TP2_Exercicio1

April 2, 2024

1 TP2 - Exercício 1

1.0.1 Autores

Afonso Ferreira - pg52669
Tiago Rodrigues - pg52705

1.0.2 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".

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

#Decode a hexadecimal string representation of the integer.
def hexi(s): return int.from_bytes(bytes.fromhex(s),byteorder="big")
```

Começamos por implementar a classe da curva de Edwards

[ ]:
```
# Classe que implementa a curva de Edwards
class EdwardsCurve(object):
    def __init__(self,p, a, d , ed): # se a = 1 entao a curva é de Edwards␣
 ↪normal e não "twisted"
        assert a != d and is_prime(p) and p > 3
        K = GF(p) # campo finito de p elementos
        self.K = K
        self.constants = {'a': a , 'd': d }
        self.l = ed['l']

    # Verifica se um ponto (x,y) pertence à curva de Edwards
    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 # copiar do notebook ax2+y2=1+dx2y2
```

De seguida, passamos à criação da classe dos pontos de Edwards

[ ]:
```
# Classe de implementação dos métodos dos pontos de edwards
class EdwardsPoint(object):
    def __init__(self,pt=None,curve=None,x=None,y=None,w=None):
        if pt != None:
            self.curve = pt.curve
            self.x = pt.x ; self.y = pt.y ; self.w = pt.w
        else:
            assert isinstance(curve,EdwardsCurve) and curve.is_edwards(x,y) #␣
 ↪verificar se o ponto pertence à curva
            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 EdwardsPoint(curve=self.curve, x=self.x, y=self.y)

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

    # Método de soma de dois pontos de Edwards
    def soma(self, other):
        #The formulas are from EFD.
        a = self.curve.constants['a']; d = self.curve.constants['d']
        # delta = d*w1*w2
        delta = d*self.w*other.w
        # x = (x1y2+y1x2)/(1+delta) e y = (y1y2-ax1x2)/(1-delta)
        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

    # Método de duplicação de um ponto de Edwards
    def duplica(self):
        #The formulas are from EFD (with assumption a=-1 propagated).
        a = self.curve.constants['a']; d = self.curve.constants['d']
        # delta = d*w1^2
        delta = d*(self.w)**2
        # x = 2w1/(1+delta) e y = (y1^2-ax1^2)/(1-delta)
        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
```

Temos agora que criar as classes que implementam o algoritmo em específico, tanto o ed25519 e o ed448

No ed25519:

[ ]:
```
class Ed25519:
    def __init__(self):
        p = 2**255-19 # The prime field
        K = GF(p)
        a = K(-1)
        d = -K(121665)/K(121666)

        ed25519 = {
        'b'  : 256,     # The coding length
        'Px' : K(hexi("216936D3CD6E53FEC0A4E231FDD6DC5C692CC76"+\
        "09525A7B2C9562D608F25D51A")),
        'Py' : K(hexi("6666666666666666666666666666666666666666"+\
        "666666666666666666666658")),
        'l'  : ZZ(hexi("1000000000000000000000000000000014def9dea2f79cd" +
                       "65812631a5cf5d3ed")), ## ordem do subgrupo primo
        'n'  : 254,     # The highest set bit
        'c'  : 3        # The logarithm of cofactor.
        }

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

        E = EdwardsCurve(p,a,d,ed=ed25519)
        B = EdwardsPoint(curve=E,x=Px,y=Py)

        self.b = ed25519['b']
        self.requested_security_strength = 128
        self.E = E
        self.B = B
        self.l = ed25519['l']
        self.n = ed25519['n']
        self.c = ed25519['c']
        self.algorithm = 'ed25519'

    def clamp(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
```

Ed448:

[ ]:
```
class Ed448:
    def __init__(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(hexi("4F1970C66BED0DED221D15A622BF36DA9E14657" +
                      "0470F1767EA6DE324A3D3A46412AE1AF72AB66511433B" +
                      "80E18B00938E2626A82BC70CC05E")) ,
        'Py' : K(hexi("693F46716EB6BC248876203756C9C7624BEA737" +
                      "36CA3984087789C1E05A0C2D73AD3FF1CE67C39C4FDBD" +
                      "132C4ED7C8AD9808795BF230FA14")) ,
        'l'  : ZZ(hexi("3fffffffffffffffffffffffffffffffffffffffffffffff" +
                    "ffffffff7cca23e9c44edb49aed63690216cc2728dc58f552378c2" +
                    "92ab5844f3")) ,
        'n'  : 447,     ## tamanho dos segredos: os dois primeiros bits são 0 e␣
 ↪o último é 1.
        'c'  : 2        # The logarithm of cofactor.
        }

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

        E = EdwardsCurve(p,a,d, ed=ed448)
        B = EdwardsPoint(curve=E,x=Px,y=Py)

        self.b = ed448['b']
        self.requested_security_strength = 224
        self.E = E
```

```
        self.B = B
        self.l = ed448['l']
        self.n = ed448['n']
        self.c = ed448['c']
        self.algorithm = 'ed448'

    def clamp(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

    # 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_octets), "08b") + context_octets, 2)
        byte_array = bits_int.to_bytes((bits_int.bit_length() + 7) // 8,␣
↪'little')

        return byte_array
```

Classe que implementa as assinaturas e os métodos pedidos:

[ ]:
```python
# Classe que implementa as assinaturas EdDSA
class EdDSA:
    storage = []

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

    # hash function for each curve ED2556 and ED448
    def hash(self,data):
        if self.Ed.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

    # 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
    # como no eddsa fornecido pelo professor
    def keyGen(self):
        bytes_length = self.Ed.b//8
        # private key
        priv = os.urandom(bytes_length)

        khash = self.digest(priv)

        a = self.Ed.clamp(khash[:bytes_length])
```

```
        # public key
        T = self.Ed.B.mult(a)

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

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

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

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

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

        # s value
        s = self.Ed.clamp(hashPK_old)

        if self.Ed.algorithm == 'ed25519':
            # (R || Q || M) hash
            hashString = self.hash(R+Q+M)
        else:
            # (dom4(0,context) || R || Q || M) hash
            hashString = self.hash(self.Ed.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.Ed.l)
        S = int(S).to_bytes(bytes_length, 'little')
```

```
        signature = R + S
        return signature

    # Verify
    def verify(self,M,A,Q, context = ''):
        bytes_length = self.Ed.b//8

        # 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.Ed.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.Ed.algorithm == 'ed25519':
            digest = self.hash(R+Q+M)
        else:
            digest = self.hash(self.Ed.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 = EdwardsPoint(curve=self.Ed.E,x=Rx,y=Ry)
        Q = EdwardsPoint(curve=self.Ed.E,x=Qx,y=Qy)

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

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

1.1 Testes

[ ]:
```
edDSA = EdDSA('ed448')
signed_message = "Esta é uma mensagem assinada!"
unsigned_message = "Esta não está assinada..."
print("Mensagem a ser assinada: " + signed_message)
privateKey, publicKey = edDSA.keyGen()
print("\nSecret Key: ")
print(privateKey)
print("Public Key: ")
print(publicKey)
print()
assinatura = edDSA.sign(dumps(signed_message), privateKey, publicKey,␣
 ↪'contexto')
print("Assinatura: ")
print(assinatura)
print()
print("Verificação da mensagem assinada:")
if edDSA.verify(dumps(signed_message), assinatura, publicKey, 'contexto')==True:
    print("Mensagem autenticada!")
else:
    print("Mensagem não autenticada...")

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

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

Secret Key:
b'\xa9\xda]\xee1Ee\xd8\xed\x18\tim\xdd\xe1S\x86j[X\xee\xadj\x00+\xf9/\x11\xa5\xf
6a\x97\xa3g\x1a\xce\xba\xa3L\xcd:\x93w\x9c0\x03/j\x9c\xea\n`\x84,\x06\xf3\xf2'
Public Key:
b'\xd4C~\xd0l\x03\xc6(\xe6\\\xa7\xf6\x0f\x9d\xb2\xa6\xa1\x9b\xd8\xc4\x0b\xe8\x98
F\xb2\xa5\xf9\x9dfW\xf6\xa4\xfd@KD|pw\xc6=n\xe0\xf7\xcd\x1d\x85)(x\x90\xd2?\xb3i
\xdc\x00'

Assinatura:
b"p\xb3\xa9}\xe4M\x1e,7\xbd\x05!\x01,\x9e\xd6\x0bz!L\x81L\xda\x9c\x85\xfcF\xb4\x
b6\xd75\x12\x03\x01\x9d\xae\x9fb\xc8Z\xcd\xe0L\x12\xb7\xb5[\xab\xd7%F\xc3\xd2\x8
5\xee\xae\x00c'\x9d\x0c\x95o\xea8\x8f\xea#\x0c\xa2\xa3\xdb\xf9\x9ei\xcaRM\xac\x8
4tP\x7f\xd4\xe9\xccu\xf7\xc6Y\xd5\xfa\xd3\xf7\x1a)\x82j\x1c0\xa9\xf9_l\xeaq\xf4E
\xab`4=\x03\x00"
```

```
Verificação da mensagem assinada:
Mensagem autenticada!

Verificação da mensagem não assinada:
Mensagem não autenticada…
```

[ ]:
```python
edDSA = EdDSA('ed25519')
signed_message = "Esta é uma mensagem assinada!"
unsigned_message = "Esta não está assinada..."
print("Mensagem a ser assinada: " + signed_message)
privateKey, publicKey = edDSA.keyGen()
print("\nSecret Key: ")
print(privateKey)
print("Public Key: ")
print(publicKey)
print()
assinatura = edDSA.sign(dumps(signed_message), privateKey, publicKey)
print("Assinatura: ")
print(assinatura)
print()
print("Verificação da mensagem assinada:")
if edDSA.verify(dumps(signed_message), assinatura, publicKey)==True:
    print("Mensagem autenticada!")
else:
    print("Mensagem não autenticada...")

print()
print("Verificação 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 é uma mensagem assinada!

Secret Key:
b'\xd7cs\xaf\xe2\x84\xc4\xe9\x9c`\x1f\xa6\xcbX\x0bf\xea2\x08;\xach\xd7\x16\xaanC
a\xea2*w'
Public Key:
b'\xb3\x1e\xdeI"\tA\x165`\xc5A6\xc0\x0e\x0c\xfcJ\xfe\xe4\xd7SD\\D\x94\xd9rP\t\x0
2\x15'

Assinatura:
b'B\xc2\xfb\xb6\xa4N\x1c\t\x82*\xa4M\x85i=\xda\x03\xf4\x90\x1c\xe6\x0b@5\x8a\xa1
\xd8\x08\x94FBm\xe3^h*$\x80%iE\x19M"\x0f\xf2~c\xbay\x97#\xf5\x1c\xf5\xe0&\x84\x9
5\xfda\x82v\x03'
```

```
Verificação da mensagem assinada:
Mensagem autenticada!

Verificação da mensagem não assinada:
Mensagem não autenticada…
```