



Project 5: SM2 的软件实现优化

学院: 网络空间安全学院
专业: 密码科学与技术
姓名: 李双平
学号: 202200180026

2025 年 8 月 15 日

目录

1 SM2 基本实现	3
1.1 实现原理	3
1.2 代码解释	3
1.3 实验结果	8
2 SM2 优化实现（固定窗口）	9
2.1 优化原理	9
2.2 代码解释	9
2.3 实验结果	10
3 SM2 优化实现（扩展欧几里得）	11
3.1 优化原理	11
3.2 代码解释	11
3.3 实验结果	11
4 SM2 优化实现（公钥压缩）	12
4.1 优化原理	12
4.2 代码解释	12
4.3 实验结果	14
5 PoC1：已知随机数 k 恢复私钥	14
5.1 背景	14
5.2 推导	14
5.3 攻击步骤	15
5.4 验证代码	15
5.5 实验结果	16
6 PoC2：同一 k 签两条消息恢复私钥	16
6.1 背景	16
6.2 推导	17
6.3 攻击步骤	17
6.4 验证代码	17
6.5 实验结果	18
7 PoC3：不同用户使用相同 k	19
7.1 背景	19
7.2 推导	19
7.3 验证代码	20
7.4 实验结果	21

8 PoC4: SM2 与 ECDSA 使用相同 k	21
8.1 背景	21
8.2 推导	21
8.3 攻击步骤	22
8.4 验证代码	22
8.5 实验结果	23
9 伪造中本聪数字签名	23
9.1 实验原理	23
9.2 代码解释	24
9.3 实验结果	26

1 SM2 基本实现

1.1 实现原理

SM2 算法是基于椭圆曲线密码学 (Elliptic Curve Cryptography, ECC) 的一种公钥密码算法，使用国密推荐的 sm2p256v1 曲线。该曲线定义在素域 \mathbb{F}_p 上，参数 (p, a, b, G, n) 均由标准规定，其中 G 是基点， n 是基点的阶。SM2 加密流程主要包括：

- 使用接收方的公钥 P 和随机数 k 生成临时点 $C_1 = kG$;
 - 计算共享点 $S = kP$, 将其坐标输入密钥派生函数 KDF 生成对称密钥流;
 - 将明文与密钥流异或得到 C_2 , 并计算哈希 C_3 用于完整性校验;
 - 输出密文 $C_1||C_3||C_2$ 。

解密流程则反向进行：用私钥 d 计算 $S = dC_1$ 得到密钥流恢复明文，并通过 C_3 验证完整性。SM2 在实现上依赖于椭圆曲线的点加法、倍点、标量乘等运算，以及 SM3 哈希函数和 KDF 派生函数。

1.2 代码解释

1. 椭圆曲线参数与模逆运算：定义 SM2 使用的椭圆曲线 sm2p256v1 的域参数及求逆函数。

```
1 from math import ceil
2 from typing import Tuple
3 import os
4 import struct
5
6 p = int("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000FFFFFFFFFFFFFF")
7     ", 16)
8 a = int("FFFFFFFFFFFFFFFFFFFFFFFFF000000000FFFFFFFFFFFFFFFC")
9     ", 16)
10 b = int("28E9FA9E9D9F5E344D5A9E4BCF6509A7F39789F515AB8F92DDBCBD414D940E93")
11     ", 16)
12 n = int("FFFFFFFFFFFFFFFFFFF7203DF6B21C6052B53BBF40939D54123")
13     ", 16)
14 Gx = int("32
15     C4AE2C1F1981195F9904466A39C9948FE30BBFF2660BE1715A4589334C74C7", 16)
16 Gy = int("BC3736A2F4F6779C59BDCEE36B692153D0A9877CC62A474002DF32E52139F0A0",
17     16)
18
19 def mod_inv(x: int, m: int = p) -> int:
20     x %= m
21     if x == 0:
22         raise ZeroDivisionError("inverse of 0")
23     return pow(x, m - 2, m)
```

说明：该部分首先定义了 SM2 椭圆曲线的素域 p 、曲线参数 a, b 、基点 G 的坐标、基点阶 n 。这些参数完全遵循 GM/T 0003 标准。随后实现了模逆运算 `mod_inv`，用于在有限域 \mathbb{F}_p 中计算 $x^{-1} \bmod m$ 。由于 p 是素数，可以直接利用费马小定理 $x^{p-2} \bmod p$ 进行快速求逆。这一函数在椭圆曲线加法、倍点等操作中是核心数学运算。

2. 椭圆曲线点运算：实现判断点是否在曲线上、点取负、点加法、标量乘法等基础操作。

```
1  def is_on_curve(P: Tuple[int,int]) -> bool:
2      if P is None:
3          return True
4      x, y = P
5      return (y * y - (x * x * x + a * x + b)) % p == 0
6
7  def point_neg(P: Tuple[int,int]):
8      if P is None:
9          return None
10     x, y = P
11     return (x, (-y) % p)
12
13 def point_add(P: Tuple[int,int], Q: Tuple[int,int]):
14     if P is None:
15         return Q
16     if Q is None:
17         return P
18     x1, y1 = P
19     x2, y2 = Q
20     if x1 == x2 and (y1 + y2) % p == 0:
21         return None
22     if P == Q:
23         lam = (3 * x1 * x1 + a) * mod_inv(2 * y1, p) % p
24     else:
25         lam = (y2 - y1) * mod_inv(x2 - x1, p) % p
26     x3 = (lam * lam - x1 - x2) % p
27     y3 = (lam * (x1 - x3) - y1) % p
28     return (x3, y3)
29
30 def scalar_mult(k: int, P: Tuple[int,int]):
31     if k % n == 0 or P is None:
32         return None
33     if k < 0:
34         return scalar_mult(-k, point_neg(P))
35     result = None
36     addend = P
37     while k:
```

```

38         if k & 1:
39             result = point_add(result, addend)
40             addend = point_add(addend, addend)
41             k >>= 1
42     return result

```

说明: `is_on_curve` 用于验证一个点是否满足曲线方程 $y^2 \equiv x^3 + ax + b \pmod{p}$ 。`point_neg` 实现有限域内的点取负，保持横坐标不变、纵坐标取模 p 的相反数。`point_add` 实现点加法和倍点公式，通过计算斜率 λ 并得出新点坐标。`scalar_mult` 则是双倍加法算法，按二进制展开逐位计算 kP 。这些函数构成 SM2 中加密、解密、签名的核心椭圆曲线算术运算。

3. SM3 哈希函数实现：实现国密 SM3 摘要算法。

```

1  IV = [
2      0x7380166F, 0x4914B2B9, 0x172442D7, 0xDA8A0600,
3      0xA96F30BC, 0x163138AA, 0xE38DEE4D, 0xB0FB0E4E
4  ]
5  T_j = [0x79CC4519 if j <= 15 else 0x7A879D8A for j in range(64)]
6
7  def rotl(x, n):
8      n &= 31
9      x &= 0xFFFFFFFF
10     if n == 0:
11         return x
12     return ((x << n) & 0xFFFFFFFF) | (x >> (32 - n))
13
14 def ff_j(x, y, z, j):
15     return (x ^ y ^ z) if j <= 15 else ((x & y) | (x & z) | (y & z))
16
17 def gg_j(x, y, z, j):
18     return (x ^ y ^ z) if j <= 15 else ((x & y) | ((~x) & z))
19
20 def p0(x):
21     return x ^ rotl(x, 9) ^ rotl(x, 17)
22
23 def p1(x):
24     return x ^ rotl(x, 15) ^ rotl(x, 23)
25
26 def sm3_hash(msg: bytes) -> bytes:
27     m = bytearray(msg)
28     l = len(m) * 8
29     m.append(0x80)
30     while ((len(m) * 8) % 512) != 448:
31         m.append(0x00)

```

```

32     m += struct.pack(">Q", 1)
33     V = IV[:]
34     for i in range(0, len(m), 64):
35         B = m[i:i+64]
36         W = [0]*68
37         W1 = [0]*64
38         for j in range(16):
39             W[j] = struct.unpack(">I", bytes(B[4*j:4*j+4]))[0]
40         for j in range(16, 68):
41             W[j] = (p1(W[j-16] ^ W[j-9] ^ rotl(W[j-3], 15)) ^ rotl(W[j-13],
42                             7) ^ W[j-6]) & 0xFFFFFFFF
43         for j in range(64):
44             W1[j] = W[j] ^ W[j+4]
45         A,Bb,C,D,E,F,G,H = V
46         for j in range(64):
47             SS1 = rotl(((rotl(A,12) + E + rotl(T_j[j], j)) & 0xFFFFFFFF),
48                         7)
49             SS2 = SS1 ^ rotl(A,12)
50             TT1 = (ff_j(A,Bb,C,j) + D + SS2 + W1[j]) & 0xFFFFFFFF
51             TT2 = (gg_j(E,F,G,j) + H + SS1 + W[j]) & 0xFFFFFFFF
52             D, C, Bb, A = C, rotl(Bb, 9), A, TT1
53             H, G, F, E = G, rotl(F, 19), E, p0(TT2)
54             V = [x ^ y for x,y in zip(V, [A,Bb,C,D,E,F,G,H])]
55     return b''.join(struct.pack(">I", x) for x in V)

```

说明: 此部分实现了 SM3 哈希算法的完整流程，包括常量初始化、消息填充、消息扩展以及 64 轮压缩函数。`rotl` 实现 32 位循环左移，`p0`、`p1` 是 SM3 的置换函数，`ff_j`、`gg_j` 是轮函数，分别用于不同的迭代区间。压缩函数内部维护 8 个 32 位寄存器，经过 64 轮运算更新链接变量 `V`。SM3 在 SM2 中用于生成密钥派生函数 KDF 的输入以及签名的哈希摘要。

4. 密钥派生函数 KDF 与整数字节转换：

```

1 def kdf(z: bytes, klen: int) -> bytes:
2     ct = 1
3     digest = b""
4     for i in range(ceil(klen / 32)):
5         msg = z + struct.pack(">I", ct)
6         digest += sm3_hash(msg)
7         ct += 1
8     return digest[:klen]
9
10 def int_to_bytes(x: int) -> bytes:
11     return x.to_bytes(32, 'big')
12

```

```

13 def bytes_to_int(b: bytes) -> int:
14     return int.from_bytes(b, 'big')

```

说明: kdf 实现 SM2 的密钥派生函数, 根据输入 Z 和所需长度 $klen$ 迭代调用 SM3 生成足够的密钥流并截取到目标长度。计数器 ct 从 1 递增并按大端序拼接到输入。`int_to_bytes` 和 `bytes_to_int` 提供整数与 32 字节大端序列的转换, 用于在有限域坐标和字节流之间切换, 这是密钥交换、加密和解密时不可缺少的辅助操作。

5. 密钥对生成:

```

1 G = (Gx, Gy)
2
3 def gen_keypair() -> Tuple[int, Tuple[int,int]]:
4     while True:
5         d = int.from_bytes(os.urandom(32), 'big') % n
6         if 1 <= d <= n-2:
7             break
8     P = scalar_mult(d, G)
9     return d, P

```

说明: 基点 G 为曲线的生成元。私钥 d 通过系统安全随机数生成器 `os.urandom` 生成 256 位随机数并取模 n , 确保 $1 \leq d \leq n - 2$ 。公钥 P 由标量乘法 $P = dG$ 得到。这一过程保证了密钥的安全性和不可预测性, 是 SM2 加密、签名等操作的基础。

6. SM2 加密流程:

```

1 def sm2_encrypt(P: Tuple[int,int], msg: bytes) -> bytes:
2     mlen = len(msg)
3     while True:
4         k = int.from_bytes(os.urandom(32), 'big') % n
5         if k == 0:
6             continue
7         C1 = scalar_mult(k, G)
8         x1, y1 = C1
9         S = scalar_mult(k, P)
10        if S is None:
11            continue
12        x2, y2 = S
13        t = kdf(int_to_bytes(x2) + int_to_bytes(y2), mlen)
14        if any(t):
15            C2 = bytes(a ^ b for a,b in zip(msg, t))
16            C3 = sm3_hash(int_to_bytes(x2) + msg + int_to_bytes(y2))
17            c1_bytes = b'\x04' + int_to_bytes(x1) + int_to_bytes(y1)
18            return c1_bytes + C3 + C2

```

说明：SM2 加密采用随机数 k 生成临时点 $C_1 = kG$ 并与接收方公钥 P 计算共享点 $S = kP$ 。取 S 的 x 、 y 坐标经 KDF 生成密钥流 t ，并将明文与 t 异或得到 C_2 。同时计算哈希 $C_3 = SM3(x_2||M||y_2)$ 用于完整性校验。最终密文为 $C_1||C_3||C_2$ ，其中 C_1 按未压缩点格式输出（前缀 0x04）。

7. SM2 解密流程：

```

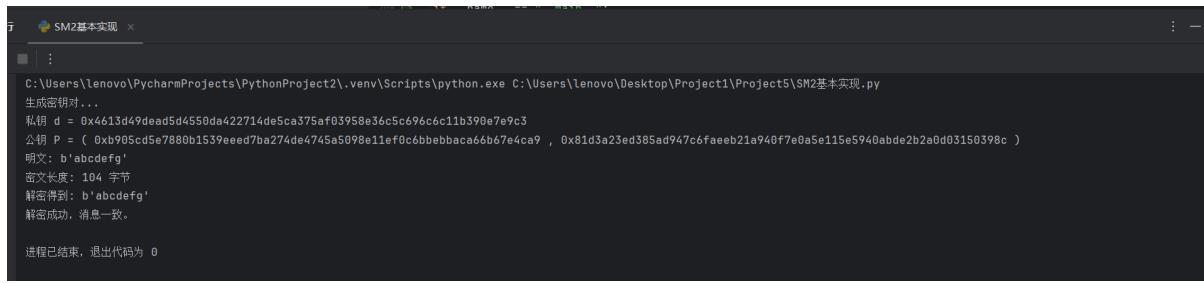
1  def sm2_decrypt(d: int, cipher: bytes) -> bytes:
2      if len(cipher) < 1 + 64 + 32:
3          raise ValueError("cipher too short")
4      if cipher[0] != 0x04:
5          raise ValueError("only uncompressed point supported")
6      c1_bytes = cipher[:1+64]
7      C3 = cipher[1+64:1+64+32]
8      C2 = cipher[1+64+32:]
9      x1 = bytes_to_int(c1_bytes[1:33])
10     y1 = bytes_to_int(c1_bytes[33:65])
11     C1 = (x1, y1)
12     if not is_on_curve(C1):
13         raise ValueError("C1 not on curve")
14     S = scalar_mult(d, C1)
15     if S is None:
16         raise ValueError("S is infinite (invalid)")
17     x2, y2 = S
18     mlen = len(C2)
19     t = kdf(int_to_bytes(x2) + int_to_bytes(y2), mlen)
20     if all(b == 0 for b in t):
21         raise ValueError("kdf result is zero -> decryption fail")
22     M = bytes(a ^ b for a,b in zip(C2, t))
23     u = sm3_hash(int_to_bytes(x2) + M + int_to_bytes(y2))
24     if u != C3:
25         raise ValueError("C3 mismatch -> decryption fail (integrity)")
26     return M

```

说明：解密时解析密文得到 C_1, C_3, C_2 ，检查 C_1 是否在曲线上并计算 $S = dC_1$ 。通过 S 的坐标调用 KDF 得到密钥流 t ，与 C_2 异或恢复明文 M 。再计算哈希 u 验证与 C_3 是否一致，确保数据未被篡改。若验证失败则抛出异常，保证 SM2 解密的安全性与完整性。

1.3 实验结果

实验结果如下：



```
C:\Users\lenovo\PycharmProjects\PythonProject2\.venv\Scripts\python.exe C:\Users\lenovo\Desktop\Project1\Project5\SM2基本实现.py
生成密钥对...
私钥 d = 0x4613d49dead5d4550da422714de5ca375af03958e36c5c696c6c11b390e7e9c3
公钥 P = ( 0xb905cd5e78801539eed7ba274de4745a5098e11ef0c6b6ebbac66b67e4ca9 , 0x81d3a23ed385ad947c6faeeb21a940f7e0a5e115e5940abde2b2a0d03150398c )
明文: b'abcdefg'
密文长度: 104 字节
解密得到: b'abcdefg'
解密成功, 消息一致。

进程已结束, 退出代码为 0
```

图 1: 实验结果一

2 SM2 优化实现（固定窗口）

2.1 优化原理

在 SM2 椭圆曲线运算中，标量乘法 kP 是最耗时的操作之一，其复杂度主要由点加与倍点运算决定。传统的“二进制方法”逐位扫描标量 k ，每一位都进行一次倍点运算，并在为 1 时额外执行一次点加。固定窗口（Fixed Window）算法通过将标量的二进制表示分成固定长度 w 的窗口来减少运算次数：在预处理阶段先计算出 $P, 2P, \dots, (2^w - 1)P$ 并存入查找表，之后在运算过程中每次读取一个窗口的值，通过多次倍点加一次查表来得到结果。这样可以减少标量乘法中的点加次数，同时更好地利用缓存局部性，提高整体性能。

2.2 代码解释

1. 固定窗口标量乘法函数 `scalar_mult`:

```

1  def scalar_mult(k: int, P: Tuple[int,int], window_size: int = 4):
2      if k % n == 0 or P is None:
3          return None
4      if k < 0:
5          return scalar_mult(-k, point_neg(P), window_size)
6
7      k = k % n
8      w = window_size
9      if w < 1:
10         result = None
11         addend = P
12         while k:
13             if k & 1:
14                 result = point_add(result, addend)
15                 addend = point_add(addend, addend)
16             k >>= 1
17         return result
18
19     table_size = (1 << w)
```

```

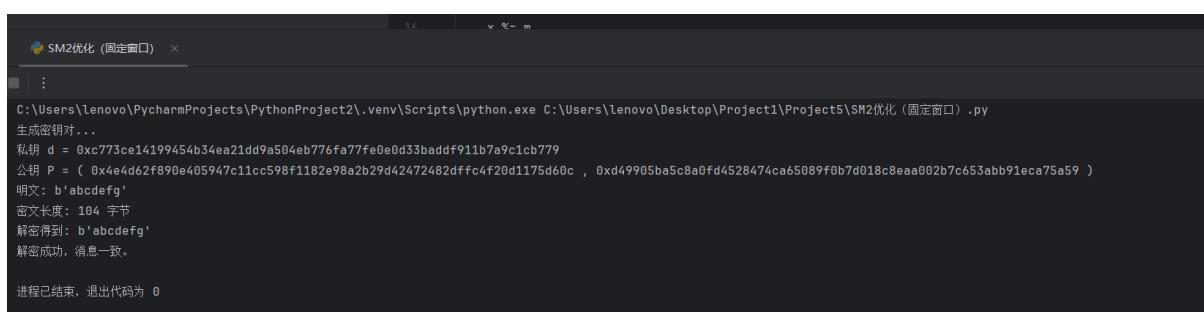
20     table = [None] * table_size
21     table[1] = P
22     for i in range(2, table_size):
23         table[i] = point_add(table[i-1], P)
24
25     kb = bin(k)[2:]
26     i = 0
27     result = None
28     L = len(kb)
29     while i < L:
30         if kb[i] == '0':
31             result = point_add(result, result)
32             i += 1
33         else:
34             l = min(w, L - i)
35             while l > 1 and kb[i:i+l][0] == '0':
36                 l -= 1
37             val = int(kb[i:i+l], 2)
38             for _ in range(l):
39                 result = point_add(result, result)
40             if val != 0:
41                 result = point_add(result, table[val])
42             i += 1
43     return result

```

说明: 该函数首先处理边界条件 ($k = 0$ 或点为空) 及负标量的取反情况。若窗口大小 $w < 1$, 退化为普通的二进制倍点加法。核心优化是构建大小为 2^w 的查找表 `table`, 预存从 P 到 $(2^w - 1)P$ 的所有倍点值。在计算时, 逐位扫描标量二进制串 `kb`, 遇到非零窗口时一次性完成 l 次倍点操作, 并查表加上对应倍点值, 从而减少了在线计算的点加次数。窗口大小的选择影响性能与内存占用, $w = 4$ 时通常能取得较好平衡。

2.3 实验结果

实验结果如下:



```

$ SM2优化 (固定窗口) ...
:
C:\Users\lenovo\PycharmProjects\PythonProject2\.venv\Scripts\python.exe C:\Users\lenovo\Desktop\Project1\Project5\SM2优化 (固定窗口).py
生成密钥对...
私钥 d = 0xc773ce1419945b34ea21dd9a504eb776fa77fe0e0d33baddf911b7a9c1cb779
公钥 P = ( 0xe4ed62f890e405947c11cc598f1182e98a2b29d42472482dffca4f20d1175d60c , 0xd49905ba5c8a0fd4528474ca65089f0b7d018c8eaa002b7c653abb91eca75a59 )
明文: b'abcdefg'
密文长度: 104 字节
解密得到: b'abcdefg'
解密成功, 消息一致。

进程已结束, 退出代码为 0

```

图 2: 实验结果二

3 SM2 优化实现（扩展欧几里得）

3.1 优化原理

在 SM2 椭圆曲线运算中，模逆运算 $\text{mod_inv}(x, m)$ 是点加与倍点运算中的核心步骤之一，其计算性能直接影响整个加密解密过程。常见的模逆计算方法是基于费马小定理 $x^{-1} \equiv x^{m-2} \pmod{m}$ ，实现上使用快速模幂算法，但需要进行 $O(\log m)$ 次模乘，开销较大。扩展欧几里得算法通过在求解 $\gcd(a, m)$ 的过程中迭代更新系数，可一次性得到 a 在模 m 下的逆元。该方法的时间复杂度为 $O(\log m)$ ，但常数项较小，且仅需整数加减与乘除，不依赖大规模模幂运算，因此在椭圆曲线标量乘法中能显著减少模逆的运行时间。

3.2 代码解释

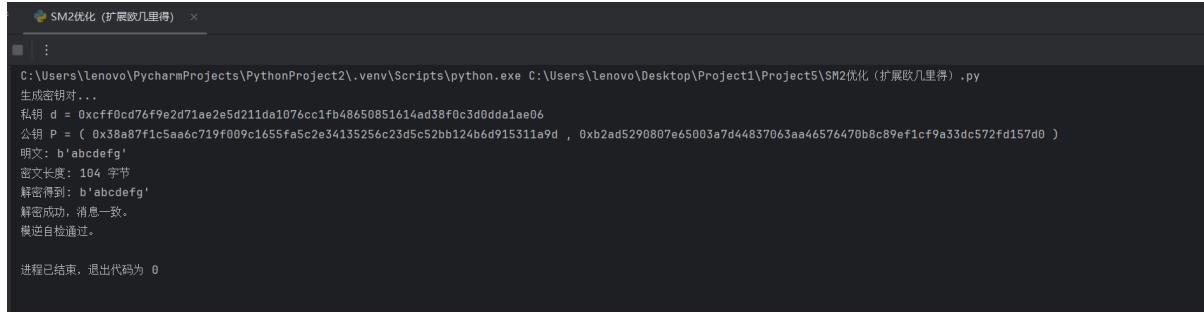
1. 扩展欧几里得模逆函数 `mod_inv`:

```
1 def mod_inv(x: int, m: int = p) -> int:
2     if m <= 0:
3         raise ValueError("modulus must be positive")
4     x = x % m
5     if x == 0:
6         raise ZeroDivisionError("inverse of 0")
7
8     a = x
9     b = m
10    lm, hm = 1, 0
11    low, high = a, b
12
13    while low > 1:
14        q = high // low
15        high, low = low, high - q * low
16        hm, lm = lm, hm - q * lm
17
18    if low != 1:
19        raise ZeroDivisionError("inverse does not exist")
20    inv = lm % m
21    return inv
```

说明：函数首先检查模数的有效性并处理 $x = 0$ 的特殊情况。变量 `lm` 与 `hm` 分别保存当前和前一轮的 Bezout 系数，`low` 与 `high` 保存当前余数和前一余数。在主循环中通过整数除法求商 q ，并更新余数与系数，直至余数降为 1，此时的 `lm` 即为 x 的模逆。该算法避免了模幂运算，使用加减与乘除即可完成，常数时间开销显著低于基于费马小定理的实现。

3.3 实验结果

实验结果如下：



```
C:\Users\lenovo\PycharmProjects\PythonProject2\.venv\Scripts\python.exe C:\Users\lenovo\Desktop\Project1\Project5\SM2优化（扩展欧几里得）.py
生成密钥对...
私钥 d = 0x... (long hex value)
公钥 P = ( 0x... (long hex value), 0x... (long hex value) )
明文: b'abcdefg'
密文长度: 104 字节
解密得到: b'abcdefg'
解密成功, 消息一致。
模逆自检通过。

进程已结束, 退出代码为 0
```

图 3: 实验结果三

4 SM2 优化实现（公钥压缩）

4.1 优化原理

在 SM2 椭圆曲线密码体制中，公钥是曲线上的一个点 $P = (x, y)$ ，采用非压缩格式时需存储 x 和 y 两个 256 位整数（共 64 字节），再加上 1 字节前缀标识，总长度为 65 字节。在许多应用中，传输或存储空间受限，直接传输完整 y 坐标会造成额外的带宽开销。椭圆曲线方程为 $y^2 \equiv x^3 + ax + b \pmod{p}$ ，因此已知 x 坐标即可通过求模平方根唯一确定 y 的两个可能值 (y 和 $-y \pmod{p}$)，根据奇偶性即可区分出具体值。压缩公钥即保存 x 坐标和 y 的奇偶性 (0x02 表示 y 为偶数，0x03 表示 y 为奇数)，可将长度由 65 字节减少到 33 字节，从而在网络传输和存储中提升效率。恢复公钥时，通过 x 计算右端 $x^3 + ax + b \pmod{p}$ 并取模平方根即可还原 y ，再依据前缀的奇偶标志调整符号。

4.2 代码解释

1. 公钥压缩函数 `point_to_bytes`:

```
1 def point_to_bytes(P: Tuple[int,int], compressed: bool = True) -> bytes:
2     if P is None:
3         raise ValueError("point is None")
4     x, y = P
5     xb = int_to_bytes(x)
6     yb = int_to_bytes(y)
7     if compressed:
8         prefix = b'\x03' if (y & 1) else b'\x02'
9         return prefix + xb
10    else:
11        return b'\x04' + xb + yb
```

说明：该函数将椭圆曲线点序列化为字节串，支持压缩与非压缩两种格式。压缩模式下，仅存储 x 坐标与奇偶标志 (0x02/0x03)，长度为 33 字节；非压缩模式下存储前缀 0x04 以及 x, y 坐标，长度为 65 字节。

2. 模平方根计算 `modular_sqrt`:

```

1 def modular_sqrt(a: int, p_mod: int = p) -> int:
2     a %= p_mod
3     if a == 0:
4         return 0
5     ls = pow(a, (p_mod - 1) // 2, p_mod)
6     if ls == p_mod - 1:
7         return None
8     if p_mod % 4 == 3:
9         y = pow(a, (p_mod + 1) // 4, p_mod)
10        if (y * y) % p_mod != a % p_mod:
11            return None
12        return y
13    raise NotImplementedError("modular_sqrt requires p % 4 == 3 or TS
algorithms implemented")

```

说明: 该函数根据 $p \bmod 4 = 3$ 的性质快速计算模平方根, 主要用于从 x 坐标恢复 y 。首先检查 a 是否为二次剩余 (勒让德符号检验), 然后用指数公式直接求解平方根。

3. 公钥恢复函数 bytes_to_point:

```

1 def bytes_to_point(pubkey_bytes: bytes) -> Tuple[int,int]:
2     prefix = pubkey_bytes[0]
3     if prefix == 0x04:
4         x = bytes_to_int(pubkey_bytes[1:33])
5         y = bytes_to_int(pubkey_bytes[33:65])
6         P = (x, y)
7         if not is_on_curve(P):
8             raise ValueError("point not on curve")
9         return P
10    elif prefix in (0x02, 0x03):
11        x = bytes_to_int(pubkey_bytes[1:33])
12        rhs = (pow(x, 3, p) + (a * x) + b) % p
13        y = modular_sqrt(rhs, p)
14        need_odd = (prefix == 0x03)
15        if (y & 1) != need_odd:
16            y = (-y) % p
17        P = (x, y)
18        if not is_on_curve(P):
19            raise ValueError("recovered point not on curve")
20        return P
21    else:
22        raise ValueError("unsupported pubkey prefix")

```

说明：该函数解析压缩或非压缩格式的公钥字节串，并恢复为 (x, y) 点对象。对于压缩格式，先用椭圆曲线方程计算 y^2 ，再取模平方根得到 y ，最后依据前缀的奇偶标志调整符号以获得唯一解。

4.3 实验结果

实验结果如下：

```
C:\> C:\Users\lenovo\PycharmProjects\PythonProject2\.venv\Scripts\python.exe C:\Users\lenovo\Desktop\Project1\Project5\SM2优化（公钥压缩）.py
生成密钥对...
私钥 d = 0x69831772ae1bc157b8e51271697fcbb5ff4f4e58daff0542502cef5bbd49e15
原公钥 P = ( 0x5377305d13e4336af0592ba1f391368198f76dcec766d64ffdb011ec1a8482c , 0xb60a1a3a425e5e5d0a81d5a2286119fd14c4e2cd91cc68d1aafe42b1ed3ac72 )
非压缩公钥：045377305d13e4336af0592ba1f391368198f76dcec766d64ffdb011ec1a8482c b60a1a3a425e5e5d0a81d5a2286119fd14c4e2cd91cc68d1aafe42b1ed3ac72
压缩公钥：025377305d13e4336af0592ba1f391368198f76dcec766d64ffdb011ec1a8482c
恢复后的公钥 P_rec = ( 0x5377305d13e4336af0592ba1f391368198f76dcec766d64ffdb011ec1a8482c , 0xb60a1a3a425e5e5d0a81d5a2286119fd14c4e2cd91cc68d1aafe42b1ed3ac72 )
压缩/恢复自检通过。
密文长度：104
加密得到：b'abcdefg'
加解密自检通过。
```

图 4: 实验结果四

5 PoC1：已知随机数 k 恢复私钥

5.1 背景

SM2 签名的核心安全性依赖于签名用一次性的随机数 k 。若同一条消息或不同消息的签名过程中， k 被泄露，则攻击者在已知签名 (r, s) 与 k 的情况下，可以显式解出私钥 d 。这属于“签名随机数误用”的典型问题，其修复要点是确保 k 的不可预测性与唯一性。

5.2 推导

SM2 签名流程（标准记号）如下：

$$\begin{aligned} e &= H(M) \bmod n, \quad (x_1, y_1) = k \cdot G, \\ r &= (e + x_1) \bmod n, \\ s &= (1 + d)^{-1} (k - rd) \bmod n, \end{aligned}$$

其中 G 为基点， n 为基点阶， d 为私钥， $H(\cdot)$ 为 SM3。

设已知 (r, s) 与随机数 k ，从 s 的定义出发：

$$s \equiv (1 + d)^{-1} (k - rd) \pmod{n}.$$

两边同乘 $(1 + d)$ ：

$$s(1 + d) \equiv k - rd \pmod{n}.$$

展开整理含 d 的项：

$$s + sd \equiv k - rd \pmod{n} \implies (s + r)d \equiv k - s \pmod{n}.$$

若 $s + r \not\equiv 0 \pmod{n}$ ，则可取逆元并直接解出 d ：

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

因此，只要签名时的 k 泄露（或可被预测），攻击者即可用上式在模 n 上一轮逆元与一轮乘法求出私钥 d 。

5.3 攻击步骤

1. **收集输入**: 获取目标的一对合法 SM2 签名 (r, s) , 并获得该次签名使用的随机数 k 。
2. **代入公式**: 计算 $\text{denom} = (s + r) \bmod n$ 并求其逆元 $\text{inv} = \text{denom}^{-1} \bmod n$ 。
3. **恢复私钥**: 输出 $d \equiv (k - s) \cdot \text{inv} \pmod{n}$ 。
4. **校验**: 用 d 计算公钥 $P' = d \cdot G$, 与目标的公钥 P 比较 (或用 d 对任意消息签名并用 P 验证), 一致则恢复成功。

5.4 验证代码

验证脚本完整实现了 SM2 所需的曲线算术与 SM3 哈希, 并在签名函数中返回签名用到的 k (模拟“ k 泄露”场景)。随后调用 `recover_d(r,s,k)` 使用上式恢复 d , 再通过计算公钥比对验证成功。

1. **`sm2sign_k_returned`**: 接收私钥 d 与消息 `msg`, 可选参数 `forced_k` 用于强制指定 k (模拟泄露场景)。先计算 $e = H(M) \bmod n$, 生成 k 并计算 $kG = (x_1, y_1)$, 得到 $r = (e + x_1) \bmod n$ 。按公式 $s = (1 + d)^{-1}(k - rd) \bmod n$ 计算签名第二分量。返回 (r, s, k) , 使后续攻击能直接利用 k 恢复私钥。

```

1  def sm2sign_k_returned(d: int, msg: bytes, forced_k: int = None) -> Tuple
2      [int, int, int]:
3          e = _int_from_msg(msg)
4          while True:
5              if forced_k is not None:
6                  k = forced_k % n
7              else:
8                  k = int.from_bytes(os.urandom(32), 'big') % n
9                  if k == 0:
10                      if forced_k is not None:
11                          raise RuntimeError("forced k invalid")
12                      continue
13                  x1y1 = scalar_mult(k, G)
14                  if x1y1 is None:
15                      if forced_k is not None:
16                          raise RuntimeError("k resulted in infinity")
17                  continue
18                  x1, y1 = x1y1
19                  r = (e + x1) % n
20                  if r == 0 or r + k == n:
21                      if forced_k is not None:
22                          raise RuntimeError("bad r for forced k")
23                      continue
24                  inv_1pd = mod_inv((1 + d) % n, n)

```

```

24         s = (inv_1pd * (k - r * d)) % n
25         if s == 0:
26             if forced_k is not None:
27                 raise RuntimeError("bad s for forced k")
28             continue
29         return r, s, k

```

2. `recover_d`: 根据推导公式 $d \equiv (k-s)(s+r)^{-1} \pmod{n}$ 恢复私钥。先计算 $\text{denom} = (s+r) \pmod{n}$ 并求逆，若分母为 0 则抛错。返回 d_{rec} ，理论上与真实 d 等价，可通过公钥比对验证。

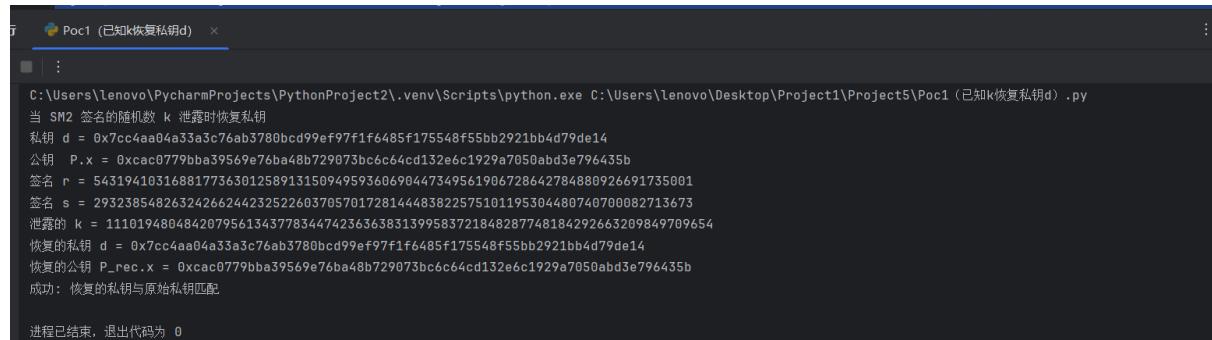
```

1 def recover_d(r: int, s: int, k: int) -> int:
2     denom = (s + r) % n
3     if denom == 0:
4         raise ZeroDivisionError("s + r == 0 (mod n), cannot invert")
5     inv = pow(denom, -1, n)
6     d_rec = ((k - s) % n) * inv % n
7     return d_rec

```

5.5 实验结果

实验结果如下：



```

Poc1 (已知k恢复私钥d) x
C:\Users\lenovo\PycharmProjects\PythonProject2\.venv\Scripts\python.exe C:\Users\lenovo\Desktop\Project1\Project5\Poc1 (已知k恢复私钥d).py
当 SM2 签名的随机数 k 泄露时恢复私钥
私钥 d = 0x7cc4aa04a33a3c76ab3780bcd99ef97f1f6485f175548f55bb2921bb4d79de14
公钥 P.x = 0xcac0779bba39569e76ba48b729073bc6c64cd132e6c1929a7050abd3e796435b
签名 r = 54319410316881773630125891315094959360690447349561906728642784880926691735001
签名 s = 29323854826324266244232522603705701728144483822575101195304480740700082713673
泄露的 k = 11101948048420795613437783447423636383139958372184828774818429263209849709654
恢复的私钥 d = 0x7cc4aa04a33a3c76ab3780bcd99ef97f1f6485f175548f55bb2921bb4d79de14
恢复的公钥 P_rec.x = 0xcac0779bba39569e76ba48b729073bc6c64cd132e6c1929a7050abd3e796435b
成功：恢复的私钥与原始私钥匹配

进程已结束，退出代码为 0

```

图 5: 实验结果一

6 PoC2: 同一 k 签两条消息恢复私钥

6.1 背景

SM2 签名安全性依赖于每次签名所用的随机数 k 必须唯一且不可预测。如果两个不同消息在签名时使用了相同的 k ，攻击者即使不知道 k 的具体值，也可以通过两组签名 (r_1, s_1) 和 (r_2, s_2) 直接恢复私钥 d 。这种漏洞常见于随机数生成器 (RNG) 输出重复、设备固件重用同一随机种子、或者调试测试阶段固定 k 的情况。此类问题的修复方式包括：使用高质量的 CSPRNG、避免跨消息复用 k 、或采用基于 RFC6979 的确定性 k 生成方法并绑定消息摘要。

6.2 推导

对消息 M_1 和 M_2 使用相同 k 签名，记：

$$\begin{aligned} e_1 &= H(M_1) \bmod n, & r_1 &= (e_1 + x_1) \bmod n, \\ e_2 &= H(M_2) \bmod n, & r_2 &= (e_2 + x_1) \bmod n, \\ s_1 &= (1 + d)^{-1}(k - r_1 d) \bmod n, \\ s_2 &= (1 + d)^{-1}(k - r_2 d) \bmod n. \end{aligned}$$

由 s_1 、 s_2 式相减：

$$(s_2 - s_1)(1 + d) \equiv (r_1 - r_2)d \pmod{n}.$$

移项得到：

$$(s_2 - s_1) \equiv [(r_1 - r_2) - (s_1 - s_2)]d \pmod{n}.$$

化简后直接解得：

$$d \equiv (s_2 - s_1) \cdot [(s_1 - s_2) + (r_1 - r_2)]^{-1} \pmod{n}.$$

因此，只要攻击者获得两组不同消息的签名且 k 相同，就能在模 n 上一次逆元 + 一次乘法恢复 d 。

6.3 攻击步骤

1. **收集输入**：获取目标对两条不同消息的合法签名 (r_1, s_1) 与 (r_2, s_2) ，并确认两次签名的 k 相同（通常由漏洞条件保证）。
2. **代入公式**：计算 $\text{denom} = (s_1 - s_2) + (r_1 - r_2) \bmod n$ ，并求其逆元 $\text{inv} = \text{denom}^{-1} \bmod n$ 。
3. **恢复私钥**：输出 $d \equiv (s_2 - s_1) \cdot \text{inv} \pmod{n}$ 。
4. **校验**：用 d 计算 $P' = d \cdot G$ ，与目标公钥比对或用其验证任意新签名。

6.4 验证代码

验证脚本完整实现 SM2 曲线运算与 SM3 哈希，并提供可控的签名函数 `sm2sign_return_k`，通过固定 `forced_k` 模拟“同一 k ”的错误使用。随后用 `recover_d` 按公式恢复私钥并验证。

1. **`sm2sign_return_k`**：输入私钥 d 与消息 `msg`，可选 `forced_k` 强制指定 k （用于模拟重复使用场景）。内部先计算 $e = H(M) \bmod n$ ，再生成或采用指定的 k ，计算 kG 并得到 $r = (e + x_1) \bmod n$ 。按 $s = (1 + d)^{-1}(k - rd) \bmod n$ 得到签名第二分量。返回 (r, s, k) 以便外部检测 k 是否相同并执行 PoC。
2. **`recover_d`**：实现推导公式 $d \equiv (s_2 - s_1) \cdot [(s_1 - s_2) + (r_1 - r_2)]^{-1} \bmod n$ 。先计算分母 `denom`，若为 0 则报错；否则取逆并乘分子 $(s_2 - s_1)$ 得到 `d_rec`。结果与真实 d 等价，可通过公钥比对验证。

```

1
2 def sm2sign_return_k(d: int, msg: bytes, forced_k: int = None) -> Tuple[int,
3     int, int]:
    e = _int_from_msg(msg)

```

```
4     while True:
5         if forced_k is not None:
6             k = forced_k % n
7         else:
8             k = int.from_bytes(os.urandom(32), 'big') % n
9         if k == 0:
10            continue
11         x1y1 = scalar_mult(k, G)
12         if x1y1 is None:
13            continue
14         x1, _ = x1y1
15         r = (e + x1) % n
16         if r == 0 or r + k == n:
17            continue
18         inv_1pd = mod_inv((1 + d) % n, n)
19         s = (inv_1pd * (k - r * d)) % n
20         if s == 0:
21            continue
22         return r, s, k
23
24 def recover_d(r1:int, s1:int, r2:int, s2:int) -> int:
25     num = (s2 - s1) % n
26     denom = ((s1 - s2) + (r1 - r2)) % n
27     if denom == 0:
28         raise ZeroDivisionError("denominator == 0 (cannot invert)")
29     inv = pow(denom, -1, n)
30     d_rec = (num * inv) % n
31     return d_rec
```

6.5 实验结果

实验结果如下：



```

运行 Poc2 (同一 k 签两条消息恢复 d) ×

C:\Users\lenovo\PycharmProjects\PythonProject2\.venv\Scripts\python.exe "C:\Users\lenovo\Desktop\Project1\Project5\Poc2 (同一 k 签两条消息恢复 d)"


SM2 重复使用同一 k 对两条消息签名 -> 恢复私钥
私钥 d = 0xb0c92ae91a708c1ec83638e80a51c77d6d9d20949b9bd82595a63d8ab0489c7f
公钥 P.x = 0xe64316385459f31405712123ffd07d7277ebdbdb85d4c4274f31ba8e0be07
固定的 k = 0x33cc088f252a961f6a513f6f09da7e57
签名1 r1 = 96842048209820647000030758951812875082500263830978236455272976432586742902353
签名1 s1 = 72919250300484753167910664014162689417308654611676555399739309926744231295187
签名2 r2 = 2122652825871147669803323892249016341258327346465550864774019136848295993824
签名2 s2 = 85983527322471453749798728728687535294109002910021271426265160336454996072895
k1 是否等于 k2 (mod n) ? True
恢复的私钥 d = 0xb0c92ae91a708c1ec83638e80a51c77d6d9d20949b9bd82595a63d8ab0489c7f
恢复的公钥 P_rec.x = 0xe64316385459f31405712123ffd07d7277ebdbdb85d4c4274f31ba8e0be07
成功：恢复的私钥与原始私钥匹配

```

图 6: 实验结果二

7 PoC3: 不同用户使用相同 k

7.1 背景

在 SM2 签名中，随机数 k 必须为每个签名独立生成，且不可预测。若两个不同的用户在签名时使用了相同的 k （例如硬件签名设备在不同账户间共享同一 RNG 种子，或多实例部署中 RNG 状态被复制），则已知其中一方的私钥 d_1 与签名 (r_1, s_1) ，攻击者可先恢复 k ，再利用另一方的签名 (r_2, s_2) 直接恢复该用户的私钥 d_2 。这种漏洞的根源在于： k 一旦相同，签名公式会在不同用户间泄漏足够的线性关系，从而导致私钥暴露。

7.2 推导

SM2 签名公式为：

$$s = (1 + d)^{-1}(k - rd) \pmod{n}.$$

已知用户 U_1 的私钥 d_1 与签名 (r_1, s_1) ，则：

$$s_1(1 + d_1) \equiv k - r_1 d_1 \pmod{n},$$

可直接解出：

$$k \equiv s_1(1 + d_1) + r_1 d_1 \pmod{n}.$$

再对用户 U_2 的签名 (r_2, s_2) 使用已知的 k ：

$$s_2(1 + d_2) \equiv k - r_2 d_2 \pmod{n},$$

整理：

$$(s_2 + r_2)d_2 \equiv k - s_2 \pmod{n},$$

于是：

$$d_2 \equiv (k - s_2)(s_2 + r_2)^{-1} \pmod{n}.$$

因此，不同用户使用相同 k 会使得知道一个用户的私钥的攻击者能够直接计算出另一方的私钥。

7.3 验证代码

验证脚本完整实现了 SM2 曲线算术和 SM3 哈希，并提供：

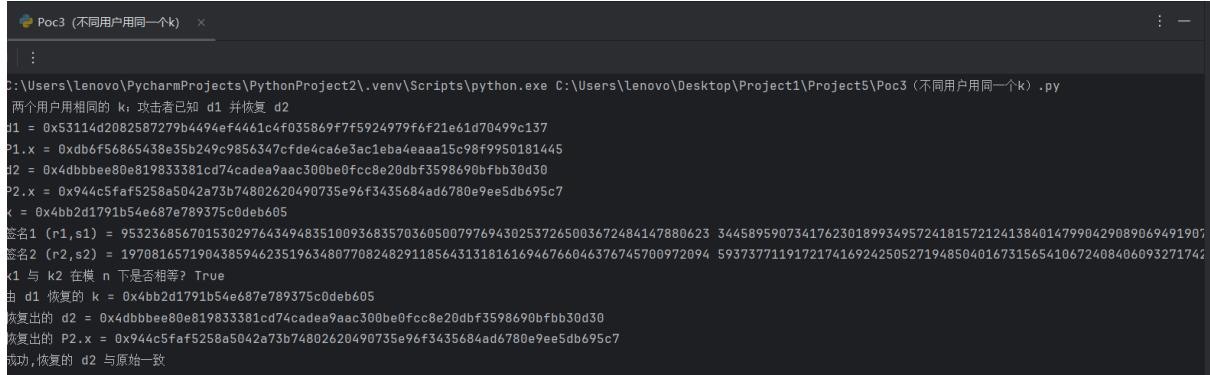
- `sm2sign_return_k`: 可选强制 `forced_k`, 用于模拟两个用户重复使用相同 k 的情况。
- `recover_d_sig`: 已知 d_1 、 r_1 、 s_1 计算 k 。
- `recover_d`: 已知 (r_2, s_2, k) 计算 d_2 。

在主程序中，随机生成两个用户密钥对，强制它们在签名时使用相同的 k ，验证是否可从 d_1 恢复 d_2 。

```
1 def sm2sign_return_k(d: int, msg: bytes, forced_k: int = None) -> Tuple[int,
2                               int]:
3     e = _int_from_msg(msg)
4     while True:
5         if forced_k is not None:
6             k = forced_k % n
7         else:
8             k = int.from_bytes(os.urandom(32), 'big') % n
9         if k == 0:
10            continue
11         x1y1 = scalar_mult(k, G)
12         if x1y1 is None:
13            continue
14         x1, y1 = x1y1
15         r = (e + x1) % n
16         if r == 0 or r + k == n:
17            continue
18         inv_1pd = mod_inv((1 + d) % n, n)
19         s = (inv_1pd * (k - r * d)) % n
20         if s == 0:
21            continue
22         return r, s, k
23
24     def recover_d_sig(d:int, r:int, s:int) -> int:
25         return ((1 + d) * s + r * d) % n
26
27     def recover_d(r:int, s:int, k:int) -> int:
28         denom = (s + r) % n
29         if denom == 0:
30             raise ZeroDivisionError("s + r == 0 mod n")
31         inv = pow(denom, -1, n)
32         d_rec = ((k - s) % n) * inv % n
33         return d_rec
```

7.4 实验结果

实验结果如下：



```

Poc3 (不同用户用同一个k)  ×
C:\Users\lenovo\PycharmProjects\PythonProject2\.venv\Scripts\python.exe C:\Users\lenovo\Desktop\Project1\Project5\Poc3 (不同用户用同一个k).py
两个用户用相同的 k, 攻击者已知 d1 并恢复 d2
d1 = 0x53114d2082587279b4494ef4461c4f035869ff7f5924979f6f21e61d70499c137
r1.x = 0xdb6f56865438e35b249c9856347cfde4ca6e3ac1eba4eaaa15c98f9950181445
d2 = 0x4dbbbe80e819833381cd74cadea9aac300be0fcc8e20dbf3598690fbff30d30
r2.x = 0x944c5faf5258a5042a73b74802620490735e96f3435684ad6780e9ee5db695c7
k = 0x4bb2d1791b54e687e789375c0deb605
签名1 (r1,s1) = 9532368567015302976434948351009368357036050079769430253726500367248147880623 3445895907341762301899349572418157212413840147990429089069491907
签名2 (r2,s2) = 19708165719043859462351963480770824829118564313181616946766046376745700972094 5937377119172174169242505271948504016731565410672408406093271742
k1 与 k2 在模 n 下是否相等? True
由 d1 恢复的 k = 0x4bb2d1791b54e687e789375c0deb605
恢复出的 d2 = 0x4dbbbe80e819833381cd74cadea9aac300be0fcc8e20dbf3598690fbff30d30
恢复出的 P2.x = 0x944c5faf5258a5042a73b74802620490735e96f3435684ad6780e9ee5db695c7
成功, 恢复的 d2 与原始一致

```

图 7: 实验结果三

8 PoC4: SM2 与 ECDSA 使用相同 k

8.1 背景

SM2 和 ECDSA 虽然签名公式不同，但都基于相同的椭圆曲线参数并依赖一次性随机数 k 保证安全性。如果同一用户在 ECDSA 和 SM2 两种算法中使用了相同的 k ，则攻击者在获取一对跨协议的签名后，可以联合利用两种算法的公式解出私钥 d 。这是一种 **跨协议密钥泄露** 攻击，根源在于两种算法对 k 的依赖是独立但共享的，一旦 k 被重用就能建立可解的方程组。

8.2 推导

ECDSA 签名公式：

$$s_1 \equiv k^{-1}(e_1 + dr_1) \pmod{n} \implies k \equiv (e_1 + dr_1)s_1^{-1} \pmod{n}.$$

SM2 签名公式：

$$s_2 \equiv (1 + d)^{-1}(k - r_2d) \pmod{n}.$$

将 k 代入 SM2 公式：

$$s_2(1 + d) \equiv (e_1 + dr_1)s_1^{-1} - r_2d \pmod{n}.$$

整理 d 项：

$$\begin{aligned} s_2 + s_2d &\equiv e_1s_1^{-1} + dr_1s_1^{-1} - r_2d \pmod{n}, \\ (s_2 - r_1s_1^{-1} + r_2)d &\equiv e_1s_1^{-1} - s_2 \pmod{n}. \end{aligned}$$

因此：

$$d \equiv \frac{e_1s_1^{-1} - s_2}{s_2 - r_1s_1^{-1} + r_2} \pmod{n}.$$

在实现时，可将 s_1^{-1} 提前计算，并按模 n 运算即可。

8.3 攻击步骤

1. **收集输入**: 获取同一私钥 d 下的 ECDSA 签名 (r_1, s_1) 与消息 M_1 , 以及 SM2 签名 (r_2, s_2) 与消息 M_2 , 两次签名使用相同的随机数 k 。
2. **计算哈希**: 对 M_1 用对应算法的哈希函数计算 $e_1 = H(M_1) \bmod n$ 。
3. **代入公式**: 使用推导得到的跨协议公式计算 d 。
4. **校验**: 用恢复的 d 计算公钥并与原公钥比对, 或对任意消息签名并验证。

8.4 验证代码

代码完整实现了 SM2 与 ECDSA 签名 (均基于同一椭圆曲线与 SM3 哈希), 并在主程序中二者使用相同的随机数 k 。随后调用 `recover_d` 函数利用跨协议公式恢复私钥并验证正确性。

```
1
2 def int_from_msg(msg: bytes) -> int:
3     return int.from_bytes(sm3_hash(msg), 'big') % n
4
5 def ecdsa_sign(d: int, msg: bytes, forced_k: int = None) -> Tuple[int,int,int]:
6     e = int_from_msg(msg)
7     while True:
8         if forced_k is not None:
9             k = forced_k % n
10        else:
11            k = int.from_bytes(os.urandom(32), 'big') % n
12            if k == 0: continue
13            x1,y1 = scalar_mult(k, G)
14            if x1 is None: continue
15            r = x1 % n
16            if r == 0: continue
17            s = (pow(k, -1, n) * (e + d * r)) % n
18            if s == 0: continue
19            return r, s, k
20
21 def sm2_sign(d: int, msg: bytes, forced_k: int = None) -> Tuple[int,int,int]:
22     e = int_from_msg(msg)
23     while True:
24         if forced_k is not None:
25             k = forced_k % n
26         else:
27             k = int.from_bytes(os.urandom(32), 'big') % n
28             if k == 0: continue
29             x1,y1 = scalar_mult(k, G)
```

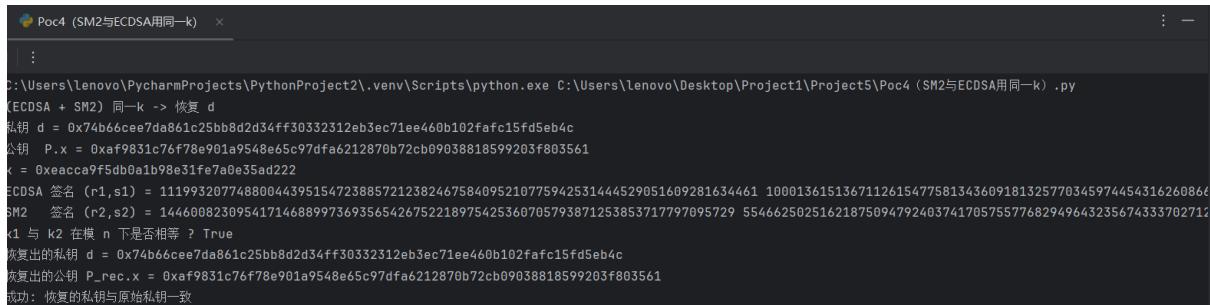
```

30         if x1 is None: continue
31         r = (e + x1) % n
32         if r == 0 or r + k == n: continue
33         inv = pow((1 + d) % n, -1, n)
34         s = (inv * (k - r * d)) % n
35         if s == 0: continue
36         return r, s, k
37
38     def recover_d(r1:int, s1:int, e1:int, r2:int, s2:int) -> int:
39         num = (s1 * s2 - e1) % n
40         denom = (r1 - (s1 * (s2 + r2) % n)) % n
41         if denom == 0:
42             raise ZeroDivisionError("denominator == 0 (cannot invert)")
43         inv = pow(denom, -1, n)
44         d_rec = (num * inv) % n
45         return d_rec

```

8.5 实验结果

实验结果如下：



```

Poc4 (SM2与ECDSA用同一k) ×
: 
d: C:\Users\lenovo\PycharmProjects\PythonProject2\.venv\Scripts\python.exe C:\Users\lenovo\Desktop\Project1\Project5\Poc4 (SM2与ECDSA用同一k) .py
(ECDSA + SM2) 同一k -> 恢复 d
私钥 d = 0x74b66cee7da861c25bb8d2d34ff30332312eb3ec71ee460b102fafc15fd5eb4c
公钥 P.x = 0xaf9831c76f78e901a9548e65c97dfa6212870b72cb09038818599203f883561
 $\epsilon$  = 0xeacca9ff5d00a1b98e31fe7ade35ad22
ECDSA 签名 (r1,s1) = 111993207748800443951547238857212382467584095210775942531444529051609281634461 1000136151367112615477581343609181325770345974454316260866
SM2 签名 (r2,s2) = 144600823095417146889973693565426752218975425360705793871253853717797095729 5546625025162187509479240374170575577682949643235674333702712
<1 与 <2 在模 n 下是否相等 ? True
恢复出的私钥 d = 0x74b66cee7da861c25bb8d2d34ff30332312eb3ec71ee460b102fafc15fd5eb4c
恢复出的公钥 P_rec.x = 0xaf9831c76f78e901a9548e65c97dfa6212870b72cb09038818599203f883561
成功：恢复的私钥与原始私钥一致

```

图 8: 实验结果四

9 伪造中本聰数字签名

9.1 实验原理

本实验基于比特币所使用的 **secp256k1** 椭圆曲线数字签名算法 (ECDSA) 展示当签名所用随机数 k 泄露时，攻击者如何恢复私钥并伪造签名。在 ECDSA 中，签名过程为：

$$\begin{aligned}
 e &= H(m) \bmod n, \quad (x_1, y_1) = k \cdot G, \\
 r &= x_1 \bmod n, \\
 s &= k^{-1}(e + d \cdot r) \bmod n,
 \end{aligned}$$

其中 d 为私钥, G 为基点, n 为基点阶。若攻击者获得了签名 (r, s) 及该次签名的 k 值, 则可直接解出私钥:

$$d \equiv (ks - e)r^{-1} \pmod{n}.$$

因此, ECDSA 安全性依赖 k 的保密性与随机性, 跨协议复用或泄露 k 将导致私钥完全暴露。

9.2 代码解释

1. 模逆运算函数 mod_inv:

```

1 def mod_inv(x: int, m: int) -> int:
2     x %= m
3     if x == 0:
4         raise ZeroDivisionError("inverse of 0")
5     return pow(x, -1, m)

```

说明: 该函数计算 x 在模 m 下的乘法逆元。首先将 x 归一化到 $[0, m - 1]$ 区间, 若 $x = 0$ 则在模运算中不存在逆元, 抛出异常。核心调用 Python 内置的 `pow` 三参数形式, 通过快速幂实现 $x^{-1} \bmod m$ 的计算。该实现利用了费马小定理的推广, 计算效率高且能处理大整数。

2. 椭圆曲线点加函数 point_add:

```

1 def point_add(P, Q):
2     if P is None:
3         return Q
4     if Q is None:
5         return P
6     x1, y1 = P
7     x2, y2 = Q
8     if x1 == x2 and (y1 + y2) % p == 0:
9         return None
10    if P == Q:
11        lam = (3 * x1 * x1 + a) * mod_inv(2 * y1, p) % p
12    else:
13        lam = (y2 - y1) * mod_inv((x2 - x1) % p, p) % p
14    x3 = (lam * lam - x1 - x2) % p
15    y3 = (lam * (x1 - x3) - y1) % p
16    return (x3, y3)

```

说明: 实现椭圆曲线加法规则, 支持点加与倍点两种情况。若任一点为 `None`, 直接返回另一点; 若 $x_1 = x_2$ 且 $y_1 = -y_2$, 结果为无穷远点 `None`。倍点时使用切线斜率公式, 普通加法时用割线斜率公式, 均通过 `mod_inv` 求模逆。最后计算新坐标 (x_3, y_3) 并返回。

3. 标量乘法函数 scalar_mult:

```

1 def scalar_mult(k: int, P):

```

```

1      if k % n == 0 or P is None:
2          return None
3
4      if k < 0:
5          return scalar_mult(-k, (P[0], (-P[1]) % p))
6
7      result = None
8      addend = P
9
10     while k:
11         if k & 1:
12             result = point_add(result, addend)
13         addend = point_add(addend, addend)
14         k >>= 1
15
16     return result

```

说明：采用“二进制展开”法实现标量乘法 kP 。先处理 $k = 0$ 或点为空的边界情况，以及负标量取反的情况。循环中每次检查最低位，若为 1 则将当前加数 `addend` 累加到结果中，然后进行倍点运算。通过不断右移 k 直至为 0，完成乘法计算。该方法实现简单，但速度较窗口法稍慢。

4. 签名函数 `sign`:

```

1  def sign(d: int, msg: bytes, forced_k: int = None):
2      e = int_from_msg_sha256(msg)
3      while True:
4          if forced_k is None:
5              k = int.from_bytes(os.urandom(32), 'big') % n
6          else:
7              k = forced_k % n
8          if k == 0:
9              continue
10         P = scalar_mult(k, G)
11         if P is None:
12             continue
13         rx = P[0] % n
14         if rx == 0:
15             continue
16         invk = mod_inv(k, n)
17         s = (invk * (e + d * rx)) % n
18         if s == 0:
19             continue
20         return rx, s, k

```

说明：实现 ECDSA 签名生成过程。首先计算消息哈希 e ，随机生成会话密钥 k （可通过 `forced_k` 固定以模拟泄露）。计算 $R = kG$ 并取其 x 坐标模 n 作为 r 。然后求 $k^{-1} \bmod n$ ，代入公式 $s = k^{-1}(e + dr) \bmod n$ 得到 s 。若 r 或 s 为 0，重新生成 k 。函数返回签名 (r, s) 及

使用的 k 。

5. 验签函数 verify:

```

1 def verify(Q, msg: bytes, sig):
2     r, s = sig
3     if not (1 <= r <= n-1 and 1 <= s <= n-1):
4         return False
5     e = int_from_msg_sha256(msg)
6     w = mod_inv(s, n)
7     u1 = (e * w) % n
8     u2 = (r * w) % n
9     P1 = scalar_mult(u1, G)
10    P2 = scalar_mult(u2, Q)
11    R = point_add(P1, P2)
12    if R is None:
13        return False
14    xR = R[0] % n
15    return xR == r

```

说明: 实现 ECDSA 验证算法。先检查 r, s 的范围, 计算消息哈希 e 。求 $w = s^{-1} \pmod{n}$, 得到 $u_1 = ew \pmod{n}$ 与 $u_2 = rw \pmod{n}$ 。计算 $P_1 = u_1G$ 与 $P_2 = u_2Q$, 相加得点 R 。若 R 无定义或 $R_x \neq r$, 则验证失败; 否则验签通过。

6. 私钥恢复函数 recover_d:

```

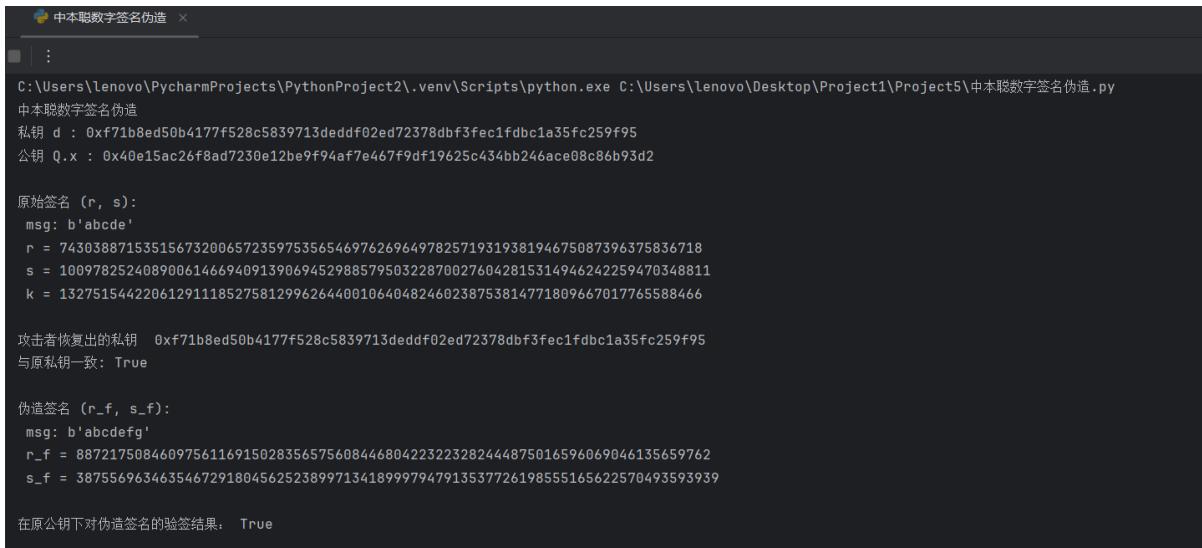
1 def recover_d(r: int, s: int, k: int, e:int) -> int:
2     denom = r % n
3     if denom == 0:
4         raise ZeroDivisionError("r == 0 mod n")
5     inv_r = mod_inv(denom, n)
6     d_rec = ((k * s - e) * inv_r) % n
7     return d_rec

```

说明: 根据 ECDSA 签名公式 $s \equiv k^{-1}(e+dr) \pmod{n}$ 反推出私钥 d 。变形得 $d \equiv (ks-e)r^{-1} \pmod{n}$, 因此已知 k, r, s 和消息哈希 e 即可恢复 d 。函数先判断 r 是否可逆, 再计算 $r^{-1} \pmod{n}$, 最后直接套用公式求得 d 。

9.3 实验结果

实验结果如下:



```
中本聪数字签名伪造 x
C:\Users\lenovo\PycharmProjects\PythonProject2\.venv\Scripts\python.exe C:\Users\lenovo\Desktop\Project1\Project5\中本聪数字签名伪造.py

原始签名 (r, s):
msg: b'abcde'
r = 74303887153515673200657235975356546976269649782571931938194675087396375836718
s = 10097825240890614669409139069452988579503228700276042815314946242259470348811
k = 13275154422061291118527581299626440010640482460238753814771809667017765588466

攻击者恢复出的私钥 0xf71b8ed50b4177f528c5839713deddf02ed72378dbf3fec1fdbca35fc259f95
与原私钥一致: True

伪造签名 (r_f, s_f):
msg: b'abcdefg'
r_f = 8872175084609756116915028356575608446804223223282444875016596069046135659762
s_f = 38755696346354672918045625238997134189997947913537726198555165622570493593939

在原公钥下对伪造签名的验签结果: True
```

图 9: 实验结果一