TP2_Ex2

March 29, 2023

1 TP2 | Exercício 2 | Grupo 15

1.0.1 Nuno Dias Mata - PG44420

1.0.2 Pedro Araújo - pg50684

Questão: 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".
3. Por aplicacao da transformada de **Fiat-Shamir** contrua um protocolo de autenticacao desafio-resposta.

O ECDSA (Elliptic Curve Digital Signature Algorithm) baseia-se, como o próprio nome indica, em Curvas Elipticas.

As curvas elipticas satisfazem a sequinte equação que é designada por equação de Weierstrass:

$$Y^2 = x^3 + Ax + B$$

[1]:
```
import os
import hashlib
```

1.1 Parâmetros

De acordo com o standar FIPS186-5, foi escolhida uma das duas curvas elipticas propostas, a Ed25519 Os parâmetros inicias para a curva Edwards25519 são:

[2]:
```
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),
'n'  : 254,
'h'  : 8
}
```

EdDSA() é a função que inicializa toda a instância e os valores globais necessários à sua execução. O **EdDSA (Edwards-curve Digital Signature Algorithm)** necessita dos seguintes 11 parâmetros:

1. Um numero primo “p”, que na curva Ed25519 é sempre 2^255 - 19.
2. Um valor “b”, multiplo de 8, para os bits da chave publica, que também é usado na assinatura EdDSA que tem 2*b bits.
3. Uma codificação de b-1 bits dos elementos do campo finito GF(p).
4. Uma função de hash criptográfica “H” que produz uma saída de 2*b bits.
5. Um inteiro “c” de valor 2 ou 3, pois os escalares secretos EdDSA são múltiplos de 2^c, este é o logaritmo base 2 do chamado cofator.
6. Um inteiro “n” com c <= n < b. Os escalares secretos EdDSA têm exatamente n + 1 bits, com o bit superior (a posição 2^n) sempre definido e os c bits inferiores sempre vom o valor 0.
7. Um elemento não-quadrático “d” de GF(p). Normalmente recomenda-se tomar como o valor mais próximo de zero que dá uma curva “aceitável”.
8. Um elemento quadrático não-zero “a” de GF(p). A recomendação usual para melhor desempenho é a = -1 se p mod 4 = 1, e a = 1 se p mod 4 = 3.
9. Um elemento B != (0,1) do conjunto E = { (x,y) é um membro de GF(p) x GF(p) tal que a * x^2 + y^2 = 1 + d * x^2 * y^2 }.
10. Um primo ímpar “L” tal que [L]B = 0 e 2^c * L = #E. O número #E (o número de pontos na curva) faz parte do dados fornecidos para uma curva elíptica E, ou pode ser calculado como cofator * ordem.
11. Uma função “prehash” PH. PureEdDSA significa EdDSA onde PH é a função identidade, ou seja, PH(M) = M. HashEdDSA significa EdDSA onde PH gera um output curt, não importa o quão longa seja a mensagem; por exemplo, PH(M) = SHA-512(M).

[3]:
```
def EdDSA():
    def __init__(self):

        self.p = 2^255 - 19
        self.K = GF(self.p)
        self.Px = self.
 ↪K(15112221349535400772501151409588531511454012693041857206046113283949847762202)
        self.Py = self.
 ↪K(46316835694926478169428394003475163141307993866256225615783033603165251855960)
        self.L = ZZ(2^252 + 27742317777372353535851937790883648493)
        self.requested_security_strength = 128
        self.a = self.K(-1)
        self.d = -self.K(121665)/self.K(121666)
```

```
        self.n = 254
        self.c = 3
        self.h = 8
        self.b = 256

        self.chave_privada = os.urandom(self.b/8)


    #H é usada durante a derivação
    #SHA512:Ed25519 ou SHAKE256:Ed448
    def H(pk):
        return hashlib.sha512(pk).digest()

    def bit(h,i):
        return ((h[int(i/8)]) >> (i%8)) & 1

    def expmod(b,e,m):
        if e == 0: return 1
        t = expmod(b,e/2,m)**2 % m
        if e & 1: t = (t*b) % m
        return t

    def inv(x):
        return expmod(x,q-2,q)

    def scalarmult(P,e):
        if e == 0: return [0,1]
        Q = scalarmult(P,e/2)
        Q = edwards(Q,Q)
        if e & 1: Q = edwards(Q,P)
        return Q

## Função que mapeia Ed para EC
    def ed2ec(x,y):
        if (x,y) == (0,1):
            return EC(0)
        z = (1+y)/(1-y) ; w = z/x
        alfa = constants['alfa']; s = constants['s']
        return 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(encoded)))

    #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 privada
        self.private_key = H(os.urandom(32))
            #3. Calcular e modificar o hdigest1: 3 primeiros bits a 0, ultimo␣
↪bit a 0 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(3,self.b-2))
            #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

    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))

    #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

    # Assinatura
    # As assinaturas EdDSA são determinísticas, a assinatura é gerada usando o␣
↪hash da chave privada e da mensagem através do procedimento abaixo (ou um␣
↪processo equivalente).
```

```
   # Inputs:
   # 1. String "M" de bits para ser assinada;
   # 2. Par de chaves pública-privada válidas (d, Q) para parâmetros de domínio D
   # 3. H: SHA-512 para Ed25519 ou SHAKE256 para Ed448
   # Output: A assinatura R || S, onde R é uma codificação de um ponto e S é um encoded value em little-endian.

   # Processo:
   # Como especificado na IETF RFC 8032, a assinatura EdDSA de uma mensagem M sob uma chave privada d é definida como a string de bits de 2b R || S.
   # As strings de octetos R e S são derivadas da seguinte forma:
   # 1. Calcula-se o hash da chave privada d, H(d) = (h0, h1, ..., h2b-1) usando SHA-512 para Ed25519. H(d) pode ser pré-calculado.
   # 2. Usando a segunda metade do resumo hdigest2 = hb || ... || h2b-1, define-se:
   # 2.1 Para Ed25519, r = SHA-512(hdigest2 || M); r terá 64 octetos.
   # octet(octetlength(c)) || c). A string "SigEd448" está em ASCII (8 octetos). O valor
   # 3. Calcula-se o ponto [r]G. A string de octetos R é a codificação do ponto [r]G.
   # 4. Derivar s de H(d) como no algoritmo de geração de par de chaves. São usadas as strings de octetos R, Q e M para definir:
   # 4.1 Para Ed25519, S = (r + SHA-512(R || Q || M) * s) mod n.
   # A string de octetos S é a codificação do inteiro resultante.
   # 5. Gera-se finalmente a assinatura como a concatenação das strings de octetos R e S."

   #EdDSA Signature Generation
   def sign(self, msg):
           #1. Calculo do 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. Calcula-se 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

    # Inputs:
    #   1. Messagem M
    #   2. Assinatura R || S, onde R e S são strings de octetos
    #   3. Chave de verificação de assinatura pretendida Q que é válida para␣
↪parâmetros de domínio D

    # Output: Aceitar ou rejeitar a assinatura sobre M como originada do␣
↪proprietário da chave pública Q
    #
    # Processo:
    #   1.Descodificar a primeira metade da assinatura como um ponto R e a␣
↪segunda metade da assinatura como um inteiro s.
    # Verificar se o inteiro s está no intervalo de 0  s < n. Descodificar a␣
↪chave pública Q num ponto Q'. Se alguma das
    # descodificações falhar, output "reject".
    #   2. Formar a string de bits HashData como a concatenação das strings de␣
↪octetos R, Q e M (ou seja, HashData = R || Q || M).
    #   3. Usando a função de hash estabelecida ou XOF,
    #       3.1 Para Ed25519, calcular o digest = SHA-512(HashData).
    #   4. Verificar se a equação de verificação [2c * S]G = [2c]R + (2c * t)Q.␣
↪Output "reject" se a verificação falhar; caso contrário output "accept".

    #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)
```

[ ]:

[ ]: