实验四 -- SM9算法的设计与实现(与SM3构建签名、密钥协商、加密和解密操作)

一、实验概述

本实验旨在实现和验证一套基于SM3哈希函数和优化椭圆曲线（Optimized Elliptic Curve）算法的密码系统，包括签名、密钥协商、加密和解密操作。该系统采用了中国国家密码标准（SM3）和双线性对（Pairing）技术。

二、实验环境

- 编程语言：Python
- 运行平台：任意支持Python的环境

三、实验原理

概述

SM9是由中国国家密码管理局发布的一种基于身份的加密算法（IBE，Identity-Based Encryption）。该算法使用用户的身份标识（如email地址、电话号码等）作为公钥，不依赖数字证书。SM9主要包含五部分内容：总则、数字签名算法、密钥交换协议、密钥封装机制与公钥加密算法、参数定义。

曲线参数

SM9基于256位BN椭圆曲线，使用素域 F_p 和有限域 F_{p^2}，双线性对采用R-ate配对。主要参数如下：

- 椭圆曲线方程：$y^2 = x^3 + b$
- 方程参数 b：05
- 基域特征 q：

  ```
  B6400000 02A3A6F1 D603AB4F F58EC745 21F2934B 1A7AEEDB E56F9B27 E351457D
  ```

- 群的阶 N：

  ```
  B6400000 02A3A6F1 D603AB4F F58EC744 49F2934B 18EA8BEE E56EE19C D69ECF25
  ```

- 群1的生成元 P_1：
  - x_{p1}：

    ```
    93DE051D 62BF718F F5ED0704 487D01D6 E1E40869 09DC3280 E8C4E481 7C66DDDD
    ```

  - y_{p1}：

    ```
    21FE8DDA 4F21E607 63106512 5C395BBC 1C1C00CB FA602435 0C464CD7 0A3EA616
    ```

- 群2的生成元 P_2：

- x_{p2}:

  ```
  (85AEF3D0 78640C98 597B6027 B441A01F F1DD2C19 0F5E93C4 54806C11 D8806141
  ,
   37227552 92130B08 D2AAB97F D34EC120 EE265948 D19C17AB F9B7213B AF82D65B
  )
  ```

- y_{p2}:

  ```
  (17509B09 2E845C12 66BA0D26 2CBEE6ED 0736A96F A347C8BD 856DC76B 84EBEB96
  ,
   A7CF28D5 19BE3DA6 5F317015 3D278FF2 47EFBA98 A71A0811 6215BBA5 C999A7C7
  )
  ```

SM9 算法主要部分

SM9算法包括密钥部分和算法部分。

密钥部分

由密钥生成中心（KGC）生成，包括主密钥对和用户私钥。

- **主密钥对**:
  - **签名主密钥对**:
    - 私钥：一个在$[1, N-1]$范围内的随机数。
    - 公钥：G2群的基点 P_2 的倍点，倍数为私钥。
  - **加密主密钥对**:
    - 私钥：一个在 $[1, N-1]$ 范围内的随机数。
    - 公钥：$G1$群的基点 P_1 的倍点，倍数为私钥。
- **用户私钥**:
  - **签名私钥**：$G1$群的基点 P_1 的倍点，仅用于签名。
  - **加密私钥**：$G2$群的基点 P_2 的倍点，用于密钥解封、解密和密钥交换。

KGC使用主私钥和用户身份标识（ID）生成用户的私钥。

算法部分

包括签名验签、密钥封装解封、加密解密和密钥交换。

- **签名算法**：使用签名主公钥和签名者的签名私钥对数据进行签名。
- **验签算法**：使用签名主公钥和签名者ID验证签名。
- **密钥封装算法**：使用加密主公钥和密钥解封者ID封装一个对称密钥。
- **密钥解封算法**：使用加密主公钥和密钥解封者ID解封对称密钥。
- **加密算法**：使用加密主公钥和解密者ID加密数据。
- **解密算法**：使用解密者的加密私钥和解密者ID解密数据。
- **密钥交换算法**：交换双方使用加密主公钥、自己的加密私钥和双方的ID协商出一个共享密钥。

用户身份标识符（ID）

在SM9算法中，ID用于私钥生成、签名验签、密钥封装解封、加密解密和密钥交换。不同应用场景下，ID有不同的用途：

- 私钥生成：ID是私钥属主的ID。
- 验签：ID是签名者的ID。
- 密钥封装解封：ID是解封者的ID。

- 加密解密：ID是解密者的ID。
- 密钥交换：双方都需要自己的ID和对方的ID。

总结

SM9算法通过使用用户的身份标识作为公钥，实现了一种基于身份的加密方案，简化了密钥管理和证书交换。其主要包含密钥生成和多种加密算法，能够应用于数字签名、密钥封装、加密解密和密钥交换等多种场景。

四、实验步骤

1. **初始化设置**（**Setup**）
   - 利用优化的椭圆曲线生成主公钥和主私钥。
2. **私钥提取**（**Extract Private Key**）
   - 根据主公钥、主私钥和用户标识提取用户私钥。
3. **公钥提取**（**Extract Public Key**）
   - 根据主公钥和用户标识提取用户公钥。
4. **签名**（**Sign**）
   - 使用用户私钥对消息进行签名。
5. **签名验证**（**Verify**）
   - 验证给定签名的真实性。
6. **生成会话密钥**（**Generate Session Key**）
   - 根据临时密钥和身份信息生成会话密钥。
7. **密钥封装**（**Key Encapsulation Mechanism, KEM**）
   - 封装密钥并生成密文。
8. **密钥解封**（**Key Decapsulation Mechanism, KEM**）
   - 解封密钥并获取会话密钥。
9. **混合加密**（**Hybrid Encryption**）
   - 使用KEM和数据加密机制（Data Encryption Mechanism, DEM）对消息进行加密。
10. **混合解密**（**Hybrid Decryption**）
    - 使用KEM和DEM对密文进行解密。

五、实验过程

1. 初始化设置

```
master_public, master_secret = setup('encrypt')
identity = "2024liuhaoran"
```

- `setup` 函数根据选择的方案（加密）初始化主公钥和主私钥。
- `identity` 代表用户标识。

2. 提取私钥

```python
Da = extract_private_key('encrypt', master_public, master_secret, identity)
```

- 使用`extract_private_key`函数提取用户的私钥。

3. 混合加密

```python
encrypted_data = kem_dem_encrypt(master_public, identity, identity, 16)
print("Encrypted data:", encrypted_data)
```

- 使用`kem_dem_encrypt`函数对消息进行加密。
- 消息内容为用户标识`identity`，验证码长度为16。

4. 混合解密

```python
decrypted_message = kem_dem_decrypt(master_public, identity, Da, encrypted_data, 16)
print("Decrypted message:", decrypted_message)
```

- 使用`kem_dem_decrypt`函数对加密的数据进行解密。
- 解密成功后应能还原原始消息内容。

六、实验结果

```
"F:\Lecture\Take it Easy\商用密码\Lab\.venv\Scripts\python.exe" "F:\Lecture\Take it Easy\商用密码\Lab\GM\sm9.py"
Encrypted data: (([15484974123574032057698198547168767974678662883809643658399193058347061996123, 17
Decrypted message: 2024liuhaoran
```

- 加密后的数据（密文）：

```
Encrypted data:
(([15484974123574032057698198547168767974678662883809643658399193
058347061996123,
17054387294821015828896977862891035178667528378337657598846656374
159783285156],
[20781687502299664114126179715454884676905979885694292478530234106289982878466,
840848539660004377408788201778047248284188974918964507838372890847632042923 5],
[49773585165778898716651566456437118882001647550915061426967333
26692380532829,
727352624354518237085371860866350834864095087137559468686514215459727615941 1]), [5, 7, 3, 80, 10, 90, 16, 88, 3, 86, 64, 4, 12], 'a7')
```

- 解密后的消息：

Decrypted message: 2024liuhaoran

结果表明解密后的消息与原始消息一致，验证了加密和解密过程的正确性。

七、代码分析

代码结构

```
./GM
-optimized_curve.py
-optimized_field_elements.py
-optimized_pairing.py
-sm3.py
-sm9.py
```

核心函数分析

1. **prime_field_inverse**：
   - 计算素域内的元素逆。

   ```python
   def prime_field_inverse(a, n):
       ...
       return lm % n
   ```

2. **hash_to_field_element**：
   - 利用SM3哈希函数，将数据哈希到有限域元素。

   ```python
   def hash_to_field_element(i, z, n):
       ...
       return (h % (n - 1)) + 1
   ```

3. **setup**：
   - 根据选择的方案初始化系统参数。

   ```python
   def setup(scheme):
       ...
       return (master_public_key, s)
   ```

4. **extract_private_key**：
   - 提取用户私钥。

   ```python
   def extract_private_key(scheme, master_public, master_secret, identity):
       ...
       return Da
   ```

5. **kem_dem_encrypt / kem_dem_decrypt**：
   - 实现混合加密和解密。

   ```python
   def kem_dem_encrypt(master_public, identity, message, v):
       ...
       return (C1, C2, C3)

   def kem_dem_decrypt(master_public, identity, D, ct, v):
       ...
       return message
   ```

八、结论

通过对SM3哈希函数和优化椭圆曲线的实现，成功地构建了一套安全的密码系统即SM9算法，涵盖了签名、密钥协商、加密和解密操作。实验结果验证了系统功能的正确性和可靠性。未来可以进一步优化算法性能，并将其应用于实际的密码学应用场景。

附录:代码

optimized_field_elements.py

```
field_modulus =
21888242871839275222246405745257275088696311157297823662689037894645226208583
FQ12_modulus_coeffs = [82, 0, 0, 0, 0, 0, -18, 0, 0, 0, 0, 0] # Implied + [1]
FQ12_mc_tuples = [(i, c) for i, c in enumerate(FQ12_modulus_coeffs) if c]

# python3 compatibility
try:
    foo = long
except:
    long = int

# Extended euclidean algorithm to find modular inverses for
# integers
def prime_field_inv(a, n):
    if a == 0:
        return 0
    lm, hm = 1, 0
    low, high = a % n, n
    while low > 1:
        r = high//low
        nm, new = hm-lm*r, high-low*r
        lm, low, hm, high = nm, new, lm, low
    return lm % n

# A class for field elements in FQ. Wrap a number in this class,
# and it becomes a field element.
class FQ():
    def __init__(self, n):
        if isinstance(n, self.__class__):
            self.n = n.n
        else:
            self.n = n % field_modulus
        assert isinstance(self.n, (int, long))

    def __add__(self, other):
        on = other.n if isinstance(other, FQ) else other
        return FQ((self.n + on) % field_modulus)

    def __mul__(self, other):
        on = other.n if isinstance(other, FQ) else other
        return FQ((self.n * on) % field_modulus)

    def __rmul__(self, other):
```

```python
        return self * other

    def __radd__(self, other):
        return self + other

    def __rsub__(self, other):
        on = other.n if isinstance(other, FQ) else other
        return FQ((on - self.n) % field_modulus)

    def __sub__(self, other):
        on = other.n if isinstance(other, FQ) else other
        return FQ((self.n - on) % field_modulus)

    def __div__(self, other):
        on = other.n if isinstance(other, FQ) else other
        assert isinstance(on, (int, long))
        return FQ(self.n * prime_field_inv(on, field_modulus) % field_modulus)

    def __truediv__(self, other):
        return self.__div__(other)

    def __rdiv__(self, other):
        on = other.n if isinstance(other, FQ) else other
        assert isinstance(on, (int, long)), on
        return FQ(prime_field_inv(self.n, field_modulus) * on % field_modulus)

    def __rtruediv__(self, other):
        return self.__rdiv__(other)

    def __pow__(self, other):
        if other == 0:
            return FQ(1)
        elif other == 1:
            return FQ(self.n)
        elif other % 2 == 0:
            return (self * self) ** (other // 2)
        else:
            return ((self * self) ** int(other // 2)) * self

    def __eq__(self, other):
        if isinstance(other, FQ):
            return self.n == other.n
        else:
            return self.n == other

    def __ne__(self, other):
        return not self == other

    def __neg__(self):
        return FQ(-self.n)

    def __repr__(self):
        return repr(self.n)

    @classmethod
    def one(cls):
        return cls(1)
```

```
    @classmethod
    def zero(cls):
        return cls(0)

# Utility methods for polynomial math
def deg(p):
    d = len(p) - 1
    while p[d] == 0 and d:
        d -= 1
    return d

def poly_rounded_div(a, b):
    dega = deg(a)
    degb = deg(b)
    temp = [x for x in a]
    o = [0 for x in a]
    for i in range(dega - degb, -1, -1):
        o[i] = (o[i] + temp[degb + i] * prime_field_inv(b[degb], field_modulus))
        for c in range(degb + 1):
            temp[c + i] = (temp[c + i] - o[c])
    return [x % field_modulus for x in o[:deg(o)+1]]

# A class for elements in polynomial extension fields
class FQP():
    def __init__(self, coeffs, modulus_coeffs):
        assert len(coeffs) == len(modulus_coeffs)
        self.coeffs = coeffs
        # The coefficients of the modulus, without the leading [1]
        self.modulus_coeffs = modulus_coeffs
        # The degree of the extension field
        self.degree = len(self.modulus_coeffs)

    def __add__(self, other):
        assert isinstance(other, self.__class__)
        return self.__class__([(x+y) % field_modulus for x,y in zip(self.coeffs,
other.coeffs)])

    def __sub__(self, other):
        assert isinstance(other, self.__class__)
        return self.__class__([(x-y) % field_modulus for x,y in zip(self.coeffs,
other.coeffs)])

    def __mul__(self, other):
        if isinstance(other, (int, long)):
            return self.__class__([c * other % field_modulus for c in
self.coeffs])
        else:
            # assert isinstance(other, self.__class__)
            b = [0] * (self.degree * 2 - 1)
            inner_enumerate = list(enumerate(other.coeffs))
            for i, eli in enumerate(self.coeffs):
                for j, elj in inner_enumerate:
                    b[i + j] += eli * elj
            # MID = len(self.coeffs) // 2
            for exp in range(self.degree - 2, -1, -1):
                top = b.pop()
                for i, c in self.mc_tuples:
                    b[exp + i] -= top * c
```

```
            return self.__class__([x % field_modulus for x in b])

    def __rmul__(self, other):
        return self * other

    def __div__(self, other):
        if isinstance(other, (int, long)):
            return self.__class__([c * prime_field_inv(other, field_modulus) %
field_modulus for c in self.coeffs])
        else:
            assert isinstance(other, self.__class__)
            return self * other.inv()

    def __truediv__(self, other):
        return self.__div__(other)

    def __pow__(self, other):
        o = self.__class__([1] + [0] * (self.degree - 1))
        t = self
        while other > 0:
            if other & 1:
                o = o * t
            other >>= 1
            t = t * t
        return o

    # Extended euclidean algorithm used to find the modular inverse
    def inv(self):
        lm, hm = [1] + [0] * self.degree, [0] * (self.degree + 1)
        low, high = self.coeffs + [0], self.modulus_coeffs + [1]
        while deg(low):
            r = poly_rounded_div(high, low)
            r += [0] * (self.degree + 1 - len(r))
            nm = [x for x in hm]
            new = [x for x in high]
            # assert len(lm) == len(hm) == len(low) == len(high) == len(nm) ==
len(new) == self.degree + 1
            for i in range(self.degree + 1):
                for j in range(self.degree + 1 - i):
                    nm[i+j] -= lm[i] * r[j]
                    new[i+j] -= low[i] * r[j]
            nm = [x % field_modulus for x in nm]
            new = [x % field_modulus for x in new]
            lm, low, hm, high = nm, new, lm, low
        return self.__class__(lm[:self.degree]) / low[0]

    def __repr__(self):
        return repr(self.coeffs)

    def __eq__(self, other):
        assert isinstance(other, self.__class__)
        for c1, c2 in zip(self.coeffs, other.coeffs):
            if c1 != c2:
                return False
        return True

    def __ne__(self, other):
        return not self == other
```

```
    def __neg__(self):
        return self.__class__([-c for c in self.coeffs])

    @classmethod
    def one(cls):
        return cls([1] + [0] * (cls.degree - 1))

    @classmethod
    def zero(cls):
        return cls([0] * cls.degree)

# The quadratic extension field
class FQ2(FQP):
    def __init__(self, coeffs):
        self.coeffs = coeffs
        self.modulus_coeffs = [1, 0]
        self.mc_tuples = [(0, 1)]
        self.degree = 2
        self.__class__.degree = 2

# The 12th-degree extension field
class FQ12(FQP):
    def __init__(self, coeffs):
        self.coeffs = coeffs
        self.modulus_coeffs = FQ12_modulus_coeffs
        self.mc_tuples = FQ12_mc_tuples
        self.degree = 12
        self.__class__.degree = 12
```

optimized_curve.py

```
from optimized_field_elements import FQ2, FQ12, field_modulus, FQ

curve_order =
21888242871839275222246405745257275088548364400416034343698204186575808495617

# Curve order should be prime
assert pow(2, curve_order, curve_order) == 2
# Curve order should be a factor of field_modulus**12 - 1
assert (field_modulus ** 12 - 1) % curve_order == 0

# Curve is y**2 = x**3 + 3
b = FQ(3)
# Twisted curve over FQ**2
b2 = FQ2([3, 0]) / FQ2([9, 1])
# Extension curve over FQ**12; same b value as over FQ
b12 = FQ12([3] + [0] * 11)

# Generator for curve over FQ
G1 = (FQ(1), FQ(2), FQ(1))
# Generator for twisted curve over FQ2
G2 =
(FQ2([10857046999023057135944570762232829481370756359578518086990519993285655852
781,
11559732032986387107991004021392285783925812861821192530917403151452391805634]),
```

```
 FQ2([84956539231234314176049732474892724384181905872636001487702806493069581019
30,
4082367875863433681332203403145435568316851327593401208105741076214120093531]),
FQ2.one())

# Check if a point is the point at infinity
def is_inf(pt):
    return pt[-1] == pt[-1].__class__.zero()

# Check that a point is on the curve defined by y**2 == x**3 + b
def is_on_curve(pt, b):
    if is_inf(pt):
        return True
    x, y, z = pt
    return y**2 * z - x**3 == b * z**3

assert is_on_curve(G1, b)
assert is_on_curve(G2, b2)

# Elliptic curve doubling
def double(pt):
    x, y, z = pt
    W = 3 * x * x
    S = y * z
    B = x * y * S
    H = W * W - 8 * B
    S_squared = S * S
    newx = 2 * H * S
    newy = W * (4 * B - H) - 8 * y * y * S_squared
    newz = 8 * S * S_squared
    return newx, newy, newz

# Elliptic curve addition
def add(p1, p2):
    one, zero = p1[0].__class__.one(), p1[0].__class__.zero()
    if p1[2] == zero or p2[2] == zero:
        return p1 if p2[2] == zero else p2
    x1, y1, z1 = p1
    x2, y2, z2 = p2
    U1 = y2 * z1
    U2 = y1 * z2
    V1 = x2 * z1
    V2 = x1 * z2
    if V1 == V2 and U1 == U2:
        return double(p1)
    elif V1 == V2:
        return (one, one, zero)
    U = U1 - U2
    V = V1 - V2
    V_squared = V * V
    V_squared_times_V2 = V_squared * V2
    V_cubed = V * V_squared
    W = z1 * z2
    A = U * U * W - V_cubed - 2 * V_squared_times_V2
    newx = V * A
    newy = U * (V_squared_times_V2 - A) - V_cubed * U2
    newz = V_cubed * W
```

```
    return (newx, newy, newz)

# Elliptic curve point multiplication
def multiply(pt, n):
    if n == 0:
        return (pt[0].__class__.one(), pt[0].__class__.one(),
pt[0].__class__.zero())
    elif n == 1:
        return pt
    elif not n % 2:
        return multiply(double(pt), n // 2)
    else:
        return add(multiply(double(pt), int(n // 2)), pt)

def eq(p1, p2):
    x1, y1, z1 = p1
    x2, y2, z2 = p2
    return x1 * z2 == x2 * z1 and y1 * z2 == y2 * z1

def normalize(pt):
    x, y, z = pt
    return (x / z, y / z)

# "Twist" a point in E(FQ2) into a point in E(FQ12)
w = FQ12([0, 1] + [0] * 10)

# Convert P => -P
def neg(pt):
    if pt is None:
        return None
    x, y, z = pt
    return (x, -y, z)

def twist(pt):
    if pt is None:
        return None
    _x, _y, _z = pt
    # Field isomorphism from Z[p] / x**2 to Z[p] / x**2 - 18*x + 82
    xcoeffs = [_x.coeffs[0] - _x.coeffs[1] * 9, _x.coeffs[1]]
    ycoeffs = [_y.coeffs[0] - _y.coeffs[1] * 9, _y.coeffs[1]]
    zcoeffs = [_z.coeffs[0] - _z.coeffs[1] * 9, _z.coeffs[1]]
    x, y, z = _x - _y * 9, _y, _z
    nx = FQ12([xcoeffs[0]] + [0] * 5 + [xcoeffs[1]] + [0] * 5)
    ny = FQ12([ycoeffs[0]] + [0] * 5 + [ycoeffs[1]] + [0] * 5)
    nz = FQ12([zcoeffs[0]] + [0] * 5 + [zcoeffs[1]] + [0] * 5)
    return (nx * w **2, ny * w**3, nz)

# Check that the twist creates a point that is on the curve
G12 = twist(G2)
assert is_on_curve(G12, b12)
```

optimized_pairing.py

```
from optimized_curve import double, add, multiply, is_on_curve, neg, twist, b,
b2, b12, curve_order, G1, normalize
from optimized_field_elements import FQ12, field_modulus, FQ
```

```
ate_loop_count = 29793968203157093288
log_ate_loop_count = 63
pseudo_binary_encoding = [0, 0, 0, 1, 0, 1, 0, -1, 0, 0, 1, -1, 0, 0, 1, 0,
                          0, 1, 1, 0, -1, 0, 0, 1, 0, -1, 0, 0, 0, 0, 1, 1,
                          1, 0, 0, -1, 0, 0, 1, 0, 0, 0, 0, 0, -1, 0, 0, 1,
                          1, 0, 0, -1, 0, 0, 0, 1, 1, 0, -1, 0, 0, 1, 0, 1, 1]

assert sum([e * 2 ** i for i, e in enumerate(pseudo_binary_encoding)]) ==
ate_loop_count


def normalize1(p):
    x, y = normalize(p)
    return x, y, x.__class__.one()


# Create a function representing the line between P1 and P2,
# and evaluate it at T. Returns a numerator and a denominator
# to avoid unneeded divisions
def linefunc(P1, P2, T):
    zero = P1[0].__class__.zero()
    x1, y1, z1 = P1
    x2, y2, z2 = P2
    xt, yt, zt = T
    # points in projective coords: (x / z, y / z)
    # hence, m = (y2/z2 - y1/z1) / (x2/z2 - x1/z1)
    # multiply numerator and denominator by z1z2 to get values below
    m_numerator = y2 * z1 - y1 * z2
    m_denominator = x2 * z1 - x1 * z2
    if m_denominator != zero:
        # m * ((xt/zt) - (x1/z1)) - ((yt/zt) - (y1/z1))
        return m_numerator * (xt * z1 - x1 * zt) - m_denominator * (yt * z1 - y1
* zt), \
               m_denominator * zt * z1
    elif m_numerator == zero:
        # m = 3(x/z)^2 / 2(y/z), multiply num and den by z**2
        m_numerator = 3 * x1 * x1
        m_denominator = 2 * y1 * z1
        return m_numerator * (xt * z1 - x1 * zt) - m_denominator * (yt * z1 - y1
* zt), \
               m_denominator * zt * z1
    else:
        return xt * z1 - x1 * zt, z1 * zt


def cast_point_to_fq12(pt):
    if pt is None:
        return None
    x, y, z = pt
    return (FQ12([x.n] + [0] * 11), FQ12([y.n] + [0] * 11), FQ12([z.n] + [0] *
11))


# Check consistency of the "line function"
one, two, three = G1, double(G1), multiply(G1, 3)
negone, negtwo, negthree = multiply(G1, curve_order - 1), multiply(G1,
curve_order - 2), multiply(G1, curve_order - 3)
```

```python
assert linefunc(one, two, one)[0] == FQ(0)
assert linefunc(one, two, two)[0] == FQ(0)
assert linefunc(one, two, three)[0] != FQ(0)
assert linefunc(one, two, negthree)[0] == FQ(0)
assert linefunc(one, negone, one)[0] == FQ(0)
assert linefunc(one, negone, negone)[0] == FQ(0)
assert linefunc(one, negone, two)[0] != FQ(0)
assert linefunc(one, one, one)[0] == FQ(0)
assert linefunc(one, one, two)[0] != FQ(0)
assert linefunc(one, one, negtwo)[0] == FQ(0)


# Main miller loop
def miller_loop(Q, P, final_exponentiate=True):
    if Q is None or P is None:
        return FQ12.one()
    R = Q
    f_num, f_den = FQ12.one(), FQ12.one()
    for b in pseudo_binary_encoding[63::-1]:
        # for i in range(log_ate_loop_count, -1, -1):
        _n, _d = linefunc(R, R, P)
        f_num = f_num * f_num * _n
        f_den = f_den * f_den * _d
        R = double(R)
        # if ate_loop_count & (2**i):
        if b == 1:
            _n, _d = linefunc(R, Q, P)
            f_num = f_num * _n
            f_den = f_den * _d
            R = add(R, Q)
        elif b == -1:
            nQ = neg(Q)
            _n, _d = linefunc(R, nQ, P)
            f_num = f_num * _n
            f_den = f_den * _d
            R = add(R, nQ)
    # assert R == multiply(Q, ate_loop_count)
    Q1 = (Q[0] ** field_modulus, Q[1] ** field_modulus, Q[2] ** field_modulus)
    # assert is_on_curve(Q1, b12)
    nQ2 = (Q1[0] ** field_modulus, -Q1[1] ** field_modulus, Q1[2] **
field_modulus)
    # assert is_on_curve(nQ2, b12)
    _n1, _d1 = linefunc(R, Q1, P)
    R = add(R, Q1)
    _n2, _d2 = linefunc(R, nQ2, P)
    f = f_num * _n1 * _n2 / (f_den * _d1 * _d2)
    # R = add(R, nQ2) This line is in many specifications but it technically does
nothing
    if final_exponentiate:
        return f ** ((field_modulus ** 12 - 1) // curve_order)
    else:
        return f


# Pairing computation
def pairing(Q, P, final_exponentiate=True):
    assert is_on_curve(Q, b2)
    assert is_on_curve(P, b)
```

```
    if P[-1] == P[-1].__class__.zero() or Q[-1] == Q[-1].__class__.zero():
        return FQ12.one()
    return miller_loop(twist(Q), cast_point_to_fq12(P),
final_exponentiate=final_exponentiate)


def final_exponentiate(p):
    return p ** ((field_modulus ** 12 - 1) // curve_order)
```

sm9.py

```
import binascii
from math import ceil, floor, log
from sm3 import sm3_key_derivation_function, sm3_hash
from random import SystemRandom
import optimized_curve as ec
import optimized_pairing as ate

FAILURE = False
SUCCESS = True


# 计算素域内的逆
def prime_field_inverse(a, n):
    """
    计算素域内元素的逆
    :param a: 被求逆元素
    :param n: 模数
    :return: 素域内元素的逆
    """
    if a == 0:
        return 0
    lm, hm = 1, 0  # 初始化 lm 和 hm
    low, high = a % n, n  # 初始化 low 和 high
    while low > 1:
        r = high // low  # 计算商
        nm, new = hm - lm * r, high - low * r  # 更新 nm 和 new
        lm, low, hm, high = nm, new, lm, low  # 交换 lm, low, hm 和 high
    return lm % n  # 返回结果


# 计算整数的二进制位长度
def bit_length(n):
    """
    计算整数的二进制位长度
    :param n: 整数
    :return: 二进制位长度
    """
    return floor(log(n, 2) + 1)  # 计算位长度


# 整数转换为定长字符串
def int_to_fixed_length_str(m, l):
    """
    整数转换为定长字符串
    :param m: 整数
    :param l: 长度
```

```python
    :return: 定长字符串
    """
    format_m = ('%x' % m).zfill(l * 2).encode('utf-8')  # 将整数转换为定长字符串
    octets = [j for j in binascii.a2b_hex(format_m)]  # 转换为字节
    octets = octets[0:l]  # 截取前 l 个字节
    return ''.join(['%02x' % oc for oc in octets])  # 转换为字符串


# 有限域元素转换为字符串
def field_element_to_str(fe):
    """
    有限域元素转换为字符串
    :param fe: 有限域元素
    :return: 字符串
    """
    fe_str = ''.join(['%x' % c for c in fe.coeffs])  # 提取有限域元素的系数
    if (len(fe_str) % 2) == 1:
        fe_str = '0' + fe_str  # 补齐字符串长度
    return fe_str  # 返回结果


# 椭圆曲线点转换为字符串
def elliptic_curve_point_to_str(P):
    """
    椭圆曲线点转换为字符串
    :param P: 椭圆曲线点
    :return: 字符串
    """
    ec_str = ''.join([field_element_to_str(fe) for fe in P])  # 转换为字符串
    return ec_str  # 返回结果


# 字符串转换为十六进制字节数组
def string_to_hex_bytes(str_in):
    """
    字符串转换为十六进制字节数组
    :param str_in: 字符串
    :return: 十六进制字节数组
    """
    return [b for b in str_in.encode('utf-8')]  # 转换为字节数组


# 哈希到有限域元素
def hash_to_field_element(i, z, n):
    """
    哈希到有限域元素
    :param i: 整数索引
    :param z: 字节串
    :param n: 有限域的阶
    :return: 有限域元素
    """
    l = 8 * ceil((5 * bit_length(n)) / 32)  # 计算长度
    msg = int_to_fixed_length_str(i, 1).encode('utf-8')  # 转换索引为字符串
    ha = sm3_key_derivation_function(msg + z, l)  # 计算哈希值
    h = int(ha, 16)  # 转换为整数
    return (h % (n - 1)) + 1  # 返回有限域元素
```

```python
# 初始化设置
def setup(scheme):
    """
    初始化设置
    :param scheme: 使用的方案
    :return: 公钥和私钥
    """
    P1 = ec.G2  # 初始化椭圆曲线点 P1
    P2 = ec.G1  # 初始化椭圆曲线点 P2

    rand_gen = SystemRandom()  # 随机数生成器
    s = rand_gen.randrange(ec.curve_order)  # 生成私钥

    if (scheme == 'sign'):  # 如果是签名方案
        Ppub = ec.multiply(P2, s)  # 计算公钥
        g = ate.pairing(P1, Ppub)  # 计算双线性对
    elif (scheme == 'keyagreement') | (scheme == 'encrypt'):  # 如果是密钥协商或加密
方案
        Ppub = ec.multiply(P1, s)  # 计算公钥
        g = ate.pairing(Ppub, P2)  # 计算双线性对
    else:
        raise Exception('Invalid scheme')  # 抛出异常

    master_public_key = (P1, P2, Ppub, g)  # 构造主公钥
    return (master_public_key, s)  # 返回主公钥和主私钥


# 提取私钥
def extract_private_key(scheme, master_public, master_secret, identity):
    """
    提取私钥
    :param scheme: 使用的方案
    :param master_public: 主公钥
    :param master_secret: 主私钥
    :param identity: 用户身份标识
    :return: 用户私钥
    """
    P1 = master_public[0]  # 提取主公钥的P1
    P2 = master_public[1]  # 提取主公钥的P2

    user_id = sm3_hash(string_to_hex_bytes(identity))  # 计算用户ID的哈希值
    m = hash_to_field_element(1, (user_id + '01').encode('utf-8'),
ec.curve_order)  # 计算有限域元素
    m = master_secret + m  # 计算中间值
    if (m % ec.curve_order) == 0:  # 检查中间值是否为零
        return FAILURE  # 返回失败
    m = master_secret * prime_field_inverse(m, ec.curve_order)  # 计算私钥

    if (scheme == 'sign'):  # 如果是签名方案
        Da = ec.multiply(P1, m)  # 计算用户私钥
    elif (scheme == 'keyagreement') | (scheme == 'encrypt'):  # 如果是密钥协商或加密
方案
        Da = ec.multiply(P2, m)  # 计算用户私钥
    else:
        raise Exception('Invalid scheme')  # 抛出异常

    return Da  # 返回用户私钥
```

```python
# 提取公钥
def extract_public_key(scheme, master_public, identity):
    """
    提取公钥
    :param scheme: 使用的方案
    :param master_public: 主公钥
    :param identity: 用户身份标识
    :return: 用户公钥
    """
    P1, P2, Ppub, g = master_public  # 提取主公钥

    user_id = sm3_hash(string_to_hex_bytes(identity))  # 计算用户ID的哈希值
    h1 = hash_to_field_element(1, (user_id + '01').encode('utf-8'),
ec.curve_order)  # 计算有限域元素

    if (scheme == 'sign'):  # 如果是签名方案
        Q = ec.multiply(P2, h1)  # 计算用户公钥
    elif (scheme == 'keyagreement') | (scheme == 'encrypt'):  # 如果是密钥协商或加密
方案
        Q = ec.multiply(P1, h1)  # 计算用户公钥
    else:
        raise Exception('Invalid scheme')  # 抛出异常

    Q = ec.add(Q, Ppub)  # 计算最终公钥

    return Q  # 返回用户公钥


# 签名
def sign(master_public, Da, msg):
    """
    签名
    :param master_public: 主公钥
    :param Da: 用户私钥
    :param msg: 消息
    :return: 签名
    """
    g = master_public[3]  # 提取主公钥中的g

    rand_gen = SystemRandom()  # 随机数生成器
    x = rand_gen.randrange(ec.curve_order)  # 生成随机数
    w = g ** x  # 计算 w

    msg_hash = sm3_hash(string_to_hex_bytes(msg))  # 计算消息哈希值
    z = (msg_hash + field_element_to_str(w

                                          )).encode('utf-8')  # 构造 z
    h = hash_to_field_element(2, z, ec.curve_order)  # 计算 h
    l = (x - h) % ec.curve_order  # 计算 l

    S = ec.multiply(Da, l)  # 计算签名 S
    return (h, S)  # 返回签名


# 验证签名
def verify(master_public, identity, msg, signature):
    """
```

```
    验证签名
    :param master_public: 主公钥
    :param identity: 用户身份标识
    :param msg: 消息
    :param signature: 签名
    :return: 验证结果
    """
    (h, S) = signature  # 提取签名

    if (h < 0) | (h >= ec.curve_order):  # 检查 h 的合法性
        return FAILURE  # 返回失败
    if not ec.is_on_curve(S, ec.b2):  # 检查 S 是否在曲线上
        return FAILURE  # 返回失败

    Q = extract_public_key('sign', master_public, identity)  # 提取用户公钥

    g = master_public[3]  # 提取主公钥中的 g
    u = ate.pairing(S, Q)  # 计算 u
    t = g ** h  # 计算 t
    wprime = u * t  # 计算 w'

    msg_hash = sm3_hash(string_to_hex_bytes(msg))  # 计算消息哈希值
    z = (msg_hash + field_element_to_str(wprime)).encode('utf-8')  # 构造 z
    h2 = hash_to_field_element(2, z, ec.curve_order)  # 计算 h2

    if h != h2:  # 检查 h 和 h2 是否相等
        return FAILURE  # 返回失败
    return SUCCESS  # 返回成功


# 生成临时密钥
def generate_ephemeral_key(master_public, identity):
    """
    生成临时密钥
    :param master_public: 主公钥
    :param identity: 用户身份标识
    :return: 临时密钥
    """
    Q = extract_public_key('keyagreement', master_public, identity)  # 提取用户公钥

    rand_gen = SystemRandom()  # 随机数生成器
    x = rand_gen.randrange(ec.curve_order)  # 生成随机数
    R = ec.multiply(Q, x)  # 计算临时密钥

    return (x, R)  # 返回临时密钥


# 生成会话密钥
def generate_session_key(idA, idB, Ra, Rb, D, x, master_public, entity, l):
    """
    生成会话密钥
    :param idA: 实体A的标识
    :param idB: 实体B的标识
    :param Ra: 实体A的临时密钥
    :param Rb: 实体B的临时密钥
    :param D: 实体的私钥
    :param x: 临时私钥
    :param master_public: 主公钥
```

```
    :param entity: 实体标识
    :param l: 密钥长度
    :return: 会话密钥
    """
    P1, P2, Ppub, g = master_public  # 提取主公钥

    if entity == 'A':
        R = Rb  # 如果实体是A，则R为Rb
    elif entity == 'B':
        R = Ra  # 如果实体是B，则R为Ra
    else:
        raise Exception('Invalid entity')  # 抛出异常

    g1 = ate.pairing(R, D)  # 计算 g1
    g2 = g ** x  # 计算 g2
    g3 = g1 ** x  # 计算 g3

    if entity == 'B':
        (g1, g2) = (g2, g1)  # 如果实体是B，则交换 g1 和 g2

    uidA = sm3_hash(string_to_hex_bytes(idA))  # 计算实体A的哈希值
    uidB = sm3_hash(string_to_hex_bytes(idB))  # 计算实体B的哈希值

    kdf_input = uidA + uidB  # 构造KDF输入
    kdf_input += elliptic_curve_point_to_str(Ra) +
elliptic_curve_point_to_str(Rb)  # 添加Ra和Rb
    kdf_input += field_element_to_str(g1) + field_element_to_str(g2) +
field_element_to_str(g3)  # 添加g1, g2, g3

    session_key = sm3_key_derivation_function(kdf_input.encode('utf-8'), l)  # 生
成会话密钥

    return session_key  # 返回会话密钥


# 加密

def key_encapsulation_mechanism_encap(master_public, identity, l):
    """
    密钥封装
    :param master_public: 主公钥
    :param identity: 用户身份标识
    :param l: 密钥长度
    :return: 密钥和密文
    """
    P1, P2, Ppub, g = master_public  # 提取主公钥

    Q = extract_public_key('encrypt', master_public, identity)  # 提取用户公钥

    rand_gen = SystemRandom()  # 随机数生成器
    x = rand_gen.randrange(ec.curve_order)  # 生成随机数

    C1 = ec.multiply(Q, x)  # 计算密文 C1
    t = g ** x  # 计算 t

    uid = sm3_hash(string_to_hex_bytes(identity))  # 计算用户ID的哈希值
    kdf_input = elliptic_curve_point_to_str(C1) + field_element_to_str(t) + uid
 # 构造KDF输入
```

```
    k = sm3_key_derivation_function(kdf_input.encode('utf-8'), l)  # 生成密钥

    return (k, C1)  # 返回密钥和密文


def key_encapsulation_mechanism_decap(master_public, identity, D, C1, l):
    """
    密钥解封
    :param master_public: 主公钥
    :param identity: 用户身份标识
    :param D: 用户私钥
    :param C1: 密文
    :param l: 密钥长度
    :return: 密钥
    """
    if not ec.is_on_curve(C1, ec.b2):  # 检查 C1 是否在曲线上
        return FAILURE  # 返回失败

    t = ate.pairing(C1, D)  # 计算 t

    uid = sm3_hash(string_to_hex_bytes(identity))  # 计算用户ID的哈希值
    kdf_input = elliptic_curve_point_to_str(C1) + field_element_to_str(t) + uid
 # 构造KDF输入
    k = sm3_key_derivation_function(kdf_input.encode('utf-8'), l)  # 生成密钥

    return k  # 返回密钥


# 混合加密
def kem_dem_encrypt(master_public, identity, message, v):
    """
    混合加密
    :param master_public: 主公钥
    :param identity: 用户身份标识
    :param message: 消息
    :param v: 验证码长度
    :return: 密文
    """
    hex_msg = string_to_hex_bytes(message)  # 将消息转换为十六进制字节
    mbytes = len(hex_msg)  # 计算消息字节数
    mbits = mbytes * 8  # 计算消息比特数

    k, C1 = key_encapsulation_mechanism_encap(master_public, identity, mbits +
v)  # 进行密钥封装
    k = string_to_hex_bytes(k)  # 转换密钥为十六进制字节
    k1 = k[:mbytes]  # 分割密钥
    k2 = k[mbytes:]  # 分割密钥

    C2 = []
    for i in range(mbytes):
        C2.append(hex_msg[i] ^ k1[i])  # 计算 C2

    hash_input = C2 + k2  # 构造哈希输入
    C3 = sm3_hash(hash_input)[:int(v / 8)]  # 计算 C3

    return (C1, C2, C3)  # 返回密文
```

```python
def kem_dem_decrypt(master_public, identity, D, ct, v):
    """
    混合解密
    :param master_public: 主公钥
    :param identity: 用户身份标识
    :param D: 用户私钥
    :param ct: 密文
    :param v: 验证码长度
    :return: 明文
    """
    C1, C2, C3 = ct  # 提取密文

    mbytes = len(C2)  # 计算消息字节数
    l = mbytes * 8 + v  # 计算长度

    k = key_encapsulation_mechanism_decap(master_public, identity, D, C1, l)  #
进行密钥解封

    k = string_to_hex_bytes(k)  # 转换密钥为十六进制字节
    k1 = k[:mbytes]  # 分割密钥
    k2 = k[mbytes:]  # 分割密钥

    hash_input = C2 + k2  # 构造哈希输入
    C3prime = sm3_hash(hash_input)[:int(v / 8)]  # 计算 C3'

    if C3 != C3prime:  # 检查 C3 和 C3' 是否相等
        return FAILURE  # 返回失败

    pt = []
    for i in range(mbytes):
        pt.append(chr(C2[i] ^ k1[i]))  # 计算明文

    message = ''.join(pt)  # 生成明文

    return message  # 返回明文


if __name__ == '__main__':
    # 加密 "2024liuhaoran"
    master_public, master_secret = setup('encrypt')
    identity = "2024liuhaoran"
    Da = extract_private_key('encrypt', master_public, master_secret, identity)
    encrypted_data = kem_dem_encrypt(master_public, identity, identity, 16)

    print("Encrypted data:", encrypted_data)

    # 解密加密的数据
    decrypted_message = kem_dem_decrypt(master_public, identity, Da,
encrypted_data, 16)

    print("Decrypted message:", decrypted_message)
```