TP2_EX2

March 28, 2023

1 TP2

1.1 Grupo 17:

PG50315 - David Alexandre Ferreira Duarte

PG51247 - João Rafael Cerqueira Monteiro

1.2 Exercício 2.

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
C. Por aplicação da transformação de Fiat-Shamir construa um protocolo de autenticação de desa
```

[74]:
```python
import hashlib
import secrets
```

classe TwistedEdwardsCurve, que define uma curva elíptica torcida de Edwards. O construtor da classe aceita um argumento ed que, se definido como 'ed25519', inicializa a curva com parâmetros correspondentes à curva ed25519 (uma curva elíptica torcida de Edwards sobre o corpo Fp, onde p é um número primo de 255 bits), caso contrário, inicializa a curva com parâmetros correspondentes à curva ed448 (uma curva elíptica torcida de Edwards sobre o corpo Fp, onde p é um número primo de 448 bits).

Os métodos da classe incluem a determinação da ordem do maior subgrupo da curva (método order), a geração de um gerador aleatório para a curva (método gen), a verificação se um ponto está na curva (método is_edwards), a transformação de um ponto da curva edwards para a curva elíptica (método ed2ec) e a transformação de um ponto da curva elíptica para a curva edwards (método ec2ed). A classe também armazena os parâmetros da curva em um dicionário de constantes.

```
1. order(): retorna a ordem do maior subgrupo da curva e seu cofator.
2. gen(): gera um ponto gerador aleatório na curva (usando o algoritmo de Hasse-Weil) e calcula
3. is_edwards(x, y): verifica se um ponto (x, y) está na forma de Edwards da curva.
4. ed2ec(x, y): mapeia um ponto (x, y) na curva Edwards para um ponto correspondente na curva d
5. ec2ed(P): mapeia um ponto P na curva de Weierstrass para um ponto correspondente na curva Ed
```

[75]:
```python
class TwistedEdwardsCurve(object):
    def __init__(self, ed = None):
        if ed == 'ed25519':
```

```
            self.p = 2^255-19
            self.K = GF(self.p)
            self.a = self.K(-1)
            self.d = -self.K(121665)/self.K(121666)
            self.ed25519 = {
                'b'  : 256,
                'Px' : self.
↪K(15112221349535400772501151409588531511454012693041857206046113283949847762202),
                'Py' : self.
↪K(46316835694926478169428394003475163141307993866256225615783033603165251855960),
                'L'  : ZZ(2^252 + 27742317777372353535851937790883648493 - 1),␣
↪## ordem do subgrupo primo
                'n'  : 254,
                'h'  : 8
            }
        else:
            # Edwards 448
            self.p = 2^448 - 2^224 - 1
            self.K = GF(self.p)
            self.a = self.K(1)
            self.d = self.K(-39081)
            self.ed448= {
                'b'  : 456,     ## tamanho das assinaturas e das chaves públicas
                'Px' : self.
↪K(22458004029592430018760433409989603624678964163256413424612546168695041546740603290902919
↪,
                'Py' : self.
↪K(29881921007848149267601793044393067343754404015408024209592824137233150618983587600353687
↪,
                'L'  : ZZ(2^446 -␣
↪13818066809895115352007386748515426880336692474882178609894547503885) ,
                'n'  : 447,     ## tamanho dos segredos: os dois primeiros bits␣
↪são 0 e o último é 1.
                'h'  : 4        ## cofactor
            }

        assert self.a != self.d and is_prime(self.p) and self.p > 3
        K        = GF(self.p)

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

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

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

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

        if ed == 'ed25519':
            self.L = self.ed25519['L']
            self.P = self.ed2ec(self.ed25519['Px'],self.ed25519['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))
```

Essa é uma classe que implementa a assinatura digital CDSA baseada em curvas elípticas. A classe tem um construtor que recebe a curva e a chave privada como parâmetros, e usa a chave privada para calcular a chave pública.

O método `sign` é usado para gerar a assinatura digital de uma mensagem. Ele gera um número aleatório k e calcula o ponto R = k * P na curva, onde P é um ponto fixo na curva (chamado de ponto base). Em seguida, ele calcula o valor r como o resto da divisão da coordenada x do ponto R pelo número primo n. Se r for zero, o processo é repetido com um novo valor de k. Em seguida, o método calcula o valor s como ((r * private_key) % n + hash(message)) * inverse_mod(k, n) % n, onde hash(message) é o hash da mensagem. O método retorna uma tupla (r, s) como a assinatura.

O método `verify` é usado para verificar a assinatura de uma mensagem. Ele recebe a mensagem e a assinatura (uma tupla (r, s)) como parâmetros. O método primeiro verifica se r e s estão no intervalo (0, n) e, em seguida, calcula o valor de w como o inverso multiplicativo de s em módulo n. Em seguida, o método calcula os valores u1 = (hash(message) * w) % n e u2 = (r * w) % n. Usando esses valores, ele calcula o ponto V = u1 * P + u2 * public_key na curva e verifica se a coordenada x de V é igual a r mod n. Se for o caso, a assinatura é considerada válida.

[81]:
```python
class EdCDSA:
    def __init__(self, curve, private_key):
        self.curve = curve
        self.private_key = private_key
        self.public_key = self.private_key * self.curve.P

    def sign(self, message):
        K = self.curve.K
        n = self.curve.L
        h = self.curve.ed25519['h']
        a = self.curve.constants['a']
        d = self.curve.constants['d']
        P = self.curve.P
        r = 0
        while r == 0:
            k = randint(1, n-1)
            R = k * P
            x = int(R[0])
            r = x % n
        s = ((r * self.private_key) % n + hash(message)) * inverse_mod(k, n) % n
        return (r, s)

    def verify(self, message, signature):
        K = self.curve.K
        n = self.curve.L
        h = self.curve.ed25519['h']
        a = self.curve.constants['a']
        d = self.curve.constants['d']
        P = self.curve.P
        r, s = signature
```

```python
        if not (0 < r < n) or not (0 < s < n):
            return False
        w = inverse_mod(s, n)
        u1 = (hash(message) * w) % n
        u2 = (r * w) % n
        V = u1 * P + u2 * self.public_key
        x = int(V[0])
        return r == x % n

    def fiat_shamir(self, challenge, secret):
        return 0
```

[77]:

```python
# Exemplo de uso
```

```
(7237005577332262213973186563042994240857116359379907606001950938285454250989,
8)
```

[79]:

```python
# Create the curve and the private key
curve = TwistedEdwardsCurve(ed='ed25519')
print(curve.order())
private_key = randint(1, curve.L-1)

# Criar chave privada e um EdCDSA object
public_key = private_key * curve.P
eddsa = EdCDSA(curve, private_key)

# Assinar
message = b"Grupo 07"
signature = eddsa.sign(message)

# Verificar
assert eddsa.verify(message, signature, public_key)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
File ~/mambaforge/envs/sage/lib/python3.9/site-packages/sage/arith/misc.py:2149
↪in inverse_mod(a, m)
   2148 try:
-> 2149     return a.inverse_mod(m)
   2150 except AttributeError:

AttributeError: 'int' object has no attribute 'inverse_mod'

During handling of the above exception, another exception occurred:

ZeroDivisionError                         Traceback (most recent call last)
Cell In[79], line 11
```

```
      9 # Sign a message
     10 message = b"Hello, world!"
---> 11 signature = eddsa.sign(message)
     13 # Verify the signature
     14 assert eddsa.verify(message, signature, public_key)

Cell In[76], line 20, in EdCDSA.sign(self, message)
     18     x = int(R[Integer(0)])
     19     r = x % n
---> 20 s = ((r * self.private_key) % n + hash(message)) * inverse_mod(k, n) % :
     21 return (r, s)

File ~/mambaforge/envs/sage/lib/python3.9/site-packages/sage/arith/misc.py:2151 
 ↪in inverse_mod(a, m)
   2149     return a.inverse_mod(m)
   2150 except AttributeError:
-> 2151     return Integer(a).inverse_mod(m)

File ~/mambaforge/envs/sage/lib/python3.9/site-packages/sage/rings/integer.pyx:
 ↪6772, in sage.rings.integer.Integer.inverse_mod (build/cythonized/sage/rings/
 ↪integer.c:42055)()
   6770 sig_off()
   6771 if r == 0:
-> 6772     raise ZeroDivisionError(f"inverse of Mod({self}, {m}) does not
 ↪exist")
   6773 return ans
   6774

ZeroDivisionError: inverse of
 ↪Mod(4288929353546667438562427604540732559223197526601359191838030664959414319503,
 ↪7237005577332262213973186563042994240857116359379907606001950938285454250988)
 ↪does not exist
```