Trabalho Prático 1 - Grupo 15

João Gonçalves - pg46535

Sara Queirós - pg47661

Exercício 3

1. Use o Sagemath para
   A. Construir uma classe Python que implemente o EdCDSA 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".

In [ ]:
```
import os
import hashlib
from sage.crypto.util import ascii_to_bin, bin_to_ascii
```

Edwards-curve Digital Signature Algorithm é uma algoritmo de assinatura digital que se baseia nas curvas de Edwards. No âmbito deste exercício será utilizada a curva "edwards25519"

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

ed25519 = {
'b'  : 256,
'Px' : K(151122213495354007725011514095885315114540126930418572060461132839
'Py' : K(463168356949264781694283940034751631413079938662562256157830336031
'L'  : ZZ(2^252 + 27742317777372353535851937790883648493), ## ordem do subgr
'n'  : 254,
'h'  : 8
}
```

Para gerar os parâmetros, foram seguidos os passos indicados partir do "standard" FIPS186-5.

- Gerar a chave privada
- Gerar a chave pública
- Gerar a assinatura
- Verificar a assinatura

Para Ed25519:

*Encoding : Por ser little endian , primeiro codifica-se a string de 32 octetos da coordenada y. O bit mais significante do octeto final é sempre 0. Para formar o ponto de encoding , copia-se o bit menos significativo de x para o mais significativo no último octeto

Geração de pares de chaves

chaves publicas -> b bits exatamente

assinaturas -> 2b bits exatamente

b é múltiplo de 8

b = 256 para Ed25519, logo a chave privada tem 32 octetos 128 bits de segurança

In [ ]:
```
class Ed(object):
    def __init__(self,p, a, d , ed = None):
        #Verificar se 'a' e 'd' não são iguais e p é primo
        assert a != d and is_prime(p) and p > 3
        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.b=ed['b']
        self.constants = {'a': a , 'd': d , 'A':A , 'B':B , 'alfa':alfa , 's
        self.EC = EllipticCurve(K,[a4,a6])
        self.n=ed['n']

        if ed != None:
            self.L = ed['L']
            #Adquirir o ponto P da curva
            self.P = self.ed2ec(ed['Px'],ed['Py'])
        else:
            self.gen()


    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 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 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(encode

    #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 priva
        self.private_key = H(os.urandom(32))
            #3. Calcular e modificar o hdigest1: 3 primeiros bits a 0, ultim
            #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(
            #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


    #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


    #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)



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))
def H(m):
        return hashlib.sha512(m).digest()

#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
```

Para testar as funções basta:

In [ ]:
```
E = Ed(p,a,d,ed25519)
E.generate_public_key()
R,s=E.sign(b'Isto e de teste')
E.verify(b'Isto e de teste',R,s)
```

```
False
```

Devido a não estarem definidas as funções de soma e multiplicação de pontos, não obtemos os resultados que eram esperados.