Ed448, com n igual à ordem da curva de edwards, $s = s_value$, Q igual à chave privada e M à mensagem. A assinatura será a *octet string* $R \,||\, S$.

Verificação: Começa-se por verificar se os valores R, S e Q são válidos. De seguida, determina-se o valor de t através da hash da string $R||Q||M$ no caso da curva Ed25519 ou $dom4(0, context)||R||Q||M$ no caso da curva Ed448. A condição para que a mensagem seja autentica é dada por $[2^c * S]G == [2^c]R + (2^c * t)Q$.

In [57]:

```
class EdDSA:
    storage = []

    def __init__(self, ed):
        if(ed=='ed25519'):
            print('Escolhida a curva Ed25519.')
            self.setup_ed25519()
        else:
            print('Escolhida a curva Ed448.')
            self.setup_ed448()

    def setup_ed25519(self):
        p = 2**255-19
        K = GF(p)
        a = K(-1)
        d = -K(121665)/K(121666)
        #

        ed25519 = {
        'b'  : 256,
        'Px' : K(151122213495354007725011514095885315114540126930418572060461
        'Py' : K(463168356949264781694283940034751631413079938662562256157830
        'L'  : ZZ(2**252 + 27742317777372353535851937790883648493), ## ordem
        'n'  : 254,
        'h'  : 8
        }

        Px = ed25519['Px']; Py = ed25519['Py']

        E = Ed(p,a,d,ed=ed25519)
        G = ed(curve=E,x=Px,y=Py)
        l = E.order()[0]

        self.b = ed25519['b']
        self.requested_security_strength = 128
        self.E = E
        self.G = G
        self.l = l
        self.algorithm = 'ed25519'


    def setup_ed448(self):
        p = 2**448 - 2**224 - 1
        K = GF(p)
        a = K(1)
        d = K(-39081)

        ed448= {
        'b'  : 456,     ## tamanho das assinaturas e das chaves públicas
        'Px' : K(224580040295924300187604334099896036246789641632564134246125
        'Py' : K(298819210078481492676017930443930673437544040154080242095928
        'L'  : ZZ(2**446 - 138180668098951153520073867485154268803366924748821
        'n'  : 447,     ## tamanho dos segredos: os dois primeiros bits são (
        'h'  : 4        ## cofactor
        }

        Px = ed448['Px']; Py = ed448['Py']
```

```
        E = Ed(p,a,d, ed=ed448)
        G = ed(curve=E,x=Px,y=Py)
        l = E.order()[0]

        self.b = ed448['b']
        self.requested_security_strength = 224
        self.E = E
        self.G = G
        self.l = l
        self.algorithm = 'ed448'

    # hash function for each curve ED2556 and ED448
    def hash(self,data):
        if self.algorithm == 'ed25519':
            return hashlib.sha512(data).digest()
        else:
            return hashlib.shake_256(data).digest(912//8)

    # private key digest
    def digest(self,d):
        h = self.hash(d)
        buffer = bytearray(h)
        return buffer

    def s_value(self,h):
        if self.algorithm == 'ed25519':
            return self.s_value_ed25519(h)
        else:
            return self.s_value_ed448(h)

    def s_value_ed25519(self,h):
        digest = int.from_bytes(h, 'little')
        bits = [int(digit) for digit in list(ZZ(digest).binary())]
        x = 512 - len(bits)
        while x != 0:
            bits = [0] + bits
            x = x-1

        bits[0] = bits[1] = bits[2] = 0
        bits[self.b-2] = 1
        bits[self.b-1] = 0

        bits = "".join(map(str, bits))

        s = int(bits[::-1], 2)
        return s

    def s_value_ed448(self,h):
        digest = int.from_bytes(h, 'little')
        bits = [int(digit) for digit in list(ZZ(digest).binary())]
        x = 512 - len(bits)
        while x != 0:
            bits = [0] + bits
            x = x-1

        bits[0] = bits[1] = 0
        bits[self.b-9] = 1
        for i in bits[self.b-8:self.b]:
```

```
            bits[i] = 0

        bits = "".join(map(str, bits))

        s = int(bits[::-1], 2)
        return s

    # point encoding
    def encoding(self,Q, n):
        x, y = Q.x, Q.y
        self.storage.insert(n,(x,y))
        return x

    # point decoding
    def decoding(self,n):
        Q = self.storage[n]
        return Q

    # KeyGen
    def keyGen(self):
        # private key
        d = os.urandom(self.b//8)
        # s value
        digest = self.digest(d)
        if self.algorithm == 'ed25519':
            bytes_length = 32
        else:
            bytes_length = 57

        hdigest1 = digest[:bytes_length]
        s = self.s_value(hdigest1)

        # public key
        T = self.G.mult(s)

        # public key encoding
        Q = self.encoding(T,0)
        Q = int(Q).to_bytes(bytes_length, 'little')
        return d, Q

    # domain separation tag
    def dom4(self, f, context):
        init_string = []
        context_octets = []

        for c in context:
            context_octets.append(format(ord(c), "08b"))
        context_octets = ''.join(context_octets)

        for c in "SigEd448":
            init_string.append(format(ord(c), "08b"))
        init_string = ''.join(init_string)

        bits_int = int(init_string + format(f, "08b") + format(len(context_oc
        byte_array = bits_int.to_bytes((bits_int.bit_length() + 7) // 8, 'lit

        return byte_array

    # Sign
```

```
    def sign(self,M,d,Q,context = ''):
        # private key hash
        digest = self.digest(d)

        if self.algorithm == 'ed25519':
            bytes_length = 32
            hashPK = digest[bytes_length:]
            hashPK_old = digest[:bytes_length]
            r = self.hash(hashPK+M)
        else:
            bytes_length = 57
            hashPK = digest[bytes_length:]
            hashPK_old = digest[:bytes_length]
            r = self.hash(self.dom4(0, context)+hashPK+M)

        # r value
        r = int.from_bytes(r, 'little')

        # calculate R and encoding it
        R = self.G.mult(r)
        Rx = self.encoding(R,1)
        R = int(Rx).to_bytes(bytes_length, 'little')

        # s value
        s = self.s_value(hashPK_old)

        if self.algorithm == 'ed25519':
            # (R || Q || M) hash
            hashString = self.hash(R+Q+M)
        else:
            # (dom4(0,context) || R || Q || M) hash
            hashString = self.hash(self.dom4(0, context)+R+Q+M)

        hashString = int.from_bytes(hashString, 'little')

        # S = (r + SHA-512(R || Q || M) * s) mod n
        S = mod(r + hashString * s,self.l)
        S = int(S).to_bytes(bytes_length, 'little')

        signature = R + S
        return signature

    # Verify
    def verify(self,M,A,Q, context = ''):
        if self.algorithm == 'ed25519':
            bytes_length = 32
        else:
            bytes_length = 57

        # get R and S from signature A
        R = A[:bytes_length]
        S = A[bytes_length:]
        s = int.from_bytes(S, 'little')

        # decoding S, R and Q
        if (s >= 0 and s < self.l):
            (Rx, Ry) = self.decoding(1)
            (Qx, Qy) = self.decoding(0)
            if(Rx != None and Qx != None):
```

```
            res = True
        else: return False
    else: return False

    # t value
    if self.algorithm == 'ed25519':
        digest = self.hash(R+Q+M)
    else:
        digest = self.hash(self.dom4(0, context)+R+Q+M)

    t = int.from_bytes(digest, 'little')

    # get variables for verifying process
    value = 2**3
    R = int.from_bytes(R, 'little')
    Q = int.from_bytes(Q, 'little')
    R = ed(curve=self.E,x=Rx,y=Ry)
    Q = ed(curve=self.E,x=Qx,y=Qy)

    # get verification conditions: [2**c * S]G == [2**c]R + (2**c * t)Q
    cond1 = self.G.mult(value*s)
    cond2 = R.mult(value)
    cond3 = Q.mult(value*t)
    cond2.soma(cond3)

    # final verification
    return cond1.eq(cond2)
```

Exemplo de teste Ed448: Este algoritmo começa por gerar o par de chaves necessário para a assinatura da mensagem (generateKeys). Assumindo que vamos assinar a mensagem 1, esta é utilizada juntamente com a chave privada e pública para gerar a assinutura (signature). Esta assinatura será utilizada juntamente com a mensagem 1 e a chave pública para verificar a autenticidade da mensagem (verify). Caso a verificação da autenticidade receba uma mensagem diferente da 1 (exemplo mensagem 2) então o resultado da verificação deve dizer que a mensagem não é autêntica. De salientar que a assinatura e a verificação precisam de um contexto.

In [58]:

```
edDSA = EdDSA('ed448')
signed_message = "Esta mensagem está assinada!"
unsigned_message = "Esta mensagem não está assinada..."
print("Mensagem a ser assinada: " + signed_message)
privateKey, publicKey = edDSA.keyGen()
print("\nSK: ")
print(privateKey)
print("PK: ")
print(publicKey)
print()
assinatura = edDSA.sign(dumps(signed_message), privateKey, publicKey, 'contex
print("Assinatura: ")
print(assinatura)
print()
print("Verificação da autenticação da mensagem assinada:")
if edDSA.verify(dumps(signed_message), assinatura, publicKey, 'contexto')==Tr
    print("Mensagem autenticada!")
else:
    print("Mensagem não autenticada...")

print()
print("Verificação da autenticação da mensagem não assinada:")
if edDSA.verify(dumps(unsigned_message), assinatura, publicKey, 'contexto')==
    print("Mensagem autenticada!")
else:
    print("Mensagem não autenticada...")
```

```
Escolhida a curva Ed448.
Mensagem a ser assinada: Esta mensagem está assinada!

SK:
b' o\xe3r\xcbi\xa9\xc0R\x93\xb2\xd6`>\xcf,-\x87\xba\x1f\xf4T\xa
d%\xe1\x19L\x16\xdcx\x08\xb1\x81\x85\x83\x16\xb34<\x9a\x02D\x8c
\x8bN(\xe4\xe9\xfc\x00\x8do\xd0mQ\xddc'
PK:
b"\x1c\xcetLB\xe7\xc5\xeal\x87eC\x8e\xc7\x800@t\xe9\x00;'\xf3;|
J\x91\x82g`\x9fN\xec\xd1\xcf\xba\xe0[{\x13\xf2\x05\x06.3\x01\x0
3\xb1\x0bh\x1d@\xb8wQ\x19\x00"

Assinatura:
b'\x84m\x19\xc8\xc9p\xea\xb6\x87\xa9\x9c\x06kZ*\xcd\xa7q\t\xf6
\xb8U\x96\x1a\xa2*{\xf7[\x92\x9cbX\x10\r\xe5\xb2\x9e\xd5\xe6\x8
2\xc1\xcbu\xc2\xc8SY\xfe?\xa6\x1a\xc9\xc4\x9f\x8e\x00rknVM\xa1>
\xe4gl\x80\xfd\xf0\xbe\xf7\x80]\xca[4\xd8Y\xbd\xd9\xeeh\x9b\xb5
\xfc\x17\xf7\x1f\x8f\x16\xdc\t\x1d\xecc@H?\xf4j\x85\x0b\x8c\xd4
\xf38\x9f\xb32\x85r6\x00'

Verificação da autenticação da mensagem assinada:
Mensagem autenticada!

Verificação da autenticação da mensagem não assinada:
Mensagem não autenticada...
```

Exemplo de teste Ed25519: Este algoritmo é idêntico ao de cima na forma de gerar as chaves, assinar e verificar, mas não necessita de um contexto.

In [59]:

```
edDSA = EdDSA('ed25519')
signed_message = "Esta mensagem está assinada!"
unsigned_message = "Esta mensagem não está assinada..."
print("Mensagem a ser assinada: " + signed_message)
privateKey, publicKey = edDSA.keyGen()
print("\nSK: ")
print(privateKey)
print("PK: ")
print(publicKey)
print()
assinatura = edDSA.sign(dumps(signed_message), privateKey, publicKey)
print("Assinatura: ")
print(assinatura)
print()
print("Verificação da autenticação da mensagem assinada:")
if edDSA.verify(dumps(signed_message), assinatura, publicKey)==True:
    print("Mensagem autenticada!")
else:
    print("Mensagem não autenticada...")

print()
print("Verificação da autenticação da mensagem não assinada:")
if edDSA.verify(dumps(unsigned_message), assinatura, publicKey)==True:
    print("Mensagem autenticada!")
else:
    print("Mensagem não autenticada...")
```

```
Escolhida a curva Ed25519.
Mensagem a ser assinada: Esta mensagem está assinada!

SK:
b'\xa0\xc5\x11W\xb7\xfb \x15\x97M\x0b|\xe9$\xc3\xc1\xc5<\xeb\xb
1\xca9\x99\x83L\xf5\x82\x87Z^\xa2\xd7'
PK:
b'\xe5<\xa7b\xe8xG\x0b\x7f\xa1\x05h}r*\x06\x1c\xd4\xb6\xda\x85=
\xb5\xcb\x9ak\x88\x0cv=\x98E'

Assinatura:
b'\x89lZ\xca\x0cP\xee_\x7f\x82Y\x9aa\xb8d3\xc1\x1f\xc1%\x07\xd9
\xa3\x1b\xc1\x0c\x9ca\x81M\xa0(`\xab`f\xe2{\xc4l\x14\xeb[7\xe3
\x84\xe5\tj\xd2\xc1r\x08(\xcd\xe7hew|\x17O\x0b\x0c'

Verificação da autenticação da mensagem assinada:
Mensagem autenticada!

Verificação da autenticação da mensagem não assinada:
Mensagem não autenticada...
```