TP2 - Exercício 3

Neste exercício foi nos pedidos para implementarmos um criptosistema baseado nos algoritmo de Boneh e Franklin. Para tal é necessário implementar 4 funções principais:

- KeyGen(λ), onde λ é o parametro de segurança
- KeyExtract(id), onde id é a identificação do cliente de destino
- Encrypt(id,x), onde id é a identificação do cliente de destino e x é o plaintext que pretendemos enviar
- Decrypt, que recebe a chave e o criptograma criado para obter o plaintext.

De maneira a ser possível implementar este algoritmo é necessário a criação de funções de hash e conversão.

Funções de “hash” e de conversão

- $\mathsf{Zr} : \mathbb{N} \rightarrow \mathbb{Z}_q$
- $f : \mathbb{F}_{p^2} \rightarrow \mathbb{Z}$
- $h : \mathsf{Bytes} \rightarrow \mathbb{Z}$
- $H : \mathbb{Z} \rightarrow \mathbb{Z}_q$
- $g : \mathbb{Z} \rightarrow \mathbb{G}$
- $\mathsf{ID} : \mathsf{Bytes} \rightarrow \mathbb{G}$

Funções Principais

KeyGen

Gera um primo com tamanho proximo ao parametro de segurança passado como input. Através deste primo é calculado um inteiro,s, no intervalo de q,o segredo que irá ser passado à entidade que decifrará a mensagem, s também que irá ser usado para calcular β, multiplicando s com o ponto gerador do grupo de torção G, que irá ser passado para cifrar a mensagem.

KeyExtract

Utilizado para gerar a chave que irá ser utilizada no processo de cifragem e decifragem. Calculado através do segredo,s, com o resultado de ID da identidade da entidade de quem irá receber a mensagem.

Encrypt

In: que processa os “inputs” e constrói uma estrutura intermédia do tipo ⟨x,v,a,μ⟩ onde x é o plaintext, v é um inteiro no intevalo de q, a é um valor no intervalo que resultante do hash do xor entre v e x e μ é o resultado do o emparelhamento de Tate entre β,d,a, onde β foi calculado no KeyGen e d é calculado através de ID(id)

Out: que constrói o output a partir da estrutura intermédia, output este da forma ⟨α,v′,x′⟩, onde α é o resultado de a∗G, onde G determinado previamente, v′ resultado do xor entre v, calculado em in, e f(μ), o resulto de transformar μ num inteiro através da função trace e do construtor ZZ, u também calculado em in, e o xor entre o plaintext e um valor no intervalo q resultante do hash de v.

Decrypt

In: que processa os “inputs” (key,α,v′,x′), onde key é obtida a partir do algoritmo **KeyExtract** e α,v′e x′ são o criptograma recebido. Reconstroi a estrutura intermédia ⟨x,v,a⟩ e contendo o a mensagem decifrada e os valores necessários para confirmar o seu valor. v é calculado através do xor entre v′ e f(μ), onde μ é calculado através de TateX(α,key,1) e o plaintext é calculado através do xor entre x′ e H(v).

Out: que recebe a estrutura intermédia e verifica se o criptograma foi bem construido verificando se o resultado de g(H(v⊕x)) é igual a α.

In [1]:
```
import hashlib

class BF_Cryptosystem():


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


        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 trace(self, x):
        return x + x^self.p

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


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

        return int_hashed_nounce


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

        z = ZZ(fp)
        return z



    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 = self.h(encoded_z)
        int_hashed_z = int(hashed_z) % 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()
        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()
        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)
        result = self.out_decrypt(alpha, v, x)

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


id = b'123123'

bf_cs = BF_Cryptosystem(4, 12345677654321)
s, beta = bf_cs.keygen()
key = bf_cs.keyextract(id, s)

x = 12345

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

```
[CORRECT DECRYPTION] 12345
```