Estruturas criptograficas: TP2 problema 1

Ed25519

Este algoritmo tem como base três funções principais: keygen, sign e verify. Desta forma, e seguindo as normas estblecidas no RFC, seguimos os seguintes passos.

KeyGen

1. Obtivémos uma string de b bits pseudo-aleatória, que serviria de chave privada **d**.
2. Usando o SHA-512, calculámos o hash dessa chave privada **d**.
3. A primeira metade do **H(d)** é usada para gerar a chave pública: Os primeiros três bits do primeiro octeto são passados a zero: o último bit do último octeto é zero; e o penúltimo bit do último octeto é 1.
4. Determinar um inteiro **s** do digest do hash, usando a conveção **little-endian**.
5. Multiplicar o escalar **s**, pelo ponto **G**, que quando aplicado o **encoding** resultará na chave pública **Q**.

Sign

1. Calcular o hash da chave privada **d**, usadno SHA-512.
2. Usando a segunda metade do digest fazemos: **r** = SHA-512(digest || **plaintext**), e interpretamos o **r**, como um inteiro little-endian de 64 octetos.
3. Calculámos o ponto [r]G. A string de octetos **R** é o **encoding** do ponto [r]G.
4. Derivar **s** do **H(d)**, como no algoritmo de geração de chaves, e definimos **digest** = SHA-512(R || Q || M).
5. Calculámos **S** = (r + digest × s) mod n. A string **S** é o resultado do **enoding** desta operação.
6. Gerámos a assinatura, pela concatenação das strings **R** e **S**.

Verify

1. Primeiramente fazemos **decoding** da primeira metade da assinatura, como um ponto **R**, e a segunda como um inteiro **t**. Verificámos se o inteiro **t** está no intervalo 0 <= t < n. Fizémos **decoding** da chave pública **Q** para um ponto **Q_**.

2. Calculámos **digest** = SHA-512(R || Q || M).

3. Para verificar a assinatura resolvemos a equação: **[t]G = R + [u]Q_**, se o resultado fosse **True**, a verificação é confirmada, e a assinatura é validada; caso contrário a verificação é rejeitada.

Casos de teste

Para testar o algoritmo desenvolvido, testamos uma verificação com a mensagem correta, e outra com a mensagem alterada. Os resultados obtidos, foram de encontro ao que era esperado.

In [1]:
```
import hashlib, os
from sage.all import *
```

Esta classe é uma adaptação de uma classe do ficheiro **eddsa2.py** fornecido pelo docente, que implementa diversas operações de inteiros com pontos: *, e +, e também de comparação, duplicação, cópia, etc.

In [2]:
```
class EdwardsPoint:

    def __init__(self, params, x, y):
        self.params = params
        self.x = x
        self.y = y
        self.w = x*y

    def copy(self):
        return EdwardsPoint(self.params, self.x, self.y)

    def zero(self):
        return EdwardsPoint(self.params, 0, 1)

    def duplica(self):
        a = self.params['a']
        d = self.params['d']
        delta = d*(self.w)**2
        self.x, self.y = (2*self.w)/(1+delta) , (self.y**2 - a*self.x**2)
        self.w = self.x*self.y

    def __add__(self, other):
        a = self.params['a']
        d = self.params['d']
        delta = d*self.w*other.w
        self.x, self.y  = (self.x*other.y + self.y*other.x)/(1+delta), (s
        self.w = self.x*self.y
        return self
```

```
    def __mul__(self, n):
        m = Mod(n, self.params['L']).lift().digits(2)
        Q = self.copy()
        A = self.zero()
        for b in m:
            if b == 1:
                A + Q
            Q.duplica()
        return A

    def __eq__(self,other):
        return self.x == other.x and self.y == other.y
```

In [3]:

```
class EdDSA:

    def __init__(self):
        self.b = 256
        self.p = 2^255 - 19
        self.K = GF(self.p)
        self.x = self.K(15112221349535400772501151409588531511454012693041857205192948904
        self.y = self.K(46316835694926478169428394003475163141307993866256225615783033603
        self.c = 3
        self.n = 254
        self.d = 37095705934669439343138083508754565189542113879843219016388785533085940
        self.a = self.K(-1)
        self.L = 2^252 + 27742317777372353535851937790883648493

        # Calular Edwards Curve
        # self.A =  2*(self.a + self.d)/(self.a - self.d)
        # self.B =  4/(self.a - self.d)

        # self.alfa = self.A/(3*self.B)
        # self.beta = self.B

        # self.a4 =  self.beta**(-2) - 3*self.alfa**2
        # self.a6 =  -self.alfa**3 - self.a4*self.alfa

        # self.EC = EllipticCurve(self.K,[self.a4,self.a6])

        self.params = {
            'd': self.d,
            'a': self.a,
            'L': self.L,
        }

        self.B = EdwardsPoint(self.params, self.x, self.y)


    # Função de hash
    def H(self, k):
        return hashlib.sha512(k).digest()


    def frombytes(self,x,b):
        rv = int.from_bytes(x, byteorder="little") % (2**(b-1))
        return self.K(rv) if rv < self.p else None

    def solve_x2(self,y):
        return ((y*y-self.K(1))/(self.d*y*y+self.K(1)))
```

```
    def sqrt(self, x):
        y = pow(x,(self.p+3)//8,self.p)

        if (y * y) % self.p == x % self.p: return y
        else:
            z = pow(2,(self.p - 1)//4, self.p)
            return (y * z) % self.p

    def decoding(self,s):
        s = bytearray(s)
        if len(s)!=self.b//8:
            return (None,None)

        xs=s[(self.b-1)//8]>>((self.b-1)&7)
        y = self.frombytes(s,self.b)

        if y is None:
            return (None,None)

        x = self.solve_x2(y)
        x = self.sqrt(x)

        if x is None or (x == 0 and xs != x % 2):
            return (None,None)

        if mod(x, 2) != xs:
            x=-x

        return (x,y)

    def encoding(self, x, y):
        s = bytearray(int(y).to_bytes(self.b//8,byteorder="little"))

        if mod(x, 2) != 0:
            s[(self.b-1)//8]|=1 << (self.b-1) % 8

        return bytes(s)

    def keygen(self):

        # gerar chave privada
        private_key = os.urandom(self.b / 8)

        # hash da chave privada
        pk = self.H(private_key)
        pk = pk[:32]

        # gera inteiro a partir do hash
        bits = int.from_bytes(pk)

        # converte inteiro para lista de bits
        bits = [int(digit) for digit in list(ZZ(bits).binary())]

        # adiciona bits a 0 para completar o tamanho da chave
        extra = self.b - len(bits)
        for i in range(extra):
            bits.insert(0, 0)

        # ajusta os bits para o formato correto
        bits[0] = bits[1] = bits[2] = 0
```

```
        bits[self.b-2] = 1
        bits[self.b-1] = 0

        # converte os bits para string
        bits = "".join(map(str, bits))

        # converte os bits para inteiro
        s = int(bits[::-1], 2)

        # calcula a chave pública
        Q = self.B * s
        Q_enc = self.encoding(Q.x, Q.y)

        Q_int = int.from_bytes(Q_enc, 'little')
        public_key = int(Q_int).to_bytes(self.b/8, 'little')

        return private_key, public_key

    #TODO: fazer o PH caso a mensagem seja muito grande
    def sign(self, private_key, public_key, message):

        # gerar hash da chave privada
        d = self.H(private_key)
        pk = d[32:]

        M = message.encode()

        # gera um hash da chave privada e da mensagem e converte para int
        r = self.H(pk + M)
        r = int.from_bytes(r, 'little')

        # calcula o ponto R
        P = self.B * r
        R = self.encoding(P.x, P.y)

        # calcula o valor de s com a primeira parte do hash da chave priv
        hd = d[:32]
        bits = int.from_bytes(hd)

        # converte inteiro para lista de bits
        bits = [int(digit) for digit in list(ZZ(bits).binary())]

        # adiciona bits a 0 para completar o tamanho da chave
        extra = self.b - len(bits)
        for i in range(extra):
            bits.insert(0, 0)

        # ajusta os bits para o formato correto
        bits[0] = bits[1] = bits[2] = 0
        bits[self.b-2] = 1
        bits[self.b-1] = 0

        # converte os bits para string
        bits = "".join(map(str, bits))

        # converte os bits para inteiro
        s = int(bits[::-1], 2)
        #-------------------------------------

        # gera um hash do ponto R (encoded), chave pública, e da mensagem
```

```
        digest = self.H(R + public_key + M)
        digest = int.from_bytes(digest, 'little')

        # calcula o valor de S
        S = mod(r + digest * s, self.L)

        print("Message signed!")

        # retorna a assinatura
        return R + int(S).to_bytes(self.b/8, 'little')

    def verify(self, M, public_key, signature):

        # separa a assinatura em R e S
        R = signature[:32]
        t = signature[32:]

        t = int.from_bytes(t, 'little')

        # verifica se t está no intervalo [0, L)
        if t < 0 or t >= self.L:
            print("Signature Invalid!")
            return

        # gera um hash do ponto R (encoded), chave pública, e da mensagem
        digest = self.H(R + public_key + M.encode())
        u = int.from_bytes(digest, 'little')

        # calcula o ponto R
        R = self.decoding(R)
        R = EdwardsPoint(self.params, R[0], R[1])

        # calcula o ponto Q
        Q = self.decoding(public_key)
        Q = EdwardsPoint(self.params, Q[0], Q[1])

        if (R.x, R.y) == (None,None) or (Q.x, Q.y) == (None,None):
            print("Invalid Signature!")
            return

        # resolve a equação [t]B = R + [u]Q', para verificar a assinatura
        b1 = self.B * t
        q1 = Q * u
        r1 = R + q1

        valid = b1 == r1

        if valid is True:
            print("Valid Signature!")
        else:
            print("Invalid Signature!")
```

Assinatura da mensagem

In [4]:
```
edsa = EdDSA()
sk, pk = edsa.keygen()
```

```
signature = edsa.sign(sk, pk, "Uma mensagem qualquer")
```

```
Message signed!
```

Verificação da mensagem com sucesso

In [5]:
```
edsa.verify("Uma mensagem qualquer", pk, signature)
```

```
Valid Signature!
```

Falha na verificação da mensagem

In [6]:
```
edsa.verify("SageMath!", pk, signature)
```

```
Invalid Signature!
```