TP2_Ex2

March 28, 2023

0.1 Trabalho prático 2 - Exercício 2

Para este exercício foi proposta a implementação de um algoritmo de assinatura **EdCDSA**. Este designa-se por *Edwards-curve Co-factor Digital Signature Algorithm*, sendo um esquema de assinatura que utiliza *Twisted Edwards-curve*, baseado no algoritmo **ECDSA**(*Elliptic Curve Digital Signature Algorithm*); bastante semelhante ao **EdDSA**.

Em três pontos distintos do enunciado, é pedido que:

- A implementação tenha funções para assinar digitalmente e verificar a assinatura.
- A implementação deverá usar uma das *Twisted Edwards Curves* definidas no *standard*, escolhidas na iniciação da classe: **Ed25519** e **Ed448**.
- Utilizar a transformação de *Fiat-Shamir* para construir um protocolo de autenticação de desafio-reposta.

Deste modo, para implementar o **EdCDSA**, é necessário utilizar uma das curvas de *Twisted Edwards* definidas no **FIPS186-5**, que pode ser a curva **edwards25519** ou **edwards448**; deixando a escolha ao utilizador.

Além disso, deve-se aplicar a transformação de **Fiat-Shamir** para a autenticação. Este é um método para transformar um esquema de assinatura de conhecimento zero interativo ($SNARK$) em uma versão não interativa; altamente utilizado na criptografia moderna. E baseado no uso de funções *hash* criptográficas.

O processo implica o provador enviar uma série de mensagens ao verificador para provar a validade de uma determinada afirmação. No protocolo de *Fiat-Shamir*, essas mensagens são substituídas por valores *hash* das mesmas. O verificador então usa os valores *hash* como se fossem as mensagens originais e continua a verificar a afirmação.

[1]:
```python
# imports
import hashlib, os
import random
from pickle import dumps
from struct import *
```

0.2 Funções auxiliares fornecidas pela equipa docente

0.2.1 Classe de implementação da curva de Edwards

O *EdCDSA* utiliza curvas de *Edwards*, que são uma classe especial de curvas elípticas que têm a forma de uma equação de *Edwards*:

```
x^2 + y^2 = 1 + dx^2y^2
```

... onde **d** é um parâmetro constante. As curvas de Edwards são conhecidas por serem mais eficientes do que as curvas elípticas tradicionais para certas operações, como a adição de pontos e a verificação de pontos na curva.

O algoritmo *EdCDSA* utiliza as curvas de *Edwards* para gerar chaves e assinaturas da mesma forma que o algoritmo *ECDSA* utiliza as curvas elípticas.

0.3 Classe *Ed*

A classe seguinte é fornecida pela equipa docente.

A classe *Ed* é usada para definir uma curva elíptica de *Edwards* sobre um **corpo finito**, e contém métodos para mapear pontos nesta curva para uma **curva elíptica**.

O construtor da classe *Ed* recebe quatro parâmetros:

- um primo p;
- os coeficientes a e d da curva de Edwards;
- o parâmetro opcional *ed*.

Se o parâmetro opcional *ed* é fornecido, ele deve ser um dicionário com as chaves *'L', 'Px' e 'Py'*, contendo respetivamente o **tamanho do maior subgrupo da curva**, e as **coordenadas x e y de um ponto na curva de *Edwards*** que será usado como **gerador do grupo**. Caso contrário, a função *gen()* é usada para gerar um ponto aleatório na curva.

No método *init()*, são calculados os coeficientes da curva de *Edwards* a partir dos coeficientes **a e d**; define-se também a curva elíptica *EC*, com os coeficientes **a4 e a6**. O método *order()* calcula a **ordem n do maior subgrupo da curva EC** e o **cofator h**; e retorna-os num tuplo *(n,h)*. O método *gen()* usa o método *order()* para gerar um ponto aleatório **P** na curva *EC* que não está no subgrupo trivial, e define **self.P** como o ponto **h*P **e** self.L **como o tamanho do maior subgrupo. O método *is_edwards(x,y)* verifica se um ponto *(x,y)* está na curva de *Edwards* definida pelos** coeficientes a e d**. O método *ed2ec(x,y)* mapeia um ponto** (x,y) **na curva de *Edwards* para um ponto na curva elíptica de *Weierstrass*** EC**. Vice-versa, o método *ec2ed(P)* mapeia um ponto** P na curva EC** para um ponto na curva de *Edwards*.

[2]:
```
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'])  # gerador do grupo
        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))
```

0.4 Classe de implementação dos métodos dos pontos de edwards

Outra classe fornecida pela equipa docente.

A classe **ed** define um ponto na curva de *Edwards* e contém métodos para operações de soma, duplicação e multiplicação escalar, para as operações necessárias nas funções a serem implementadas pela classe **EdCDSA**.

[3]:
```
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)   ## obter a representação␣
↪binária do argumento "n"
        Q = self.copy() ; A = self.zero()
```

```
        for b in m:
            if b == 1:
                A.soma(Q)
            Q.duplica()
        return A
```

0.5 Classe EdCDSA

Como referido no enunciado, é pedida que a implementação seja com base no *standard* **FIPS186-5**. Neste caso, segue-se os passos para desenvolver o algoritmo de assinatura **ECDSA**, substituindo a utilização de curvas elípticas por *Twisted Edwards curves*.

Por esse motivo, tal como na maior parte dos algoritmo de assinatura, são implementadas **3 funcionalidades** base:

- a geração de chaves (*generateKeys*);
- assinatura (*signature*);
- verificação da assinatura (*verify*).

Antes de tudo é realizado um *setup* para a classe, que tem como objetivo preparar as variáveis globais necessárias para as operações com a curva escolhida, **edwards25519** ou **edwards448**. Por este motivo, este recebe a **curva** a usar, na forma de uma *string*, e ,recorrendo ao código fornecido pela equipa docente sobre a inicialização e preparação de uma *Twisted Edwards curve*, estabelece-se parâmetros importantes como a **instância Ed da curva** e a **instância ed de um ponto base da curva**, entre outras.

Geração de chaves: A assinatura de uma qualquer mensagem será através de uma **chave privada** e a sua respetiva verificação através da **chave pública** correspondente.

Deste modo, o primeiro passo é a geração de um par de chaves **pública-privada**.

Segundo o *standard*, no que toca ao algoritmo **ECDSA**, a chave privada é um **número inteiro aleatório**, geralmente escolhido com ajuda de um gerador de números pseudo-aleatórios criptográficos.

... o número aleatório criado para ser chave privada deve encontrar-se no *range [1-(n//h)-1]*. É utilizado o **co-fator** para gerar a chave privada para reduzir a probabilidade de gerar uma chave **fraca**; facilmente quebrada por um atacante. Deste modo, previne-se que o **sub-grupo** do co-fator seja utilizado; pelo que os pontos aqui são mais repetitivos.

A chave pública é um ponto na curva de *Edwards*, que é obtido através da **multiplicação da chave privada pelo ponto gerador da curva**. O ponto gerador é um ponto pré-definido na curva, que é escolhido de tal forma que sua ordem seja um número primo grande, para garantir a segurança do algoritmo (como se verificou na classe **Ed**). O processo deverá repetir-se na ocasião da chave pública calculada, um ponto, não encontre-se na **curva definida**.

No final, na forma das variáveis d e Q, retorna-se as chaves **privada** e **pública**, respetivamente.

Assinatura: Para assinar uma mensagem, o remetente precisa seguir os seguintes passos:

- **Cálculo do hash da mensagem:** a mensagem é transformada em um valor hash criptográfico usando uma função hash segura, como SHA-512, se for a curva **edwards25519**, ou SHAKE256, caso seja a **edwards448**;
- **Cálculo do valor e:** Este, no caso do valor de (log(n)) ser maior que o tamanho das *hash*, pode ser o valor da *hash*. Caso contrário, será os *leftmost* (log(n)) *bits* da *hash*.
- **Escolha de um número aleatório k:** um número aleatório k é escolhido, novamente com a ajuda de um gerador de números pseudo-aleatórios criptográficos. O valor de k deve ser mantido em segredo e não deve ser reutilizado para outras assinaturas;
- **Cálculo do ponto de assinatura R:** o ponto de assinatura R é obtido multiplicando o valor de k pelo ponto gerador da curva. Esse ponto é um ponto na curva de *Edwards* e é utilizado para calcular/gerar a **assinatura**.
- **Cálculo do valor s:** o valor s é calculado usando a chave privada do remetente e o hash da mensagem. Ele é dado por s = $(r + h)/k(modn))$, onde r é a coordenada x do ponto rx, d é a chave privada e h é o valor hash da mensagem. Posteriormente, os valores K, e asua respetiva inversa modular, devem ser destruídas, de modo a preservar a segurança do processo.
- **Cálculo da assinatura:** a assinatura é um par de valores (R,s), que é enviado junto com a mensagem; sendo esto o retorno do método *signature*.

O processo é todo reptido enquanto que a assinatura (r, s) é nula (=0.) Posteriormente, o valor **r** (compromisso) será usado para comparar, no âmbito da verificação, com o valor R lá calculado (sendo este a partir da chave pública criada). O valor **s** tem como objetivo garantir integridades e autenticidade da mensagem assinada.

Verificação da assinatura: Para verificar a assinatura, o destinatário precisa seguir os seguintes passos:

- **Verificar se a assinatura é válida:** Antes de tudo é verificado se o compromisso **r** e a verificação da integridade da mensagem **s**, estão **não nulos** (encontram-se não válidos); caso estejam, um erro é lançado.
- **Cálculo do hash da mensagem:** a mensagem é transformada em um valor hash criptográfico usando a mesma função hash segura utilizada pelo remetente.
- **Cálculo do valor e:** Este, no caso do valor de (log(n)) ser maior que o tamanho das *hash*, pode ser o valor da *hash*. Caso contrário, será os *leftmost* (log(n)) *bits* da *hash*.
- **Cálculo do valor rx:** Para este é necessário o cálculo das variáveis **u** ($u = es^-1(modn)$) e **v** ($(v = rs - 1(modn))$), culminando no em rx = [u]G + [v]Q (mod n); rejeitar o *output* caso **rx** seja a identidade.
- **Comparar se r = rx (mod n):** No caso de ser verdade, a assinatura encontra-se corretamente validada, caso contrário deve ser rejeitada.

Em concreto, o algoritmo **EdCDSA** é uma implementação do **ECDSA** que usa curvas elípticas torcidas de *Edwards*, como tinha sido ref erido em diferentes pontos do *notebook*, sendo a estratégia seguida basicamente igual.

```
[4]: class EdCDSA():

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

         #Parâmetros das curvas de ED25519 ou ED448
         def setup(self, curve):

             if curve == "ed25519":
                 p = 2^255-19               # número primo que define o corpo sobre o␣
     ↪qual a curva elíptica é definida
                 K = GF(p)                  # Corpo finito de ordem p
                 a = K(-1)                  # Coeficiente da equação da curva elíptica
                 d = -K(121665)/K(121666)   # Coeficiente da equação da curva elíptica

                 ed25519 = {
                 'b'  : 256, # integer representing the bit-length of the prime field␣
     ↪p
                 'Px' :␣
     ↪K(15112221349535400772501151409588531511454012693041857206046113283949847762202),␣
     ↪# x-coordinate of a point on the edwards25519 curve
                 'Py' :␣
     ↪K(46316835694926478169428394003475163141307993866256225615783033603165251855960),␣
     ↪# y-coordinate of a point on the edwards25519 curve
                 'L'  : ZZ(2^252 + 27742317777372353535851937790883648493), # order␣
     ↪of a subgroup of the edwards25519 curve
                 'n'  : 254, # integer representing the bit-length of L
                 'h'  : 2^3  # integer representing the cofactor of the edwards25519␣
     ↪curve
                 }

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

                 E = Ed(p,a,d,ed=ed25519)  # an instance of the Ed class representing␣
     ↪the edwards25519 curve
                 b = ed25519['b']
                 G = ed(curve=E,x=Bx,y=By) # a point on the edwards25519 curve used␣
     ↪as a generator for the EdCDSA algorithm
                 l = E.order()[0]          # the order of the edwards25519 curve

             else:
                 p = 2^448 - 2^224 - 1
                 K = GF(p)
                 a = K(1)
                 d = K(-39081)
```

```
        ed448= {
        'b'  : 456,     ## tamanho das assinaturas e das chaves públicas
        'Px' :␣
↪K(2245800402959243001876043340998960362467896416325641342461254616869504154674060329090291928
↪,
        'Py' :␣
↪K(2988192100784814926760179304439306734375440401540802420959282413723315061898358760035368786
↪,
        'L'  : ZZ(2^446 -␣
↪13818066809895115352007386748515426880336692474882178609894547503885) ,
        'n'  : 447,     ## tamanho dos segredos: os dois primeiros bits são␣
↪0 e o último é 1.
        'h'  : 4        ## cofactor
        }

        Bx = ed448['Px']; By = ed448['Py']

        E = Ed(p,a,d,ed=ed448)
        b = ed448['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
def hash512(self,data):

    return hashlib.sha512(data).digest()

def hash256(self,data):

    return hashlib.shake_256(data).digest(114)

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

    if self.b == 256:
        h = self.hash512(d)
    else:
        h = self.hash256(d)

    return h

# Método para gerar o par de chaves pública/privada
def generateKeys(self):
```

```
        # Ordem da curva
        (n, h) = self.E.order()

        print(n)

        # Variável para verificar se o ponto pertence à curva
        is_ed = False

        while is_ed == False:

            # Gerar private key
            d = random.randint(1, (n//h) -1)

            # Gerar public key
            Q = self.G.mult(d)

            if self.E.is_edwards(Q.x, Q.y):
                is_ed = True

        return d, Q

    # Método para gerar a assinatura de uma mensagem
    def signature(self,m,d):

        # Ordem n da curva
        (n, h) = self.E.order()

        # Calcular hash da mensagem
        h = self.digest(m)

        if self.b == 256:
            hash_size = 512
        else:
            hash_size = 256

        if log(n,2).n() >= hash_size:
            E = h
            e = int.from_bytes(E, 'little')
        else:
            E = h[:floor(log(n,2).n() / 8)]
            e = int.from_bytes(E, 'little')

        s = 0
        r = 0
        while s==0 or r==0:
            # Obter inteiro aleatório entre 1 e n-1
            k = random.randint(1, n-1)
```

```
            #Inversa modular de k
            inv_mod_k = inverse_mod(k, n)

            # Calcular ponto na curva (x,y) = k*G
            R = self.G.mult(k)

            # Cálculo de r
            r = int(R.x) % n

            # Cálculo de s
            s = (inv_mod_k * (e + r * d)) % n

            # Destruir K e a sua inversa modular
            k = None
            inv_mod_k = None

        return (r,s)

    # Método para verificar a mensagem da assinatura
    def verify(self,m,A,Q):

        (r,s) = A

        # Ordem n da curva
        (n, _) = self.E.order()

        if 1 <= r <= n-1 and 1 <= s <= n-1:

            # Calcular hash da mensagem
            h = self.digest(m)

            if self.b == 256:
                hash_size = 512
            else:
                hash_size = 256

            if log(n,2).n() >= hash_size:
                E = h
                e = int.from_bytes(E, 'little')
            else:
                E = h[:floor(log(n,2).n() / 8)]
                e = int.from_bytes(E, 'little')

            # Obter a inversa modular de s
            inv_s = inverse_mod(s, n)
```

```
            u = (e * inv_s) % n
            v = (r * inv_s) % n

            # Calcular o ponto da curva
            ug = self.G.mult(u)
            vq = Q.mult(v)

            vq.soma(ug)

            if vq.w == 0:
                raise ValueError("Signature Error")

            r1 = int(vq.x)
            rx = r1 % n

            if r == rx:
                return True
            else:
                return False
        else:
            raise ValueError("Signature Error")
```

0.6 Testes criados

Seguem-se os testes para ambas as curvas - **edwards25519** e **edwards448**.

E, como foi pedido num terceiro ponto, o processo de assinatura (de autenticação) vai seguir o esquema (ou protocolo) **Fiat-Shamir**; de modo a construir um protocolo de autenticação **desafio-resposta**.

Concretamente, o esquema apresenta um modo de exprimir a seguinte secção de testes, utilizando para isso uma transformação que recorrerá aos métodos criados previamente.

0.6.1 Esquema de Fiat-Shamir

Um esquema de **assinaturas digitais** é fundamentalmente uma forma particular de prova de conhecimento não-interativa. A **FST** é um mecanismo genérico para transformar um sigma protocolo (prova de conhecimento interativa) numa prova de conhecimento não-interativa.

Deste modo, o esquema de assinaturas é formado por **dois geradores probabilísticos** e uma **decisão determinística**. Deste modo:

- **KeyGen** - Gerar chaves pública e privada, a partir dos parâmetros definidores do esquema.
- **Sign** - Gera a assinatura de uma mensagem m usando a chave privada;
- **Verify** - Verifica a correção da assinatura com a mensagem e a chave pública.

… estes encontram-se criados, na classe **EdCDSA**, nos formatos: - *generateKeys*; - *signature*; - *verify*.

Com as chaves criadas, será escolhido um valor **t**, arbitrário, para executar o processo de assinatura t vezes; sendo este valor o nível de segurança que se quer ter no esquema.

Na variável *sign*, vão encontrar-se as t assinaturas criadas.

Estas vão ser posteriormente passados pelo *verify*, para construir as decisões determinísticas. Com todas criadas, verifica-se se o resultado é o **positivo**, *True*, pelo que o processo dá-se como terminado.

Este algoritmo irá ajudar a confirmar a segurança do protocolo, devido ao espírito probabilístico da função *signature*, pelo que os seus *outputs* **deverão** ser diferentes entre si.

0.6.2 Exemplos de teste

Para ambas as curvas foram criadas células para testar a capacidade de autenticação dos métodos criados. Ambos são idênticos, sendo a única mudança o tipo de curva usada (**ed25519** e **ed448**).

0.6.3 Edwards25519

[5]:
```
edcdsa = EdCDSA("ed25519")              # Classe EdCDSA
message1 = "Mensagem a ser assinada"    # Mensagem a assinar
t = 5                                   # Nível de segurança

print("Iniciar programa com mensagem" + message1)

# Processo de assinatura (KeyGen(L,λ))
prv_key, pub_key = edcdsa.generateKeys()
print("Private key: ")
print(prv_key)
print()
print("Public key: ")
print(pub_key)
print()

# Processo de assinatura (Sign(x,w,m,t))
i = 0       # variável para iterar
sign = []   # variável para guardar as assinaturas

while i < t:

    assinatura = edcdsa.signature(dumps(message1), prv_key)

    print("Assinatura nº${i}: ")
    print(assinatura)
    print()

    sign.insert(i, assinatura)

    i += 1
```

```python
# Verify(x,m,sign)

is_correct = True # Se uma assinatura não tiver correta -> False
j = 0             # variável para iterar

while j < t and is_correct:

    if edcdsa.verify(dumps(message1), sign[j], pub_key):
        print("Mensagem autenticada!")
    else:
        print("Mensagem não autenticada!")
        is_correct = False

    j += 1

print()

if(is_correct):
    print("Verificação Fiat-Shamir completa - Com sucesso!")

else:
    print("Verificação Fiat-Shamir completa - Sem sucesso!")
```

```
Iniciar programa com mensagemMensagem a ser assinada
7237005577332262213973186563042994240857116359379907606001950938285454250989
Private key:
665431671916301390281835320381298422511718061847272732538351824201282072243

Public key:
<__main__.ed object at 0x7fc5999736a0>

Assinatura nº${i}:
(4431932896055662749694489659021645961873827747988163470403334798083079444903,
2234436805087015887268933164299107908291685850943336856115329595535138775706)

Assinatura nº${i}:
(1041582818283963812121973103756703520841479304116111482541285106412477945450,
112488850785967946327111388601541821922074507114291740500106362808865954832)

Assinatura nº${i}:
(4202274250617037674734613974335338966445118929230642972171196907587714204060,
3380448508893453328030578194139962558425852437453766534956542245958880949248)

Assinatura nº${i}:
(6816869756725112507413351289661967825046471952864300941527735241861708167616,
4819982286565192806109268650504533296700386248900132685873777443771253554444)
```

```
Assinatura nº${i}:
(171194769013885555750863075674895063442352542922428077250310819594451770041 2,
946883561250372234362561370626852427909586472588319292045662809694780983898)

Mensagem autenticada!
Mensagem autenticada!
Mensagem autenticada!
Mensagem autenticada!
Mensagem autenticada!

Verificação Fiat-Shamir completa - Com sucesso!
```

0.6.4 Edwards448

[6]:
```python
edcdsa = EdCDSA("ed448")               # Classe EdCDSA
message1 = "Mensagem a ser assinada"    # Mensagem a assinar
t = 5                                   # Nível de segurança

print("Iniciar programa com mensagem" + message1)

# Processo de assinatura (KeyGen(L,λ))
prv_key, pub_key = edcdsa.generateKeys()
print("Private key: ")
print(prv_key)
print()
print("Public key: ")
print(pub_key)
print()

# Processo de assinatura (Sign(x,w,m,t))
i = 0       # variável para iterar
sign = []   # variável para guardar as assinaturas

while i < t:

    assinatura = edcdsa.signature(dumps(message1), prv_key)

    print("Assinatura nº${i}: ")
    print(assinatura)
    print()

    sign.insert(i, assinatura)

    i += 1

# Verify(x,m,sign)
```

```python
is_correct = True # Se uma assinatura não tiver correta -> False
j = 0             # variável para iterar

while j < t and is_correct:

    if edcdsa.verify(dumps(message1), sign[j], pub_key):
        print("Mensagem autenticada!")
    else:
        print("Mensagem não autenticada!")
        is_correct = False

    j += 1

print()

if(is_correct):
    print("Verificação Fiat-Shamir completa - Com sucesso!")

else:
    print("Verificação Fiat-Shamir completa - Sem sucesso!")
```

```
Iniciar programa com mensagemMensagem a ser assinada
18170968107390172263733095197200113358841034017182951507037254979514600396153958
5716195755291692375963310293709091662304773755859649779
Private key:
53328581435890060107908859355181628767978219129673482557474134428583090975183163
6487742194582490126025541955970184811497307937081O176

Public key:
<__main__.ed object at 0x7fc598500d90>

Assinatura nº${i}:
(1061109447261332183071009997850775996167911560591794083129358229571904665208591
36229310947635916531916769105468011736866482397879773397, 1792176388808389011568
80106961814945226494367490058384495196891726359201625370450400784592992926508592
789425449770543246860904156410583)

Assinatura nº${i}:
(1062462900705379580315174629078192925855341936844647463528059603854073644359859
9142043884208612108386198826003640330521301139125534249, 28068721714379341304806
38292287203889487171489067261002007243051066057402685856512274369458839340977711
1792712167665800431574408721509)

Assinatura nº${i}:
(1135689088152538985842635445122499387730111838851140332195922198021923655182660
40395637255550313802717308209023836781813533502045460759, 6369760235352629300696
60950971514523692773432802962103873484816203868540299950144101493023859002947999
```

```
616393632372666341776517377284444)

Assinatura nº${i}:
(6546212352767867378564543302134879932209623175509723463169756462922728566648425
3139458366058991861607909464037817589556236678257535352, 1722446706019658819762924
2438047180285704344086861594942151089701572323645658319063278371210611706321875 0
80476612753672687992491095485566)

Assinatura nº${i}:
(1144805685533957018839460252757004320241202649549808272607315419648523833925820
9341638742717589221650424008839378477533735931161802741 2, 1305696215894658684148
5091826513463150318214668071696023163136832175330684600108366125305112037118411 3
30397787979641912412004375477 8425)

Mensagem autenticada!
Mensagem autenticada!
Mensagem autenticada!
Mensagem autenticada!
Mensagem autenticada!

Verificação Fiat-Shamir completa - Com sucesso!
```