baseSM2 算法：

1.椭圆曲线：

基于有限域上椭圆曲线的离散对数问题：

$y^2 = x^3 + ax + b \pmod p$

其中 a，b 为系数，p 为大素数，构成有限域 GF(p)

2.SM2 密钥生成：

随机生成私钥 d，满足 $1 \leqslant d \leqslant n-1$

计算公钥 $Q = d \cdot G$，其中·表示椭圆曲线上的点乘运算

3.SM2 签名算法：

计算用户标识杂凑值 Z。

对消息 M 和 Z 进行处理，计算 $e = H(Z \| M)$，其中 H 为 SM3 哈希函数。随机生成 k，满足 $1 \leqslant k \leqslant n-1$。

计算点 $(x_1, y_1) = k \cdot G$。计算 $r = (e + x_1) \bmod n$，若 $r=0$ 或 $r+k=n$ 则重新选择 k。

计算 $s = [(1+d)^{-1} \cdot (k - r \cdot d)] \bmod n$，若 $s=0$ 则重新选择 k。

签名结果为(r, s)。

数学表达式：

$$r = (e + x_1) \bmod n$$

$$s = (k - r \cdot d) \cdot (1 + d)^{-1} \bmod n$$

4.SM2 验证算法：

计算用户标识杂凑值 Z

对消息 M 和 Z 进行处理，计算 $e = H(Z \| M)$

验证 r 和 s 是否满足 $1 \leqslant r, s \leqslant n-1$，若不满足则验证失败

计算 $t = (r + s) \bmod n$，若 $t=0$ 则验证失败

计算点 $(x_2, y_2) = s \cdot G + t \cdot Q$

计算 $R = (e + x_2) \bmod n$

若 $R = r$ 则验证通过，否则失败

数学表达式：

$(e + x_2) \bmod n = r$

其中 $(x_2, y_2) = s \cdot G + t \cdot Q$ 且 $t = (r + s) \bmod n$

5.SM2 加密算法：

随机生成 k，满足 $1 \leqslant k \leqslant n-1$

计算点 $C_1 = k \cdot G$

计算点 $(x_2, y_2) = k \cdot Q$

计算 $t = H(x_2 \,||\, y_2)$，若 t 为全 0 则重新选择 k

计算 $C_2 = M \oplus t$，其中⊕为异或运算

计算 $C_3 = H(x_2 \,||\, M \,||\, y_2)$

密文为 $C = C_1 \,||\, C_2 \,||\, C_3$

6.SM2 解密算法：

解析密文得到 C_1，C_2，C_3

从 C_1 中解析点 (x_1, y_1) 并验证其是否在椭圆曲线上

计算点 $(x_2, y_2) = d \cdot (x_1, y_1)$

计算 $t = H(x_2 \,||\, y_2)$，若 t 为全 0 则解密失败

计算 $M = C_2 \oplus t$

验证 $H(x_2 \| M \| y_2)$是否等于 C_3，若不等则解密失败

返回明文 M

代码整体概述：

1.椭圆曲线点加运算：

点加运算用于计算两个点 P 和 Q 的和 R = P + Q：

```
def point_add(p1, p2):
    if p1.infinity:
        return p2
    if p2.infinity:
        return p1
    if p1.x == p2.x and p1.y != p2.y:
        return Point(0, 0, True)
        # 无穷远点
    # 计算斜率λ
    if p1 != p2:
        lam = (p2.y - p1.y) * pow(p2.x - p1.x, p-2, p) % 
p
    else:
        # 点加倍，P = Q
        lam = (3 * p1.x * p1.x + a) * pow(2 * p1.y, p-2, 
p) % p
```

```
    # 计算结果点坐标
    x3 = (lam * lam - p1.x - p2.x) % p
    y3 = (lam * (p1.x - x3) - p1.y) % p
    return Point(x3, y3)
```

数学原理：

对于不同点 $P(x_1,y_1)$ 和 $Q(x_2,y_2)$，斜率 $\lambda=(y_2-y_1)/(x_2-x_1)$

对于同一点的加倍，斜率 $\lambda=(3x_1^2+a)/(2y_1)$

结果点 $R(x_3,y_3)$ 坐标计算：

$x_3 = \lambda^2 - x_1 - x_2$

$y_3 = \lambda(x_1 - x_3) - y_1$

2.椭圆曲线点乘运算：

点乘运算用于计算 k •P，即点 P 的 k 倍，通过二进制扩展法实现：

```
def point_mul(k, p):
 result = Point(0, 0, True)  # 初始化为无穷远点
 current = p
 while k > 0:
     if k % 2 == 1:
         result = point_add(result, current)
     current = point_add(current, current)  # 计算 2 倍点
     k = k // 2
 return result
```

3.密钥生成实现：

```
   def generate_key_pair():
    # 生成 1 ≤ d ≤ n-1 的随机私钥
    d = int.from_bytes(os.urandom(32), byteorder='big') %
(n-1) + 1
    # 计算公钥 Q = d·G
    Q = point_mul(d, Point(Gx, Gy))
    return d, Q
```

4.签名与验证实现：

```
def sm2_sign(d, M, Z):
    # Z 为用户标识的杂凑值
    M_prime = Z + M
    e = int(sm3_hash(bytes.fromhex(M_prime)), 16)
    while True:
        k = int.from_bytes(os.urandom(32), byteorder='big') %
(n-1) + 1
        kG = point_mul(k, Point(Gx, Gy))
        r = (e + kG.x) % n
        if r == 0 or r + k == n:
            continue
        s = (pow(1 + d, n-2, n) * (k - r * d)) % n
        if s != 0:
```

```
            break
    return (r, s)
def sm2_verify(Q, M, Z, signature):
    r, s = signature
    if not (1 <= r < n and 1 <= s < n):
        return False
    M_prime = Z + M
    e = int(sm3_hash(bytes.fromhex(M_prime)), 16)
    t = (r + s) % n
    if t == 0:
        return False
    sG = point_mul(s, Point(Gx, Gy))
    tQ = point_mul(t, Q)
    P = point_add(sG, tQ)
    if P.infinity:
        return False
    return (e + P.x) % n == r
```

5.加解密实现：

```
   def sm2_encrypt(Q, M):
    # 生成随机数k
    k = int.from_bytes(os.urandom(32), byteorder='big') %
(n-1) + 1
```

```
    # 计算kG和kQ
    kG = point_mul(k, Point(Gx, Gy))
    kQ = point_mul(k, Q)
    # 计算x2 || y2
    x2y2 = format(kQ.x, '064x') + format(kQ.y, '064x')
    # 计算t = SM3(x2 || y2)
    t = sm3_hash(bytes.fromhex(x2y2))
    if t == '0' * 64:
        return None
    # 计算C1 = kG
    C1 = format(kG.x, '064x') + format(kG.y, '064x')
    # 计算C2 = M ^ t
    M_bytes = bytes.fromhex(M)
    t_bytes = bytes.fromhex(t)
    C2 = bytes([a ^ b for a, b in zip(M_bytes, t_bytes)]).hex()
    # 计算C3 = SM3(x2 || M || y2)
    x2 = format(kQ.x, '064x')
    y2 = format(kQ.y, '064x')
    C3 = sm3_hash(bytes.fromhex(x2 + M + y2))
    return C1 + C2 + C3
def sm2_decrypt(d, C):
```

```
# 解析密文
C1_len = 128  # 64字节x + 64字节y
C3_len = 64   # SM3哈希结果长度
C1 = C[:C1_len]
C2 = C[C1_len:-C3_len]
C3 = C[-C3_len:]
# 从C1中解析x1和y1
x1 = int(C1[:64], 16)
y1 = int(C1[64:], 16)
P1 = Point(x1, y1)
# 验证P1是否在椭圆曲线上
if (y1 * y1 - (x1 * x1 * x1 + a * x1 + b)) % p != 0:
    return None
# 计算dP1
dP1 = point_mul(d, P1)
x2 = dP1.x
y2 = dP1.y
# 计算t = SM3(x2 || y2)
x2y2 = format(x2, '064x') + format(y2, '064x')
t = sm3_hash(bytes.fromhex(x2y2))
if t == '0' * 64:
    return None
```

```
    # 计算 M = C2 ^ t
    C2_bytes = bytes.fromhex(C2)
    t_bytes = bytes.fromhex(t)
    M = bytes([a ^ b for a, b in zip(C2_bytes, t_bytes)]).hex()
    # 验证 C3 是否正确
    x2_hex = format(x2, '064x')
    y2_hex = format(y2, '064x')
    if sm3_hash(bytes.fromhex(x2_hex + M + y2_hex)) != C3:
        return None
    return M
```

SM2 算法优化：

1.射影坐标优化：

在仿射坐标中，点加和点乘运算需要多次计算模逆，这是非常耗时的操作。通过使用射影坐标可以显著减少模逆运算的次数。

在射影坐标中，点(x, y)表示为(X, Y, Z)，其中 $x = X/Z^2$，$y = Y/Z^3$。这种表示法可以将点加运算中的模逆操作推迟到最后进行。

```
def point_add_projective(p1, p2):
    # 转换为射影坐标 (X, Y, Z)
    X1, Y1, Z1 = p1.x, p1.y, 1
    X2, Y2, Z2 = p2.x, p2.y, 1
```

```
# 计算中间变量
U1 = (X1 * pow(Z2, 2, p)) % p
U2 = (X2 * pow(Z1, 2, p)) % p
V1 = (Y1 * pow(Z2, 3, p)) % p
V2 = (Y2 * pow(Z1, 3, p)) % p
if U1 == U2:
    if V1 != V2:
        return Point(0, 0, True)  # 相反点，和为无穷远点
    # 点加倍
    S = (2 * V1 * Z1 * Z2) % p
    M = (3 * U1 * U1 + a * pow(Z1 * Z2, 4, p)) % p
    T = (M * M - 2 * S * U1) % p
    X3 = (S * T) % p
    Y3 = (M * (S * U1 - T) - 2 * S * S * V1) % p
    Z3 = (S * Z1 * Z2) % p
else:
    # 点加法
    W = (U2 - U1) % p
    R = (V2 - V1) % p
    T = (R * R) % p
    M = (U1 * W * W) % p
    S = (V1 * W * W * W) % p
```

```
        U3 = (T - 2 * M) % p
        X3 = (W * U3) % p
        Y3 = (R * (M - U3) - S) % p
        Z3 = (W * W * W) % p
    # 转换回仿射坐标
    Z3_inv = pow(Z3, p-2, p)
    x3 = (X3 * pow(Z3_inv, 2, p)) % p
    y3 = (Y3 * pow(Z3_inv, 3, p)) % p
    return Point(x3, y3)
```

2.窗口法点乘优化：

点乘运算 k·P 是 ECC 中最耗时的操作之一。窗口法通过预计算一些点来减少点加操作的次数，从而加速点乘运算。

```
def point_mul_window(k, p, window_size=4):
    # 预计算窗口表
    window_table = [Point(0, 0, True)] * (1 << window_size)
    window_table[1] = p
    # 预计算 2P, 3P, ..., (2^window_size - 1)P
    for i in range(2, 1 << window_size):
        window_table[i] = point_add_projective(window_table[i-1], p)
    # 处理负系数
    for i in range(1, 1 << (window_size - 1)):
```

```
        neg_i = (1 << window_size) - i
        window_table[neg_i] = Point(window_table[i].x, (-window_table[i].y) % p)
    # 将k转换为二进制，并按窗口分组
    k_bits = bin(k)[2:].zfill(((len(bin(k)) - 2 + window_size - 1) // window_size) * window_size)
    num_windows = len(k_bits) // window_size
    result = Point(0, 0, True)
    for i in range(num_windows):
        # 从高位到低位处理
        window = k_bits[i * window_size : (i + 1) * window_size]
        digit = int(window, 2)
        if digit != 0:
            result = point_add_projective(result, window_table[digit])
        # 每处理一个窗口，结果乘以2^window_size
        if i != num_windows - 1:
            for _ in range(window_size):
                result = point_add_projective(result, result)
    return result
```

窗口法的核心思想是将标量 k 分解为基数为 2^w 的数字表示，通

过预计算这些数字对应的点，将点乘运算的复杂度从 O(log k)降低到 O(log k / w)。

SM2 签名算法误用

1.漏洞原理及其数学推导：

如果重复使用相同的 k 值对不同消息进行签名，攻击者可以恢复私钥 d，原理如下：

根据签名公式：

$$s1 = (k - r \cdot d) \cdot (1 + d)^{-1} \bmod n$$

$$s2 = (k - r \cdot d) \cdot (1 + d)^{-1} \bmod n$$

整理可得：

$$s1 \cdot (1 + d) = k - r \cdot d \bmod n$$

$$s2 \cdot (1 + d) = k - r \cdot d \bmod n$$

两式相减：

$$(s1 - s2) \cdot (1 + d) = (e1 - e2) \bmod n$$

进一步推导：

$$1 + d = (e1 - e2) \cdot (s1 - s2)^{-1} \bmod n$$

$$d = [(e1 - e2) \cdot (s1 - s2)^{-1} - 1] \bmod n$$

这表明攻击者可以利用两个使用相同 k 的签名计算出私钥 d。

2.POC 验证实现：

```
def recover_private_key(M1, Z1, sig1, M2, Z2, sig2):
 r1, s1 = sig1
```

```
r2, s2 = sig2
# 确保两个签名使用了相同的r
if r1 != r2:
    print("r1 != r2，可能没有使用相同的随机数k")
    return None
r = r1
e1 = int(sm3_hash(bytes.fromhex(Z1 + M1)), 16)
e2 = int(sm3_hash(bytes.fromhex(Z2 + M2)), 16)
# 计算d = (s1 - s2) / (s2*r - s1*r + e1 - e2) mod n
numerator = (s1 - s2) % n
denominator = (s2 * r - s1 * r + e1 - e2) % n
if denominator == 0:
    print("无法计算，分母为0")
    return None
d = (numerator * pow(denominator, n-2, n)) % n
return d
```

测试结果表明，利用两个使用相同 k 的签名，确实可以成功恢复出私钥，进而伪造任意消息的签名。

伪造中本聪数字签名的分析

1.核心思路：

如果中本聪在签名过程中重复使用了随机数 k，攻击者可以利用

类似SM2中的攻击方法恢复其私钥，进而伪造签名。

2.代码实现：

```
def recover_ecdsa_private_key(public_key, message1, sig1, message2, sig2):
    # 解析签名
    r1, s1 = sigdecode_string(sig1, curve.order)
    r2, s2 = sigdecode_string(sig2, curve.order)
    # 确保r相同（表明可能使用了相同的k）
    if r1 != r2:
        print("r值不同，无法恢复私钥")
        return None
    r = r1
    # 计算消息哈希
    e1 = int.from_bytes(hashlib.sha256(message1).digest(), byteorder='big') % curve.order
    e2 = int.from_bytes(hashlib.sha256(message2).digest(), byteorder='big') % curve.order
    # 计算私钥d
    numerator = (s1 - s2) % curve.order
    denominator = (s2 * r - s1 * r + e1 - e2) % curve.order
    if denominator == 0:
        print("无法计算，分母为0")
```

```
        return None
    d = (numerator * pow(denominator, curve.order - 2, curve.order)) % curve.order
    return ecdsa.SigningKey.from_secret_exponent(d, curve=curve, hashfunc=hashlib.sha256)
```

该函数可以从两个使用相同 k 的 ECDSA 签名中恢复私钥。得到私钥后，攻击者可以伪造任意消息的签名，包括伪造中本聪的签名。