Estruturas criptograficas: TP2 problema 3

Para a realização deste problema, definimos as funções auxiliares (ex. ID, g, h, H, ...), tal como estão definidas nos apontamentos do docente na secção do criptosistema de Boneh-Franklin.

Geração de chaves

Para a geração de um segredo administrativo **s** e uma chave pública administrativa **beta**, foi gerado um nonce de forma pseudo-aleatória, e posteriormente aplicada uma função de **hash**, (previamente definida no construtor da classe). De seguida, o resultado dessa função de hash foi transformado em inteiro e aplicado um mod com o valor de **q**, resultando desta forma, no segredo **s**. Para obter a chave pública **beta** multiplicámos a chave **s** com o ponto **G**, pertencente à curva elíptica definida através dos parâmetros públicos establecidos, que mais uma vez foram definidos no construtor da classe.

Cifragem

O processo de cifragem, consiste na transformação da chave pública, (gerada através do id do emissor) e do plaintext, num criptograma pronto a ser enviado para o recetor. Desta forma, o recetor poderá associar uma mensagem, a uma identidade criptográfica.

Para cifrar a mensagem, seguimos todos os passos descritos tanto nos apontamentos da cadeira, como no RFC5091.

Extração de chave

O algoritmo de extração de chave consiste, na multiplicação no segredo administrativo **s** que é transmitido ao recetor aravés de um canal seguro (privado), com a chave públia **id**, obtendo desta forma uma chave capaz de decifrar o criptograma e consequentemente confirmar a identidade do emissor.

Decifragem

Para deifrar o criptograma, são executadas as operações descritas no RFC, envolvendo o criptograma e a chave obtida através da função KeyExtract, com o objetivo obter a mensagem criada e enviada pelo emissor.

Casos de Teste

Para testar o bom funcionamento do algoritmo de cifragem orientada à identidade, ciframos uma mensagem qualquer, com uma identidade qualquer, neste caso "Universidade do Minho", e decifamos a mesma mensagem com a identidade correta e outra com a identidade incorreta. Como seria de esperar, obtivémos os resultados pretendidos.

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

In [2]:
```
class BF:

    def __init__(self):
        self.version = 2
        self.n = 1024
        self.n_p = 512
        self.n_q = 160
        self.hashfnc = hashlib.sha1

        self.q = random_prime(2^self.n_q-1,lbound=2^(self.n_q-1))

        # print("q:" + str(q))

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

        self.p = t - 1

        # print("p:" + str(p))

        self.Fp     = GF(self.p)                  # corpo primo com "p" e
        R.<z>  = self.Fp[]                    # anel dos polinomios em "z"
        f     = R(z^2 + z + 1)
        Fp2.<z> = GF(self.p^2, modulus=f)
        self.E2 = EllipticCurve(Fp2, [0,1])

        # ponto arbitrário  de ordem "q" em E2
        cofac = (self.p + 1)// self.q
        self.G = cofac * self.E2.random_point()
        # print("G:" + str(G.xy()))


    def ID(self, id):
        id_int = self.h(id)
        return self.g(id_int)

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

    def h(self, a):
        return int.from_bytes(a.encode(), byteorder='big')

    def H(self, a):
        return mod(a, self.q)

    def KeyGen(self):
        s = 0
        while s <= 2:
```

```
            nounce = os.urandom(16)
            s = self.hashfnc(nounce).digest()
            s = int.from_bytes(s, byteorder='big')
            s = s % self.q

        beta = self.g(s)

        return s, beta

    def KeyExtract(self, id, s):
        d = self.ID(id)

        key = s * d

        return key

    def phi(self, P):              # a isogenia que mapeia  (x,y)  ->  (z*
        R.<z>  = self.Fp[]
        f      = R(z^2 + z + 1)
        Fp2.<z> = GF(self.p^2, modulus=f)

        (x,y) = P.xy()
        return self.E2(z*x,y)

    def TateX(self, P, Q, l=1):      # o emparelhamento de Tate generaliz
        return P.tate_pairing(self.phi(Q), self.q, 2)^l

    def trace(self, x):       # função linear que mapeia Fp2  em  Fp
        return x + x^self.p

    def Encrypt(self, beta, id, x):
        x = self.h(x)

        #IN
        d = self.ID(id)

        nounce = os.urandom(16)
        v = self.hashfnc(nounce).digest()
        v = int.from_bytes(v, byteorder='big')
        v = v % self.q

        a = self.H(v ^^ x)

        mu = self.TateX(beta, d, a)


        #OUT
        alfa = self.g(a)

        v_ = self.trace(mu)
        v_ = int(v) ^^ int(v_)

        x_ = int(x) ^^ int(self.H(v))

        print("Encryption finished!")

        return alfa, v_, x_


    def Decrypt(self, alfa, v_, x_, key):
```

```
        #IN
        mu = self.TateX(alfa, key)

        v = self.trace(mu)
        v = int(v_) ^^ int(v)

        x = int(x_) ^^ int(self.H(v))

        #OUT
        a = self.H(v ^^ x)

        if alfa == self.g(a):
            m = (int(x).to_bytes((int(x.bit_length()) + 7) // 8, byteorde
            print("Decryption successful!\nMessage: " + m)
            return m
        else:
            print("Decryption failed!")
            return None
```

Geração de chave e Cifragem

In [3]:
```
bf = BF()

s, beta = bf.KeyGen()

alfa, v_, x_ = bf.Encrypt(beta, "Universidade do Minho", "Uma mensagem qu
```

```
Encryption finished!
```

Decifragem Bem Sucedida

In [4]:
```
key = bf.KeyExtract("Universidade do Minho", s)

m = bf.Decrypt(alfa, v_, x_, key)
```

```
Decryption successful!
Message: Uma mensagem qualquer
```

Decifragem com uma identidade incorreta

In [5]:
```
key = bf.KeyExtract("Universidade de Coimbra", s)

m = bf.Decrypt(alfa, v_, x_, key)
```

```
Decryption failed!
```