



山东大学  
SHANDONG UNIVERSITY

## Project2 图片水印嵌入和提取

学院 网络空间安全

专业 网络空间安全

学号 202200460149

班级姓名 网安 22.1 班张弛

2025 年 8 月 8 日

# 目录

<b>1</b>	<b>实验任务</b>	<b>1</b>
<b>2</b>	<b>SM2 python 实现和优化</b>	<b>2</b>
2.1	SM2 基础知识 . . . . .	2
2.2	SM2 基础实现 . . . . .	4
2.3	SM2 优化 1: 雅可比坐标 . . . . .	7
2.4	SM2 优化 2 . . . . .	10
2.5	优化对比 . . . . .	11
<b>3</b>	<b>SM2 签名误用</b>	<b>12</b>
3.1	SM2 signature: leaking k . . . . .	12
3.2	SM2 signature: reusing k . . . . .	13
3.3	SM2 signature: reusing k by different users . . . . .	15
3.4	SM2 signature: same d and k with ECDSA . . . . .	17
<b>4</b>	<b>SM2 伪造中本聪签名</b>	<b>19</b>
<b>5</b>	<b>参考链接</b>	<b>21</b>

# 1 实验任务

Project 5: SM2 的软件实现优化

- a). 考虑到 SM2 用 C 语言来做比较复杂, 大家看可以考虑用 python 来做 sm2 的基础实现以及各种算法的改进尝试
- b). 20250713-wen-sm2-public.pdf 中提到的关于签名算法的误用分别基于做 poc 验证, 给出推导文档以及验证代码
- c). 伪造中本聪的数字签名

## 2 SM2 python 实现和优化

### 2.1 SM2 基础知识

SM2 是我国采用的一种公开密钥加密标准，由国家密码管理局于 2010 年 12 月 17 日发布，相关标准为“GM/T 0003-2012 《SM2 椭圆曲线公钥密码算法》”。2016 年，成为中国国家密码标准（GB/T 32918-2016）。

在商用密码体系中，SM2 主要用于替换 RSA 加密算法，其算法公开。据国家密码管理局表示，SM2 基于 ECC，其效率较高，安全性与 NIST Prime256 相当。

SM2 主要包括三部分：签名算法、密钥交换算法、加密算法，在这次的项目中，我们主要关注加密和签名算法，实现并进行优化。

在此之前我们需要了解椭圆曲线相关的原理：

- 有限域  $F_q$  上的椭圆曲线是由点组成的集合。在仿射坐标系下，椭圆曲线上的点  $P$ （非无穷远点）的坐标表示为  $P = (x_p, y_p)$ ，其中  $x$  和  $y$  为满足一定方程的域元素，分别称为点  $P$  的  $x$  坐标和  $y$  坐标。
- 定义在  $F_p$ （ $p$  是大于 3 的素数）上的椭圆曲线方程为：

$$y^2 = x^3 + ax + b, \quad a, b \in F_p, \quad \text{且} \quad (4a^3 + 27b^2) \bmod p \neq 0.$$

- 椭圆曲线  $E(F_p)$  定义为：

$$E(F_p) = \{(x, y) \mid x, y \in F_p, \text{ 且满足上述方程}\} \cup \{O\}, \quad \text{其中 } O \text{ 是无穷远点。}$$

椭圆曲线  $E(F_p)$  上的点的数目用  $\#E(F_p)$  表示，称为椭圆曲线  $E(F_p)$  的阶。

椭圆曲线  $E(F_p)$  上的点按照下面的加法运算规则，构成一个交换群：

- $O + O = O$ ;
- $\forall P = (x, y) \in E(F_p) \setminus \{O\}, \quad P + O = P$ ;
- $\forall P = (x, y) \in E(F_p) \setminus \{O\}, \quad P$  的逆元素  $-P = (x, -y), \quad P + (-P) = O$ ;

两个非互逆的不同点相加的规则：

设  $P_1 = (x_1, y_1) \in E(F_p) \setminus \{O\}, P_2 = (x_2, y_2) \in E(F_p) \setminus \{O\}$ , 且  $x_1 \neq x_2$ 。

设  $P_3 = (x_3, y_3) = P_1 + P_2$ , 则

$$x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1,$$

其中,

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}.$$

设  $P_1 = (x_1, y_1) \in E(F_p) \setminus \{O\}$ , 且  $y_1 \neq 0$ , 设  $P_3 = (x_3, y_3) = P_1 + P_1$ , 则

$$x_3 = \lambda^2 - 2x_1, \quad y_3 = \lambda(x_1 - x_3) - y_1,$$

其中,

$$\lambda = \frac{3x_1^2 + a}{2y_1}.$$

求点  $P$  的  $k$  倍点  $[k]P$ , 将  $k$  用  $l$  长比特串形式表示  $k = \sum_{j=0}^{l-1} k_j 2^j, k_j \in \{0, 1\}$ 。

可按以下步骤进行求解:

1. 置  $Q = O$ ;
2. 从  $l - 1$  下降到 0 执行:  $Q = [2]Q$ ;
3. 若  $k_j = 1$ , 则  $Q = Q + P$ ;
4. 执行结束后得到的  $Q$  即为  $[k]P$ 。

国密推荐曲线参数如下 (十六进制):

```

1 #推荐使用素数域256位椭圆曲线。
2 #椭圆曲线方程:  $y^2 = x^3 + ax + b$ 。
3 #曲线参数:
4 p = FFFFFFFE FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 00000000 FFFFFFFF FFFFFFFF
5 a = FFFFFFFE FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 00000000 FFFFFFFF FFFFFFFC
6 b = 28E9FA9E 9D9F5E34 4D5A9E4B CF6509A7 F39789F5 15AB8F92 DDBCBD41 4D940E93
7 n = FFFFFFFE FFFFFFFF FFFFFFFF FFFFFFFF 7203DF6B 21C6052B 53BBF409 39D54123
8 Gx = 32C4AE2C 1F198119 5F990446 6A39C994 8FE30BBF F2660BE1 715A4589 334C74C7
9 Gy = BC3736A2 F4F6779C 59BDCEE3 6B692153 D0A9877C C62A4740 02DF32E5 2139F0A0
    
```

## 2.2 SM2 基础实现

在正式定义加密解密函数之前，我们先定义椭圆曲线计算的函数：

```

1 def inverse_mod(k, p):
2     return pow(k, -1, p)
3
4 def point_add(P, Q):
5     if P is None: return Q
6     if Q is None: return P
7     x1, y1 = P
8     x2, y2 = Q
9     if x1 == x2 and y1 != y2:
10         return None
11     if P == Q:
12         lam = (3 * x1 * x1 + a) * inverse_mod(2 * y1, p) % p
13     else:
14         lam = (y2 - y1) * inverse_mod(x2 - x1, p) % p
15     x3 = (lam * lam - x1 - x2) % p
16     y3 = (lam * (x1 - x3) - y1) % p
17     return (x3, y3)
18
19 def scalar_mult(k, P):
20     R = None
21     while k:
22         if k & 1:
23             R = point_add(R, P)
24             P = point_add(P, P)
25             k >>= 1
26     return R
    
```

### 1. 密钥生成：

随机选择一个私钥（随机数），通常是一个 256 位的随机数。使用椭圆曲线上的点运算，将私钥与基点（椭圆曲线上的固定点）相乘，得到公钥。公钥是一个曲线上的点，可以表示为 (x, y) 坐标。

```

1 def generate_keypair():
2     d = random.randint(1, n - 2)
3     P = scalar_mult(d, G)
4     return d, P
    
```

### 2. 加密：

随机选择一个临时私钥（临时随机数），通常是一个 256 位的随机数。使用

临时私钥与基点相乘，得到临时公钥。将明文数据转换为椭圆曲线上的点（编码）。生成一个随机数  $k$ ，与临时公钥进行点运算，得到  $C1$  点。使用接收方的公钥进行点运算，将  $C1$  点与明文数据进行异或运算，得到  $C2$  点。使用临时私钥与  $C1$  点相乘，得到一个数值。对  $C2$  点和该数值进行哈希运算，得到  $C3$  点。将  $C1$ 、 $C2$  和  $C3$  点组成密文。

```

1  #字节处理
2  def int_to_bytes(x: int, size=32) -> bytes:
3      return x.to_bytes(size, 'big')
4
5  def bytes_to_int(b: bytes) -> int:
6      return int.from_bytes(b, 'big')
7
8  #KDF函数
9  def KDF(Z: bytes, klen: int) -> bytes:
10     ct = 1
11     v = 256
12     Ha = b''
13     for _ in range((klen + v - 1) // v):
14         msg = Z + ct.to_bytes(4, 'big')
15         digest_hex = sm3.sm3_hash(func.bytes_to_list(msg))
16         Ha += bytes.fromhex(digest_hex)
17         ct += 1
18     return Ha[:klen // 8]
19
20 def sm2_encrypt(message: bytes, pubkey):
21     k = random.randint(1, n - 1)
22     C1 = scalar_mult(k, G)
23     S = scalar_mult(k, pubkey)
24     x2, y2 = S
25     x2_bytes = int_to_bytes(x2)
26     y2_bytes = int_to_bytes(y2)
27     t = KDF(x2_bytes + y2_bytes, len(message) * 8)
28     if int.from_bytes(t, 'big') == 0:
29         raise ValueError("KDF derived all-zero key, aborting")
30     C2 = bytes([m ^ t[i] for i, m in enumerate(message)])
31     C3 = bytes.fromhex(sm3.sm3_hash(func.bytes_to_list(x2_bytes + message +
32                                     y2_bytes)))
33     return C1, C2, C3

```

### 3 解密：

使用私钥与  $C1$  点相乘，得到一个数值。对  $C2$  点和该数值进行哈希运算，得

到 C3 点。将 C1、C2 和 C3 点组成密文。使用接收方的私钥与 C1 点相乘，得到临时公钥。使用临时公钥进行点运算，将 C1 点与 C2 点进行异或运算，得到明文数据。

```

1 def sm2_decrypt(C1, C2, C3, privkey):
2     S = scalar_mult(privkey, C1)
3     x2, y2 = S
4     x2_bytes = int_to_bytes(x2)
5     y2_bytes = int_to_bytes(y2)
6     t = KDF(x2_bytes + y2_bytes, len(C2) * 8)
7     if int.from_bytes(t, 'big') == 0:
8         raise ValueError("KDF derived all-zero key, aborting")
9     M = bytes([c ^ t[i] for i, c in enumerate(C2)])
10    u = bytes.fromhex(sm3.sm3_hash(func.bytes_to_list(x2_bytes + M +
11                                     y2_bytes)))
12    if u != C3:
13        raise ValueError("Decryption failed: hash does not match")
14    return M

```

#### 4. 数字签名：

对待签名数据进行哈希运算，得到哈希值。随机选择一个数值 k，与基点相乘，得到点 (x1, y1)。将 x1 的值与哈希值进行异或运算，得到一个数值。计算该数值的模反函数，得到另一个数值。将哈希值与另一个数值进行相乘，得到一个数值。使用私钥与该数值相乘，得到一个数值。使用点 (x1, y1) 与该数值进行点运算，得到点 (x2, y2)。将 x2 的值与哈希值进行比较，如果相等，则签名有效。

```

1 def sm2_sign(message: bytes, privkey):
2     e = int(sm3.sm3_hash(func.bytes_to_list(message)), 16) % n
3     while True:
4         k = random.randint(1, n - 1)
5         P1 = scalar_mult(k, G)
6         r = (e + P1[0]) % n
7         if r == 0 or r + k == n:
8             continue
9         s = (inverse_mod(1 + privkey, n) * (k - r * privkey)) % n
10        if s != 0:
11            break
12        return (r, s)
13
14 def sm2_verify(message: bytes, signature, pubkey):
15     r, s = signature

```



```

16     if not (1 <= r <= n - 1) or not (1 <= s <= n - 1):
17         return False
18     e = int(sm3.sm3_hash(func.bytes_to_list(message)), 16) % n
19     t = (r + s) % n
20     if t == 0:
21         return False
22     P1 = scalar_mult(s, G)
23     P2 = scalar_mult(t, pubkey)
24     Rxy = point_add(P1, P2)
25     if Rxy is None:
26         return False
27     x1, y1 = Rxy
28     R = (e + x1) % n
29     return R == r

```

基础实现结果：

```

(base) zhangchi@zhangchi-virtual-machine:~/桌面$ python3 SM2_baseline.py
b'Hello, this is a message encrypted using SM2.'
私钥: 0xb98b6015139e9930c1f21df6fb7e4657fee681c8f7d694c1d51b2788a535500f
公钥:
x = 0xb4f63e25a6928da0f2fda089a001868655cf1bd7ae7b10b62feb7038f0f0e0000
y = 0x7ec54cf8f745c86d26adec48cb93bc250c7bfc65d74969f755fff06de196d47b
加密结果:
C1:
x = 0x72af4a1e9c4b2bbf09e4afd8420b6d5dab70768f3eebb6b15ad2142bfe2800ed
y = 0x6f3c4b4a2514d7a0cb15f6de0f2b91e23517213feef71795f0c968ec5526209
C2: e72b21aa06922bf50f5f52b9f3ad5d4a8d9b71af49ac7c387f832faaaadadb5ed9fdf5fdd49fabaf0a9360b52f
C3: 2aa95dc5945d7d408f51852fb93b8c8c4c7865eb0f714e3e3538ae22d793bdd5
加密时间: 0.009706 秒
解密还原明文:
Hello, this is a message encrypted using SM2.
解密时间: 0.004991 秒
签名结果:
r = 0x68776dc7b3568387c2ad1af575f35b73912ee5a5b2a952fee2a19076f8aac125
s = 0xee774c568387d17a9f93d7cc2bb6ae2350fe5eb7db0bb743685210d9505ddcbe
签名时间: 0.005456 秒
验签结果: 成功
验签时间: 0.009725 秒
(base) zhangchi@zhangchi-virtual-machine:~/桌面$

```

图 1 基础实现结果

## 2.3 SM2 优化 1：雅可比坐标

核心思路：ECC 坐标变换

标准为仿射坐标系 (X,Y) :  $Y^2 = X^3 + aX + b$

- jacobian 坐标系 (x,y,z) :  $y^2 = x^3 + ax.Z^4 + bz^6$
- $X = x/z^2, Y = y/z^3$
- jacobian 坐标系计算复杂度低

我们定义两种坐标的转换：

```

1 def to_jacobian(P):
2     x, y = P

```

```

3     return (x, y, 1)
4
5 def from_jacobian(P):
6     X, Y, Z = P
7     if Z == 0:
8         return (0, 0)
9     Z_inv = inverse_mod(Z, p)
10    Z_inv2 = (Z_inv * Z_inv) % p
11    Z_inv3 = (Z_inv2 * Z_inv) % p
12    x = (X * Z_inv2) % p
13    y = (Y * Z_inv3) % p
14    return (x, y)

```

定义了雅可比坐标下的乘积和加法：

```

1 def jacobian_double(P):
2     X1, Y1, Z1 = P
3     if Y1 == 0 or Z1 == 0:
4         return (0, 1, 0) # 无穷远点雅可比坐标表示
5     A = (X1 * X1) % p
6     B = (Y1 * Y1) % p
7     C = (B * B) % p
8     D = (2 * ((X1 + B) * (X1 + B) - A - C)) % p
9     E = (3 * A) % p
10    F = (E * E) % p
11    X3 = (F - 2 * D) % p
12    Y3 = (E * (D - X3) - 8 * C) % p
13    Z3 = (2 * Y1 * Z1) % p
14    return (X3, Y3, Z3)
15
16 def jacobian_add(P, Q):
17     X1, Y1, Z1 = P
18     X2, Y2, Z2 = Q
19
20     if Z1 == 0:
21         return (X2, Y2, Z2)
22     if Z2 == 0:
23         return (X1, Y1, Z1)
24
25     Z1Z1 = (Z1 * Z1) % p
26     Z2Z2 = (Z2 * Z2) % p
27     U1 = (X1 * Z2Z2) % p
28     U2 = (X2 * Z1Z1) % p
29     S1 = (Y1 * Z2 * Z2Z2) % p

```

```

30     S2 = (Y2 * Z1 * Z1Z1) % p
31
32     if U1 == U2:
33         if S1 != S2:
34             return (0, 1, 0) # 无穷远点
35         else:
36             return jacobian_double(P)
37
38     H = (U2 - U1) % p
39     I = (2 * H) % p
40     I = (I * I) % p
41     J = (H * I) % p
42     RR = (2 * (S2 - S1)) % p
43     V = (U1 * I) % p
44
45     X3 = (RR * RR - J - 2 * V) % p
46     Y3 = (RR * (V - X3) - 2 * S1 * J) % p
47     Z3 = ((Z1 + Z2) * (Z1 + Z2) - Z1Z1 - Z2Z2) * H % p
48
49     return (X3, Y3, Z3)
50
51
52 def scalar_mult(k, P):
53     R = (0, 1, 0) # 无穷远点的雅可比坐标表示
54     P_j = to_jacobian(P)
55     for i in bin(k)[2:]:
56         R = jacobian_double(R)
57         if i == '1':
58             R = jacobian_add(R, P_j)
59     return from_jacobian(R)

```

最终运行测试结果:

```

(base) zhangchi@zhangchi-virtual-machine:~/桌面$ python3 SM2_opt1.py
原文消息: b'Hello, this is a message encrypted using SM2.'
私钥: 0x37d923f4c3e6b2e6b4380258214d2878fa37b729af6e80fc4b18974e5965104f
公钥:
x = 0x5ee8e4b5ea1e62b50dd6238d59e7c949ac7cfe1a536056bf2df711d271fa5e5d
y = 0xf26f1d0d964458da0a0e4ba903b4a87f5fe7b123540d8c6eedd5b3abe5a60704
加密时间: 0.003708 秒

解密结果: Hello, this is a message encrypted using SM2.
解密时间: 0.003098 秒

签名r: 0x2833fce31cf8654cea411dd22e6c7726fde3ee888d9d96791961bb59efecd916
签名s: 0xcb4d1da5d754096e8c3b34e7932443e25898bf73ac5981f7f74842c6a22b27f6
签名时间: 0.004252 秒
验签时间: 0.004286 秒
验证结果: 成功
(base) zhangchi@zhangchi-virtual-machine:~/桌面$

```

图 2 SM2 优化 1 结果

## 2.4 SM2 优化 2

核心思想是：

ECC 中固定点点乘  $kG$ ：预计算表

• 运算中  $G$  为固定点，可预计算  $nG$  并存储为表

```

1  # ----- 固定点 G 的预计算加速 -----
2
3  G_TABLE = []
4
5  def precompute_G_table():
6      global G_TABLE
7      G_TABLE = []
8      P = G
9      for _ in range(256):
10         G_TABLE.append(P)
11         P = point_add(P, P)
12
13  def fixed_scalar_mult(k):
14      R = None
15      for i in range(256):
16         if (k >> i) & 1:
17             R = point_add(R, G_TABLE[i])
18      return R
    
```

然后再 main 函数中使用：

```

1  precompute_G_table() # 预计算
    
```

就可以继续使用后面的加密解密函数。

优化 2 运行结果如下所示：

```
(base) zhangchi@zhangchi-virtual-machine:~/桌面$ python3 SM2_opt2.py
b'Hello, this is a message encrypted using SM2.'
私钥: 0x167a1dc00a70c5c2ce1721cee114a93fe55005e3c2fa2b1eb2f4750b4b715fdd
公钥:
x = 0x385813538acfd6acd51288897186a159ecbd30f623f0b6732dce9ad1997802df
y = 0x4df2d5a5efb7181b632e371ecc74490b6bbdf9b8588fb0b1e415186e7f4e3884

加密结果:
c1:
x = 0xa377554a3b2801be48d9efc03ee876ed85defa763cd2e89b4628feeda3e1630b
y = 0xa7b3749b19e6def2ac98ddd65945db4a5fe2493ed3cfc10eecbfa5cdb7cb48bc
c2: 78cd3a2fb0f68ec9f069178b858d8913d758441df5f6db70c6877a1ecadf49be178f88dadf7c91e9c82464646e
c3: 5b42019aa3ce89dea1f477062268549bdbc4d2904d6b22f872c943d1593d16f7
加密时间: 0.008546 秒

解密还原明文:
Hello, this is a message encrypted using SM2.
解密时间: 0.009520 秒

签名结果:
r = 0xbbc395334505668132c2e06502ed818b06bdf6cc60ab98cd4eb5ce5b4a03a60f
s = 0x49785d3aa1d1010e1fd572a73a411966a8a4749e47aa7a69742e9e494e6895a2
签名时间: 0.003761 秒

验证结果: 成功
验签时间: 0.009922 秒
(base) zhangchi@zhangchi-virtual-machine:~/桌面$
```

图 3 优化 2 运行结果

## 2.5 优化对比

根据上面的结果，我们可以列出时间表格：

实现模式	加密时间 (s)	解密时间 (s)	签名时间 (s)	验签时间 (s)
基础实现	0.009708	0.004991	0.005456	0.009725
优化 1	0.003708	0.003098	0.004252	0.004286
优化 2	0.008546	0.009520	0.003761	0.009922

我们可以看到雅可比坐标下的优化更加高效，预计算 G 表优化有限，并且在解密方面没基础实现好。

## 3 SM2 签名误用

ppt 中提到了两种情况我们分别来进行模拟：

### 3.1 SM2 signature: leaking k

推导：

- 设消息哈希  $e = H_v(Z_A || M)$ 。
- 生成随机数  $k \in Z_n^*$ ，计算点  $kG = (x_1, y_1)$ 。
- $r = (e + x_1) \bmod n$ 。
- $s = (1 + d_A)^{-1} \cdot (k - r \cdot d_A) \bmod n$ 。

我们要解  $d_A$ 。

从上式：

$$s(1 + d_A) \equiv k - rd_A \pmod{n}$$

展开：

$$s + sd_A \equiv k - rd_A \pmod{n}$$

移项把  $d_A$  收集：

$$sd_A + rd_A \equiv k - s \pmod{n}$$

$$d_A(s + r) \equiv k - s \pmod{n}$$

只要  $s + r \neq 0 \pmod{n}$ ，就能求得：

$$d_A \equiv (s + r)^{-1} \cdot (k - s) \pmod{n}$$

说明当单次签名的  $k$  泄露时，可直接恢复私钥。

相应代码如下所示：

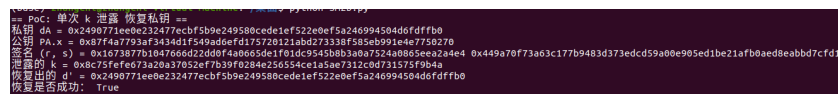
```
1 def recover_d_from_k(r: int, s: int, k: int) -> int:
2     """
3     d = (s + r)^{-1} * (k - s) mod n
```

```

4     """
5     denom = (s + r) % n
6     inv = inv_mod(denom, n)
7     d = (inv * ((k - s) % n)) % n
8     return d
9
10 # ----- PoC 演示 -----
11 def demo_single_k_leak():
12     print("== PoC: 单次 k 泄露 恢复私钥 (gmssl SM3 版本) ==")
13     dA, PA = keygen()
14     print("私钥 dA =", hex(dA))
15     print("公钥 PA.x =", hex(PA[0]))
16     # ID_A 示例
17     ID_A = b'1234567812345678'
18     ZA = compute_ZA(ID_A, PA)
19     M = b"Hello SM2 with SM3 (gmssl) PoC"
20     to_sign = ZA + M
21
22     r, s, k, e = sign(dA, to_sign, k=None) # 随机 k, 但我们拿到 k (模拟泄
        露)
23     print("签名 (r, s) =", hex(r), hex(s))
24     print("泄露的 k =", hex(k))
25
26     d_rec = recover_d_from_k(r, s, k)
27     print("恢复出的 d' =", hex(d_rec))
28     print("恢复是否成功: ", d_rec == dA)
29     assert d_rec == dA

```

运行结果如下所示：



```

== PoC: 单次 k 泄露 恢复私钥 ==
私钥 dA = 0x2490771ee0e232477ecbf5b9e249580cedef522e0ef5a246994504d6dffb0
公钥 PA.x = 0x87f4a7793af3434d1f549ad6ef17572012abd27338f58eb991e4e7750270
签名 (r, s) = 0x1073877b1047666d22d0f4a0065de1f01dc9545b0b30a7524a0065eea2a4e4 0x449a70f773a63c177b9483d373edcd59a00e905ed1be21afbae8eabbd7cfd1
泄露的 k = 0x8c75fefe072a20a37052ef7839f0204e25654cc1a5ae7312c0d721575f94b
恢复出的 d' = 0x2490771ee0e232477ecbf5b9e249580cedef522e0ef5a246994504d6dffb0
恢复是否成功: True

```

图 4 误用 1

可以看到恢复私钥成功。

## 3.2 SM2 signature: reusing k

详细推导如下所示：

- **Precompute:**  $Z_A = H_{256}(ENTL_A || ID_A || a || b || x_G || y_G || x_A || y_A)$
- **Key Generation:**  $P_A = d_A \cdot G$ , order is  $n$

- **Sign( $Z_A, M$ ):**  $Sign_{d_A}(M, Z_A) \rightarrow (r, s)$ 
  - Set  $M = Z_A || M$
  - $e = H_v(M)$
  - $k \leftarrow Z_n^*, kG = (x_1, y_1)$
  - $r = (e + x_1) \bmod n$
  - $s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A)) \bmod n$
  - Signature is  $(r, s)$
- **Verify:**  $(r, s)$  of  $M$  with  $P_A$ 
  - $Z_A = H_{256}(ENTL_A || ID_A || a || b || x_G || y_G || x_A || y_A)$
  - Set  $M = Z_A || M, e = H_v(M)$
  - $t = (r + s) \bmod n$
  - $(x_1, y_1) = sG + tP_A$
  - $R = (e + x_1) \bmod n$ , Verify  $R = r$
- **Signing message  $M_1$  with  $d_A$ :**
  - Randomly select  $k \in [1, n - 1], kG = (x, y)$
  - $r_1 = (Hash(Z_A || M_1) + x) \bmod n$
  - $s_1 = ((1 + d_A)^{-1} \cdot (k - r_1 \cdot d_A)) \bmod n$
- **Signing message  $M_2$  with  $d_A$ :**
  - Reuse the same  $k, kG = (x, y)$
  - $r_2 = (Hash(Z_A || M_2) + x) \bmod n$
  - $s_2 = ((1 + d_A)^{-1} \cdot (k - r_2 \cdot d_A)) \bmod n$
- **恢复  $d_A$  :** 利用 2 个签名  $(r_1, s_1), (r_2, s_2)$ :
  - $s_1(1 + d_A) = (k - r_1 \cdot d_A) \bmod n$
  - $s_2(1 + d_A) = (k - r_2 \cdot d_A) \bmod n$
  - $d_A = \frac{s_2 - s_1}{s_1 - s_2 + r_1 - r_2} \bmod n$

根据上述推导，相应代码如下所示：

```

1 def demo_k_reuse_two_sigs():
2     print("\n== PoC: k 重用 两个不同消息 恢复私钥 (gmssl SM3 版本) ==")
3     dA, PA = keygen()
4     ID_A = b'1234567812345678'
5     ZA = compute_ZA(ID_A, PA)

```



```

6     M1 = b"Message one"
7     M2 = b"Another message"
8     # 故意使用相同 k
9     k_shared = secrets.randbelow(n - 1) + 1
10    r1, s1, k1, e1 = sign(dA, ZA + M1, k=k_shared)
11    r2, s2, k2, e2 = sign(dA, ZA + M2, k=k_shared)
12    print("签名1 (r1,s1) =", hex(r1), hex(s1))
13    print("签名2 (r2,s2) =", hex(r2), hex(s2))
14    print("共享 k =", hex(k_shared))
15
16    # 已知 k: 直接恢复
17    d_rec1 = recover_d_from_k(r1, s1, k_shared)
18    print("已知 k 时恢复 d =", hex(d_rec1), " 成功?", d_rec1 == dA)
19
20    # 若未知 k, 但有两签名: 用两签名推导
21    # 推导: (s1 - s2) * (1 + d) = (r2 - r1) * d
22    # => d * (r2 - r1 - s1 + s2) = s1 - s2
23    numer = (s1 - s2) % n
24    denom = (r2 - r1 - s1 + s2) % n
25    try:
26        d_rec2 = (numer * inv_mod(denom, n)) % n
27        print("未知 k 时用两签名恢复 d =", hex(d_rec2), " 成功?", d_rec2 ==
            dA)
28    except ZeroDivisionError:
29        print("系数不可逆, 无法用该公式恢复 (极少数情况)")

```

运行结果如下所示:

```

== Poc: k 重用 两个不同消息 恢复私钥 (gmsl SM3 版本) ==
签名1 (r1,s1) = 0x70e94d99c11302af893e4fc6d437407678d7338d1e16cef480c325c0d58f2 0xefef1c75905b4b5d3be4b2e0bd9eac4e585cb61d15e5f0cb79b10b270e648c
9c9
签名2 (r2,s2) = 0xb30762e53ce4152baa541fc222b3e93d866a8f898933d1cc4d8ba4ef6a12e83 0x3a473b6a9999f0414d039b0c801d1edb2a07e79da284e7b9ecccfd22104b
639
共享 k = 0x20b9a7ea46cc82e8b1a720fd89dc5d57c2b7f7d16b0cf9e9add1191801e08aa1ee
已知 k 时恢复 d = 0x4eb5fd7aea7d37049ee45c71c0bb87515dc4d4608af3f23960218999e758fd 成功? True
未知 k 时用两签名恢复 d = 0x4eb5fd7aea7d37049ee45c71c0bb87515dc4d4608af3f23960218999e758fd 成功? True

```

图 5 误用 2

可以看到也能恢复私钥成功。

### 3.3 SM2 signature: reusing k by different users

详细推导如下所示:

- 预计算:

$$Z_A = H_{256}(ENTL_A || ID_A || a || b || x_G || y_G || x_A || y_A)$$

• 密钥生成:

$$P_A = d_A \cdot G, \text{ 阶为 } n$$

• 签名过程  $(Z_A, M)$ :

$$\text{Sign}_{d_A}(M, Z_A) \rightarrow (r, s)$$

- 设置  $M = Z_A || M$
- 计算  $e = H_v(M)$
- 随机选择  $k \in Z_n^*$ , 计算  $kG = (x_1, y_1)$
- 计算  $r = (e + x_1) \bmod n$
- 计算  $s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A)) \bmod n$
- 签名为  $(r, s)$

• 验证过程:

- 计算  $Z_A = H_{256}(ENTL_A || ID_A || a || b || x_G || y_G || x_A || y_A)$
- 设置  $M = Z_A || M$ , 计算  $e = H_v(M)$
- 计算  $t = (r + s) \bmod n$
- 计算  $(x_1, y_1) = sG + tP_A$
- 计算  $R = (e + x_1) \bmod n$ , 验证  $R = r$

• Alice 签名消息  $M_1$ :

- 随机选择  $k \in [1, n - 1]$ , 计算  $kG = (x, y)$
- 计算  $r_1 = (Hash(Z_A || M_1) + x) \bmod n$
- 计算  $s_1 = ((1 + d_A)^{-1} \cdot (k - r_1 \cdot d_A)) \bmod n$

• Bob 签名消息  $M_2$  (重用  $k$ ):

- 计算  $kG = (x, y)$
- 计算  $r_2 = (Hash(Z_B || M_2) + x) \bmod n$
- 计算  $s_2 = ((1 + d_B)^{-1} \cdot (k - r_2 \cdot d_B)) \bmod n$

• 密钥推导:

- Alice 可推导 Bob 私钥:

$$d_B = \frac{k - s_2}{s_2 + r_2} \bmod n$$

– Bob 可推导 Alice 私钥:

$$d_A = \frac{k - s_1}{s_1 + r_1} \mod n$$

运行结果如下所示, 可以看到恢复成功:

```
== PoC: 不同用户重用同一 k 恢复各自私钥 ==
[用户A] 原私钥 = 0x2e62ffba02143f3e99b88b352e731efdefdf5c9d0b5a1a228c73a0c5994898bb
恢复 = 0x2e62ffba02143f3e99b88b352e731efdefdf5c9d0b5a1a228c73a0c5994898bb 成功? True
[用户B] 原私钥 = 0x991c286c5e8b950b6e21c52e7798e9b15e92f17456b66dc4bf7076b4e024f5d
恢复 = 0x991c286c5e8b950b6e21c52e7798e9b15e92f17456b66dc4bf7076b4e024f5d 成功? True
(base) zhangchi@zhangchi-virtual-machine:~/桌面$
```

图 6 误用 3

### 3.4 SM2 signature: same d and k with ECDSA

详细推导如下:

• ECDSA 签名关系:

$$s_1 = k^{-1}(e_1 + dr_1) \mod n \implies k = s_1^{-1}(e_1 + dr_1) \mod n$$

• SM2 签名关系:

$$s_2 = (1 + d)^{-1}(k - dr_2) \mod n \implies k = s_2(1 + d) + dr_2 \mod n$$

令两个关于  $k$  的表达式相等:

$$s_1^{-1}(e_1 + dr_1) = s_2(1 + d) + dr_2 \mod n$$

两边乘以  $s_1$  消去倒数:

$$e_1 + dr_1 = s_1s_2 + s_1s_2d + s_1r_2d \mod n$$

整理含  $d$  的项:

$$e_1 - s_1s_2 = d(s_1s_2 + s_1r_2 - r_1) \mod n$$

最终解得私钥  $d$ :

$$d = (e_1 - s_1 s_2) \cdot (s_1 s_2 + s_1 r_2 - r_1)^{-1} \mod n$$

相应代码如下所示:

```

1  # ----- 跨算法攻击恢复私钥 -----
2  def recover_d_from_ecdsa_sm2(r1, s1, e1, r2, s2):
3      # 根据公式:
4      #  $s_1^{-1} \cdot (e_1 + d r_1) = s_2 \cdot (1 + d) + d r_2$ 
5      # 两边乘  $s_1$ :
6      #  $e_1 + d r_1 = s_1 s_2 + s_1 s_2 d + s_1 r_2 d$ 
7      # 整理:
8      #  $e_1 - s_1 s_2 = d (s_1 s_2 + s_1 r_2 - r_1)$ 
9      #  $d = (e_1 - s_1 s_2) \cdot \text{inv}(s_1 s_2 + s_1 r_2 - r_1) \mod n$ 
10     numerator = (e1 - s1 * s2) % n
11     denominator = (s1 * s2 + s1 * r2 - r1) % n
12     denom_inv = inv_mod(denominator, n)
13     d = (numerator * denom_inv) % n
14     return d
15
16 # ----- PoC 演示: 跨算法同  $d$  和  $k$  导致私钥泄露 -----
17 def demo_cross_algorithm_attack():
18     print("\n== PoC: 跨算法同  $d$  和  $k$  导致私钥泄露 ==")
19     # 用户密钥和公钥
20     d, P = keygen()
21     print(f"[用户] 私钥 d = {hex(d)}")
22
23     # 共享随机数  $k$ 
24     k = secrets.randbelow(n - 1) + 1
25     print(f"[用户] 共享随机数 k = {hex(k)}")
26
27     # 消息
28     M_ecdsa = b"Message signed by ECDSA"
29     M_sm2 = b"Message signed by SM2"
30
31     # ECDSA 签名
32     r1, s1, e1 = sign_ecdsa(d, M_ecdsa, k)
33     print(f"ECDSA 签名: r1={hex(r1)}, s1={hex(s1)}, e1={hex(e1)}")
34
35     # SM2 签名
36     ID_A = b'User-ID-12345678'

```

```

37     ZA = compute_ZA(ID_A, P)
38     to_sign_sm2 = ZA + M_sm2
39     r2, s2, k2, e2 = sign_sm2(d, to_sign_sm2, k=k)
40     print(f"SM2 签名: r2={hex(r2)}, s2={hex(s2)}, e2={hex(e2)}")
41
42     # 恢复私钥
43     d_rec = recover_d_from_ecdsa_sm2(r1, s1, e1, r2, s2)
44     print(f"恢复的私钥 d = {hex(d_rec)}")
45     print("恢复成功?", d_rec == d)

```

运行结果如下所示，成功得出私钥：

```

== PoC: 跨算法同 d 和 k 导致私钥泄露 ==
[用户] 私钥 d = 0xbdcfbfcef0e8fd4cdcfa75e3b1a0e1642b645e05fc7d5fd84e60fde7af6defd
[用户] 共享随机数 k = 0x8ef69cb7367707e31d86362316824db435fb106b14ae39db99b5b11ea54f4e84
ECDSA 签名: r1=0x7f5d7db7e9d24910cca8fe1f633c44d3a693bf778761200ea18fdce53c91ae82, s1=0x
e6f0d8fba25657aa7aec7b0266fe17479e6f034ffe8a126477fae85cd01206, e1=0x18d89ed637c14df55
c0754dd1de1a10d07b77595d609dc675a30f088b7623c4
SM2 签名: r2=0x1daa507911d14a0c0cbd7ee93a46d104be6a0bddd5e22fb9cce0dcfc581abd09, s2=0xda
33b23f4cb3579bf81e1b4831358f5b9e9028aa2b2b2997b21652bb2f1d75a7, e2=0x9e4cd2c027ff00fb401
480c9d70a8c3089da2bd1704714d67f0cf420555e4faa
恢复的私钥 d = 0xbdcfbfcef0e8fd4cdcfa75e3b1a0e1642b645e05fc7d5fd84e60fde7af6defd
恢复成功? True
(base) zhangchi@zhangchi-virtual-machine:~/桌面$

```

图 7 误用 4

## 4 SM2 伪造中本聪签名

我们在不获取私钥的情况下伪装他人的签名：

核心思路：

伪造的是一个消息哈希值  $e$  + 签名  $(r, s)$  三元组，满足标准 ECDSA/SM2 验签公式：

- 伪造的  $(r, s, e)$  不是针对真实消息的签名，而是构造的满足验签方程的“伪造消息哈希 + 签名”组合
- 即伪造出一组满足验签数学关系的假“消息摘要”与对应签名对

具体实现方法：

- 不计算真实消息的哈希  $e = H(M)$
- 随机选择两个数  $u, v$ ，利用公钥和基点构造点：

$$R = uG + vP$$

- 基于  $R$  的横坐标构造  $r$

- 通过数学推导，利用  $u, v$  计算出对应的  $s$  和  $e$ ，使验签方程成立
- 这样伪造的签名对  $(r, s)$  和“消息哈希”在数学上是合法的

数学推导如下：

### 1. 随机数选择与初始计算：

- 选择随机数  $u, v \in [1, n-1]$
- 计算点：

$$R = uG + vP = (x_r, y_r)$$

### 2. 伪造签名参数定义：

$$r = x_r \mod n$$

$$s = r \cdot v^{-1} \mod n$$

$$e = r \cdot u \cdot v^{-1} \mod n$$

### 3. 验证伪造签名的正确性：

- 计算逆元：

$$w = s^{-1} = (r \cdot v^{-1})^{-1} = v \cdot r^{-1} \mod n$$

- 计算中间量：

$$u_1 = e \cdot w = (r \cdot u \cdot v^{-1}) \cdot (v \cdot r^{-1}) = u \mod n$$

$$u_2 = r \cdot w = r \cdot v \cdot r^{-1} = v \mod n$$

- 验签点计算：

$$R' = u_1G + u_2P = uG + vP = R$$

- 最终验证：

$$R'_x \mod n = r \quad \text{满足验签条件}$$

相应代码如下所示：

```
1 # ----- 伪造ECDSA/SM2签名 -----
2 def forge_signature(P):
3     """
```

```

4     无私钥伪造签名
5
6     选择随机 u, v, 计算:
7         R = uG + vP
8         r = R.x mod n
9         s = r * v-1 mod n
10        e = r * u * v-1 mod n
11
12    返回 (r, s, e)
13    """
14    while True:
15        u = secrets.randbelow(n - 1) + 1
16        v = secrets.randbelow(n - 1) + 1
17
18        R = point_add(scalar_mul(u, (Gx, Gy)), scalar_mul(v, P))
19        if R is 0:
20            continue
21        r = R[0] % n
22        if r == 0:
23            continue
24        v_inv = inv_mod(v, n)
25        s = (r * v_inv) % n
26        if s == 0:
27            continue
28        e = (r * u * v_inv) % n
29        return (r, s, e)

```

最终，我们可以成功伪造签名：

```

(base) zhongchi@zhongchi-virtual-machine:~/桌面$ python3 SM2c.py
== SM2/ECDSA伪造签名示例 ==
生成消息哈希: 私钥 d = 0xdac278e2b385f590df8be1b43ee159731915ebd6a1d2a48a049267a8585d
公钥 P = (0xb80b6978f61560f830e2df99141472b2423bc01ac5b4164db299556fa4229d, 0x82a029a5e3404558a35bb25320ff4f9a80bf04db17c0b705f14dde7f3135a033)
伪造签名结果:
r = 0xdf9d27cc346d65553441d050ed8027aede3f6a80eff34c017cc67f95769fb37
s = 0xb7b60c33345300568d4a3e814c83ceffdefdb199eb389ef09f7d241632b95f9c
消息哈希 e = 0xf02af2a90402087e0c5f0abb6435105761ad727e3eb7111e080648ea01d06d7e
伪造签名验证结果: 成功
(base) zhongchi@zhongchi-virtual-machine:~/桌面$

```

图 8 伪造成功

## 5 参考链接

### 1.CSDN 帖子《密码学中的 SM2》