TP1-Problema3

March 28, 2022

1 TRABALHO PRÁTICO 1 - GRUPO 14

1.1 Problema 3

Neste problema era proposto a implementação de um algoritmo de assinatura de mensagens que usa as *twisted edwards curves*. Para isso implementou-se o algoritmo EdDSA apartir do paper RFC8032, sendo que as curvas escolhidas para esta implementação foram as ED2556. O primeiro passo consistiu em usar a classe `Ed`, fornecida pelo o docnete, para transformar as curvas ED2556 na curva de edward biracional Curve2556. De seguida implementou-se o algoritmo de geração de chaves, assinatura e verificação na classe `EdDSA`. Para as operações entre pontos foi utilizada a classe `ed` que implementa métodos de soma, multiplicação, igualdade entre pontos na curva de edwards.

1.1.1 Resolução do Problema

Imports

[10]:
```
import hashlib, os
from pickle import dumps
from struct import *
```

Classe de implementação da curva de Edwards A classe `Ed`, fornecida pelo o docente, tem como função transformar curvas ED2556 na curva elíptica de edwards biracional Curve2556. Desta forma é possivel que apartir dos parâmetros da curva ED2556 criar uma curva eliptica isomórfica.

[11]:
```
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 != None:
            self.L = ed['L']
            self.P = self.ed2ec(ed['Px'],ed['Py'])
        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,x,y):         ## mapeia EC --> Ed
        #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`, fornecida pelo docente, tem como função implementar funções de multiplicação entre inteiros e pontos (`mult`), soma entre dois pontos (`soma`), igualdade entre dois pontos (`eq`) para as operações necessárias nas funções a serem implementadas pela classe EdDSA.

[12]:
```
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.
↪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)
        P = self.copy() ; A = self.zero()
        for b in m:
            if b == 1:
                A.soma(P)
            P.duplica()
        return A
```

Classe que implementa as assinaturas EdDSA Nesta classe são implementadas 3 funcionalidades base dos algoritmos de assinatura: a geração de chaves (`generateKey`), assinatura

(`signature`) e verificação da assinatura (`verify`).

Geração de chaves: Nesta função começa-se por gerar a chave privada apartir de uma função que gera 32 bytes de forma pseudo-aleatória. Esta chave é utilizada na função de hash usada nas curvas de ED2556, SHA-512 implementada pela função`hash512` e utilizada pela função `digest`. O valor desta hash é depois processada apartir da função de `s_value` , originando o valor de s a ser múltiplicado pelo ponto G que é dado como parâmetro nas curvas ED2556. A chave pública será o valor x do ponto gerado.

Assinatura: Nesta função utiliza-se a mensagem a assinar, a chave privada e a chave pública de forma gerar a assinatura respetiva. Começamos por determinar a hash da chave privada que será usada juntamente com a mensagem para determinar o valor r. Este valor vai ser multiplicado pelo o ponto G sendo o resultado igual ao ponto R. De forma a determinar o valor S utilizamos a expressão `S = (r + SHA-512(R || Q || M) * s) mod n, com n = ordem da curva de edwards, s = s_value, Q = chave privada e M = mensagem`. A assinatura será a *octet string* R || S.

Verificação: Como forma de verificar se a mensagem é autêntica, começamos por verificar se os valores R, S e Q são válidos segundo o *encoding* e *decoding* aplicados. De seguida vamos determinar o valor de t através da hash da string R || S || Q. A condição para que a mensagem seja autentica é dada por `[2^c * S]G == [2^c]R + (2^c * t)Q`

Para além destas funções principais também foi implementado um método de *encoding* e *decoding* de forma a conseguir comprimir e decomprimir, respetivamente, um ponto.

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` para que através de um ponto P e de um índice i armazene o ponto P no indíce i no array *storage*. Para decompressão do ponto utilizamos o `decoding` onde através de um índice retornamos o ponto armazenado no array *storage* nesse índice. Exemplo, a chave privada estará no índice 0, pois é o primeiro ponto a ser gerado, já o ponto R ao ser o segundo ponto a ser gerado estará no índice 1.

[13]:
```
#Array de armazenamento de pontos utilizados nos métodos
#   Q -> indice 0
#   R -> indice 1
storage = []

class EdDSA:

    #Função de inicialização das variaveis a usar nos métodos
    def __init__(self):
        self.E, self.G, self.b, self.l, self.p = self.setup()

    #Parâmetros das curvas de ED2556 - Transformação de ED2556 em Curve2556
    def setup(self):
        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'  : 2^3
        }

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

        E = Ed(p,a,d,ed=ed25519)
        b = ed25519['b']
        G = ed(curve=E,x=Bx,y=By)
        l = E.order()[0]

        return E, G, b, l, p

    #Função de hash a ser utilizada nas curvas de ED2556 - SHA-512
    def hash512(self,data):

        return hashlib.sha512(data).digest()

    #Função que determina a hash da chave privada
    def digest(self,d):

        h = self.hash512(d)

        buffer = bytearray(h)

        return buffer

    #Função que determina o valor s usado em várias operações dos métodos de␣
↪EdDSA
    def s_value(self,h):

        #Passar o valor recebido (octet string) para inteiro little endian
        digest = int.from_bytes(h, 'little')
        buffer = [int(digit) for digit in list(Z(digest).binary())]
        x = 512 - len(buffer)

        while x != 0:
            buffer = [0] + buffer
```

```
            x = x-1

        #Manipulação dos bits do inteiro segundo o algoritmo
        buffer[0] = buffer[1] = buffer[2] = 0
        buffer[self.b-2] = 1
        buffer[self.b-1] = 0

        #Junção de todos os bits
        buffer = "".join(map(str, buffer))

        #valor em inteiro em formato little endian
        s = int(buffer[::-1], 2)

        return s

    #Função de encoding - compressão de ponto
    def encoding(self,Q, n):

        x, y = Q.x, Q.y

        #armazena ponto completo no indice que recebeu
        storage.insert(n,(x,y))

        return x

    #Função de decoding - descompressão de ponto
    def decoding(self,n):

        #devolve o ponto que estava no índice que recebeu como parâmetro
        Q = storage[n]

        return Q

    #Funçaõ de geração do par de chaves
    def generateKeys(self):

        #Gerar a private key
        d = os.urandom(self.b//8)

        #Geração do valor s
        digest = self.digest(d)
        s = self.s_value(digest[:32])

        #Gerar a chave pública
        T = self.G.mult(s)

        #Compressão da chave publica - passagem para octet string
```

```
        Q = self.encoding(T,0)
        Q = int(Q).to_bytes(32, 'little')

        return d, Q

    #Função de assinatura de uma mensagem
    def signature(self,M,d,Q):

        #Determina hash da chave privada
        digest = self.digest(d)
        hashPK = digest[32:]
        hashPK_old = digest[:32]

        #Determinar valor r
        r = self.hash512(hashPK+M)
        r = int.from_bytes(r, 'little')

        #Determinar ponto R,comprimi-lo e transforma em octet string
        R = self.G.mult(r)
        Rx = self.encoding(R,1)
        R = int(Rx).to_bytes(32, 'little')

        #Determinar valor s
        s = self.s_value(hashPK_old)

        #Determinar hash da octet string R||Q||M
        hashString = self.hash512(R+Q+M)
        hashString = int.from_bytes(hashString, 'little')

        # Expressão para determinar S - S = (r + SHA-512(R || Q || M) * s) mod n
        multHash = hashString*s
        addHash = r + multHash
        S = mod(addHash,self.l)

        #Valor de S em octet string
        S = int(S).to_bytes(32, 'little')

        #Assinatura final
        signature = R+S

        return signature

    #Função de verificação de uma mensagem
    def verify(self,M,A,Q):

        #Retira valores R e S da assinatura A
        R = A[:32]
```

```
        S = A[32:]

        s = int.from_bytes(S, 'little')

        #Verificação dos processos de decoding das variaveis S, R e 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

        #Determinar valor t
        digest = self.hash512(R+Q+M)
        t = int.from_bytes(digest, 'little')

        #Determinar variaveis a serem usadas na expressão de verificação
        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)

        #Determinar valores da condição de verificação - [2^c * S]G == [2^c]R +␣
↪(2^c * t)Q
        part1 = self.G.mult(value*s)
        part2 = R.mult(value)
        part3 = Q.mult(value*t)
        part2.soma(part3)

        #Verificação da condição de verificação
        if part1.eq(part2):
            res = True
        else :
            res = False


        return res
```

Exemplo de Teste 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.

[14]:
```
edDSA = EdDSA()
message1 = "Mensagem a ser assinada"
message2 = "Outra mensagem teste"

print("Iniciar programa com mensagem " + message1)

privateKey, publicKey = edDSA.generateKeys()
print("Private Key: ")
print(privateKey)
print("Public Key: ")
print(publicKey)
print()

assinatura = edDSA.signature(dumps(message1), privateKey, publicKey)
print("Assinatura: ")
print(assinatura)
print()

print("Verificação da autenticação da mensagem enviada")
if edDSA.verify(dumps(message1), assinatura, publicKey)==True:
    print("Mensagem autenticada!")
else:
    print("Mensagem não autenticada!")

print()

print("Verificação da autenticação da mensagem não enviada")
if edDSA.verify(dumps(message2), assinatura, publicKey)==True:
    print("Mensagem autenticada!")
else:
    print("Mensagem não autenticada!")
```

```
Iniciar programa com mensagem Mensagem a ser assinada
Private Key:
b'\x805w79\xcb\x11\x90\xa8b\x83
>\xff\xafTf\x01\x0f*\xe0\xd8y\xbd/\xce\xfb\x8e\x1b\xc7\xd7\xda'
Public Key:
b'h<\x08\x82\xb1\x01\x98/=Q\xc0\xf2m\x16A6\xbc\xbe\x8d\x19V\x13D\x15\x13YPu5\r4V
'

Assinatura:
b'\x18\r9\x1c\x12\x15\xc1\xae\xab\xa0g\x06F\xc5\x1et\x1e/\xae>N\x17\xfeW\xbc\x0c
\xb4#|\xcb\x02\x10\xe0;\xb7\x04\xd1\x16\x90r\xd6BV\xa3:F\x9fU\xe5\xfd\x12\xa6\x9
5\xa3%\x19\\u\xd0Q\xc8\xd8G\t'

Verificação da autenticação da mensagem enviada
```

```
Mensagem autenticada!

Verificação da autenticação da mensagem não enviada
Mensagem não autenticada!
```