Exercício 1 - Trabalho Prático 2

Grupo 6:

Ruben Silva - pg57900

Luís Costa - pg55970

Problema:

1. Pretende-se construir em torno de uma cifra assimétrica um conjunto de técnicas criptográficas destinadas a fins distintos. Apesar de todas as alíneas do problema poderem ser respondidas com a maioria das cifras assimétricas clássicas ou pós-quânticas, neste problema vamos exemplificar o processo com uma técnica simples da família Diffie-Hellman nomeadamente a cifra assimétrica ElGamal com parâmetros de segurança λ.
   A. Implemente um esquema PKE $\mathsf{ElGamal}(\lambda)$ (ver Capítulo 4) num subgrupo de ordem prima q, com $|q| \geq \lambda$, do grupo multiplicativo $\mathbb{F}_p^*$ com p um primo que verifica $|p| \geq \lambda \times |\lambda|$. Identifique o gerador de chaves e os algoritmos de cifra de decifra neste esquema. Identifique o núcleo deterministico do algoritmo de cifra.
   B. Supondo que a cifra que implementou é IND-CPA segura (de novo Capítulo 4), usando a transformação de Fujisaki-Okamoto implemente um PKE que seja IND-CCA seguro.
   C. A partir de **2** construa um esquema de KEM que seja IND-CCA seguro.
   D. A partir de **2** construa uma implementação de um protocolo autenticado de "Oblivious Transfer" κ-out-of-n.

Implementação do Problema

In [8]:
```python
from sage.all import *
from cryptography.hazmat.primitives import hashes
import secrets
import random
```

Parte I

Foi seguida a seguinte informação presente no material da UC

EL GAMAL

Qualquer PKE é determinado por três algoritmos: geração de chaves, cifra e decifra. No contexto de ataques `IND-CPA` apenas os dois primeiros são relevantes e apenas a chave pública relevante.

$\text{GenKeys}(\lambda)$ ... λ é o parâmetro de segurança

1. gerar aleatoriamente um primo $q \approx 2^\lambda$
2. gerar um primo p tal que $\mathbb{F}_p^*$ tem um sub-grupo de ordem q
   * calcular um gerador g desse sub-grupo
3. gerar aleatoriamente $0 < s < q$, a chave privada
4. calcular e revelar a chave pública **pk** $\equiv \langle p, q, g, g^s \rangle$

$\text{Enc}(pk, m)$ ... a mensagem m é um elemento de $\mathbb{F}_p^*$

1. obter elementos públicos $p, q, g, g^s \leftarrow$ **pk**
2. gerar aleatoriamente $0 < \omega < q$
3. calcular $\gamma \leftarrow g^\omega$ e $\kappa \leftarrow (g^s)^\omega$.
4. construir o criptograma $\mathbf{c} \leftarrow \langle \gamma , m \times \kappa \rangle$

Note-se que se verifica $\kappa = \gamma^s$

É forte contra `IND-CPA` porque:

A criptografia ElGamal, com seu uso de γ e ω, é resistente a ataques de IND-CPA porque, sem a chave secreta s, o atacante não consegue obter informações sobre a mensagem original. O uso de ω (um valor aleatório) e a estrutura de κ garantem que a cifra seja não-determinística. Ou seja, mesmo se o atacante cifrar a mesma mensagem várias vezes, ele obterá criptogramas diferentes devido à aleatoriedade de ω.

Funções

1. É definido a função **find_p_q_g** que retorna p,q,g
   A. É gerado um q tal que $q \approx 2^\lambda$
   B. É gerado um p tal que $p = 2 * q + 1$ que claramente sastifaz $|p| \geq \lambda \times |\lambda|$
   C. Verificamos se esse p é primo.
   D. Descubrir o subgrupo

In [9]:
```
def find_p_q_g(lambda_bits):
    while True:
        q = random_prime(2**(lambda_bits-1), 2**lambda_bits)

        p = 2*q + 1

        if is_prime(p):
            # Encontrar o gerador
            for g in range(2, p):
                if pow(g, q, p) != 1 and pow(g, 2, p) != 1:
                    return p, q, g
```

1. É definido a função **string_to_int** que retorna o inteiro da string
2. É definido a função **int_to_string** que retorna a string apartir de uma inteiro

In [10]:
```
def string_to_int(s, p):
    message_bytes = s.encode('utf-8')

    # Converte bytes para inteiro
    message_int = int.from_bytes(message_bytes, byteorder='big')

    if message_int >= p -1:
        print("Mensagem muito grande para o campo Fp")
        print("Tente uma mensagem menor")
        print("ou")
        print("Aumente o tamanho do campo Fp (Aumentar lambda)")
        return None

    # Garantir que o número esteja dentro de Fp*
    return message_int % (p - 1)

def int_to_string(n):
    try:
        # Calcula o número de bytes necessários para representar n
        num_bytes = (n.bit_length() + 7) // 8
        byte_data = n.to_bytes(num_bytes, byteorder="big")

        return byte_data.decode('utf-8')
    except Exception as e:
        return f"Erro na decodificação: {e}"
```

1. É definido a função **ElGamal_GenKeys** que retorna a `public_key` e a `private_key`
   A. É gerado um q, p e g usando a função *find_p_q_g(λ)*
   B. É gerado a $0 <$ `private_key` $< q$
   C. É gerado o tuplo correspondente à `public_key` $\equiv \langle p, q, g, g^s \rangle$
   D. É devolvido o par de chaves

In [11]:
```
def ElGamal_GenKeys(lambdaa):
    p, q, g = find_p_q_g(lambdaa)

    # Private key: nr entre 0 and q-1
    private_key = secrets.randbelow(q)

    # Public key: (p, q, g, g^s)
    public_key = (p, q, g, pow(g, private_key, p))

    return public_key, private_key
```

1. É definido a função **ElGamal_Enc** que retorna o `criptograma`
   A. obter elementos públicos $p, q, g, g^s \leftarrow$ pk
   B. gerar aleatoriamente $0 < \omega < q$
   C. calcular $\gamma \leftarrow g^\omega$ e $\kappa \leftarrow (g^s)^\omega$.
   D. construir o criptograma $\mathbf{c} \leftarrow \langle \gamma\, ,\, m \times \kappa \rangle$

In [12]:
```
def ElGamal_Enc(public_key, message):
    p, q, g, gs = public_key

    message_int = string_to_int(message, p)

    # gerar w entre 0 e q-1
    w = secrets.randbelow(q)

    # Calcular Gamma e Omega corretamente
    gamma = pow(g, w, p)
    omega = pow(gs, w, p)

    # Calcular C
    c = (gamma, (omega * message_int) % p)
    return c
```

1. É definido a função **ElGamal_Dec** que retorna a mensagem decifrada
   A. obter p a partir da `public_key`
   B. obter γ e o ciphertext a partir do `criptograma`
   C. Fazer os cálculos inversos aplicados no **ElGamal_Enc**

In [13]:
```
def ElGamal_Dec(secret_key, public_key, criptograma):
    p = public_key[0]
    gamma, ciphertext = criptograma

    # Calcular kappa = gamma^secret_key mod p
    kappa = pow(gamma, secret_key, p)

    # Calcular inverso modular
    kappa_inv = inverse_mod(kappa, p)

    # Decifrar a mensagem
    message_int = (ciphertext * kappa_inv) % p

    return int_to_string(message_int)
```

Correr Ex1

In [14]:
```
def ElGamal_ex1(lambdaa, mensagem):
    print("\n==== INICIANDO CIFRAGEM ELGAMAL ====\n")

    # Verificar se a mensagem pode ser representada em Fp*
    if string_to_int(mensagem, 2**lambdaa) is None:
        print("Erro: A mensagem não pode ser representada no espaço de Fp*. Aume
        return

    print(f"Lambda selecionado: {lambdaa}-bits")

    # Gerar chaves pública e privada
    public_key, private_key = ElGamal_GenKeys(lambdaa)

    print("\n==== CHAVES GERADAS ====")
    print(f"Chave Pública (p, q, g, g^s): \n{public_key}")
    print(f"Chave Privada (s): {private_key}")
    print("===================\n")
```

```python
    # Cifrar a mensagem
    criptograma = ElGamal_Enc(public_key, mensagem)

    print("\n==== CIFRAGEM ====")
    print(f"Mensagem original: {mensagem}")
    print(f"Criptograma gerado (γ, m * κ): {criptograma}")
    print("==================\n")

    # Decifrar a mensagem
    mensagem_decifrada = ElGamal_Dec(private_key, public_key, criptograma)

    print("\n==== DECIFRAGEM ====")
    print(f"Mensagem Decifrada: {mensagem_decifrada}")


    # Verificar a Cifragem
    if mensagem_decifrada == mensagem:
        print("\nFuncionou! A mensagem foi recuperada corretamente.")
    else:
        print("\nErro! A decifração falhou.")

    print("\n==== Nucleo Deterministico ====")
    print(f"c -> {criptograma} \nElGamal_Dec(private_key, public_key, criptogram
```

In [15]:

```python
lambdaa = 100
message = "Hello World"
ElGamal_ex1(lambdaa, message)
```

```
==== INICIANDO CIFRAGEM ELGAMAL ====

Lambda selecionado: 100-bits

==== CHAVES GERADAS ====
Chave Pública (p, q, g, g^s):
(1248139363892158849556220127679, 624069681946079424778110063839, 13, 54591848980
0967976683817338128)
Chave Privada (s): 350883497988038238129680359342
==================



==== CIFRAGEM ====
Mensagem original: Hello World
Criptograma gerado (γ, m * κ): (515506813419386164759758667661, 50302423805096192
3541091891775)
==================



==== DECIFRAGEM ====
Mensagem Decifrada: Hello World

Funcionou! A mensagem foi recuperada corretamente.

==== Nucleo Deterministico ====
c -> (515506813419386164759758667661, 503024238050961923541091891775)
ElGamal_Dec(private_key, public_key, criptograma)
```

Part II

Foi seguida a seguinte informação presente no material da UC

Fujisaki-Okamoto

1. **Aleatoriedade**
   A. Escolhe-se um Valor Aleatorio r de tamanho λ em bits
2. **Ofuscação da Mensagem** x
   A. Gerar y tal que $y = x \oplus g(r)$
3. **Derivação da Aleatoriedade Determinística**
   A. r é misturado com y numa outra hash h tal que $r' = h(r, y)$
4. **Cifra Assimétrica**
   A. O par (y, c) é o `novo criptograma`
   B. $c = f(r, h(r, y))$ ou $c = f(r, r')$

NOTAS

1. g é uma "hash pseudoaleatória" que mapeia r para o mesmo tamanho x
2. h é uma "hash" que deriva o r e y dando origem a r'
3. f é um Núcleo Determinístico

Funções

1. É definido a função **func_g** mapeia r para o mesmo tamanho x

In [16]:
```
def func_g(r, length):
    digest = hashes.Hash(hashes.SHA256())
    digest.update(str(r).encode('utf-8'))
    hash_output = digest.finalize()
    return int.from_bytes(hash_output, 'big') % (2 ^ length)
```

1. É definido a função **func_h** deriva r e y

In [17]:
```
def func_h(r, y, length):
    digest = hashes.Hash(hashes.SHA256())
    digest.update(str(r).encode('utf-8'))
    digest.update(str(y).encode('utf-8'))
    hash_output = digest.finalize()
    return int.from_bytes(hash_output, 'big') % (2 ^ length)
```

1. É definido a função **ElGamal_Enc_FO** que retorna o `criptograma` utilizando o metodo `Fujisaki-Okamoto (FO)`
   A. obter elementos públicos $p, q, g, g^s \leftarrow$ pk
   B. gerar aleatoriamente r tal que $|r| = |\lambda|$
   C. gerar y tal que $y = mensagem \oplus g(r)$
   D. derivar $r' = h(r, y)$
   E. calcular $c = f(r, r')$

F. construir o criptograma $(y, f(r, r')) = (y, c)$

In [18]:
```
def ElGamal_Enc_FO(public_key, message, lambdaa):
    p, q, g, gs = public_key

    message_int = string_to_int(message, p)

    # Gerar um número aleatório r
    r = secrets.randbits(lambdaa)
    r = (r % (q-1)) + 1

    # Calcular g(r) e y = message ⊕ g(r)
    gr = func_g(r, message_int.bit_length())
    y = message_int ^ gr

    # Derivar r' = h(r, y)
    next_r = func_h(r, y, q)

    # Usar o core determinístico do ElGamal para cifrar r usando r'
    #f(r,r')
    gamma = pow(g, next_r, p)
    kappa = pow(gs, next_r, p)
    c = (gamma, (kappa * r) % p)

    # O criptograma final é (y, c)
    return (y, c)
```

1. É definido a função **ElGamal_Dec_FO** que retorna a mensagem original gerada pelo decifrador em `Fujisaki-Okamoto (FO)`
   A. obter elementos públicos $p, q, g, g^s \leftarrow$ pk
   B. obter y, c a partir do criptograma
   C. fazer o inverso da crifragem feita em **ElGamal_Enc_FO**
   D. verificar se a cifra é válida
   E. se sim, retorna-la

In [19]:
```
def ElGamal_Dec_FO(private_key, public_key, criptograma):
    p, q, g, gs = public_key
    y, c = criptograma

    # Decifrar r usando o ElGamal básico
    gamma, enc_r = c
    kappa = pow(gamma, private_key, p)
    kappa_inv = inverse_mod(kappa, p)
    r = (enc_r * kappa_inv) % p

    # Calcular g(r)
    gr = func_g(r, y.bit_length())

    # Recuperar a mensagem original
    message_int = y ^ gr

    # Verificar se a cifra é válida
    r_prime = func_h(r, y, q)
    expected_gamma = pow(g, r_prime, p)

    if gamma != expected_gamma:
```

```
        raise ValueError("Criptograma inválido: verificação de integridade falho

    return int_to_string(message_int)
```

Correr Ex2

In [20]:
```
def ElGamal_Ex2(lambdaa, message):
    print("\n==== INICIANDO CIFRAGEM ELGAMAL ====\n")

    # Verificar se a mensagem pode ser representada em Fp*
    if string_to_int(message, 2**lambdaa) is None:
        print("Erro: A mensagem não pode ser representada no espaço de Fp*. Aume
        return

    print(f"Lambda selecionado: {lambdaa}-bits")

    # Gerar chaves pública e privada
    public_key, private_key = ElGamal_GenKeys(lambdaa)

    print("\n==== CHAVES GERADAS ====")
    print(f"Chave Pública (p, q, g, g^s): \n{public_key}")
    print(f"Chave Privada (s): {private_key}")
    print("===================\n")

    # Cifrar a mensagem
    criptograma = ElGamal_Enc_FO(public_key, message, lambdaa)

    print("\n==== CIFRAGEM ====")
    print(f"Mensagem original: {message}")
    print(f"Criptograma gerado (y, c): {criptograma}")
    print("===================\n")

    # Decifrar a mensagem
    mensagem_decifrada = ElGamal_Dec_FO(private_key, public_key, criptograma)

    print("\n==== DECIFRAGEM ====")
    print(f"Mensagem Decifrada: {mensagem_decifrada}")

    # Verificar a Cifragem
    if mensagem_decifrada == message:
        print("\nFuncionou! A mensagem foi recuperada corretamente.")
    else:
        print("\nErro! A decifração falhou.")
```

In [21]:
```
lambdaa = 150
message = "Hello World FO"
ElGamal_Ex2(lambdaa, message)
```

```
==== INICIANDO CIFRAGEM ELGAMAL ====

Lambda selecionado: 150-bits

==== CHAVES GERADAS ====
Chave Pública (p, q, g, g^s):
(771891897421248003712694909221213449131224403, 3859459487106240018563474546106066
724565612201, 2, 129327235076526384777270866596214932034713221)
Chave Privada (s): 407764068183618582308738645955382974079440 47
===================



==== CIFRAGEM ====
Mensagem original: Hello World FO
Criptograma gerado (y, c): (14683690913466894680576260770545 33, (1016057455213589
412621779384078109633081671 26, 679468179947861130872090465598498206879224222))
===================



==== DECIFRAGEM ====
Mensagem Decifrada: Hello World FO

Funcionou! A mensagem foi recuperada corretamente.
```

Part III

Foi seguida a seguinte informação presente no material da UC

KEM-IND-CCA

1. **Aleatoriedade**
   A. Escolhe-se um Valor Aleatorio r de tamanho λ em bits
2. **Ofuscação da Mensagem** x
   A. gerar y tal que $y = x \oplus g(r)$
3. **Gerar** e **e** k
   A. gerar o par $(e, k) = f(y||r)$
4. **Ofuscação do** k **com** r
   A. gerar c tal que $c = k \oplus r$
5. **Criptograma**
   A. o tuplo (y, e, c) é o novo criptograma

NOTAS

1. g é uma "hash pseudoaleatória" que mapeia r para o mesmo tamanho x
2. f é um Núcleo Determinístico

Notas Importantes

1. **KEM** é uma função tal que:
   A. $KEM = f(r)$ onde r é um ouput aleatorio
2. **KRev** é uma função tal que:

A. $KRev(e) \approx k \Leftrightarrow (\forall r | (e,k) = f(r)$

Funções

1. É definido a função **ElGamal_Enc_KEM_IND_CCA** que retorna o `criptograma` utilizando o metodo de `encapsulamento` e `seguro a CCA`
   A. obter elementos públicos $p, q, g, g^s \leftarrow$ pk
   B. gerar aleatoriamente r tal que $|r| = h$
   usando uma hash h aleatória
   C. gerar y tal que $y = mensagem \oplus g(r)$
   D. gerar o par $(e, k) = f(y||r)$
   E. gerar c tal que $c = k \oplus r$
   F. gerar o criptograma (y, e, c)

In [22]:
```
def ElGamal_Enc_KEM_IND_CCA(public_key, message, lambdaa):
    p, q, g, gs = public_key

    message_int = string_to_int(message, p)

    # Gerar um número aleatório r
    r = secrets.randbits(lambdaa)
    r = (r % (q-1)) + 1
    r = func_g(r, secrets.randbits(lambdaa))

    # Calcular g(r) e y = message ⊕ g(r)
    gr = func_g(r, message_int.bit_length())
    y = message_int ^ gr

    # gerar (e,k)
    yORr = (y << (r.bit_length())) | r
    e = pow(g, yORr, p)
    k = pow(gs, yORr, p)

    # gerar c = k ⊕ r
    c = k ^ r

    #Obter o criptograma final
    return (y,e, c)
```

1. É definido a função **ElGamal_Dec_KEM_IND_CCA** que retorna a mensagem original gerada pelo decifrador do método de encapsulamento e segura CCA
   A. obter elementos públicos $p, q, g, g^s \leftarrow$ pk
   B. KRev(e)
   C. recuperar r
   D. fazer o inverso da crifragem feita em **ElGamal_Enc_FO**
   E. verificar se a cifra é válida
   F. se sim, retorna-la decifrada

In [23]:
```
def ElGamal_Dec_KEM_IND_CCA(public_key, private_key, criptograma):
    p, q, g, gs = public_key
```

```
    y, e, c = criptograma

    #KRev(e)
    # Calcular k = e^s mod p
    k = pow(e, private_key, p)

    # Recuperar r a partir de c
    r = c ^ k

    # Validar a integridade
    yORr = (y << (r.bit_length())) | r
    if (e,k) != (pow(g, yORr, p), pow(gs, yORr, p)):
        raise ValueError("Criptograma inválido: verificação de integridade falho


    # Obter a mensagem decifrada
    gr = func_g(r, y.bit_length())
    message_int = y ^ gr
    message = int_to_string(message_int)


    return message
```

Correr Ex3

In [24]:

```
def ElGamal_Ex3(lambdaa, message):
    print("\n==== INICIANDO CIFRAGEM ELGAMAL ====\n")

    # Verificar se a mensagem pode ser representada em Fp*
    if string_to_int(message, 2**lambdaa) is None:
        print("Erro: A mensagem não pode ser representada no espaço de Fp*. Aume
        return

    print(f"Lambda selecionado: {lambdaa}-bits")

    # Gerar chaves pública e privada
    public_key, private_key = ElGamal_GenKeys(lambdaa)

    print("\n==== CHAVES GERADAS ====")
    print(f"Chave Pública (p, q, g, g^s): \n{public_key}")
    print(f"Chave Privada (s): {private_key}")
    print("==================\n")

    # Cifrar a mensagem
    criptograma = ElGamal_Enc_KEM_IND_CCA(public_key, message, lambdaa)

    print("\n==== CIFRAGEM ====")
    print(f"Mensagem original: {message}")
    print(f"Criptograma gerado (y, e, c): {criptograma}")
    print("==================\n")

    # Decifrar a mensagem
    mensagem_decifrada = ElGamal_Dec_KEM_IND_CCA(public_key, private_key, cripto

    print("\n==== DECIFRAGEM ====")
    print(f"Mensagem Decifrada: {mensagem_decifrada}")

    # Verificar a Cifragem
    if mensagem_decifrada == message:
```

```
        print("\nFuncionou! A mensagem foi recuperada corretamente.")
    else:
        print("\nErro! A decifração falhou.")
```

In [25]:
```
lambdaa = 150
message = "Hello World KEM"
ElGamal_Ex3(lambdaa, message)
```

```
==== INICIANDO CIFRAGEM ELGAMAL ====

Lambda selecionado: 150-bits

==== CHAVES GERADAS ====
Chave Pública (p, q, g, g^s):
(410277827537132413058315308670759877478991819, 205138913768566206529157654335379
938739495909, 2, 341029145909774165989002300610431525377027111)
Chave Privada (s): 196416548491400370778904146357193040758530796
===================


==== CIFRAGEM ====
Mensagem original: Hello World KEM
Criptograma gerado (y, e, c): (37590248738475250382275227572628821‌3, 175087209005
25363601083136176705823209747970‌8, 6226943751206584626010851009238036466701191‌41)
===================


==== DECIFRAGEM ====
Mensagem Decifrada: Hello World KEM

Funcionou! A mensagem foi recuperada corretamente.
```

Part IV

Foi seguida a seguinte informação presente no material da UC

Oblivious Transfer κ-out-of-n

O protocolo de 'Oblivious Transfer' (OT) κ-out-of-n é um mecanismo criptográfico que permite a um Receiver obter exatamente κ mensagens de um conjunto de n mensagens fornecidas por um Provider, sem que o Provider saiba quais foram escolhidas.

1. **Critério**

   A. O Provider gera o critério $\mathcal{C}_{k,n}$
   B. O Provider envia o critério ao Receiver

2. **Gerar as Chaves**

   A. O Receiver escolhe um conjunto $\mathcal{I} \subset \{1,n\}$ de tamanho $\#I = \kappa$, que identifica os índices das mensagens que pretende recolher, Seja e a enumeração de I: a função crescente $e : \{1,\kappa\} \to \{1,n\}$ cuja imagem é I.
      a. Gera aleatoriamente um segredo s e, usando um *XOF* com s como "seed", constroi κ chaves privadas $s_1, \ldots, s_\kappa$

b. $\forall i \in \{1, \kappa\}$, gera chaves públicas $pk(s_i) \to v_i$ e atribui v_i à componenete de ordem $e(i)$ do vector p

c. Gera uma "tag" de autenticação para a seleção I e o segredo s

$$hash(I, s) \to \tau$$

B. Em seguida completa a def. de p atribuindo à componentes $\{p_j\}_{j \notin I}$ valores tais que o vetor de chaves públicas p seja aceite pelo $\mathcal{C}_{k,n}$

C. Finalmente o Receiver enviar ao Provider a "tag" τ e o vetor p

3. O Provider determina $\mathcal{C}_{k,n}(p)$; se p não for aceite pelo critério então aborta o protocolo. Se p for aceite, então usa o seguinte método de cifragem $E'_p(x, \tau)$:

   A. $r \leftarrow \{0,1\}^\lambda$
   B. $y \leftarrow x \oplus g(r)$
   C. $r' \leftarrow h(r, y, \tau)$
   D. $c \leftarrow f_p(r, r')$
   E. (y, c)

cuja característica específica é o facto de se incluir o "tag" τ no "hash" $h(r, y, \tau)$ usado para construir a pseudo-aleatoriedade r'.

Usando esta cifra o Provider constrói n cripotrogramas $(y_i, c_i) \leftarrow E'_{p_i}(m_i)$ com $i \in \{1, n\}$ que envia para o Receiver

4. O Receiver decifra da seguinte forma $D'_s(y, c, \tau)$:
   A. $r \leftarrow D_s(c)$
   B. $r' \leftarrow h(r, y, \tau)$
   C. if $c! = f_p(r, r')$ then $\perp$
   D. else $y \oplus g(r)$

uma vez mais a única caracteristica particular deste algoritmo é o uso "tag" de atutenticação τ na construção da pseudo-aleatoriedade $r' \leftarrow h(r, y, \tau)$

5. O agente Receiver:
   A. conhece, porque criou, a "tag" τ que autentica o conjunto de mensagens escolhidas I e o respetivo conjunto de chaves públicas ("boas chaves")
   B. conhece, porque gerou e armazenou num passo anterior, as chaves privadas s_i para todos $i \in I$
   C. conhece, porque recebeu do Receiver, todos os criptogramas $\{(y_i, c_i)\}_{i \in \{1,n\}}$

Então, $\forall i \in I$, pode recuperar a mensagem $m_i \leftarrow D_{s_i}(y_i, c_i, \tau)$

NOTAS

1. O Receiver não pode decifrar criptogramas que não "pediu"
2. O Provider não sabe quais mensagens enviou visto que enviou todas
3. É possível fazer-se uma prova de honestidade contra `verificadores desonestos`

Funções

1. É definido a função **func_h** deriva r e y e τ

In [26]:
```
def func_h_ot(r, y, tau, length):
    digest = hashes.Hash(hashes.SHA256())
    digest.update(str(r).encode('utf-8'))
    digest.update(str(y).encode('utf-8'))
    digest.update(str(tau).encode('utf-8'))
    hash_output = digest.finalize()
    return int.from_bytes(hash_output, 'big') % (2 ^ length)
```

1. É definido a função **func_h** deriva I e s dando origem à "tag" τ

In [27]:
```
def func_tau(I,s):
    digest = hashes.Hash(hashes.SHA256())
    digest.update(str(I).encode('utf-8'))
    digest.update(s)
    return int.from_bytes(digest.finalize(), 'big')
```

1. É definido a função **gerar_chaves_OT** que retorna o `o vetor com chaves boas`, `o vetor com todas as chaves`, `tag tau` e o `grupo matematico`
   A. obter os parametros globais do grupo para gerar chaves
   B. gerar κ chaves usando a função de ElGamal
   C. gerar "tag" τ
   D. gerar o vetor p com todas as keys (boas e más)

In [28]:
```
def gerar_chaves_OT(n, kappa, I, lambdaa):
    p, q, g = find_p_q_g(lambdaa)  # Parâmetros globais do grupo para gerar chav
    keys = []

    # Gerar kappa chaves
    for _ in range(kappa):
        _, private_key = ElGamal_GenKeys(lambdaa)  # Gera kappa chaves
        public_key = (p, q, g, pow(g, private_key, p))
        keys.append((public_key, private_key))

    # Gera "tag" tau
    s = secrets.token_bytes(16)
    tau = func_tau(I, s)

    # Gerar vetor  p
    p_vector = [0] * n
    for idx, i in enumerate(I):
        p_vector[i-1] = keys[idx][0][3]  # Usa g^s das chaves "boas"

    # Preencher o vetor com chaves "más"
    for j in range(n):
        if p_vector[j] == 0:
            p_vector[j] = secrets.randbelow(q)  # Placeholder para chaves "más"

    return keys, p_vector, tau, (p, q, g)
```

1. É definido a função **ElGamal_Enc_FO_OT** que retorna o `criptograma` utilizando o metodo `Fujisaki-Okamoto (FO)`
   A. obter elementos públicos $p, q, g, g^s \leftarrow$ pk
   B. gerar aleatoriamente r tal que $|r| = |\lambda|$
   C. gerar y tal que $y = mensagem \oplus g(r)$
   D. derivar $r' = h(r, y)$ incluindo a "tag" τ `OT`
   E. calcular $c = f(r, r')$
   F. construir o criptograma $(y, f(r, r')) = (y, c)$

In [29]:
```
def ElGamal_Enc_FO_OT(public_key, message, lambdaa, tau):
    p, q, g, gs = public_key
    message_int = string_to_int(message, p)
    if message_int is None:
        return None

    r = secrets.randbits(lambdaa) % (q-1) + 1
    gr = func_g(r, message_int.bit_length())
    y = message_int ^ gr

    next_r = func_h_ot(r, y, tau, q)  # Inclui tau
    gamma = pow(g, next_r, p)
    kappa = pow(gs, next_r, p)
    c = (gamma, (kappa * r) % p)
    return (y, c)
```

1. É definido a função **ElGamal_Dec_FO_OT** que retorna a mensagem original gerada pelo decifrador em `Fujisaki-Okamoto (FO)`
   A. obter elementos públicos $p, q, g, g^s \leftarrow$ pk
   B. obter y, c a partir do criptograma
   C. fazer o inverso da crifragem feita em **ElGamal_Enc_FO** usando também a "tag" τ `OT`
   D. verificar se a cifra é válida
   E. se sim, retorna-la

In [30]:
```
def ElGamal_Dec_FO_OT(private_key, public_key, criptograma, tau):
    p, q, g, gs = public_key
    y, c = criptograma
    gamma, enc_r = c

    kappa = pow(gamma, private_key, p)
    kappa_inv = inverse_mod(kappa, p)
    r = (enc_r * kappa_inv) % p

    gr = func_g(r, y.bit_length())
    message_int = y ^ gr

    r_prime = func_h_ot(r, y, tau, q)  # Inclui tau
    expected_gamma = pow(g, r_prime, p)
    if gamma != expected_gamma:
        return None  # Criptograma inválido

    return int_to_string(message_int)
```

Correrr Ex4

In [31]:
```
def oblivious_transfer_elgamal(n, kappa, I ,messages, lambdaa):
    print(f"\n==== INICIANDO OT {kappa}-out-of-{n} ====\n")

    # Receiver escolhe I (exemplo fixo)
    keys, p_vector, tau, grupo_math = gerar_chaves_OT(n, kappa, I, lambdaa)

    #Print
    print(f"Parâmetros globais (p, q, g): {grupo_math}")
    print(f"Vetor p: {p_vector}")
    print(f"Tag tau: {tau}")

    # Provider cifra mensagens
    # grupo_math = (p, q, g)
    criptogramas = []
    for i, m in enumerate(messages):
        public_key_i = (grupo_math[0], grupo_math[1], grupo_math[2], p_vector[i]
        c_i = ElGamal_Enc_FO_OT(public_key_i, m, lambdaa, tau)
        criptogramas.append(c_i)

    # Mostrar Criptogramas
    print("\n==== CRIPTOGRAMAS ====")
    for idx, i in enumerate(I):
        print(f"Criptograma {i}: {criptogramas[idx]}")

    # Receiver decifra
    recovered_messages = []
    for idx, i in enumerate(I):
        private_key_i = keys[idx][1]
        public_key_i = (grupo_math[0], grupo_math[1], grupo_math[2], p_vector[i-
        m_i = ElGamal_Dec_FO_OT(private_key_i, public_key_i, criptogramas[i-1],
        recovered_messages.append(m_i)

    # Mostrar mensagens recuperadas e Verificar
    print("\n==== MENSAGENS RECUPERADAS ====")
    for idx, i in enumerate(I):
        if recovered_messages[idx] == messages[i-1]:
            print(f"Mensagem {i} OK: {recovered_messages[idx]}")
        else:
            print(f"Erro na mensagem {i}.")
```

In [32]:
```
# Teste
lambdaa = 100
I = [2,7,3,1] #mensagens a serem decifradas
messages = ["Msg1", "Msg2", "Msg3", "Msg4", "Msg5", "Msg6", "Msg7"] #todas as me

kappa = len(I)
n = len(messages)
oblivious_transfer_elgamal(n, kappa, I,messages, lambdaa)
```

```
==== INICIANDO OT 4-out-of-7 ====

Parâmetros globais (p, q, g): (265127790856487397546272288279, 132563895428243698773136144139, 17)
Vetor p: [157961865890575542613420104822, 102469778732385666792140682439, 221184220010339473365945015234, 51022154631339141650527313846, 954886956146589164615779 46431, 51581304155811971976917225272, 71562749482044250160469728063]
Tag tau: 5442686499080169604133849875496639885450212960883279295975654028715857006 2011

==== CRIPTOGRAMAS ====
Criptograma 2: (1299408676, (166753763441990808491028453259, 150324547387892866101115778158))
Criptograma 7: (1299408686, (70711369854782437862600022196, 207699576152305595604818057460))
Criptograma 3: (1299408702, (167355220012916278093970680826, 171952339847332613194329364340))
Criptograma 1: (1299408703, (234114017913851973474162652283, 263846739002411341775168669856))

==== MENSAGENS RECUPERADAS ====
Mensagem 2 OK: Msg2
Mensagem 7 OK: Msg7
Mensagem 3 OK: Msg3
Mensagem 1 OK: Msg1
```