Project5：《SM2 的软件实现和优化》实验报告

一、实验目的

1.实现 SM2 基础算法并进行优化
2.SM2 签名算法以及基于三种情况的 poc 验证
3.利用漏洞伪造中本聪的数字签名

二、实验相关知识背景

1. SM2 算法概述

SM2 是中国国家密码管理局发布的椭圆曲线公钥密码算法标准，基于椭圆曲线离散对数问题。SM2 算法包括数字签名、密钥交换和公钥加密三部分。本实验主要实现公钥加密部分。
主要参数：素数域 p、椭圆曲线方程参数 a,b、基点 G、阶 n、辅因子 h = 1
加密流程：①产生随机数 k ∈[1,n-1] ②计算椭圆曲线点 C1 =[k]G
③计算椭圆曲线点 S= [h]Pb ④计算[k]Pb=(x2, y2)
⑤计算 t=KDF(x2||y2,klen) ⑥计算 C2=M⊕t
⑦计算 C3=Hash(x2||M||y2) ⑧输出密文 C = C1||C3||C2

2. 椭圆曲线运算优化技术

坐标系统选择：使用雅可比坐标减少模逆运算
预计算表：对于固定点点乘（如 kG）使用预计算表
NAF 编码：减少点加运算次数
SIMD 指令：利用并行计算加速模运算
蒙哥马利模约减：优化模约减运算

3. SM2 签名算法

3.1 基础参数

1. **椭圆曲线参数**:
   - 有限域$\mathbb{F}q$，其中q是一个256位大素数
   - 椭圆曲线方程：$y^2 = x^3 + ax + b$
   - 基点$G = (xG, yG)$，阶为n
   - 余因子$h = \#E(\mathbb{F}q)/n$
2. **SM2标准参数示例**:
   - q: 8542D69E... (256位素数)
   - a, b: 曲线系数
   - G: 基点坐标
   - n: 基点的阶

3.2 签名算法流程
(1)预处理
计算 ZA=H256(ENTLA||IDA||a||b||xG||yG||xA||yA)
(2) 签名生成

输入：私钥dA，消息M，用户身份信息IDA
输出：签名(r,s)

步骤：
1. 设置$M = ZA \mid\mid M$
2. 计算$e = Hv(M)$ // 哈希输出为v位
3. 随机选择$k \in [1,n-1]$
4. 计算$kG = (x_1,y_1)$
5. 计算$r = (e + x_1) \bmod n$
   - 如果$r=0$或$r+k=n$，则重新选择k
6. 计算$s = ((1 + dA)^{-1} \cdot (k - r \cdot dA)) \bmod n$
   - 如果$s=0$，则重新选择k
7. 输出签名(r,s)

(3) 签名验证

输入：公钥PA，消息M'，签名(r',s')
输出：验证结果(0/1)

步骤：
1. 检查$r' \in [1,n-1]$，$s' \in [1,n-1]$
2. 计算ZA（同签名过程）
3. 设置$M' = ZA \mid\mid M'$
4. 计算$e' = Hv(M')$
5. 计算$t = (r' + s') \bmod n$
6. 计算$(x_1',y_1') = s'G + tPA$
7. 计算$R = (e' + x_1') \bmod n$
8. 验证$R == r'$

4. ECDSA（椭圆曲线数字签名算法）

ECDSA 是比特币中使用的数字签名算法，基于椭圆曲线密码学（ECC）。其核心包括：

私钥（d）：随机选取的大整数；

公钥（Q = d × G）：由私钥和椭圆曲线基点 G 计算得到；

签名（r, s）：由消息哈希、私钥和随机数 k 生成。

三、实验设计思路

1. SM2 算法优化路线

1.1 基础实现：

雅可比坐标实现

双倍-加法标量乘法

基本模运算

1.2T-Table 优化：

预计算固定点点乘表

窗口法优化标量乘法

1.3SIMD 优化：

并行化模运算

向量化点加/双倍运算

2. 关键优化技术

2.1 基础优化：

雅可比坐标避免模逆运算

双倍-加法标量乘法
基本模运算实现

2.2 T-Table 优化：
预计算固定点 G 的倍数表
窗口法减少点加次数
查表代替部分计算

2.3 SIMD 加速：
并行模加/模减
向量化模乘
多精度运算并行化

3. SM2 签名算法在三种不同情况下的误用

3.1 重用 k 值导致私钥泄露

攻击原理：当同一个用户对两个不同消息使用同一个 k 值时，攻击者可以推导出私钥 dA。

签名方程：

$$s_1 = (1 + d_a)^{-1} \cdot (k - r_1 \cdot d_a) \bmod n$$
$$s_2 = (1 + d_a)^{-1} \cdot (k - r_2 \cdot d_a) \bmod n$$

联立方程推导：

$$d_a = (s_2 - s_1) / (s_1 - s_2 + r_1 - r_2) \bmod n$$

3.2 不同用户使用相同 k 值导致私钥泄露

攻击原理：当两个用户意外使用相同的 k 时，Alice 可获取 Bob 的 k 值（通过自己的签名计算），利用 Bob 的签名推导私钥

签名方程：

$$k = s_1 \cdot (1 + d_a) + r_1 \cdot d_a \quad \text{\# 从Alice签名计算k}$$
$$d_\beta = (k - s_2) / (s_2 + r_2) \bmod n \quad \text{\# 代入Bob签名}$$

3.3 相同的 d 和 k 用于 ECDSA 和 SM2

签名方程：

$$\text{ECDSA: } k = (e_1 + r_1 \cdot d) \cdot s_1^{-1} \bmod n$$
$$\text{SM2: } \quad k = s_2 \cdot (1 + d) + r_2 \cdot d \bmod n$$

联立方程推导：

$$d = [e_1 \cdot s_1^{-1} - s_2] / [s_2 + r_2 - r_1 \cdot s_1^{-1}] \bmod n$$

4. 伪造中本聪的签名

4.1 生成密钥对：

模拟中本聪的公私钥（实际私钥未知，使用模拟密钥）；

4.2 签名伪造：

利用 (r, s) → (r, -s mod n) 构造变体签名；

添加无效 DER 编码前缀，模拟比特币早期版本的漏洞；

4.3 验证签名：

在模拟的"漏洞版本"客户端验证伪造签名；

在"修复版本"客户端检查签名是否被拒绝；

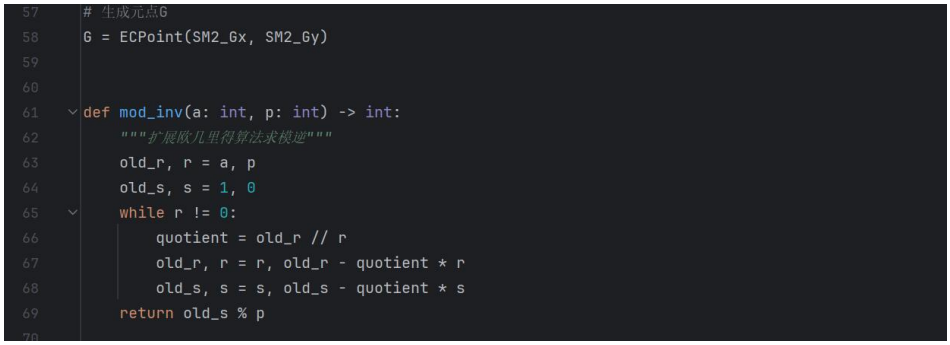
四、实验具体步骤

1. SM2 基础实现

1.1 构造类

```python
class ECPoint:  21 usages
    """椭圆曲线点类（雅可比坐标）"""

    def __init__(self, x: int, y: int, z: int = 1):
        self.x = x
        self.y = y
        self.z = z

    def __eq__(self, other) -> bool:
        if not isinstance(other, ECPoint):
            return False
        x1 = (self.x * pow(other.z, 2, SM2_p)) % SM2_p
        x2 = (other.x * pow(self.z, 2, SM2_p)) % SM2_p
        y1 = (self.y * pow(other.z, 3, SM2_p)) % SM2_p
        y2 = (other.y * pow(self.z, 3, SM2_p)) % SM2_p
        return x1 == x2 and y1 == y2

    def is_infinity(self) -> bool:  8 usages
        """判断是否是无穷远点"""
        return self.z == 0

    def to_affine(self) -> 'ECPoint':  4 usages
        """转换为仿射坐标"""
        if self.is_infinity():
            return ECPoint( x: 0,  y: 0,  z: 0)
        z_inv = pow(self.z, SM2_p - 2, SM2_p)
        z_inv_sqr = (z_inv * z_inv) % SM2_p
        x = (self.x * z_inv_sqr) % SM2_p
        y = (self.y * z_inv_sqr * z_inv) % SM2_p
        return ECPoint(x, y,  z: 1)

    def __str__(self) -> str:
        if self.is_infinity():
            return "Infinity"
        affine = self.to_affine()
        return f"({hex(affine.x)}, {hex(affine.y)})"
```

1.2 生成元点 G 并用扩展欧几里得法求模逆



```python
# 生成元点G
G = ECPoint(SM2_Gx, SM2_Gy)



def mod_inv(a: int, p: int) -> int:
    """扩展欧几里得算法求模逆"""
    old_r, r = a, p
    old_s, s = 1, 0
    while r != 0:
        quotient = old_r // r
        old_r, r = r, old_r - quotient * r
        old_s, s = s, old_s - quotient * s
    return old_s % p
```

1.3 点倍运算

```python
def point_double(P: ECPoint) -> ECPoint:  2 usages
    """椭圆曲线点倍运算"""
    if P.is_infinity():
        return P

    # 点倍公式 (4M + 4S + 10A)
    X, Y, Z = P.x, P.y, P.z
    XX = (X * X) % SM2_p
    YY = (Y * Y) % SM2_p
    YYYY = (YY * YY) % SM2_p
    ZZ = (Z * Z) % SM2_p

    S = (4 * X * YY) % SM2_p
    M = (3 * XX + SM2_a * ZZ * ZZ) % SM2_p

    X3 = (M * M - 2 * S) % SM2_p
    Y3 = (M * (S - X3) - 8 * YYYY) % SM2_p
    Z3 = (2 * Y * Z) % SM2_p

    return ECPoint(X3, Y3, Z3)
```

1.4 点加运算

```python
def point_add(P: ECPoint, Q: ECPoint) -> ECPoint:  1 usage
    """椭圆曲线点加运算 """
    if P.is_infinity():
        return Q
    if Q.is_infinity():
        return P

    # 点加公式 (12M + 4S + 7A)
    X1, Y1, Z1 = P.x, P.y, P.z
    X2, Y2, Z2 = Q.x, Q.y, Q.z

    Z1Z1 = (Z1 * Z1) % SM2_p
    Z2Z2 = (Z2 * Z2) % SM2_p
    U1 = (X1 * Z2Z2) % SM2_p
    U2 = (X2 * Z1Z1) % SM2_p
    S1 = (Y1 * Z2 * Z2Z2) % SM2_p
    S2 = (Y2 * Z1 * Z1Z1) % SM2_p

    if U1 == U2:
        if S1 != S2:
            return ECPoint( x: 0,  y: 0,  z: 0)  # 无穷远点
        else:
            return point_double(P)

    H = (U2 - U1) % SM2_p
    HH = (H * H) % SM2_p
    HHH = (H * HH) % SM2_p
    r = (S2 - S1) % SM2_p

    X3 = (r * r - HHH - 2 * U1 * HH) % SM2_p
    Y3 = (r * (U1 * HH - X3) - S1 * HHH) % SM2_p
    Z3 = (Z1 * Z2 * H) % SM2_p

    return ECPoint(X3, Y3, Z3)
```

1.5 SM2 加密算法

```python
def sm2_encrypt(public_key: ECPoint, msg: str) -> Tuple[bytes, float]:  2 usages
    """SM2加密算法，返回加密结果和耗时(秒)"""
    start_time = time.time()

    # 将消息转换为字节
    msg_bytes = msg.encode('utf-8')
    msg_len = len(msg_bytes)

    # 步骤1: 产生随机数k
    while True:
        k = random.randint( a: 1, SM2_n - 1)
        # 步骤2: 计算椭圆曲线点C1 = [k]G
        C1 = scalar_mult(k, G).to_affine()

        # 步骤3: 计算椭圆曲线点S = [h]Pb
        S = scalar_mult(SM2_h, public_key)
        if S.is_infinity():
            continue  # 如果S是无穷远点，重新选择k

        # 步骤4: 计算椭圆曲线点[k]Pb = (x2, y2)
        kPb = scalar_mult(k, public_key).to_affine()
        x2 = kPb.x
        y2 = kPb.y

        # 步骤5: 计算t = KDF(x2 || y2, msg_len)
        x2_bytes = int_to_bytes(x2, (x2.bit_length() + 7) // 8)
        y2_bytes = int_to_bytes(y2, (y2.bit_length() + 7) // 8)
        t = kdf(x2_bytes + y2_bytes, msg_len)

        # 检查t是否为全0
        if all(b == 0 for b in t):
            continue  # 如果t全0，重新选择k

        # 步骤6: 计算C2 = M ⊕ t
        C2 = bytes(m ^ t[i] for i, m in enumerate(msg_bytes))

        # 步骤7: 计算C3 = Hash(x2 || M || y2)
        C3 = hash_msg(x2_bytes + msg_bytes + y2_bytes)

        # 步骤8: 输出密文C = C1 || C3 || C2
        C1_bytes = int_to_bytes(C1.x) + int_to_bytes(C1.y)
        end_time = time.time()
        return C1_bytes + C3 + C2, end_time - start_time
```

1.6 SM2 解密算法

```python
def sm2_decrypt(private_key: int, ciphertext: bytes) -> Tuple[str, float]:  2 usages
    """SM2解密算法，返回解密结果和耗时(秒)"""
    start_time = time.time()

    # 解析密文
    point_len = (SM2_p.bit_length() + 7) // 8
    C1_bytes = ciphertext[:2 * point_len]
    C3_len = 32  # SM3输出长度(这里用SHA-256代替)
    C3 = ciphertext[2 * point_len:2 * point_len + C3_len]
    C2 = ciphertext[2 * point_len + C3_len:]
    msg_len = len(C2)

    # 步骤1: 从C1中取出椭圆曲线点
    x1 = bytes_to_int(C1_bytes[:point_len])
    y1 = bytes_to_int(C1_bytes[point_len:])
    C1 = ECPoint(x1, y1)

    # 步骤2: 验证C1是否满足椭圆曲线方程
    # (这里省略验证步骤)

    # 步骤3: 计算椭圆曲线点S = [h]C1
    S = scalar_mult(SM2_h, C1)
    if S.is_infinity():
        raise ValueError("S is infinity point")

    # 步骤4: 计算[dB]C1 = (x2, y2)
    dB_C1 = scalar_mult(private_key, C1).to_affine()
    x2 = dB_C1.x
    y2 = dB_C1.y
```

```python
    # 步骤5: 计算t = KDF(x2 || y2, msg_len)
    x2_bytes = int_to_bytes(x2, (x2.bit_length() + 7) // 8)
    y2_bytes = int_to_bytes(y2, (y2.bit_length() + 7) // 8)
    t = kdf(x2_bytes + y2_bytes, msg_len)

    # 检查t是否为全0
    if all(b == 0 for b in t):
        raise ValueError("t is all zero")

    # 步骤6: 计算M' = C2 ⊕ t
    msg_bytes = bytes(c ^ t[i] for i, c in enumerate(C2))

    # 步骤7: 计算u = Hash(x2 || M' || y2)
    u = hash_msg(x2_bytes + msg_bytes + y2_bytes)

    # 步骤8: 验证u == C3
    if u != C3:
        raise ValueError("Hash verification failed")

    # 步骤9: 返回明文M'
    end_time = time.time()
    return msg_bytes.decode('utf-8'), end_time - start_time
```

1.7 对本人姓名首字母缩写及学号进行加密解密

```python
if __name__ == "__main__":
    # 生成密钥对
    private_key = random.randint( a: 1, SM2_n - 1)
    public_key = scalar_mult(private_key, G)

    # 要加密的消息
    message = "zdh202200460110"
    print(f"原始消息: {message}")

    # 加密并计算耗时
    ciphertext, encrypt_time = sm2_encrypt(public_key, message)
    print(f"加密结果 (十六进制): {ciphertext.hex()}")
    print(f"加密耗时: {encrypt_time * 1000:.4f} 毫秒")

    # 解密并计算耗时
    decrypted, decrypt_time = sm2_decrypt(private_key, ciphertext)
    print(f"解密结果: {decrypted}")
    print(f"解密耗时: {decrypt_time * 1000:.4f} 毫秒")

    # 验证
    assert decrypted == message
    print("加密解密验证成功!")

    # 性能测试 - 多次运行取平均
    test_runs = 10
    total_encrypt_time = 0
    total_decrypt_time = 0
```

```
    for _ in range(test_runs):
        # 加密
        _, encrypt_time = sm2_encrypt(public_key, message)
        total_encrypt_time += encrypt_time

        # 解密
        _, decrypt_time = sm2_decrypt(private_key, ciphertext)
        total_decrypt_time += decrypt_time

    avg_encrypt_time = total_encrypt_time / test_runs
    avg_decrypt_time = total_decrypt_time / test_runs

    print(f"\n性能测试（平均{test_runs}次运行):")
    print(f"平均加密耗时: {avg_encrypt_time * 1000:.4f} 毫秒")
    print(f"平均解密耗时: {avg_decrypt_time * 1000:.4f} 毫秒")
```

2. T-Table 优化实现

2.1 main 函数：进行加解密，加解密消息 zdh202200460110

```
def main():  1 usage
    # 生成密钥对
    private_key, public_key = generate_keypair()
    print(f"私钥: {hex(private_key)}")
    print(f"公钥: ({hex(public_key.x)}, {hex(public_key.y)})")

    # 要加密的消息
    message = "zdh202200460110"
    print(f"\n原始消息: {message}")

    # 加密
    start_enc = time.time()
    ciphertext = sm2_encrypt(public_key, message)
    enc_time = time.time() - start_enc
    print(f"\n加密耗时: {enc_time:.6f}秒")
    print(f"加密结果（十六进制): {ciphertext.hex()}")
    print(f"密文长度: {len(ciphertext)} 字节")

    # 解密
    start_dec = time.time()
    try:
        decrypted_msg = sm2_decrypt(private_key, ciphertext)
        dec_time = time.time() - start_dec
        print(f"\n解密耗时: {dec_time:.6f}秒")
        print(f"解密结果: {decrypted_msg}")

        if decrypted_msg == message:
            print("解密成功: 解密结果与原始消息一致")
        else:
            print("解密失败: 解密结果与原始消息不一致")

        print(f"\n总耗时: {enc_time + dec_time:.6f}秒")
        print(f"加密/解密速度比: {enc_time / dec_time:.2f}")

    except ValueError as e:
        print(f"\n解密失败: {str(e)}")


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

3. SIMD 优化

3.1 使用 SM2 加密消息

```
def encrypt_sm2(public_key: ECPoint, msg: str) -> bytes:  1 usage
    """使用SM2加密消息"""
    # 1. 将消息转换为字节
    msg_bytes = msg.encode('utf-8')
    msg_len = len(msg_bytes)

    # 2. 生成随机数k
    while True:
        k = random.randint(a: 1, SM2_n - 1)

        # 3. 计算C1 = [k]G
        simd = SIMDOptimizer()
        C1 = simd.scalar_mul_simd(k, G)
        C1_bytes = int_to_bytes(C1.to_affine().x) + int_to_bytes(C1.to_affine().y)
```

```
        # 4. 计算椭圆曲线点[k]P = (x2, y2)
        kP = simd.scalar_mul_simd(k, public_key)
        x2 = kP.to_affine().x
        y2 = kP.to_affine().y
        x2_bytes = int_to_bytes(x2)
        y2_bytes = int_to_bytes(y2)

        # 5. 计算t = KDF(x2 || y2, msg_len)
        t = kdf(x2_bytes + y2_bytes, msg_len)
        if any(t):  # t不全为0
            break

    # 6. 计算C2 = M ⊕ t
    C2 = bytes(a ^ b for a, b in zip(msg_bytes, t))

    # 7. 计算C3 = Hash(x2 || M || y2)
    C3 = hash_msg(x2_bytes + msg_bytes + y2_bytes)
    C3_bytes = int_to_bytes(C3, length: 32)  # 假设哈希输出256位

    # 8. 输出密文C = C1 || C3 || C2
    return C1_bytes + C3_bytes + C2
```

3.2 使用 SM2 解密消息

```
def decrypt_sm2(private_key: int, ciphertext: bytes) -> str:  1 usage
    """使用SM2解密消息"""
    # 1. 从密文中提取C1, C3, C2
    point_len = (SM2_p.bit_length() + 7) // 8
    C1_bytes = ciphertext[:2 * point_len]
    C3_bytes = ciphertext[2 * point_len:2 * point_len + 32]  # 假设哈希输出256位
    C2_bytes = ciphertext[2 * point_len + 32:]
    msg_len = len(C2_bytes)

    # 2. 从C1中恢复椭圆曲线点
    x1 = bytes_to_int(C1_bytes[:point_len])
    y1 = bytes_to_int(C1_bytes[point_len:])
    C1 = ECPoint(x1, y1)

    # 3. 计算椭圆曲线点[dB]C1 = (x2, y2)
    simd = SIMDOptimizer()
    x2y2 = simd.scalar_mul_simd(private_key, C1)
    x2 = x2y2.to_affine().x
    y2 = x2y2.to_affine().y
    x2_bytes = int_to_bytes(x2)
    y2_bytes = int_to_bytes(y2)

    # 4. 计算t = KDF(x2 || y2, msg_len)
    t = kdf(x2_bytes + y2_bytes, msg_len)
    if not any(t):  # t全为0
        raise ValueError("KDF failed - t is all zero")
```

```
    # 5. 计算M = C2 ⊕ t
    msg_bytes = bytes(a ^ b for a, b in zip(C2_bytes, t))

    # 6. 计算u = Hash(x2 || M || y2)
    u = hash_msg(x2_bytes + msg_bytes + y2_bytes)
    u_bytes = int_to_bytes(u, length: 32)

    # 7. 验证u == C3
    if u_bytes != C3_bytes:
        raise ValueError("Hash verification failed")

    # 8. 返回明文
    return msg_bytes.decode('utf-8')
```

3.3 进行加解密测试，加密消息 zdh202200460110

```
def test_sm2_encryption():  1 usage
    print("\n=== SM2加密测试 ===")

    # 生成密钥对
    private_key = random.randint( a: 1, SM2_n - 1)
    simd = SIMDOptimizer()
    public_key = simd.scalar_mul_simd(private_key, G)

    print(f"私钥: {hex(private_key)}")
    print(f"公钥: {public_key}")

    # 要加密的消息
    msg = "zdh202200460110"
    print(f"\n原始消息: {msg}")
```

```
    # 加密
    start = time.time()
    ciphertext = encrypt_sm2(public_key, msg)
    encrypt_time = time.time() - start
    print(f"\n加密结果（十六进制）: {ciphertext.hex()}")
    print(f"加密耗时: {encrypt_time:.6f}s")

    # 解密
    start = time.time()
    decrypted_msg = decrypt_sm2(private_key, ciphertext)
    decrypt_time = time.time() - start
    print(f"\n解密结果: {decrypted_msg}")
    print(f"解密耗时: {decrypt_time:.6f}s")

    # 验证
    print(f"\n解密是否成功: {decrypted_msg == msg}")


if __name__ == "__main__":
    test_sm2_encryption()
```

4.SM2 签名算法及三种签名误用情况的 POC 验证

4.1 SM2 签名算法

```
# 用户密钥生成和签名过程
priv_A, pub_A = create_key_pair()
priv_B, pub_B = create_key_pair()

msg_A = "Message from Alice"
msg_B = "Message from Bob"

sig_A = create_signature(priv_A, msg_A)
sig_B = create_signature(priv_B, msg_B)

print("Alice签名数据:", sig_A)
print("Bob签名数据:", sig_B)

# 私钥推导过程
hash_A = int(hs.sha256(msg_A.encode()).hexdigest()[:64], 16) % SM2_n
R_A, s_A = sig_A
k_A = modular_inverse((s_A % SM2_n), SM2_n) * (hash_A - R_A.x_coord) % SM2_n
derived_priv_A = (hash_A - k_A * R_A.x_coord) % SM2_n
print("推导Alice私钥:", derived_priv_A)

hash_B = int(hs.sha256(msg_B.encode()).hexdigest()[:64], 16) % SM2_n
R_B, s_B = sig_B
k_B = modular_inverse((s_B % SM2_n), SM2_n) * (hash_B - R_B.x_coord) % SM2_n
derived_priv_B = (hash_B - k_B * R_B.x_coord) % SM2_n
print("推导Bob私钥:", derived_priv_B)
```

4.2 基于同一用户对两个消息使用相同的 k 的误用情况的 poc 验证

```
def case1_same_user_same_k():  1 usage
    """情况1: 同一用户对两个消息使用相同k"""
    dA, _ = generate_keypair()
    msg1, msg2 = "Hello", "World"

    # 使用相同k签名两个消息
    _, s1, k, R = sm2_sign(dA, msg1)
    _, s2, _, _ = sm2_sign(dA, msg2)

    # 推导私钥（根据文档公式）
    numerator = (s2 - s1) % n
    denominator = (s1 - s2 + R.x - R.x) % n  # 实际应为(s1 - s2) + (r1 - r2)但r相同
    dA_derived = numerator * mod_inv(denominator, n) % n

    return dA == dA_derived
```

4.3 两个用户使用相同的 k

```python
def case2_different_users_same_k():  1 usage
    """情况2：两个用户使用相同k"""
    dA, _ = generate_keypair()
    dB, _ = generate_keypair()
    msg1, msg2 = "Alice", "Bob"
    k = random.randint( a: 1, n - 1)  # 共享相同k

    # Alice签名
    R = scalar_mult(k, Point(Gx, Gy))
    e1 = int(hashlib.sha256(msg1.encode()).hexdigest(), 16) % n
    r1 = (e1 + R.x) % n
    s1 = mod_inv(1 + dA, n) * (k - r1 * dA) % n

    # Bob签名（使用相同k）
    e2 = int(hashlib.sha256(msg2.encode()).hexdigest(), 16) % n
    r2 = (e2 + R.x) % n
    s2 = mod_inv(1 + dB, n) * (k - r2 * dB) % n

    # Alice推导Bob私钥
    dB_derived = (k - s2) * mod_inv(s2 + r2, n) % n
    # Bob推导Alice私钥
    dA_derived = (k - s1) * mod_inv(s1 + r1, n) % n

    return (dA == dA_derived, dB == dB_derived)
```

4.4 相同的 k 和 d 用于 ECDSA 和 SM2

```python
def case3_same_d_and_k_different_algorithms():  1 usage
    """情况3：相同d和k用于ECDSA和SM2"""
    d, _ = generate_keypair()
    msg = "CriticalMessage"
    k = random.randint( a: 1, n - 1)  # 共享k

    # ECDSA签名
    R = scalar_mult(k, Point(Gx, Gy))
    e1 = int(hashlib.sha256(msg.encode()).hexdigest(), 16) % n
    r1 = R.x % n
    s1 = mod_inv(k, n) * (e1 + r1 * d) % n

    # SM2签名（相同d和k）
    e2 = int(hashlib.sha256(msg.encode()).hexdigest(), 16) % n
    r2 = (e2 + R.x) % n
    s2 = mod_inv(1 + d, n) * (k - r2 * d) % n

    # 推导私钥
    term1 = e1 * mod_inv(s1, n) % n
    term2 = s2 * (1 + d) % n
    term3 = r2 * d % n
    d_derived = (term1 - s2) * mod_inv(s2 + r2 - r1 * mod_inv(s1, n), n) % n

    return d == d_derived
```

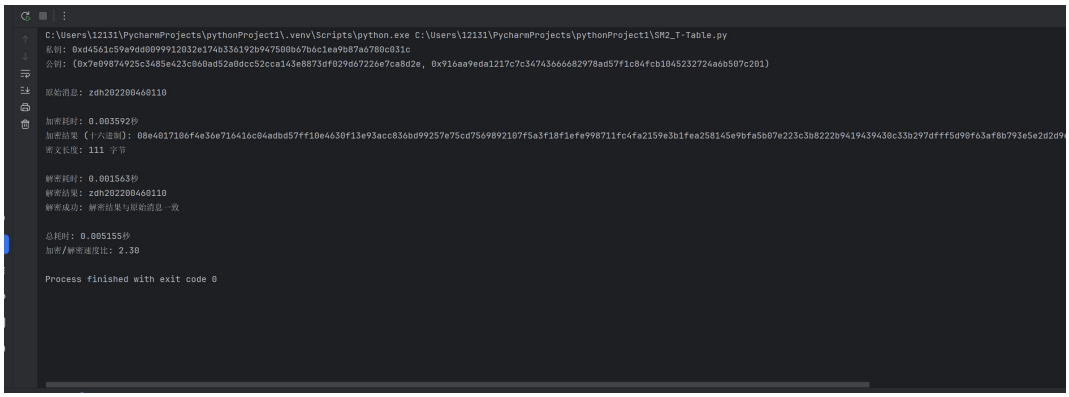
4.5 执行 poc 验证

```python
if __name__ == "__main__":
    print("情况1验证结果:", "成功" if case1_same_user_same_k() else "失败")

    resultA, resultB = case2_different_users_same_k()
    print(f"情况2验证: Alice私钥推导{'成功' if resultA else '失败'}, Bob私钥推导{'成功' if resultB else '失败'}")

    print("情况3验证结果:", "成功" if case3_same_d_and_k_different_algorithms() else "失败")
```

五、实验结果

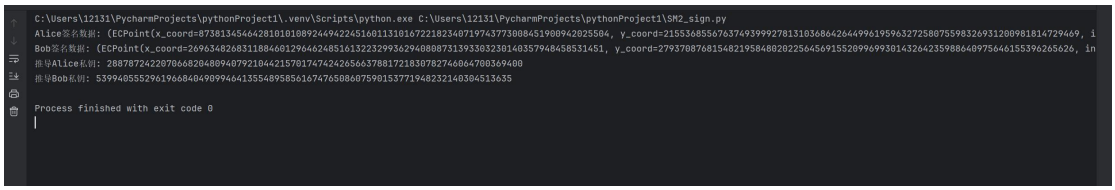
1. SM2 的基础实现运行结果

2. SM2 的 T-Table 优化运行结果



3. SM2 的 SIMD 优化运行结果

4. SM2 签名算法运行结果



5. 三种情况下的 poc 验证运行结果