Exercício 1

Enunciado do Problema

Construir uma classe Python que implemente um KEM - ElGamal. A classe deve:

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

Descrição do problema

O objetivo deste problema é desenvolver uma classe em Python que implemente um KEM-ElGamal. O ElGamal é composto por dois algoritmos principais, o Key Encapsulation Method (KEM) e o Public Key Encryption (PKE). O KEM permite a geração de uma chave secreta compartilhada entre duas partes, enquanto o PKE permite a cifragem e decifragem de mensagens.

A classe desenvolvida deve ser capaz de inicializar cada instância com o parâmetro de segurança (tamanho em bits da ordem do grupo cíclico) e gerar as chaves pública e privada para cada instância. Além disso, deve conter métodos para encapsulamento e revelação da chave gerada pelo KEM. Para garantir a segurança IND-CCA do PKE, será necessário implementar a transformação de Fujisaki-Okamoto.

A segurança IND-CCA assegura que, mesmo com acesso às informações cifradas e a capacidade de escolher quais textos cifrados serão decifrados, o atacante não será capaz de distinguir entre mensagens diferentes. A implementação da transformação de Fujisaki-Okamoto é fundamental para garantir esta propriedade na classe que está a ser desenvolvida.

Abordagem e Código Implementado

Para este trabalho serão implementadas 3 abordagens distintas. A primeira será o KEM-ELGamal "original", ou seja, sem uma abordagem que utilize curva elípticas. Na segunda abordagem iremos implementar o KEM-ELGamal com utilização de curvas elípticas e na terceira abordagem iremos implementar um KEM-ELGamal com a utilização da curva elíptica P-256, uma das curvas elípticas aprovadas no NIST.

Na primeira abordagem o código implementa o KEM-ElGamal usando a transformação de Fujisaki-Okamoto. Ele também inclui métodos para assinatura de mensagens usando o algoritmo de assinatura digital Ed25519. O código usa as bibliotecas Sage, os e cryptography para realizar as suas operações.

As funcionalidades da implementação são as seguintes:

Classe KEMElGamal:
__init__(self, security_parameter): o construtor da classe inicializa o objeto com o parâmetro de segurança security_parameter. Em seguida, chama o método generate_keys() para gerar as chaves pública e privada.

generate_safe_prime(self, lower_bound, upper_bound): este método gera um número primo seguro com um número específico de bits. O método itera até encontrar um número primo seguro.

generate_keys(self): este método é responsável por gerar as chaves pública e privada. Ele gera um número primo seguro p e um gerador primitivo g no corpo finito GF(p). Em seguida, gera uma chave privada aleatória private_key e a chave pública correspondente public_key. O método retorna p, g, private_key e public_key.

hkdf_extract_shared_key(self, c2): este método extrai a chave compartilhada a partir de c2, que é um componente do encapsulamento da chave. Ele utiliza o HKDF (HMAC-based Key Derivation Function) para derivar uma chave simétrica a partir do c2.

encapsulate(self): este método encapsula a chave simétrica gerada aleatoriamente em uma chave pública para permitir que ela seja trocada de forma segura entre as partes. Ele gera um número aleatório r, calcula c1 = g^r e c2 = public_key^r. Em seguida, chama o método hkdf_extract_shared_key() para extrair a chave compartilhada a partir de c2 e a retorna juntamente com c1.

reveal(self, c1): este método é responsável por revelar a chave simétrica encapsulada usando a chave privada correspondente. Ele calcula c2 = c1^private_key e chama o método hkdf_extract_shared_key() para extrair a chave compartilhada a partir de c2. Em seguida, retorna a chave compartilhada.

Classe FujisakiOkamotoElGamal (KEMElGamal):

__init__(self, security_parameter): Este método construtor inicializa um objeto FujisakiOkamotoElGamal com um parâmetro de segurança especificado. Ele chama o construtor da classe KEMElGamal para gerar as chaves pública e privada.

encrypt(self, plaintext, public_key_pem): Este método cifra o texto fornecido (plaintext) usando a transformação Fujisaki-Okamoto e a chave pública fornecida em formato PEM (public_key_pem). Ele executa as seguintes etapas:

Chama o método encapsulate() para obter a chave encapsulada (c1) e a chave compartilhada do remetente.
Gera um valor aleatório de 12 bytes (nonce) para usar no algoritmo de

criptografia AES-GCM.
Utiliza a chave compartilhada do remetente e o nonce para cifrar o texto simples usando o algoritmo AES-GCM.
Chama o método sign() para assinar o texto cifrado com a chave privada Ed25519 em formato PEM.
Retorna o conjunto (c1, nonce, ciphertext, signature).

decrypt(self, c1, nonce, ciphertext, signature, public_key_pem): Este método decifra o texto cifrado fornecido (ciphertext) usando a transformação Fujisaki-Okamoto e verifica a assinatura antes de decifrar. Ele executa as seguintes etapas:

Verifica se a assinatura fornecida (signature) é válida para o texto cifrado fornecido, usando a chave pública Ed25519 em formato PEM (public_key_pem).
Chama o método reveal() com o valor c1 para obter a chave compartilhada do receptor.
Utiliza a chave compartilhada do receptor e o nonce fornecido para decifrar o texto cifrado usando o algoritmo AES-GCM.
Retorna o texto decifrado.

generate_ed25519_key_pair(self): Este método gera um par de chaves Ed25519 (private_key_pem, public_key_pem) no formato PEM. Ele utiliza a biblioteca cryptography.hazmat.primitives.asymmetric.ed25519 para gerar a chave privada e, em seguida, extrai a chave pública correspondente. As chaves são serializadas no formato PEM usando a biblioteca cryptography.hazmat.primitives.serialization.

sign(self, data, private_key_pem): Este método assina os dados fornecidos (data) usando a chave privada Ed25519 em formato PEM (private_key_pem). Ele carrega a chave privada a partir do PEM e cria a assinatura usando a função sign() da biblioteca cryptography.
Retorna a assinatura.
verify_signature(self, data, signature, public_key_pem): Verifica se a assinatura é válida para os dados fornecidos usando a chave pública Ed25519 em formato PEM. Levanta uma exceção "Invalid signature!" se a assinatura for inválida.
save_keys_to_file(self, private_key_pem, public_key_pem, private_key_file, public_key_file): Guarda as chaves privada e pública em formato PEM nos arquivos especificados.
load_keys_from_file(self, private_key_file, public_key_file): Carrega as chaves privada e pública em formato PEM dos arquivos especificados e retorna (private_key_pem, public_key_pem).

As funcionalidades implementadas no código são utilizadas para realizar operações criptográficas, como gerar chaves, cifrar e decifrar mensagens, assinar e verificar assinaturas digitais. A combinação do KEM-ElGamal com a transformação Fujisaki-Okamoto fornece um sistema criptográfico híbrido com segurança reforçada. O algoritmo

de assinatura digital Ed25519 é usado para garantir a autenticidade e integridade das mensagens trocadas entre os utilizadores.

O código usa também as seguintes bibliotecas:

```
os: para gerar números aleatórios e manipulação de arquivos.
time: para medir o tempo de execução de operações.
hashlib: para os hashes criptográficos.
sage.all: para realizar operações matemáticas, como operações com
números primos e operações com corpos.
cryptography.hazmat.primitives: para implementar algoritmos
criptográficos, como AES-GCM e Ed25519.
cryptography.exceptions: para tratar exceções específicas de
criptografia.
```

Temos então o código implementado para a primeira abordagem:

```
import os
import time
import hashlib
from sage.all import GF, randint
from sage.crypto.util import random_prime
from cryptography.hazmat.primitives.asymmetric import ed25519
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives.serialization import
load_pem_private_key, load_pem_public_key
from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
import hashlib

class KEMElGamal:
    def __init__(self, security_parameter):
        self.security_parameter = security_parameter
        self.p, self.g, self.private_key, self.public_key =
self.generate_keys()

    def generate_safe_prime(self, lower_bound, upper_bound):
        """Função para gerar um primo seguro."""
        num_bits = self.security_parameter

        while True:
            p = random_prime(2 ** num_bits - 1, lbound=2 ** (num_bits
- 1), proof=True)
            q = (p - 1) // 2
            if is_prime(q) and lower_bound <= p < upper_bound:
                return p

    def generate_keys(self):
```

```
        print("Gerando chaves, aguarde...")
        start_time = time.time()
        p = self.generate_safe_prime(2**(self.security_parameter - 1),
2**self.security_parameter)
        F = GF(p)
        g = F.primitive_element()
        private_key = randint(1, p - 2)
        public_key = g ** private_key
        print(f"Chaves geradas em {time.time() - start_time:.2f}
segundos.")
        return p, g, private_key, public_key

    def hkdf_extract_shared_key(self, c2):
        hkdf = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=None,
        )
        shared_key = hkdf.derive(c2.encode())
        return shared_key

    def encapsulate(self):
        r = randint(1, self.p - 2)
        c1 = self.g ** r
        c2 = self.public_key ** r
        shared_key = self.hkdf_extract_shared_key(str(c2))
        return c1, shared_key

    def reveal(self, c1):
        c2 = c1 ** self.private_key
        shared_key = self.hkdf_extract_shared_key(str(c2))
        return shared_key

class FujisakiOkamotoElGamal(KEMElGamal):
    def __init__(self, security_parameter):
        super().__init__(security_parameter)

    def encrypt(self, plaintext):
        c1, shared_key_sender = self.encapsulate()
        nonce = os.urandom(12)
        aes_gcm = AESGCM(shared_key_sender)
        ciphertext = aes_gcm.encrypt(nonce, plaintext.encode(), None)
        return c1, nonce, ciphertext

    def decrypt(self, c1, nonce, ciphertext):
        shared_key_receiver = self.reveal(c1)
        aes_gcm = AESGCM(shared_key_receiver)
        plaintext = aes_gcm.decrypt(nonce, ciphertext, None).decode()
        return plaintext
```

```
    def generate_ed25519_key_pair(self):
        private_key = ed25519.Ed25519PrivateKey.generate()
        public_key = private_key.public_key()
        private_pem = private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.NoEncryption()
        )
        public_pem = public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )
        return private_pem, public_pem

    def sign(self, data, private_key_pem):
        private_key = load_pem_private_key(private_key_pem, None)
        signature = private_key.sign(data)
        return signature

    def verify_signature(self, data, signature, public_key_pem):
        public_key = load_pem_public_key(public_key_pem)
        try:
            public_key.verify(signature, data)
        except InvalidSignature:
            raise ValueError("Invalid signature!")

    def save_keys_to_file(self, private_key_pem, public_key_pem,
private_key_file, public_key_file):
        with open(private_key_file, 'wb') as f:
            f.write(private_key_pem)
        with open(public_key_file, 'wb') as f:
            f.write(public_key_pem)
        print("Chaves guardadas nos arquivos.")

    def load_keys_from_file(self, private_key_file, public_key_file):
        with open(private_key_file, 'rb') as f:
            private_key_pem = f.read()
        with open(public_key_file, 'rb') as f:
            public_key_pem = f.read()
        return private_key_pem, public_key_pem
```

Na segunda abordagem é implementado o KEM-ELGamal com curvas elípticas e a transformação de Fujisaki-Okamoto. O código também inclui métodos para lidar com assinaturas digitais usando o algoritmo Ed25519. O código utiliza várias bibliotecas, como Sage, os e cryptography, para executar as operações criptográficas e matemáticas necessárias.

As funcionalidades implementadas no código são:

Classe KEMElGamal:

__init__(self, security_parameter): O construtor da classe inicializa um objeto KEMElGamal com um parâmetro de segurança especificado e gera as chaves pública e privada, chamando o método generate_keys().

generate_safe_prime(self, lower_bound, upper_bound): Este método gera um safe prime no intervalo especificado. Um safe prime é um primo p tal que (p-1)/2 também é primo.

generate_keys(self): Este método gera as chaves pública e privada baseadas em curvas elípticas. Primeiro, ele gera um número safe prime e cria um corpo finito com o primo gerado. Em seguida, ele cria uma curva elíptica válida sobre o corpo finito, escolhendo aleatoriamente os coeficientes a e b. Depois, ele gera um ponto aleatório 'g' na curva e uma chave privada como um número inteiro aleatório. A chave pública é calculada multiplicando a chave privada pelo ponto 'g'. Retorna os valores (curve, g, private_key, public_key).

hkdf_extract_shared_key(self, c2): Este método extrai uma chave compartilhada usando o HKDF (HMAC-based Key Derivation Function) com o algoritmo SHA-256. Ele recebe como entrada um valor 'c2' e codifica-o antes de derivar a chave compartilhada. Retorna a chave compartilhada.

encapsulate(self): Este método realiza o encapsulamento da chave na classe ElGamal baseado em curvas elípticas. Primeiro, ele gera um número inteiro aleatório 'r'. Em seguida, calcula 'c1' como 'r' multiplicado pelo ponto 'g' e 'c2' como 'r' multiplicado pela chave pública. A chave compartilhada é extraída usando o método hkdf_extract_shared_key() com 'c2' como input. Retorna o par (c1, shared_key).

reveal(self, c1): Este método revela a chave compartilhada usando o valor 'c1' e a chave privada. Calcula 'c2' multiplicando a chave privada pelo valor 'c1' e extrai a chave compartilhada usando o método hkdf_extract_shared_key() com 'c2' como entrada. Retorna a chave compartilhada.

Classe FujisakiOkamotoElGamal (KEMElGamal):

__init__(self, security_parameter): Este método construtor inicializa um objeto FujisakiOkamotoElGamal com um parâmetro de segurança especificado. Ele chama o construtor da classe KEMElGamal para gerar as chaves pública e privada.

encrypt(self, plaintext, public_key_pem): Este método cifra o texto fornecido (plaintext) usando a transformação Fujisaki-Okamoto e a chave pública fornecida em formato PEM (public_key_pem). Ele executa as seguintes etapas:

Chama o método encapsulate() para obter a chave encapsulada (c1) e a chave compartilhada do remetente.
Gera um valor aleatório de 12 bytes (nonce) para usar no algoritmo de criptografia AES-GCM.
Utiliza a chave compartilhada do remetente e o nonce para cifrar o texto simples usando o algoritmo AES-GCM.
Chama o método sign() para assinar o texto cifrado com a chave privada Ed25519 em formato PEM.
Retorna o conjunto (c1, nonce, ciphertext, signature).

decrypt(self, c1, nonce, ciphertext, signature, public_key_pem): Este método decifra o texto cifrado fornecido (ciphertext) usando a transformação Fujisaki-Okamoto e verifica a assinatura antes de decifrar. Ele executa as seguintes etapas:

Verifica se a assinatura fornecida (signature) é válida para o texto cifrado fornecido, usando a chave pública Ed25519 em formato PEM (public_key_pem).
Chama o método reveal() com o valor c1 para obter a chave compartilhada do receptor.
Utiliza a chave compartilhada do receptor e o nonce fornecido para decifrar o texto cifrado usando o algoritmo AES-GCM.
Retorna o texto decifrado.

generate_ed25519_key_pair(self): Este método gera um par de chaves Ed25519 (private_key_pem, public_key_pem) no formato PEM. Ele utiliza a biblioteca cryptography.hazmat.primitives.asymmetric.ed25519 para gerar a chave privada e, em seguida, extrai a chave pública correspondente. As chaves são serializadas no formato PEM usando a biblioteca cryptography.hazmat.primitives.serialization.

sign(self, data, private_key_pem): Este método assina os dados fornecidos (data) usando a chave privada Ed25519 em formato PEM (private_key_pem). Ele carrega a chave privada a partir do PEM e cria a assinatura usando a função sign() da biblioteca cryptography.
Retorna a assinatura.
verify_signature(self, data, signature, public_key_pem): Verifica se a assinatura é válida para os dados fornecidos usando a chave pública Ed25519 em formato PEM. Levanta uma exceção "Invalid signature!" se a assinatura for inválida.
save_keys_to_file(self, private_key_pem, public_key_pem, private_key_file, public_key_file): Guarda as chaves privada e pública em formato PEM nos arquivos especificados.
load_keys_from_file(self, private_key_file, public_key_file): Carrega as chaves privada e pública em formato PEM dos arquivos especificados e retorna (private_key_pem, public_key_pem).

Este código usa o KEM-ELGAMAL baseado em curvas elípticas com a transformação Fujisaki-Okamoto, além de permitir a criação, verificação e armazenamento de assinaturas digitais usando o algoritmo Ed25519. Ele utiliza várias bibliotecas para realizar as operações criptográficas necessárias de maneira eficiente e segura.

Temos então o código implementado para a segunda abordagem:

```
import os
import time
from sage.all import GF, randint, EllipticCurve, is_prime
from sage.crypto.util import random_prime
from hashlib import sha3_256
from cryptography.hazmat.primitives.asymmetric import ed25519
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives.serialization import
load_pem_private_key, load_pem_public_key
from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
import hashlib

class KEMElGamal:
    def __init__(self, security_parameter):
        self.security_parameter = security_parameter
        self.curve, self.g, self.private_key, self.public_key =
self.generate_keys()

    def generate_safe_prime(self, lower_bound, upper_bound):
        """Função para gerar um primo seguro."""
        num_bits = self.security_parameter

        while True:
            p = random_prime(2 ** num_bits - 1, lbound=2 ** (num_bits
- 1), proof=True)
            q = (p - 1) // 2
            if is_prime(q) and lower_bound <= p < upper_bound:
                return p

    def generate_keys(self):
        print("Gerando chaves, aguarde...")
        start_time = time.time()
        p = self.generate_safe_prime(2**(self.security_parameter - 1),
2**self.security_parameter)
        F = GF(p)
        while True:
            a = F.random_element()
            b = F.random_element()
            try:
                curve = EllipticCurve(F, [a, b])
```

```
                break
            except ValueError:
                continue
        g = curve.random_point()
        private_key = randint(1, p - 2)
        public_key = private_key * g
        print(f"Chaves geradas em {time.time() - start_time:.2f}
segundos.")
        return curve, g, private_key, public_key

    def hkdf_extract_shared_key(self, c2):
        hkdf = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=None,
        )
        shared_key = hkdf.derive(c2.encode())
        return shared_key

    def encapsulate(self):
        r = randint(1, self.curve.order() - 1)
        c1 = r * self.g
        c2 = r * self.public_key
        shared_key = self.hkdf_extract_shared_key(str(c2))
        return c1, shared_key

    def reveal(self, c1):
        c2 = self.private_key * c1
        shared_key = self.hkdf_extract_shared_key(str(c2))
        return shared_key

class FujisakiOkamotoElGamal(KEMElGamal):
    def __init__(self, security_parameter):
        super().__init__(security_parameter)

    def encrypt(self, plaintext, public_key_pem):
        c1, shared_key_sender = self.encapsulate()
        nonce = os.urandom(12)
        aes_gcm = AESGCM(shared_key_sender)
        ciphertext = aes_gcm.encrypt(nonce, plaintext.encode(), None)
        signature = self.sign(ciphertext, public_key_pem)
        return c1, nonce, ciphertext, signature

    def decrypt(self, c1, nonce, ciphertext, signature,
public_key_pem):
        self.verify_signature(ciphertext, signature, public_key_pem)
        shared_key_receiver = self.reveal(c1)
        aes_gcm = AESGCM(shared_key_receiver)
        plaintext = aes_gcm.decrypt(nonce, ciphertext, None).decode()
```

```
        return plaintext

    def generate_ed25519_key_pair(self):
        private_key = ed25519.Ed25519PrivateKey.generate()
        public_key = private_key.public_key()
        private_pem = private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.NoEncryption()
        )
        public_pem = public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )
        return private_pem, public_pem

    def sign(self, data, private_key_pem):
        private_key = load_pem_private_key(private_key_pem, None)
        signature = private_key.sign(data)
        return signature

    def verify_signature(self, data, signature, public_key_pem):
        public_key = load_pem_public_key(public_key_pem)
        try:
            public_key.verify(signature, data)
        except InvalidSignature:
            raise ValueError("Invalid signature!")

    def save_keys_to_file(self, private_key_pem, public_key_pem,
private_key_file, public_key_file):
        with open(private_key_file, 'wb') as f:
            f.write(private_key_pem)
        with open(public_key_file, 'wb') as f:
            f.write(public_key_pem)
        print("Chaves guardadas nos arquivos.")

    def load_keys_from_file(self, private_key_file, public_key_file):
        with open(private_key_file, 'rb') as f:
            private_key_pem = f.read()
        with open(public_key_file, 'rb') as f:
            public_key_pem = f.read()
        return private_key_pem, public_key_pem
```

Para a terceira abordagem foi utilizado o código da segunda abordagem mas com modificações de maneira a que a curva elíptica gerada fosse a curva P-256.

Temos então o seguinte código:

```
import os
import time
```

```
from sage.all import GF, randint, EllipticCurve, is_prime
from sage.crypto.util import random_prime
from hashlib import sha3_256
from cryptography.hazmat.primitives.asymmetric import ed25519
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives.serialization import
load_pem_private_key, load_pem_public_key
from cryptography.exceptions import InvalidSignature
import hashlib
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes

class KEMElGamal:
    def __init__(self, security_parameter):
        self.security_parameter = security_parameter
        self.curve, self.g, self.private_key, self.public_key =
self.generate_keys()

    # Função modificada para usar a curva P-256
    def generate_keys(self):
        print("Gerando chaves, aguarde...")
        start_time = time.time
        # Parâmetros da curva P-256
        p =
11579208921035624876269744694940757353008614341529031419553363130886709
97853951
        a = -3
        b =
41058363725152142129326129780047268409114441015993725554835256314039467
401291
        gx =
48439561293906451759052585252797914202762949526041747995844080717082404
635286
        gy =
36134250956749795798585127919587881956611106672985015071877198253568414
405109
        n =
11579208921035624876269744694940757352999695522413576134201427583856440
340431

        F = GF(p)
        curve = EllipticCurve(F, [a, b])
        g = curve(gx, gy)
        private_key = randint(1, n - 1)
        public_key = private_key * g
        print(f"Chaves geradas em {time.time() - start_time():.2f}
segundos.")
        return curve, g, private_key, public_key
```

```
    def hkdf_extract_shared_key(self, c2):
        hkdf = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=None,
        )
        shared_key = hkdf.derive(c2.encode())
        return shared_key

    def encapsulate(self):
        r = randint(1, self.curve.order() - 1)
        c1 = r * self.g
        c2 = r * self.public_key
        shared_key = self.hkdf_extract_shared_key(str(c2))
        return c1, shared_key

    def reveal(self, c1):
        c2 = self.private_key * c1
        shared_key = self.hkdf_extract_shared_key(str(c2))
        return shared_key


class FujisakiOkamotoElGamal(KEMElGamal):
    def __init__(self, security_parameter):
        super().__init__(security_parameter)

    def encrypt(self, plaintext, public_key_pem):
        c1, shared_key_sender = self.encapsulate()
        nonce = os.urandom(12)
        aes_gcm = AESGCM(shared_key_sender)
        ciphertext = aes_gcm.encrypt(nonce, plaintext.encode(), None)
        signature = self.sign(ciphertext, public_key_pem)
        return c1, nonce, ciphertext, signature

    def decrypt(self, c1, nonce, ciphertext, signature,
public_key_pem):
        self.verify_signature(ciphertext, signature, public_key_pem)
        shared_key_receiver = self.reveal(c1)
        aes_gcm = AESGCM(shared_key_receiver)
        plaintext = aes_gcm.decrypt(nonce, ciphertext, None).decode()
        return plaintext

    def generate_ed25519_key_pair(self):
        private_key = ed25519.Ed25519PrivateKey.generate()
        public_key = private_key.public_key()
        private_pem = private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
```

```
            encryption_algorithm=serialization.NoEncryption()
        )
        public_pem = public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )
        return private_pem, public_pem

    def sign(self, data, private_key_pem):
        private_key = load_pem_private_key(private_key_pem, None)
        signature = private_key.sign(data)
        return signature

    def verify_signature(self, data, signature, public_key_pem):
        public_key = load_pem_public_key(public_key_pem)
        try:
            public_key.verify(signature, data)
        except InvalidSignature:
            raise ValueError("Invalid signature!")

    def save_keys_to_file(self, private_key_pem, public_key_pem,
private_key_file, public_key_file):
        with open(private_key_file, 'wb') as f:
            f.write(private_key_pem)
        with open(public_key_file, 'wb') as f:
            f.write(public_key_pem)
        print("Chaves guardadas nos arquivos.")

    def load_keys_from_file(self, private_key_file, public_key_file):
        with open(private_key_file, 'rb') as f:
            private_key_pem = f.read()
        with open(public_key_file, 'rb') as f:
            public_key_pem = f.read()
        return private_key_pem, public_key_pem
```

Exemplo de aplicação da primeira abordagem

```
if __name__ == "__main__":
    # Cria uma instância da classe FujisakiOkamotoElGamal com um
parâmetro de segurança de 256 bits
    fujisaki_elgamal = FujisakiOkamotoElGamal(256)

    # Exibe os números primos gerados
    print(f"p: {fujisaki_elgamal.p}")
    print(f"g: {fujisaki_elgamal.g}")

    # Gere um par de chaves EdCDSA
    ecdsa_private_key_pem, ecdsa_public_key_pem =
fujisaki_elgamal.generate_ed25519_key_pair()

    # Guarda as chaves EdCDSA em arquivos
```

```
    fujisaki_elgamal.save_keys_to_file(ecdsa_private_key_pem,
ecdsa_public_key_pem, "private_key.pem", "public_key.pem")

    # Carregua as chaves EdCDSA a partir dos arquivos
    loaded_ecdsa_private_key_pem, loaded_ecdsa_public_key_pem =
fujisaki_elgamal.load_keys_from_file("private_key.pem",
"public_key.pem")

    # Define a mensagem a ser cifrada
    message = "Unidade curricular de Estruturas Criptográficas."

    # Cifra a mensagem usando o esquema Fujisaki-Okamoto ElGamal
    encrypted_message = fujisaki_elgamal.encrypt(message)
    print("Mensagem cifrada:", encrypted_message)

    # Decifra a mensagem cifrada
    decrypted_message = fujisaki_elgamal.decrypt(*encrypted_message)
    print("Mensagem decifrada:", decrypted_message)

    # Assina a mensagem original usando a chave privada EdCDSA
    signature = fujisaki_elgamal.sign(message.encode(),
loaded_ecdsa_private_key_pem)
    print("Assinatura:", signature)

    # Verifique a assinatura usando a chave pública EdCDSA
    try:
        fujisaki_elgamal.verify_signature(message.encode(), signature,
loaded_ecdsa_public_key_pem)
        print("A assinatura é válida!")
    except ValueError as e:
        print("A assinatura é inválida:", e)

Gerando chaves, aguarde...
Chaves geradas em 29.57 segundos.
p:
898713872869764474741459535390063582920649930444978129733174241625272
94354727
g: 5
Chaves guardadas nos arquivos.
Mensagem cifrada:
(33214201954569559110929965828197368968148516357358507052373615940117
932420282, b'@=\xe8S\xae\xd7Z\x97$\x1b\xde\xca', b't8V\xa4\xbfw\x08}\
xcb\xabjl\xeb\xe3oP\xe8\x9c,\x1c\x90\r\x14\x1fr\x94\xeb\xaddrLb\x13\
xadl0\xd4\xe8s.\x14\x9eDh\xd4\xab\xfe\xcbs\xb5@p\x02\xefK\x9b%\x01\
x10\x19K\xf3\x98\xddz')
Mensagem decifrada: Unidade curricular de Estruturas Criptográficas.
Assinatura: b'wWu\x03\xaa\xa1\xfe\r\x0czf\x10\x86 yf;\xd0\x06\xc6\
xa6h\xd9\xf4\x1bO)~\x88}O\xd8\x0fT\x1a\x0c\x141$0\xa9\x9d\x06\xe5\
xa1d\xc3\x1cS\x9b\xafo\x90j$\xb7\x11Q\xc7D\xb0\\ \x02'
A assinatura é válida!
```

Exemplo de aplicação da segunda abordagem

```
def main():
    security_parameter = 256

    # Instanciar a classe
    fo_elgamal = FujisakiOkamotoElGamal(security_parameter)

    # Exibição dos números primos
    print(f"p: {fo_elgamal.curve.base_ring().order()}")
    print(f"q: {(fo_elgamal.curve.base_ring().order() - 1) // 2}")

    # Gera um par de chaves Ed25519
    private_key_pem, public_key_pem =
fo_elgamal.generate_ed25519_key_pair()

    # Salva chaves em arquivos
    private_key_file = "private_key.pem"
    public_key_file = "public_key.pem"
    fo_elgamal.save_keys_to_file(private_key_pem, public_key_pem,
private_key_file, public_key_file)

    # Carrega chaves de arquivos
    loaded_private_key_pem, loaded_public_key_pem =
fo_elgamal.load_keys_from_file(private_key_file, public_key_file)

    # Mensagem a ser cifrada
    plaintext = "Unidade curricular de Estruturas Criptográficas."

    # cifrar a mensagem
    c1, nonce, ciphertext, signature = fo_elgamal.encrypt(plaintext,
loaded_private_key_pem)
    print("Mensagem cifrada:", ciphertext)

    # Decifrar a mensagem
    decrypted_text = fo_elgamal.decrypt(c1, nonce, ciphertext,
signature, loaded_public_key_pem)
    print("Mensagem decifrada:", decrypted_text)

if __name__ == "__main__":
    main()
```

```
Gerando chaves, aguarde...
Chaves geradas em 7.87 segundos.
p:
832423376058504749085860495600089258818420262104726838110627143468399
0930547
q:
416211688029252374542930247800044629409210131052363419055313571734199
5465273
```

```
Chaves guardadas nos arquivos.
Mensagem cifrada: b'W2\xeb\xfc\x8eM\x9a373\xd4#6G\xa6\xde(l\xad\xfe\
xdc\xeb~p\xf4\xe7\x91V\xe7\x80\xa1\r\xc4\xeclM\x05\x83\x97\xa38"\x97\
xd9\xc6\xf7Lz\xc67Z(\xe0\x16\xba\xac^\xba\x88\x92\x9fr\xfd*_'
Mensagem decifrada: Unidade curricular de Estruturas Criptográficas.
```

Exemplo de aplicação da terceira abordagem

```
def main():

    security_parameter = 256

    # Inicializa a classe
    fujisaki_okamoto = FujisakiOkamotoElGamal(security_parameter)

    # Gera um par de chaves EdCDSA para assinatura
    private_key_pem, public_key_pem =
fujisaki_okamoto.generate_ed25519_key_pair()

    # Guarda as chaves EdCDSA em arquivos
    fujisaki_okamoto.save_keys_to_file(private_key_pem,
public_key_pem, "private_key.pem", "public_key.pem")

    # Carrega as chaves EdCDSA dos arquivos
    loaded_private_key_pem, loaded_public_key_pem =
fujisaki_okamoto.load_keys_from_file("private_key.pem",
"public_key.pem")

    # Mensagem a ser cifrada
    plaintext = "Unidade curricular de Estruturas Criptográficas."

    # Cifra a mensagem
    c1, nonce, ciphertext, signature =
fujisaki_okamoto.encrypt(plaintext, loaded_private_key_pem)

    # Decifra a mensagem
    decrypted_plaintext = fujisaki_okamoto.decrypt(c1, nonce,
ciphertext, signature, loaded_public_key_pem)

    # Verifica se a mensagem decifrada corresponde à mensagem original
    if decrypted_plaintext == plaintext:
        print("A decifragem foi bem-sucedida!")
        print("Mensagem cifrada:", ciphertext)
        print("Mensagem original:", plaintext)
        print("Mensagem decifrada:", decrypted_plaintext)
    else:
        print("A decifragem falhou!")

if __name__ == "__main__":
    main()
```

```
Gerando chaves, aguarde...
Chaves geradas em -0.00 segundos.
Chaves guardadas nos arquivos.
A decifragem foi bem-sucedida!
Mensagem cifrada: b'[N\xc4\xed\n=C\xff=\xec\xcalv\xd8$\xce\xa7\x8a\
xc5\tE\xb5\xff\xf8\x0e\x06\xe8\xad\x1di\xa42\x899^\xf5S\xce\xac\x85\
x17fsHTv_\xdc\xdb`P\x99\xd2\x1e\x00\x8a\x89\x98pP\x17\xda\xda7\xb8'
Mensagem original: Unidade curricular de Estruturas Criptográficas.
Mensagem decifrada: Unidade curricular de Estruturas Criptográficas.
```