Estruturas Criptográficas

Trabalho Prático 2 - Exercício 3

José de Matos Moreira - PG53963

Pedro Freitas - PG52700

Enunciado do problema

O algoritmo de **Boneh e Franklin** (BF), discutido no **Capítulo 5b: Curvas Elípticas e sua Aritmética**, é uma técnica fundamental na chamada **"Criptografia Orientada à Identidade"**. Seguindo as orientações definidas nesse texto, pretende-se construir, usando **Sagemath**, uma classe **Python** que implemente este criptossistema.

Resolução

Em primeiro lugar, apresentam-se os *imports* efetuados.

In [3]:
```
from sage.all import *
import hashlib
import random
```

BF

Tendo como base a teoria e o código presentes no **Capítulo 5b**, elaborou-se todo este exercício dentro da classe **BF**. Aqui, desenvolveram-se as várias funções responsáveis, não só, pela aplicação do algoritmo, mas, também, todas as funções auxiliares de *hash* e conversão. Deste modo, passa-se, assim, a explicar toda a resolução do presente exercício:

- **__init__**: o construtor da classe inicializa os parâmetros necessários para o esquema criptográfico. Ele recebe um parâmetro **lmbda**, que é usado na geração dos parâmetros de segurança e, posteriormente, na criação do valor primo **q**. Posteriormente e, recorrendo ao código fornecido, definiram-se as variáveis correspondentes à curva elítica e ao grupo de torção
- **phi**: representa a isogenia responsável pelo mapeamento **(x, y) -> (z * x, y)**, recorrendo à curva elítica definida em fase de construção da classe
- **ex**: função responsável pelo emparelhamento de **Tate**

- **trace**: função auxiliar que soma um ponto **P** com o resultado do mesmo elevado ao primo **p**
- **Zr**: **PRG** que, a partir de um *nounce*, gera um número pertencente a $\mathbb{Z}_q$ ($\mathbb{N}$ -> $\mathbb{Z}_q$)
- **f**: função que, recorrendo à função **trace**, cria um valor em $\mathbb{Z}$, a partir de um ponto pertencente a $\mathbb{F}_{p^2}$
- **h**: função responsável pela conversão Bytes -> $\mathbb{Z}$
- **H**: algoritmo que, recorrendo a uma função de *hash*, aplica a conversão $\mathbb{Z}$ -> $\mathbb{Z}_q$
- **g**: função que, recorrendo a um inteiro, calcula um elemento pertencente ao grupo de torção $\mathbb{G}$ ($\mathbb{Z}$ -> $\mathbb{G}$)
- **id**: recorrendo às funções **g** e **h**, anteriormente mencionadas, transforma um valor em **bytes** num valor pertencente ao grupo de torção
- **keygen**: algoritmo que gera um segredo administrativo e uma chave pública administrativa
- **keyextract**: função que usa a informação de administração para extrair a chave privada associada à chave pública
- **in_encrypt**: responsável por preparar os dados de entrada para o processo de cifragem
- **out_encrypt**: função que cifra os dados de entrada
- **encrypt**: algoritmo que, recorrendo às duas funções anteriores, realiza o processo completo de cifragem
- **in_decrypt**: função que, analogamente, prepara os dados de entrada para o processo de decifragem
- **out_decrypt**: função que, a partir dos dados anteriormente gerados, realiza o processo de decifragem
- **decrypt**: algoritmo que, a partir das duas funções anteriormente descritas, aplica o processo completo de decifragem as dados cifrados

In [11]:
```
class BF():

    def __init__(self, lmbda):
        self.lmbda = lmbda
        self.bq = 2 ^ (self.lmbda - 1)
        self.bp = 2 ^ self.lmbda - 1
        self.q = random_prime(self.bp, lbound = self.bq)


        t = self.q * 3 * 2 ^ (self.bp - self.bq)
        while not (t - 1).is_prime():
            t = t << 1

        self.p = t - 1
        Fp = GF(self.p)
        R.<z> = Fp[]
        f = R(z ^ 2 + z + 1)
        Fp2.<z> = GF(self.p ^ 2, modulus=f)
```

```
        self.z = z
        self.E2 = EllipticCurve(Fp2, [0,1])

        cofac = (self.p + 1) // self.q
        self.G = cofac * self.E2.random_point()


    def phi(self, P):
        (x, y) = P.xy()

        return self.E2(self.z * x, y)


    def ex(self, P, Q, l=1):
        return P.tate_pairing(self.phi(Q), self.q, 2) ^ l


    def trace(self, P):
        return P + P ^ self.p


    def Zr(self, nounce):
        encoded_nounce = str(nounce).encode()
        hashed_nounce = hashlib.sha256(encoded_nounce).digest()
        int_hashed_nounce = int.from_bytes(hashed_nounce, byteorder='big') %

        return int_hashed_nounce


    def f(self, P):
        tp = self.trace(P)

        return ZZ(tp)


    def h(self, bts):
        hash_object = hashlib.sha256()
        hash_object.update(bts)
        hex_hash = hash_object.hexdigest()

        return Integer('0x' + hex_hash)


    def H(self, z):
        encoded_z = str(z).encode()
        hashed_z = hashlib.sha256(encoded_z).digest()
        int_hashed_z = int.from_bytes(hashed_z, byteorder='big') % self.q

        return int_hashed_z


    def g(self, s):
        return s * self.G


    def id(self, bts):
```

```
        return self.g(self.h(bts))


    def keygen(self):
        s = self.Zr(random.randint(self.bq, self.bp))
        beta = self.g(s)

        return s, beta


    def keyextract(self, id, s):
        d = self.id(id)

        return s * d


    def in_encrypt(self, id, x, beta):
        d = self.id(id)
        v = self.Zr(random.randint(self.bq, self.bp))
        a = self.H(v ^^ x)
        u = self.ex(beta, d, a)

        return x, v, a, u


    def out_encrypt(self, x, v, a, u):
        alpha = self.g(a)
        vl = v ^^ self.f(u)
        xl = x ^^ self.H(v)

        return alpha, vl, xl


    def encrypt(self, id, x, beta):
        x, v, a, u = self.in_encrypt(id, x, beta)

        return self.out_encrypt(x, v, a, u)


    def in_decrypt(self, alpha, vl, xl, key):
        u = self.ex(alpha, key, 1)
        v = vl ^^ self.f(u)
        x = xl ^^ self.H(v)

        return alpha, v, x


    def out_decrypt(self, alpha, v, x):
        a = self.H(v ^^ x)
        if alpha != self.g(a):
            return None

        return x


    def decrypt(self, key, alpha, vl, xl):
```

```
        alpha, v, x = self.in_decrypt(alpha, vl, xl, key)

        return self.out_decrypt(alpha, v, x)
```

Testes de aplicação

In [12]:
```
id = b'messi'
x = 24061987
bf_cs = BF(7)
s, beta = bf_cs.keygen()
key = bf_cs.keyextract(id, s)


alpha, vl, xl = bf_cs.encrypt(id, x, beta)
decryption = bf_cs.decrypt(key, alpha, vl, xl)

if decryption == None:
    print('[ERROR] decryption failed')
else:
    print(f'[CORRECT DECRYPTION] {decryption}')
```

```
[CORRECT DECRYPTION] 24061987
```