

```
In [ ]:
import random
from sage.all import *
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
import hashlib
import numpy as np
import os
```

Geração dos parâmetros **p,g** e **g** de acordo com as regras do enunciado, ou seja, **q** é um número primo e **p >= lambda * |lambda|** e tudo isto enquanto trabalhamos num grupo multiplicativo **Fp**.

Fora isso, também se criou o método de decrypt e encrypt em CPA usando o pdf como base. Quanto ao núcleo determinístico deve-se ao facto de se usar o **plaintext** para se gerar o **c**.

```
In [ ]:
class PKE_ElGamal_CPA:
    def __init__(self, lambd):
        self.lambd = lambd

    def keygen(self):
        q = random_prime(2**self.lambd, True, 2**(self.lambd-1)) # q será sempre um número primo igual ou maior
        #print("q: ", q)
        i = 0
        while True:
            p = q * 2**i + 1
            if (p.bit_length() >= self.lambd * abs(self.lambd)) and (is_prime(p)):
                break
            i += 1
        #print ("p: ", p)
        while True:
            g_temp = Zmod(p).random_element()
            g = g_temp**((p-1)//q)
            if g**q == 1 and g != 1:
                break
        #print ("g: ", g)

        privatekey = randrange (1, q) # no enunciado diz que devia ser 0, mas não pode ser 0 a chave privada
        publickey = power_mod(g, privatekey, p)

        #print("privatekey: ", privatekey)
        #print("publickey: ", publickey)

        return (p, q, g, privatekey, publickey)

    def encrypt(self, p, q, g, publickey, plaintext):
        w = randrange(1, q) # w não pode ser 0, porque se for 0, o c[1] vai ser 0 e o c[0] vai ser 1
        y = power_mod(g, w, p)
        k = power_mod(publickey, w, p)
        c = (y, plaintext * k % p)
        return c

    def decrypt(self, c, privatekey, p):
        k = power_mod(c[0], privatekey, p)
        plaintext = c[1] * inverse_mod(int(k), p) # inverse_mod não esta a funcionar
        return plaintext
```

Vamos agr começar a fazer o PKE ElGamal CCA

Aqui há o uso do que foi definido acima, tal como nos é apresentado no pdf foram criadas duas hashs, a **hash g** que é responsável por mascarar o **r** (elemento random de tamanho lambda) e a **hash h** que cria mais uma camada para mascarar o **r**. Depois de se fazer o processo todo do pdf, chega a uma parte onde se tem de aplicar um função que no pdf é denominada por **f**. Assumimos que seria o **PKE_ElGamal_CPA** implementado acima, já que o nosso enunciado inferia que teríamos que usar o que fora implementado acima.

```
In [ ]:
class PKE_ElGamal_CCA:
    def __init__(self, lambd, q_bytes_len=None):
        self.lambd = lambd
        self.r_bytes_len = (self.lambd + 7) // 8
        self.h_bytes_len = hashlib.sha256().digest_size
        self.h_plus_r_size = self.r_bytes_len + self.h_bytes_len
        self.q_bytes_len = q_bytes_len

    def g_hash(self, r_bytes, output_len):
        digest = hashlib.sha256(r_bytes).digest()
        return digest[:output_len]

    def h_hash(self, r_bytes, y, tau=None):
        if tau is None:
```

```

        h = hashlib.sha256(r_bytes + y).digest()
    else:
        #print ("[H_HASH] tau: ", tau)
        tau_bytes = int(tau).to_bytes(self.q_bytes_len, "big")
        h = hashlib.sha256(r_bytes + y + tau_bytes).digest()
    return h

def X0R(self, a, b):
    return bytes(x ^ y for x, y in zip(a, b))

def encrypt_CCA(self, p, q, g, publickey, plaintext, tau=None):
    r = ZZ_random_element(2**self.lambd - 1), 2**self.lambd - 1)
    r_bytes = int(r).to_bytes(self.r_bytes_len, "big")

    #print ("[ENCRYPT] r_bytes: ", r_bytes)

    gr = self.g_hash(r_bytes, len(plaintext))
    y = self.X0R(plaintext, gr)
    h = self.h_hash(r_bytes, y, tau)

    #print ("[ENCRYPT] h: ", h)

    r_plus_h = r_bytes + h
    r_plus_h_int = int.from_bytes(r_plus_h, "big")
    print ("[ENCRYPT] r_plus_h_int: ", r_plus_h_int)

    c = PKE_ElGamal_CPA(self.lambd).encrypt(p, q, g, publickey, r_plus_h_int)
    #print ("[ENCRYPT] c: ", c)
    return (y, c)

def decrypt_CCA(self, ciphertext, privatekey, p, tau=None):
    (y, c) = ciphertext
    #print ("[DECRYPT] c: ", c)
    r_plus_h_int = PKE_ElGamal_CPA(self.lambd).decrypt(c, privatekey, p)
    print ("[DECRYPT] r_plus_h_int: ", r_plus_h_int)

    r_plus_h_bytes_needed = self.r_bytes_len + self.h_bytes_len
    r_plus_h_bytes = int(r_plus_h_int).to_bytes(r_plus_h_bytes_needed, "big")

    r_bytes = r_plus_h_bytes[-r_plus_h_bytes_needed:][:self.r_bytes_len]
    h_original = r_plus_h_bytes[-r_plus_h_bytes_needed:][self.r_bytes_len:]

    #print("[DECRYPT] r_bytes: ", r_bytes, "len:", len(r_bytes))
    #print("[DECRYPT] h_original: ", h_original, "len:", len(h_original))

    if tau is not None:
        tau = int(tau)
        #print("[DECRYPT] tau:", tau)

        h_by_us = self.h_hash(r_bytes, y, tau)
        #print("[DECRYPT] h_by_us: ", h_by_us, "len:", len(h_by_us))

        if h_original != h_by_us:
            #print("[ERROR] Falha na verificação de integridade")
            #print(f"h_original (hex): {h_original.hex()}")
            #print(f"h_by_us (hex): {h_by_us.hex()}")
            raise ValueError("Integridade comprometida")

        gr = self.g_hash(r_bytes, len(y))
        plaintext = self.X0R(y, gr)
    return plaintext

```

```

-----
Vamos agr começar a fazer o KEM ElGamal CCA
-----

```

Aqui foi algo simples e, simplesmente, se usou a informação do pdf diretamente.

```

In [ ]: x
class KEM_ElGamal_CCA:
    def __init__(self, lambd):
        self.lambd = lambd
        self.pke_cc = PKE_ElGamal_CCA(lambd)

    def KDF_hash(self, shared_secret, output_len):
        kdf = HKDF(
            algorithm=hashes.SHA256(),
            length=output_len,
            salt=None,
            info=None,
        )
        key = kdf.derive(shared_secret)
        return key

```

```

def encapsulate (self, p, q, g, publickey):
    x = ZZ.random_element(2**self.lambd - 1), 2**self.lambd - 1)
    x_bytes = int(x).to_bytes((self.lambd + 7) // 8, "big")
    x_cifrado = self.pke_ccc.encrypt_CCA(p, q, g, publickey, x_bytes)
    k = self.KDF_hash(x_bytes, 32)
    return (x_cifrado, k)

def decapsulate (self, x_cifrado, privatekey, p):
    x_bytes = self.pke_ccc.decrypt_CCA(x_cifrado, privatekey, p)
    k = self.KDF_hash(x_bytes, 32)
    return k

```

'''

Vamos agr começar a fazer o "Oblivious Transfer" k-out-of-n

'''

Esta função foi criada para ser usada na resolução do OT para tentar resolver o problema na hora da cifrar e decifrar as **k** mensagens.

Qual o intuito desta função?

-> Como se sabe o OT no nosso trabalho vai trabalhar sobre Z_q com q um número primo, ou seja, usamos q sendo o valor que se calcula no Keygen(). Ou seja, vamos trabalhar num espaço finito de q , consequentemente todos os resultados serão **mod q** contudo, na parte do PKE_CCA e PKE_CPA na parte das chaves públicas usa-se uma abordagem em **mod p** e para as chaves privadas uma abordagem em **mod q**. Então, concluimos que seria necessário fazer esta mudança de espaço finito para conseguirmos obter através a partir do output dado em **decrypt_CCA** a resposta depois de fazermos **encrypt_CCA**, contudo falhamos neste ponto, pois não conseguimos implementar de forma correta o **k-out-of-n** completamente, visto que a mensagem dá sempre como comprometida.

Apesar de não termos conseguido resolver o problema, podemos presumir algumas das razões pelas quais isto possa estar a acontecer. A *primeira hipótese* surgiu depois de analisar os outputs verificamos que a mensagem antes de cifrar é sempre muito maior do que a mensagem depois de decifrar, o que não deveria ser possível, visto que deveria dar o mesmo resultado, o que nos leva a inferir se **p** é que não está a suportar a quantidade de bits necessárias para armazenar a mensagem? Contudo, descartamos estar ideia porque **p** é um número enorme e mesmo aumentando lambda para termos um **p** maior, continua sem funcionar. A *segunda hipótese* que colocamos seria o facto de o código não estar completamente modular, contudo, também, não encontramos evidências de ser o caso. A *terceira hipótese* que supusemos foi que a hash **h** não estaria a lidar bem com o tau, visto que foi uma das poucas mudanças entre a alínea (b) e a alínea (d). Apesar de todas estas hipóteses parecem credíveis aos nossos olhos e mesmo depois de tentar resolver cada problema individualmente e em conjunto continuamos sem conseguir obter um resultado satisfatório.

```
In [1]: def embed_key(x, g, p):
    if hasattr(x, 'lift'):
        key_int = int(x.lift())
    elif isinstance(x, int):
        key_int = x
    else:
        key_int = int(x)

    return power_mod(g, key_int, p)
```

'''

P R O V I D E C O D E

'''

Depois de apresentar o problema no markdown acima, não há muito a falar, visto que a única coisa que foi feita foi aplicar o conteúdo do powerpoint, tanto para o provider como para o receiver.

```
In [ ]: class ProviderOT:
    def __init__(self, lambd):
        self.lambd = lambd
        self.n = 0
        self.k = 0
        self.OC = None

        self.pke_cpa = PKE_ElGamal_CPA(lambd)
        self.p, self.q, self.g, self.privatekey, self.publickey = self.pke_cpa.keygen()

        self.pke_ccc = PKE_ElGamal_CCA(lambd, (self.q.nbits() + 7) // 8)
```

```

    self.messages = []
    self.vetor_p = None
    self.tau = None
    #self.rho = None

    self.encrypted_messages = None

def set_messages(self, msgs):
    """
    Set the messages for the OT protocol
    """
    self.messages = msgs
    self.n = len(msgs)

def setK(self, k):
    """
    Set the number of messages k for the OT protocol
    """
    self.k = k

def receive(self, p, tau):
    self.vetor_p = p
    self.tau = tau
    #print ("Provider recebeu vetor p: ", self.vetor_p)
    #print ("Provider recebeu tau: ", self.tau)

def getParams(self):
    """
    Get the parameters p, q, g for the OT protocol
    """
    return self.p, self.q, self.g

"""

----- PROVIDER CODE - MAIN FUNCTIONS -----


"""

def generate_OC(self):
    """
    Generate the Oblivious Criterion using a seed and XOF
    """
    F = Zmod(self.q)

    seed_rho = os.urandom(32)

    xof_A = hashlib.shake_256(seed_rho + b"A")
    xof_u = hashlib.shake_256(seed_rho + b"u")

    attempts_A = 0
    while attempts_A < 1000:
        attempts_A += 1

        bytes_needed = self.n * (self.n - self.k) * ((self.q.nbits() + 7) // 8)
        random_bytes = xof_A.digest(bytes_needed)

        A_elements = []
        byte_index = 0
        for i in range(self.n):
            row = []
            for j in range(self.n - self.k):
                bytes_per_element = (self.q.nbits() + 7) // 8
                element_bytes = random_bytes[byte_index:byte_index + bytes_per_element]
                byte_index += bytes_per_element

                element = int.from_bytes(element_bytes, byteorder='little') % self.q
                row.append(F(element))
            A_elements.append(row)

        A = matrix(F, A_elements)

        if A.rank() == min(self.n, self.n - self.k):
            break

    attempts_u = 0
    while attempts_u < 1000:
        attempts_u += 1

        bytes_needed = (self.n - self.k) * ((self.q.nbits() + 7) // 8)
        random_bytes = xof_u.digest(bytes_needed)

        u_elements = []
        byte_index = 0
        for i in range(self.n - self.k):
            bytes_per_element = (self.q.nbits() + 7) // 8
            element_bytes = random_bytes[byte_index:byte_index + bytes_per_element]
            byte_index += bytes_per_element

```

```

        element = int.from_bytes(element_bytes, byteorder='little') % self.q
        u_elements.append(F(element))

    u = vector(F, u_elements)

    if not u.is_zero():
        break

#self.seed_rho = seed_rho
self.OC = (A, u)
return self.OC

def verify_OC(self):
    F = Zmod(self.q)
    print ("F: ", F)

    print ("Vetor p: ", self.vetor_p)

    p_vec = vector(F, self.vetor_p)

    A_mat = Matrix(F, self.OC[0])
    u_vec = vector(F, self.OC[1])

    product = p_vec * A_mat

#print("Vetor p (reduzido):", p_vec)
#print("Matriz A:", A_mat)
#print("Vetor u:", u_vec)
#print("Produto p * A:", product)

    return product == u_vec

def execute_OT(self):
    """
    Execute the OT protocol by encrypting all messages with respective public keys
    """
    if not self.verify_OC():
        print("OC verification failed")
        return None

    print("OC verified successfully")

    self.encrypted_messages = []

    for i in range(self.n):
        message = self.messages[i]
        if self.vetor_p[i] is None:
            print(f"Error: public key at index {i} is None")
            return None

        public_key = embed_key(self.vetor_p[i], self.g, self.p)

        if isinstance(message, str):
            message_bytes = message.encode()
        elif isinstance(message, int):
            message_bytes = str(message).encode()
        else:
            message_bytes = message
#print ("p: ", self.p)
#print ("tau: ", self.tau)
        encrypted_message = self.pke_cca.encrypt_CCA(
            self.p, self.q, self.g,
            public_key, message_bytes,
            self.tau,
        )

#print ("Mensagem encriptada i: ", encrypted_message)

        self.encrypted_messages.append(encrypted_message)

    return self.encrypted_messages

```

"""

R E C E I V E R C O D E

"""

Na função **complete_p** tivemos muitos problemas porque estamos a tentar usar uma biblioteca de sage que estava a calcular mal o produto escalar, ou a fazer arredondamentos. Não percebemos quais dos dois casos foi. A função em

questão foi o **solve_right** e, por isso, tivemos muito tempo encravados nesta parte, visto que na maior parte das vezes dava validação errada quando fazíamos $pA = u^*$, mas algumas vezes dava bem, ou seja, deveria ser arredondamentos.

Depois de usarmos a função atual de sage **inverse()** funcionou imediatamente, contudo às vezes ainda falha (algo muito excepcional) devido ao facto da matriz não ser passível de se inverter, deveríamos ter tido cuidado nestes casos e criar uma contra medida, por exemplo, avisar o provider que a solução que ele deu não tem resposta e esperar que ele gera-se um novo A e u. Poderíamos ter iterado a função **generate_p** de novo para conseguirmos novas chaves publicas e assim ao calcular o **p_known** conseguíramos, talvez, algo que não fosse impossível de resolver.

In [2]:

```
class ReceiverOT:
    def __init__(self, lambd, k):
        self.lambd = lambd
        self.n = 0
        self.k = k
        self.I = []
        self.OC = None
        self.s = None
        self.p = None
        self.tau = None

        self.p_prov = None
        self.q_prov = None
        self.g_prov = None

        self.chaves_privadas = None

        self.received_encrypted_messages = None

    def setN(self, n):
        self.n = n

    def setOC(self, OC):
        self.OC = OC
        #print ("A: ", OC[0])
        #print ("u: ", OC[1])

    def setProviderParams(self, p, q, g):
        self.p_prov = p
        self.q_prov = q
        self.g_prov = g

    def receive_encrypted_messages(self, encrypted_messages):
        """
        Receive the encrypted messages from the provider
        """
        #print ("Mensagens encriptadas recebidas: ", encrypted_messages)
        self.received_encrypted_messages = encrypted_messages

    """
    -----
    R E C E I V E R   C O D E - M A I N   F U N C T I O N S
    -----
    """

    def xof(self, seed, k):
        """
        XOF function to generate k keys from a seed
        """
        keys = []
        base_hash = hashlib.sha256(str(seed).encode()).digest()

        for i in range(k):
            key_hash = hashlib.sha256(base_hash + str(i).encode()).hexdigest()
            key_to_add = int(key_hash, 16) % self.q_prov
            #print (f"Chave {i+1}: {key_to_add}")
            keys.append(key_to_add)

        return keys

    def pk(self, si):
        F = Zmod(self.q_prov)
        # A operação power_mod já pode ser feita ou, se preferir, converta raw para um elemento de F:
        raw = power_mod(self.g_prov, si, self.p_prov)
        return F(raw)

    def generate_p(self):
        """
        Generate the vector p according to the protocol
        """
        if self.n == 0 or self.OC is None:
            raise ValueError("n and OC must be set before generating p")

        #print ("n: ", self.n)
```

```

    #print ("k: ", self.k)
    self.I = sorted(np.random.choice(range(0, self.n), size=self.k, replace=False).tolist()) #aumentar o
    print(f"Conjunto I escolhido: {self.I}")

    self.s = ZZ.random_element(1, self.q_prov-1)
    #print(f"Segredo s gerado: {self.s}")

    self.chaves_privadas = self.xof(self.s, self.k)
    #print(f"Chaves privadas geradas: {self.chaves_privadas}")

    self.p = [None] * self.n

    #chaves_publicas = []
    for idx, i in enumerate(self.I):
        v_i = self.pk(self.chaves_privadas[idx])
        self.p[i] = v_i

    self.complete_p()

    self.tau = self.generate_tau()

    return self.p, self.tau

def complete_p(self):
    A, u = self.OC

    F = Zmod(self.q_prov)

    A_matrix = matrix(F, A)
    u_vector = vector(F, u)

    known_indices = self.I
    unknown_indices = [i for i in range(self.n) if i not in self.I]

    for i in known_indices:
        self.p[i] = F(self.p[i])

    A_known = A_matrix.matrix_from_rows(known_indices)
    A_unknown = A_matrix.matrix_from_rows(unknown_indices)

    p_known = vector(F, [self.p[i] for i in known_indices])
    partial_result = p_known * A_known

    right_side = u_vector - partial_result

    if not A_unknown.is_invertible():
        print("A_unknown não é invertível")
        return None
    X_vec = right_side * A_unknown.inverse()

    for idx, j in enumerate(unknown_indices):
        self.p[j] = X_vec[idx]

    vector_p = vector(F, self.p)
    product = vector_p * A_matrix
    print("Vetor p completo (reduzido):", vector_p)
    print("u_vector:", u_vector)
    print("Produto p * A:", product)
    print(f"Verificação p * A == u: {product == u_vector}")

    return self.p

def generate_tau(self):
    """
    Generate the authentication tag τ = hash(I,s)
    """
    if self.I is None or self.s is None:
        raise ValueError("I and s must be set before generating tau")

    # Combine I and s into a string and hash it
    combined = str(self.I) + str(self.s)
    self.tau = int(hashlib.sha256(combined.encode()).hexdigest(), 16) % self.q_prov
    print(f"Tag de autenticação τ gerado: {self.tau}")

    return self.tau

def decrypt_messages(self):
    """
    Decrypt the received messages corresponding to the selected indices
    """
    if self.received_encrypted_messages is None:
        raise ValueError("No encrypted messages received")

    decrypted_messages = {}

    for idx, i in enumerate(self.I):
        try:

```

```

        if i >= len(self.received_encrypted_messages):
            print(f"Index {i} is out of bounds for received messages")
            decrypted_messages[i] = None
            continue

        encrypted_message = self.received_encrypted_messages[i]
        if encrypted_message is None:
            print(f"No message received for index {i}")
            decrypted_messages[i] = None
            continue

        private_key = self.chaves_privadas[idx]
        q_len = self.q_prov.nbits() + 7 // 8
        pke_cca = PKE_ElGamal_CCA(self.lambd, q_len)

        #print ("p_prov: ", self.p_prov)
        #print ("tau: ", self.tau)
        decrypted = pke_cca.decrypt_CCA(
            encrypted_message,
            private_key,
            self.p_prov,
            self.tau
        )

        print ("-----")
        print ("Mensagem encriptada i: ", encrypted_message)
        print ("-----")

        if isinstance(decrypted, bytes):
            try:
                decrypted = decrypted.decode('utf-8')
            except UnicodeDecodeError:
                pass

        decrypted_messages[i] = decrypted
        print(f"Successfully decrypted message at index {i}")

    except Exception as e:
        print(f"Failed to decrypt message {i}: {str(e)}")
        decrypted_messages[i] = None

    return decrypted_messages

```

```

"""
-----
MAIN
-----
"""

```

A main está estranho por 2 motivos, primeiro, o código foi feito em ficheiro .py e não em notebook, pois não conseguimos colocar o notebook a funcionar com sage. O outro motivo que levará a inferir que está estranho é o facto de a main ser na sua maioria gerada pelo chatgpt, visto que temos essa facilidade e a main nada mais é de uma forma de testar, com a ajuda do chatgpt conseguimos ter uma main objetiva, organizada e que nos dá uma perspetiva do que está mal e do que está bem.

```

In [ ]: # o user vai dar o input
user_input = ""
while True:
    user_input = input(
        '\n'
        'Escolha uma das seguintes opções:\n'
        '1 - PKE ElGamal CPA\n'
        '2 - PKE ElGamal CCA\n'
        '3 - KEM CCA\n'
        '4 - Oblivious Transfer k-out-of-n\n'
        '5 - Sair\n'
        'Opção:\n'
    )

    if (user_input == "1"):
        print("\n== PKE ElGamal CPA ==\n")

        scheme_cpa = PKE_ElGamal_CPA(32)
        p, q, g, privatekey, publickey = scheme_cpa.keygen()

        message = ("Just Testing CPA Integrity.")
        plaintext_bytes = message.encode("utf-8")

```

```

plaintext_int = int.from_bytes(plaintext_bytes, "big")

# Teste Funcional: Encriptação e Decifragem válidas
print("Caso Funcional")
try:
    ciphertext = scheme_cpa.encrypt(p, q, g, publickey, plaintext_int)
    decrypted = scheme_cpa.decrypt(ciphertext, privatekey, p)
    decrypted_int = int(decrypted)
    decrypted_bytes = decrypted_int.to_bytes((decrypted_int.bit_length() + 7) // 8, "big")
    recovered_text = decrypted_bytes.decode("utf-8")
    print(" Desencriptação bem-sucedida:")
    print(recovered_text)
except Exception as e:
    print(" Erro no fluxo funcional:", e)

# Ataque 1: Alteração de y
print("\nAlteração de y (integridade comprometida)")
try:
    y, c_val = ciphertext
    y_attack = bytearray(y)
    y_attack[0] ^= 0xFF
    ciphertext_attack = (bytes(y_attack), c_val)
    _ = scheme_cpa.decrypt(ciphertext_attack, privatekey, p)
    print(" Erro: Decifragem com y adulterado ocorreu sem erro!")
except Exception as e:
    print(" Ataque em y bloqueado com sucesso:")
    print("Detalhes:", e)

# Ataque 2: Alteração de c[1]
print("\nAlteração de c[1] (integridade comprometida)")
try:
    y, c_val = ciphertext
    if isinstance(c_val, tuple):
        c_list = list(c_val)
        c_list[1] = c_list[1] + 1
        c_attack = tuple(c_list)
    else:
        c_attack = c_val + 1
    ciphertext_attack = (y, c_attack)
    _ = scheme_cpa.decrypt(ciphertext_attack, privatekey, p)
    print(" Erro: Decifragem com c adulterado ocorreu sem erro!")
except Exception as e:
    print(" Ataque em c[1] bloqueado com sucesso:")
    print("Detalhes:", e)

# Ataque 3: Uso de chaves inválidas
print("\nGeração de chaves inválidas")
try:
    invalid_private = 0
    invalid_public = power_mod(g, invalid_private, p)
    ciphertext = scheme_cpa.encrypt(p, q, g, invalid_public, plaintext_int)
    _ = scheme_cpa.decrypt(ciphertext, invalid_private, p)
    print(" Erro: Decifragem com chave privada inválida ocorreu sem erro!")
except Exception as e:
    print(" Chave inválida bloqueada com sucesso:")
    print("Detalhes:", e)

...
-----
Vamos agr começar a fazer a alinea (b)
-----
...

if (user_input == "2"):
    print("\n==== Teste do PKE ElGamal IND-CCA (com F0) ====\n")
    lambd = 32

    pke_cpa = PKE_ElGamal_CPA(lambd)
    p, q, g, privatekey, publickey = pke_cpa.keygen()

    plaintext = "Just Testing F0 Integrity.".encode("utf-8")
    print("Plaintext original:")
    print(plaintext.decode("utf-8"))

    pke_cca = PKE_ElGamal_CCA(lambd)

    print("\nFluxo Funcional")
    try:
        ciphertext = pke_cca.encrypt_CCA(p, q, g, publickey, plaintext)
        decrypted = pke_cca.decrypt_CCA(ciphertext, privatekey, p)
        print(" Decifragem bem-sucedida:")
        print(decrypted.decode("utf-8"))
    except Exception as e:
        print(" Erro no fluxo funcional:", e)

    # Ataque: Alteração do componente y
    print("\nAlteração de y (parte do ciphertext)")
    try:
        (y, c) = ciphertext

```

```

y_attack = bytearray(y)
y_attack[0] ^= 0xFF
ciphertext_attack = (bytes(y_attack), c)
_ = pke_cca.decrypt_CCA(ciphertext_attack, privatekey, p)
print(" Erro: Decifragem com y adulterado ocorreu sem erro!")
except Exception as e:
    print(" Ataque em y detectado (integridade acionada):")
    print("Detalhes do erro:", e)

# Ataque: Alteração do componente c (ciphertext do encapsulamento)
print("\nAlteração do componente c (ciphertext do encapsulamento)")
try:
    (y, c) = ciphertext
    if isinstance(c, tuple):
        c_list = list(c)
        c_list[0] = c_list[0] + 1
        c_attack = tuple(c_list)
    else:
        c_attack = c + 1
    ciphertext_attack = (y, c_attack)
    _ = pke_cca.decrypt_CCA(ciphertext_attack, privatekey, p)
    print(" Erro: Decifragem com c adulterado ocorreu sem erro!")
except Exception as e:
    print(" Ataque em c detectado (integridade acionada):")
    print("Detalhes do erro:", e)

...
-----
Vamoos agr começar a fazer a alinea (c)
-----
...

if (user_input == "3"):
    print("\n== Teste do KEM ElGamal IND-CCA ==\n")

    lambd = 32
    pke_cpa = PKE_ElGamal_CPA(lambd)
    p, q, g, privatekey, publickey = pke_cpa.keygen()

    kem = KEM_ElGamal_CCA(lambd)

    # --- Encapsulação e Decapsulação válidas ---
    print("Caso Funcional: Encapsulação e Decapsulação válidas")
    try:
        (ciphertext, k_enc) = kem.encapsulate(p, q, g, publickey)
        k_dec = kem.decapsulate(ciphertext, privatekey, p)
        if k_enc == k_dec:
            print(" Sucesso: A chave encapsulada foi corretamente decapsulada.")
        else:
            print(" Falha: As chaves derivadas não correspondem!")
    except Exception as e:
        print(" Exceção no fluxo funcional:", e)

    # --- Adulteração de uma parte do ciphertext ---
    print("\nAtaque: Adulteração do ciphertext (modificando y)")
    try:
        (y, c) = ciphertext
        y_attack = bytearray(y)
        y_attack[0] ^= 0xFF
        ciphertext_attack = (bytes(y_attack), c)
        _ = kem.decapsulate(ciphertext_attack, privatekey, p)
        print(" Erro: Decifragem com ciphertext adulterado (y modificado) ocorreu sem erro!")
    except Exception as e:
        print(" Ataque em y detectado (integridade acionada):")
        print("Detalhes do erro:", e)

    # --- Ataque 3: Adulteração do componente c (ciphertext cifrado) ---
    print("\nAtaque: Alteração do ciphertext (modificando c)")
    try:
        (y, c) = ciphertext
        if isinstance(c, tuple):
            c_list = list(c)
            c_list[1] = c_list[1] + 1
            c_attack = tuple(c_list)
        else:
            c_attack = c + 1
        ciphertext_attack = (y, c_attack)
        _ = pke_cca.decrypt_CCA(ciphertext_attack, privatekey, p)
        print(" Erro: Decifragem com ciphertext adulterado (c modificado) ocorreu sem erro!")
    except Exception as e:
        print(" Ataque em c detectado (integridade acionada):")
        print("Detalhes do erro:", e)

...
-----
Vamoos agr começar a fazer a alinea (d)
-----
...

```

```

if (user_input == "4"):
    lambd = 16
    k = 2

    print("\n==== Protocolo OT k-out-of-n via PKE ElGamal IND-CCA ===\n")

    provider = ProviderOT(lambd)
    msgs = [
        b"Mensagem 1: segredo A",
        b"Mensagem 2: segredo B",
        #b"Mensagem 3: segredo C",
        #b"Mensagem 4: segredo D",
        #b"Mensagem 5: segredo E",
        #b"Mensagem 6: segredo F",
        #b"Mensagem 7: segredo G",
        #b"Mensagem 8: segredo H",
        #b"Mensagem 9: segredo I",
        #b"Mensagem 10: segredo J",
        #b"Mensagem 11: segredo K",
        b"Mensagem 12: segredo L",
        b"Mensagem 13: segredo M",
    ]
    provider.set_messages(msgs)
    provider.setK(k)
    OC = provider.generate_OC()
    receiver = ReceiverOT(lambd, k)
    receiver.setN(len(msgs))
    p,q,g = provider.getParams()
    receiver.setProviderParams(p,q,g)

    receiver.setOC(OC)
    p_vector, tau = receiver.generate_p()
    provider.receive(p_vector, tau)
    print(provider.verify_OC())
    encrypted_messages = provider.execute_OT()
    if encrypted_messages is not None:
        receiver.receive_encrypted_messages(encrypted_messages)
        decrypted_messages = receiver.decrypt_messages()
        print("\nMensagens decifradas:")
        for i, msg in decrypted_messages.items():
            print(f"{i}: {msg}")

    if (user_input == "5"):
        break

print("\nFim do script")

```