

# Estruturas Criptográficas 2022/23

## TP2. Problema 1

### Grupo 7. Leonardo Berteotti e Paulo R. Pereira

Pretende-se a construção de uma classe Python que implemente um **KEM - El Gamal**.

Em primeiro lugar, a classe deve inicializar cada instância recebendo o parâmetro de segurança (tamanho em bits da ordem do grupo cíclico). O **KEM - El Gamal** utiliza os parâmetros públicos do protocolo Diffie-Hellman. Por sua vez, as técnicas da família Diffie-Hellman utilizam as propriedades de um grupo cíclico multiplicativo  $G \equiv \mathbb{Z}_p^*$  em que  $p$  é número primo grande. A ordem deste grupo é  $p - 1$ . Assim, é necessário encontrar um primo  $p$  cuja ordem tenha um tamanho igual ou superior ao parâmetro de segurança. Além disso,  $p$  tem um divisor primo  $q$  grande (de forma a que DLP seja complexo). Assim,

1. começa-se por gerar o primo  $q$  tal que o seu tamanho é igual ou superior a 160 bits (20 bytes);
2. gerar sucessivamente  $p_i = q \times 2^i + 1$  até que  $p_i - 1$  tenha tamanho igual ou superior ao critério de segurança (passado como parâmetro);
3. é gerado  $g \in \mathbb{Z}_p^*$  de ordem  $q$ ;
4. obtém-se o tuplo  $(p, q, g)$ .

Para gerar  $g$ , uma vez que todo o grupo multiplicativo da forma  $\mathbb{Z}_p^*$  é um grupo cíclico de ordem  $p - 1$ , um gerador deste grupo pode-se determinar por tentativas percorrendo os pequenos primos  $(2, 3, 5, \dots)$  e determinando a ordem de cada um. Um primo cuja ordem seja  $p - 1$  é um gerador. No entanto, queremos que  $g$  tenha ordem  $q$ . Assim, podemos usar o algoritmo de encontrar geradores de grupos cíclicos, baseado no facto de que se  $g$  é um gerador de  $\mathbb{Z}_p^*$ , então  $g^{(p-1)/q}$  é um gerador de ordem  $q$ .

O algoritmo consiste em gerar aleatoriamente um elemento  $g$  de  $\mathbb{Z}_p^*$  e calcular  $h = g^{(p-1)/q}$ . Se  $h^q = 1$  e se  $h \neq 1$ , então  $h$  é um gerador de ordem  $q$ .

Deste modo, são implementados os seguintes métodos:

- KeyGen :
  1. Parâmetros públicos  $p, q, g$  como no protocolo DH
  2. A chave privada é  $a \neq 0 \in \mathbb{Z}_q$  gerada aleatoriamente;
  3. A chave pública é  $\beta \equiv g^a \pmod{p}$
- KEM ( $\beta$ )  $\equiv \vartheta r \leftarrow \mathbb{Z}_q \setminus 0 \cdot \vartheta \text{ key} \leftarrow \beta^r \pmod{p} \cdot \vartheta \text{ enc} \leftarrow g^r \pmod{p} \cdot (\text{key}, \text{enc})$
- KRev ( $a, \text{enc}$ )  $\equiv \text{enc}^a \pmod{p}$

Os métodos DEM e DRev usam a primitiva Authenticated symmetric encryption ChaCha20Poly1305. Neste caso em particular, como a chave simétrica precisa de ter 32 bytes, foi usado um **KDF** para, a partir da chave acordada, gerar a nova chave acordada de 32 bytes. Note-se que para chave ser igual nos dois agentes, os parâmetros que conferem a aleatoriedade devem ser iguais e por isso são passados por argumento.

In [3]:

```
from sage.all import *
from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes

def find_generator(p, q):
    while True:
        g = Zmod(p).random_element()
        h = g**((p-1)//q)
        if h**q == 1 and h != 1:
            return h

def generate_private_key(q):
    # generate a random integer between 1 and q-1 (inclusive)
    a = randrange(1, q)
    # ensure that the element is non-zero
    while a == 0:
        a = randrange(1, q)
    return a
```

In [4]:

```
class KEM_ElGamal:
    def __init__(self, s):
        self.s = s

    def key_gen(self):
        q = random_prime(2**160-1, True, 2**(160-1))
        i = 0
        while True:
            p = q * 2**i + 1
            if (p-1).bit_length() >= self.s and is_prime(p):
                break
            i += 1
        g = find_generator(p, q)

        # generate_private_key(q) generates a random non-zero element in Zq
        sk = generate_private_key(q)
        pk = power_mod(g, sk, p)
        return p, q, g, sk, pk

    # generate key and key encapsulation
    def KEM(self, pk, p, q, g):
        r = generate_private_key(q)
        key = power_mod(pk, r, p)
        enc = power_mod(g, r, p)
        return key, enc

    # reveals key
    def KRev(self, sk, enc, p):
        return power_mod(enc, sk, p)

    # encapsulates plaintext using the primitive of
    # authenticated encryption ChaCha20Poly1305.
    def DEM(self, plaintext, key, nonce):
        key_bytes = str(key).encode('utf-8')
        hkdf = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=b"salt",
            info=b"additional info",
        )
        cipher_key = hkdf.derive(key_bytes)
        chacha = ChaCha20Poly1305(cipher_key)
        aad = b"authenticated but unencrypted data"
        ciphertext = chacha.encrypt(nonce, plaintext, aad)
        return ciphertext

    def DRev(self, ciphertext, sk, enc, p, nonce):
        key = self.KRev(sk, enc, p)
        key_bytes = str(key).encode('utf-8')
        hkdf = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=b"salt",
            info=b"additional info",
        )
        cipher_key = hkdf.derive(key_bytes)
        chacha = ChaCha20Poly1305(cipher_key)
```

```

aad = b"authenticated but unencrypted data"
plaintext = chacha.decrypt(nonce, ciphertext, aad)
return plaintext

```

In [5]:

```

# Testing
kem = KEM_ElGamal(1024)
p, q, g, sk, pk = kem.key_gen()

k, e = kem.KEM(pk, p, q, g)

nonce = os.urandom(12)

plaintext = 'hello there :)'
print("Plaintext:\n" + plaintext)
ciphertext = kem.DEM(plaintext.encode('utf-8'), k, nonce)
print("\nCiphertext:")
print(ciphertext.decode('unicode_escape'))
print("\nDecrypted ciphertext:")
decrypted_ciphertext = kem.DRev(ciphertext, sk, e, p, nonce)
print(decrypted_ciphertext.decode('unicode_escape'))

```

Plaintext:  
hello there :)

Ciphertext:  
½ ÈÊ,cÑU ç]yZ m 9÷Djï îÃ ÿMòÇO

Decrypted ciphertext:  
hello there :)

Por fim, pretende-se, a partir do **KEM** já definido, e usando a transformação de Fujisaki-Okamoto, um **PKE** que seja *IND-CCA* seguro.

É então construído um esquema assimétrico  $E', D'$  através de

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

Portanto, têm-se o seguinte:

1. gerar um *random\_generated*  $r$  que é resultado do hash a um número pseudo-aleatório;
2. calcular  $g(r)$  a partir do novo hash  $g$ ;
3. efetuar o XOR entre o *plaintext*  $x$  e o  $g(r)$  do ponto 2 de forma a obter  $y$ ;
4. concatenar  $y$  com  $r$  e obter a chave  $e$  e o encapsulamento da chave  $k$ ,  $e$  tal como no método KEM da classe **KEM\_ElGamal**;
5. efetuar o XOR da chave  $k$  com o  $r$  para obter uma ofuscação da chave  $c$ .

In [6]:

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

def encryptFOT (pk, plaintext, p, q, g):
    r = hash(ZZ.random_element(0, p-1))
    gr = hash(str(r))
    plaintext_bytes = plaintext.encode('utf-8')
    gr_bytes = str(gr).encode('utf-8')
    y = xor(plaintext_bytes, gr_bytes)
    y_int = int.from_bytes(y, "big")
    conc = str(y_int) + str(r)
    k = power_mod(pk, int(conc), p)
    enc = power_mod(g, int(conc), p)
    k_bytes = str(k).encode('utf-8')
    r_bytes = str(r).encode('utf-8')
    c = xor(k_bytes, r_bytes)
    return y, enc, c

plaintext = "secret message"
print("Plaintext:\n" + plaintext)
y, enc, c = encryptFOT(pk, plaintext, p, q, g)
print("\nCiphertext:")
print(y.decode('utf-8'))
```

Plaintext:  
secret message

Ciphertext:  
JWSE\F UWDARPV

Para decifrar a mensagem, o algoritmo será

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

Este algoritmo é o "inverso" do descrito anteriormente.

In [7]:

```
def decryptFOT (pk, sk, y, e, c, p):
    k = kem.KRev(sk, e, p)
    k_bytes = str(k).encode('utf-8')
    r = xor(c, k_bytes)
    r = r.decode('utf-8')
    gr = hash(r)
    gr = str(gr).encode('utf-8')

    y_int = int.from_bytes(y, "big")
    conc = str(y_int) + str(r)

    derived_k = power_mod(pk, int(conc), p)
    derived_e = power_mod(g, int(conc), p)

    if derived_e == e and derived_k == k:
        return xor (y, gr)
    else:
        print("ERROR! The key doesn't match!")

decrypted_ciphertext = decryptFOT(pk, sk, y, enc, c, p)
print("Decrypted message:")
print(decrypted_ciphertext.decode('utf-8'))
```

Decrypted message:  
secret message