

TP2-1

March 28, 2023

1 Trabalho Prático 2

Trabalho realizado pelo grupo 11:

- Beatriz Fernandes Oliveira, PG50942
- Bruno Filipe Machado Jardim, PG49997

1.1 Problema 1

1. Construir uma classe Python que implemente um KEM - ElGamal. A classe deve:
 - Inicializar cada instância recebendo o parâmetro de segurança (tamanho em bits da ordem do grupo cíclico) e gere as chaves pública e privada.
 - Conter funções para encapsulamento e revelação da chave gerada.
 - Construir, a partir deste KEM e usando a transformação de Fujisaki-Okamoto, um PKE que seja IND-CCA seguro.

1.1.1 A

É necessário garantir que o DLP seja complexo, para isto não basta que o p seja grande, é necessário que o maior fator primo de $(p - 1)$ é também grande:

Para garantir estas condições o primo p tem de ser gerado de uma determinada forma: - É gerado um primo q , com pelo menos 160 bits de tamanho - Geram-se sucessivamente inteiros $p = q * 2 + 1$ e q até que p seja um primo suficientemente grande para satisfazer o parâmetro de segurança

Optamos por este método pois quando utilizamos o método descrito no [Capítulo 3a: Introdução à Álgebra Abstrata \(continuação\)](#), isto causou com que o parâmetro p se tornasse extremamente grande e causasse problemas na implementação da transformação Fujisaki-Okamoto.

1.1.2 B

As funções de encapsulamento e revelação foram definidas da seguinte forma:

- $KEM(\beta) \equiv \vartheta r \leftarrow \mathbb{Z}_q \setminus 0 \cdot \vartheta key \leftarrow \beta^r \bmod p \cdot \vartheta enc \leftarrow g^r \bmod p \cdot (key, enc)$
- $KRev(a, enc) \equiv enc^a \bmod p$

```
[1]: class ElGamal:
    def __init__(self, size):
    def genPrime():
        q_size = 160
```

```

        q = random_prime(ZZ.random_element(2^(q_size-1),2^q_size-1))
        pi = 2*q + 1

        while not is_prime(pi) and len(pi.binary()) < self.size:
            q = next_prime(q)
            pi = 2*q + 1

        return pi,q

    def genParams():
        p,q = genPrime()
        R = GF(p)
        g = R.multiplicative_generator()
        return (p, q, g)

    self.size = size
    self.p, self.q, self.g = genParams()

    def keyGen(self):
        a = ZZ.random_element(2, (self.q)-1)
        beta = power_mod(self.g, a, self.p)
        return a, beta

    def KEM(self, beta, r=None):
        if r is None:
            r = ZZ.random_element(1, (self.q)-1)

        key = power_mod(beta, r, self.p)
        enc = power_mod(self.g, r, self.p)
        return key, enc

    def KRev(self,a ,enc):
        return power_mod(enc, a, self.p)

```

```

[2]: # Generation of the keys
eg = ElGamal(1024)
alice_pvk, alice_pbk = eg.keyGen()
bob_pvk, bob_pbk = eg.keyGen()
print("Alice's keys: ", alice_pvk, alice_pbk)
print("Bob's keys: ", bob_pvk, bob_pbk)

```

```

Alice's keys:  88954212864795234711183281976193608245606840828
345194281402849483193736564786774756074963423310

```

Bob's keys: 10727876611202689404025585254945538564741967408
130292690589652448018258306229434629834791848156

```
[3]: # Sharing of the keys

alice_key, alice_enc = eg.KEM(bob_pbk)
print("Alice: ", alice_key)
bob_key = eg.KRev(bob_pvk, alice_enc)
print("Bob: ", bob_key)
```

Alice: 260980077287140300146424847704917592233955327643
Bob: 260980077287140300146424847704917592233955327643

1.1.3 C

Para esta última alínea tínhamos que transformar o nosso *KEM* num *PKE-IND-CCA*

Dado isto construímos a classe *FO_ElGamal*, que constrói o seu próprio *KEM-ElGamal* que depois utiliza nas funções *encrypt* e *decrypt*.

A função de cifração é definida da seguinte forma:

$$E'(x) \equiv \vartheta r \leftarrow h \cdot \vartheta y \leftarrow x \oplus g(r) \cdot (e, k) \leftarrow f(y||r) \cdot \vartheta c \leftarrow k \oplus r \cdot (y, e, c)$$

A função de decifração será da seguinte forma:

$$D'(y, e, c) \equiv \vartheta k \leftarrow \text{KRev}(e) \cdot \vartheta r \leftarrow c \oplus k \cdot \text{if } (e, k) \neq f(y||r) \text{ then } \perp \text{ else } y \oplus g(r)$$

```
[4]: import hashlib
import os
```

```
[5]: def xor(b, a):
    return bytes([a^b for a,b in zip(a,b)])
```

```
[6]: class FO_ElGamal:
    def __init__(self, size):
        self.kem = ElGamal(size)

    def encrypt(self, m, key):
        x = m.encode()

        r = ZZ.random_element(2, (self.kem.q)-1)
        r = str(r).encode()
        # Cipher r
        gr = hashlib.sha256(r).digest()
        # XOR
        y = xor(x, gr)
        # Concat r and y
        yr = y + r
        yr = int.from_bytes(yr, byteorder='big')
        # KEM
```

```

        k, e = self.kem.KEM(key, yr)
        k_ = int(k).to_bytes(len(r), byteorder='big')
        # XOR k and r
        c = xor(k_,r)

        return y,e,c

def decrypt(self, y, e, c, pvk, pbk):
    # KRev
    k = self.kem.KRev(pvk, e)
    k_ = int(k).to_bytes(len(c), byteorder='big')

    # XOR k and c
    r = xor(k_,c)

    # Check if the decryption can be done
    yr = y + r
    yr = int.from_bytes(yr, byteorder='big')

    if (k,e) != (self.kem.KEM(pbk, yr)):
        return "Decryption failed"

    # Decryption
    g = hashlib.sha256(r).digest()
    pt = xor(g,y)

    return pt.decode()

```

```
[7]: fo = FO_ElGamal(1024)
```

```

# Gen Keys
apvk, apbk = fo.kem.keyGen()
bpvk, bpbk = fo.kem.keyGen()

```

```

[8]: y,e,c = fo.encrypt("TP2 Fujisaki-Okamoto", bpbk)
     pt = fo.decrypt(y,e,c, bpvk, bpbk)
     pt

```

```
[8]: 'TP2 Fujisaki-Okamoto'
```