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

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：

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

- 椭圆曲线 $E(F_p)$ 定义为：

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

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 $k=\sum_{j=0}^{l-1}k_j2^j,k_j\in\{0,1\}$。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）:

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

```
def inverse_mod(k, p):
    return pow(k, -1, p)

def point_add(P, Q):
    if P is None: return Q
    if Q is None: return P
    x1, y1 = P
    x2, y2 = Q
    if x1 == x2 and y1 != y2:
        return None
    if P == Q:
        lam = (3 * x1 * x1 + a) * inverse_mod(2 * y1, p) % p
    else:
        lam = (y2 - y1) * inverse_mod(x2 - x1, p) % p
    x3 = (lam * lam - x1 - x2) % p
    y3 = (lam * (x1 - x3) - y1) % p
    return (x3, y3)

def scalar_mult(k, P):
    R = None
    while k:
        if k & 1:
            R = point_add(R, P)
        P = point_add(P, P)
        k >>= 1
    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。

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

```
#字节处理
def int_to_bytes(x: int, size=32) -> bytes:
    return x.to_bytes(size, 'big')

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

#KDF函数
def KDF(Z: bytes, klen: int) -> bytes:
    ct = 1
    v = 256
    Ha = b''
    for _ in range((klen + v - 1) // v):
        msg = Z + ct.to_bytes(4, 'big')
        digest_hex = sm3.sm3_hash(func.bytes_to_list(msg))
        Ha += bytes.fromhex(digest_hex)
        ct += 1
    return Ha[:klen // 8]

def sm2_encrypt(message: bytes, pubkey):
    k = random.randint(1, n - 1)
    C1 = scalar_mult(k, G)
    S = scalar_mult(k, pubkey)
    x2, y2 = S
    x2_bytes = int_to_bytes(x2)
    y2_bytes = int_to_bytes(y2)
    t = KDF(x2_bytes + y2_bytes, len(message) * 8)
    if int.from_bytes(t, 'big') == 0:
        raise ValueError("KDF derived all-zero key, aborting")
    C2 = bytes([m ^ t[i] for i, m in enumerate(message)])
    C3 = bytes.fromhex(sm3.sm3_hash(func.bytes_to_list(x2_bytes + message +
        y2_bytes)))
    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。

```
def sm2_decrypt(C1, C2, C3, privkey):
    S = scalar_mult(privkey, C1)
    x2, y2 = S
    x2_bytes = int_to_bytes(x2)
    y2_bytes = int_to_bytes(y2)
    t = KDF(x2_bytes + y2_bytes, len(C2) * 8)
    if int.from_bytes(t, 'big') == 0:
        raise ValueError("KDF derived all-zero key, aborting")
    M = bytes([c ^ t[i] for i, c in enumerate(C2)])
    u = bytes.fromhex(sm3.sm3_hash(func.bytes_to_list(x2_bytes + M +
        y2_bytes)))
    if u != C3:
        raise ValueError("Decryption failed: hash does not match")
    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。

```
def sm2_sign(message: bytes, privkey):
    e = int(sm3.sm3_hash(func.bytes_to_list(message)), 16) % n
    while True:
        k = random.randint(1, n - 1)
        P1 = scalar_mult(k, G)
        r = (e + P1[0]) % n
        if r == 0 or r + k == n:
            continue
        s = (inverse_mod(1 + privkey, n) * (k - r * privkey)) % n
        if s != 0:
            break
    return (r, s)

def sm2_verify(message: bytes, signature, pubkey):
    r, s = signature
```

```
    if not (1 <= r <= n - 1) or not (1 <= s <= n - 1):
        return False
    e = int(sm3.sm3_hash(func.bytes_to_list(message)), 16) % n
    t = (r + s) % n
    if t == 0:
        return False
    P1 = scalar_mult(s, G)
    P2 = scalar_mult(t, pubkey)
    Rxy = point_add(P1, P2)
    if Rxy is None:
        return False
    x1, y1 = Rxy
    R = (e + x1) % n
    return R == r
```

基础实现结果：

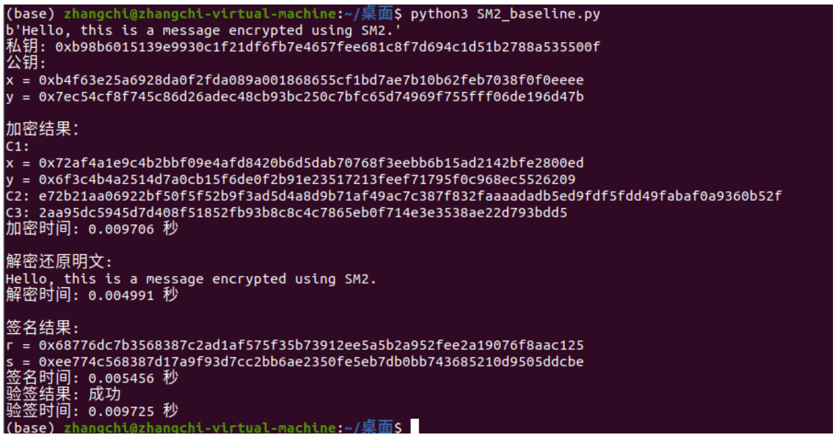

图 1 基础实现结果

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）：$y2 = x^3 + ax.Z^4 + bz^6$
- $X = x/z^2, Y = y/z^3$
- jacobian 坐标系计算复杂度低

我们定义两种坐标的转换：

```
def to_jacobian(P):
    x, y = P
```

```
    return (x, y, 1)

def from_jacobian(P):
    X, Y, Z = P
    if Z == 0:
        return (0, 0)
    Z_inv = inverse_mod(Z, p)
    Z_inv2 = (Z_inv * Z_inv) % p
    Z_inv3 = (Z_inv2 * Z_inv) % p
    x = (X * Z_inv2) % p
    y = (Y * Z_inv3) % p
    return (x, y)
```

定义了雅可比坐标下的乘积和加法：

```
def jacobian_double(P):
    X1, Y1, Z1 = P
    if Y1 == 0 or Z1 == 0:
        return (0, 1, 0)  # 无穷远点雅可比坐标表示
    A = (X1 * X1) % p
    B = (Y1 * Y1) % p
    C = (B * B) % p
    D = (2 * ((X1 + B) * (X1 + B) - A - C)) % p
    E = (3 * A) % p
    F = (E * E) % p
    X3 = (F - 2 * D) % p
    Y3 = (E * (D - X3) - 8 * C) % p
    Z3 = (2 * Y1 * Z1) % p
    return (X3, Y3, Z3)

def jacobian_add(P, Q):
    X1, Y1, Z1 = P
    X2, Y2, Z2 = Q

    if Z1 == 0:
        return (X2, Y2, Z2)
    if Z2 == 0:
        return (X1, Y1, Z1)

    Z1Z1 = (Z1 * Z1) % p
    Z2Z2 = (Z2 * Z2) % p
    U1 = (X1 * Z2Z2) % p
    U2 = (X2 * Z1Z1) % p
    S1 = (Y1 * Z2 * Z2Z2) % p
```

```
    S2 = (Y2 * Z1 * Z1Z1) % p

    if U1 == U2:
        if S1 != S2:
            return (0, 1, 0)  # 无穷远点
        else:
            return jacobian_double(P)

    H = (U2 - U1) % p
    I = (2 * H) % p
    I = (I * I) % p
    J = (H * I) % p
    RR = (2 * (S2 - S1)) % p
    V = (U1 * I) % p

    X3 = (RR * RR - J - 2 * V) % p
    Y3 = (RR * (V - X3) - 2 * S1 * J) % p
    Z3 = ((Z1 + Z2) * (Z1 + Z2) - Z1Z1 - Z2Z2) * H % p

    return (X3, Y3, Z3)


def scalar_mult(k, P):
    R = (0, 1, 0)  # 无穷远点的雅可比坐标表示
    P_j = to_jacobian(P)
    for i in bin(k)[2:]:
        R = jacobian_double(R)
        if i == '1':
            R = jacobian_add(R, P_j)
    return from_jacobian(R)
```

最终运行测试结果：

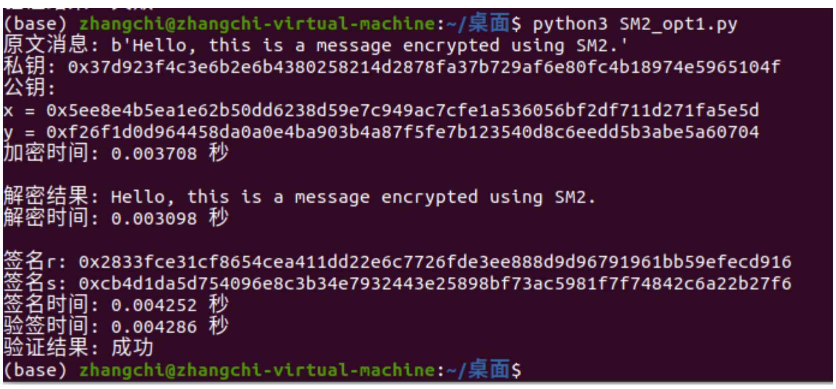

图 2 SM2 优化 1 结果

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

```
# ------------------ 固定点 G 的预计算加速 ------------------

G_TABLE = []

def precompute_G_table():
    global G_TABLE
    G_TABLE = []
    P = G
    for _ in range(256):
        G_TABLE.append(P)
        P = point_add(P, P)

def fixed_scalar_mult(k):
    R = None
    for i in range(256):
        if (k >> i) & 1:
            R = point_add(R, G_TABLE[i])
    return R
```

然后再 main 函数中使用:

```
    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:

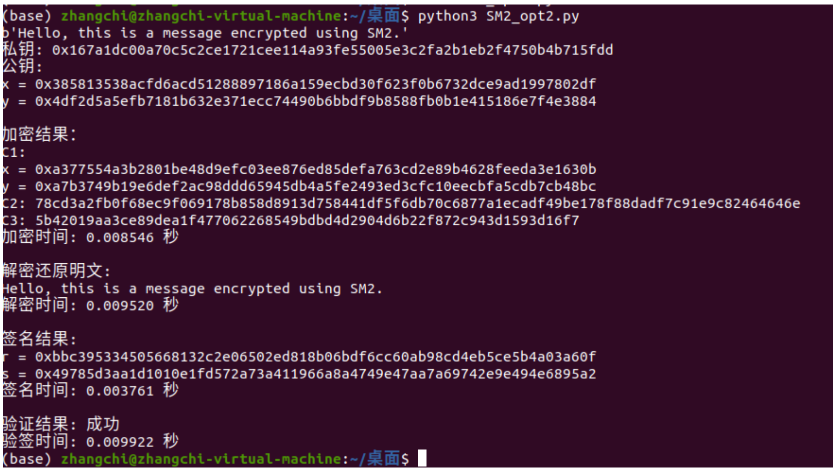

图 3 优化 2 运行结果

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 $kG = (x_1, y_1)$。
- $r = (e + x_1) \mod n$。
- $s = (1 + d_A)^{-1} \cdot (k - r \cdot d_A) \mod n$。

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 $s + r \neq 0$（模 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：

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

```
    """
    denom = (s + r) % n
    inv = inv_mod(denom, n)
    d = (inv * ((k - s) % n)) % n
    return d

# --------------------- PoC 演示 ---------------------
def demo_single_k_leak():
    print("== PoC: 单次 k 泄露 恢复私钥 (gmssl SM3 版本) ==")
    dA, PA = keygen()
    print("私钥 dA =", hex(dA))
    print("公钥 PA.x =", hex(PA[0]))
    # ID_A 示例
    ID_A = b'1234567812345678'
    ZA = compute_ZA(ID_A, PA)
    M = b"Hello SM2 with SM3 (gmssl) PoC"
    to_sign = ZA + M

    r, s, k, e = sign(dA, to_sign, k=None)  # 随机 k, 但我们拿到 k (模拟泄露)
    print("签名 (r, s) =", hex(r), hex(s))
    print("泄露的 k =", hex(k))

    d_rec = recover_d_from_k(r, s, k)
    print("恢复出的 d' =", hex(d_rec))
    print("恢复是否成功: ", d_rec == dA)
    assert d_rec == dA
```

运行结果如下所示：

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)**: $Sign_{d_A}(M, Z_A) \to (r, s)$
  - Set $M = Z_A||M$
  - $e = H_v(M)$
  - $k \leftarrow Z_n^*, kG = (x_1, y_1)$
  - $r = (e + x_1) \mod n$
  - $s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A)) \mod n$
  - 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 $M = Z_A||M$, $e = H_v(M)$
  - $t = (r + s) \mod n$
  - $(x_1, y_1) = sG + tP_A$
  - $R = (e + x_1) \mod n$, Verify $R = r$
- **Signing message** M_1 with d_A:
  - Randomly select $k \in [1, n-1]$, $kG = (x, y)$
  - $r_1 = (Hash(Z_A||M_1) + x) \mod n$
  - $s_1 = ((1 + d_A)^{-1} \cdot (k - r_1 \cdot d_A)) \mod n$
- **Signing message** M_2 with d_A:
  - Reuse the same k, $kG = (x, y)$
  - $r_2 = (Hash(Z_A||M_2) + x) \mod n$
  - $s_2 = ((1 + d_A)^{-1} \cdot (k - r_2 \cdot d_A)) \mod n$
- **恢复** d_A： 利用 2 个签名 $(r_1, s_1), (r_2, s_2)$:
  - $s_1(1 + d_A) = (k - r_1 \cdot d_A) \mod n$
  - $s_2(1 + d_A) = (k - r_2 \cdot d_A) \mod n$
  - $d_A = \dfrac{s_2 - s_1}{s_1 - s_2 + r_1 - r_2} \mod n$

根据上述推导，相应代码如下所示:

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

```
    M1 = b"Message one"
    M2 = b"Another message"
    # 故意使用相同 k
    k_shared = secrets.randbelow(n - 1) + 1
    r1, s1, k1, e1 = sign(dA, ZA + M1, k=k_shared)
    r2, s2, k2, e2 = sign(dA, ZA + M2, k=k_shared)
    print("签名1 (r1,s1) =", hex(r1), hex(s1))
    print("签名2 (r2,s2) =", hex(r2), hex(s2))
    print("共享 k =", hex(k_shared))

    # 已知 k: 直接恢复
    d_rec1 = recover_d_from_k(r1, s1, k_shared)
    print("已知 k 时恢复 d =", hex(d_rec1), " 成功?", d_rec1 == dA)

    # 若未知 k, 但有两签名: 用两签名推导
    # 推导: (s1 - s2) * (1 + d) = (r2 - r1) * d
    # => d * (r2 - r1 - s1 + s2) = s1 - s2
    numer = (s1 - s2) % n
    denom = (r2 - r1 - s1 + s2) % n
    try:
        d_rec2 = (numer * inv_mod(denom, n)) % n
        print("未知 k 时用两签名恢复 d =", hex(d_rec2), " 成功?", d_rec2 ==
            dA)
    except ZeroDivisionError:
        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：

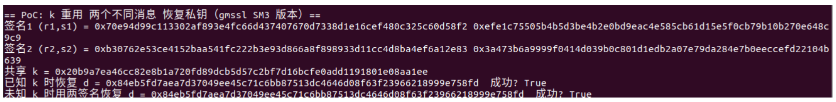

图 5 误用 2

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）：

$$Sign_{d_A}(M, Z_A) \to (r, s)$$

  - 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 $r = (e + x_1) \mod n$
  - 计算 $s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A)) \mod n$
  - 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 $t = (r + s) \mod n$
  - 计算 $(x_1, y_1) = sG + tP_A$
  - 计算 $R = (e + x_1) \mod n$，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 $r_1 = (Hash(Z_A || M_1) + x) \mod n$
  - 计算 $s_1 = ((1 + d_A)^{-1} \cdot (k - r_1 \cdot d_A)) \mod n$

- **Bob 签名消息** M_2（重用 k）：

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

- **密钥推导**：

  - Alice 可推导 Bob 私钥：

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

– 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:

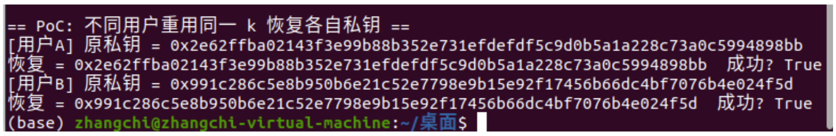

图 6 误用 3

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:

$$e_1 + dr_1 = s_1 s_2 + s_1 s_2 d + s_1 r_2 d \mod n$$

整理含 d 的项:

$$e_1 - s_1 s_2 = d(s_1 s_2 + s_1 r_2 - r_1) \mod n$$

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:

```
# --------------------- 跨算法攻击恢复私钥 ---------------------
def recover_d_from_ecdsa_sm2(r1, s1, e1, r2, s2):
    # 根据公式:
    # s1^{-1} * (e1 + d r1) = s2 * (1 + d) + d r2
    # 两边乘 s1:
    # e1 + d r1 = s1 s2 + s1 s2 d + s1 r2 d
    # 整理:
    # e1 - s1 s2 = d (s1 s2 + s1 r2 - r1)
    # d = (e1 - s1 s2) * inv(s1 s2 + s1 r2 - r1) mod n
    numerator = (e1 - s1 * s2) % n
    denominator = (s1 * s2 + s1 * r2 - r1) % n
    denom_inv = inv_mod(denominator, n)
    d = (numerator * denom_inv) % n
    return d

# --------------------- PoC 演示: 跨算法同 d 和 k 导致私钥泄露 ---------------------
def demo_cross_algorithm_attack():
    print("\n== PoC: 跨算法同 d 和 k 导致私钥泄露 ==")
    # 用户密钥和公钥
    d, P = keygen()
    print(f"[用户] 私钥 d = {hex(d)}")

    # 共享随机数 k
    k = secrets.randbelow(n - 1) + 1
    print(f"[用户] 共享随机数 k = {hex(k)}")

    # 消息
    M_ecdsa = b"Message signed by ECDSA"
    M_sm2 = b"Message signed by SM2"

    # ECDSA 签名
    r1, s1, e1 = sign_ecdsa(d, M_ecdsa, k)
    print(f"ECDSA 签名: r1={hex(r1)}, s1={hex(s1)}, e1={hex(e1)}")

    # SM2 签名
    ID_A = b'User-ID-12345678'
```

```python
    ZA = compute_ZA(ID_A, P)
    to_sign_sm2 = ZA + M_sm2
    r2, s2, k2, e2 = sign_sm2(d, to_sign_sm2, k=k)
    print(f"SM2 签名: r2={hex(r2)}, s2={hex(s2)}, e2={hex(e2)}")

    # 恢复私钥
    d_rec = recover_d_from_ecdsa_sm2(r1, s1, e1, r2, s2)
    print(f"恢复的私钥 d = {hex(d_rec)}")
    print("恢复成功？", d_rec == d)
```

运行结果如下所示，成功得出私钥：

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**：
   $$\begin{aligned} r &= x_r \mod n \\ s &= r \cdot v^{-1} \mod n \\ e &= r \cdot u \cdot v^{-1} \mod n \end{aligned}$$
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：
   $$\begin{aligned} u_1 &= e \cdot w = (r \cdot u \cdot v^{-1}) \cdot (v \cdot r^{-1}) = u \mod n \\ u_2 &= r \cdot w = r \cdot v \cdot r^{-1} = v \mod n \end{aligned}$$
   - 验签点计算：
   $$R' = u_1 G + u_2 P = uG + vP = R$$
   - 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：

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

```
    无私钥伪造签名

    选择随机 u, v, 计算:
        R = uG + vP
        r = R.x mod n
        s = r * v^{-1} mod n
        e = r * u * v^{-1} mod n

    返回 (r, s, e)
    """
    while True:
        u = secrets.randbelow(n - 1) + 1
        v = secrets.randbelow(n - 1) + 1

        R = point_add(scalar_mul(u, (Gx, Gy)), scalar_mul(v, P))
        if R is 0:
            continue
        r = R[0] % n
        if r == 0:
            continue
        v_inv = inv_mod(v, n)
        s = (r * v_inv) % n
        if s == 0:
            continue
        e = (r * u * v_inv) % n
        return (r, s, e)
```

最终，我们可以成功伪造签名：

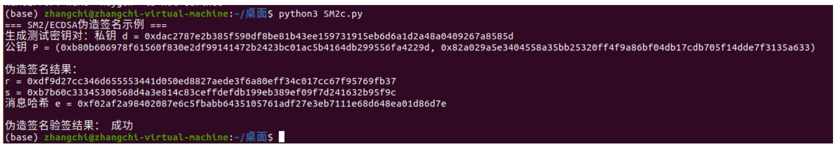

图 8 伪造成功

5 参考链接

1.CSDN 帖子《密码学中的 SM2》