Implementação do ElGamal em Curvas Elípticas : IND-CPA e IND-CCA

Ao contrário da implementação anterior baseada em aritmética modular em $\mathbb{Z}_p$, esta implementação explora as propriedades das **curvas elípticas**, proporcionando a **mesma segurança** com **chaves menores** e operações aritméticas **mais eficientes**.

1. Curvas Elípticas

As implementações utilizam a curva **Edwards25519**, uma curva elíptica definida pela equação:

$$ax² + y² = 1 + dx²y²$$

Onde:

- $a = -1$
- $d = -121665/121666$
- $p = 2^255 - 19$ (campo finito)

2. ElGamal em Curvas Elípticas

O algoritmo ElGamal em curvas elípticas é uma adaptação do ElGamal tradicional, em vez da aritmética modular em grupos multiplicativos, usamos **aritmética em curvas elípticas**:

1. **Gerar as Chaves**:
   - Escolhemos um número aleatório **s** como **chave privada**
   - Calcular **H = s·G** onde **G** é o ponto **gerador da curva**
2. **Encode da mensagem**:
   - Transformamos a mensagem em um ponto da curva com o método de **Koblitz**
3. **Cifrar**:
   - Escolher um valor **aleatório r** (**omega** na versão **IND-CPA**)
   - Calcular gamma = r·G
   - Calcular S = r·H (onde H é a chave pública)
   - Calcular C = M + S (onde M é a mensagem encoded como ponto)
4. **Decifrar**:
   - Calcular S = s·gamma (onde s é a chave privada)
   - Calcular M = C - S (via adição do inverso S_inv)
   - Descodificar M para recuperar a mensagem original

2.1 Encode das mensagens

Um aspecto crucial da implementação é o mecanismo para dar encode a mensagens para pontos da curva:

```
def encode_message(self, message):
    m_int = Integer(int.from_bytes(message.encode('utf-8'), 'big'))
    k_bits = self.curve.p.bit_length()
    if m_int.bit_length() > (k_bits - 1 - self.ell):
        raise ValueError("Message too long to encode in one block.")
    x0 = m_int << self.ell  # Append ell zero bits.
    for i in range(2**self.ell):
        x = x0 + i
        if x >= self.curve.p:
            break
        # Compute f(x) = x^3 + a*x + b mod p.
        f_val = self.curve.K(x**3 + self.curve.constants['a4']*x + self.curve.constants['a6'])
        if f_val.is_square():
            y = f_val.sqrt()
            ec_point = self.curve.EC(x, y)
            ed_x, ed_y = self.curve.ec2ed(ec_point)
            return EdPoint(ed_x, ed_y, self.curve)
    raise ValueError("Non-encodable message: tried 2^ell possibilities.")
```

O processo de encode segue os seguintes passos:

1. **Conversão para inteiro**: A mensagem é convertida em bytes e depois em um inteiro

2. **Verificação de tamanho**: Garante que a mensagem cabe dentro do espaço disponível
3. **Aplicação de padding**: Desloca os bits da mensagem à esquerda em `ell` (8) posições
4. **Busca por coordenada x válida**: Testa até 2^ell possibilidades para encontrar um x que gera um ponto válido na curva
5. **Teste de resíduo quadrático**: Verifica se f(x) = x³ + a4*x + a6 é um resíduo quadrático no campo finito
6. **Conversão para ponto na curva Edwards**: Converte o ponto da forma Weierstrass para a forma Edwards

Este método é inspirado no **método de Koblitz**, que mapeia mensagens de forma deterministica, para pontos na curva. O decode segue o processo inverso, recuperar o valor x do ponto na curva Weierstrass e remove o padding.

3. El Gammal IND-CPA

A implementação **IND-CPA** (arquivo `cpa.sage`) fornece segurança básica contra ataques em que o adversário tem acesso ao **oráculo de cifrar**:

```
def encrypt_message(self, public_key, plaintext):
    M = self.encode_message(plaintext)
    omega = random.randint(1, int(self.L) - 1)
    Gamma = self.G.mult(omega)
    S = public_key.mult(omega)
    C = M.add(S)
    return ((int(Gamma.x), int(Gamma.y)), (int(C.x), int(C.y)))

def decrypt_message(self, private_key, encrypted_data):
    (gamma_x, gamma_y), (c_x, c_y) = encrypted_data
    Gamma = self.curve.create_point(self.curve.K(gamma_x), self.curve.K(gamma_y))
    C = self.curve.create_point(self.curve.K(c_x), self.curve.K(c_y))
    S = Gamma.mult(private_key)
    S_inv = S.sim()
    M = C.add(S_inv)
    return self.decode_message(M)
```

Características:

- Utiliza um valor aleatório `omega` para adicionar **aleatoriedade** ao ciphertext
- O ponto codificado é **"mascarado"** pela adição de um ponto S = omega·H
- A cifra é composta por dois pontos da curva: Gamma (omega·G) e C (M + S)

Vulnerabilidades:

- **Não** fornece **integridade dos dados**
- Vulnerável a ataques de manipulação do ciphertext

4. El Gammal IND-CCA

A implementação IND-CCA (arquivo `cca.sage.py`) inclui a transformação Fujisaki-Okamoto para fornecer segurança contra ataques em que o adversário tem acesso ao oráculo de cifrar e de decifrar:

```
def encrypt_message(self, public_key, plaintext):
    m_int = int.from_bytes(plaintext.encode('utf-8'), 'big')
    max_message_bits = 182
    if m_int.bit_length() > max_message_bits:
        raise ValueError(f"Message too long. Must be <= {max_message_bits} bits.")

    r_bits = 64
    r = random.randint(1, (1 << r_bits) - 1)
    combined = (r << max_message_bits) + m_int
    M = self.encode_message(combined)

    # ElGamal encryption
    Gamma = self.G.mult(r)
    Kappa = public_key.mult(r)
    C = M.add(Kappa)

    c_2 = self.H(r)
    c_1 = ((int(Gamma.x), int(Gamma.y)), (int(C.x), int(C.y)))

    return (c_1, c_2)

def decrypt_message(self, private_key, ciphertext):
    c_1, c_2 = ciphertext
    (gamma_x, gamma_y), (c_x, c_y) = c_1

    Gamma = self.curve.create_point(self.curve.K(gamma_x), self.curve.K(gamma_y))
```

```
        C = self.curve.create_point(self.curve.K(c_x), self.curve.K(c_y))

        Kappa = Gamma.mult(private_key)
        Kappa_inv = Kappa.sim()
        M = C.add(Kappa_inv)

        combined = self.decode_message(M)

        r_bits = 64
        max_message_bits = 182
        r = combined >> max_message_bits
        m_int = combined & ((1 << max_message_bits) - 1)

        r_hash_calculated = self.H(r)
        if r_hash_calculated != c_2:
            raise ValueError("Ciphertext integrity check failed.")

        byte_length = (m_int.bit_length() + 7) // 8
        m_bytes = m_int.to_bytes(byte_length, 'big')
        return m_bytes.decode('utf-8')
```

Melhorias:

1. **Transformação de Fujisaki-Okamoto**:
   - Gera valor aleatório `r` de 64 bits (tivemos que limitar o tamanho)
   - Combina `r` com a mensagem via deslocamento de bits: `combined = (r << max_message_bits) + m_int`
   - Esta combinação é então codificada como um ponto na curva
2. **Verificação de Integridade**:
   - Inclui um hash de r (c_2) para verificar a integridade do ciphertext
   - Ao decifrar, o valor de r é recuperado e seu hash comparado com o valor recebido

Esta transformação impede que um atacante possa modificar o ciphertext sem ser detetado, uma vez que seria necessário encontrar um r' diferente que produza o mesmo hash que r.

5. Comparação com a implementação anterior do ElGamal

Aspecto	ElGamal exercicio 1	ElGamal exercicio 3
Operações aritméticas	Multiplicação e exponenciação modular	Adição de pontos e multiplicação escalar
Tamanho das chaves	Maior	Menor
Eficiência computacional	Menor	Maior
Codificação da mensagem	Direta, mensagem como elemento do grupo	Complexa, requer mapeamento para pontos da curva
Base de segurança	Problema do logaritmo discreto em $\mathbb{Z}_p$	Problema do logaritmo discreto em curvas elípticas (ECDLP)

A principal vantagem das curvas elípticas é proporcionar o mesmo nível de segurança com chaves significativamente menores, tornando-as ideais para ambientes com menos recursos.