



山东大学

SHANDONG UNIVERSITY

Project 5: SM2 的实现优化

姓 名: 谢星豪
学 号: 202100161170
专 业: 网络空间安全
班 级: 网安一班

2025 年 8 月 3 日

目录

1	实验内容	1
2	SM2 基础算法实现	1
2.1	实验原理	1
2.2	实验流程	3
2.3	实验结果	3
3	SM2 优化算法实现	4
3.1	实验流程	4
3.2	实验结果	9
4	POC 验证	10
4.1	实验流程	10
4.2	实验结果	15
5	中本聪签名伪造实验	16
5.1	实验流程	16
5.2	实验结果	17

1 实验内容

Project 5: SM2 的软件实现优化

- a). 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 基础算法实现

2.1 实验原理

椭圆曲线与系统参数

SM2 算法基于 $GF(p)$ 上的椭圆曲线, 其方程为:

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

其中 a, b 为曲线参数, p 为大素数。

SM2 标准中推荐使用的素数域 256 位椭圆曲线参数为:

- 素数域 p : FFFFFFFFE FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 00000000
FFFFFFFF FFFFFFFF
- 参数 a : FFFFFFFFE FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 00000000
FFFFFFFF FFFFFFFC
- 参数 b : 28E9FA9E 9D9F5E34 4D5A9E4B CF6509A7 F39789F5 15AB8F92 DDBCBD41
4D940E93
- 基点 G 的坐标:
 - x_G : 32C4AE2C 1F198119 5F990446 6A39C994 8FE30BBF F2660BE1 71A689DA
8BB17202
 - y_G : B975E668 F8764835 B264D069 F560CA12 3B489679 17DD5EE8 571A9A9E
334B41A8
- 基点 G 的阶 n : FFFFFFFE FFFFFFFF FFFFFFFF FFFFFFFF 7203DF6B
21C6052B 53BBF409 39D54123

密钥对生成

1. 生成私钥 d_A : 随机生成一个 $d_A \in [1, n - 1]$ 的整数。
2. 计算公钥 P_A :

$$P_A = [d_A]G$$

其中 $[d_A]G$ 表示椭圆曲线上的点 G 进行 d_A 次点加运算。

签名算法

假设签名者私钥为 d_A ，公钥为 P_A ，待签名消息为 M ，用户身份标识为 ID_A 。1. 计算 Z_A :

$$Z_A = \text{SM3}(\text{ENTL}_A \parallel ID_A \parallel a \parallel b \parallel x_G \parallel y_G \parallel x_A \parallel y_A)$$

其中 ENTL_A 是 ID_A 的比特长度， \parallel 表示连接。2. 计算哈希值 e :

$$e = \text{SM3}(Z_A \parallel M)$$

3. 生成随机数 k : 随机选择一个 $k \in [1, n-1]$ 的整数。4. 计算椭圆曲线点 R :

$$R = [k]G = (x_1, y_1)$$

5. 计算签名值 r :

$$r = (e + x_1) \pmod{n}$$

若 $r = 0$ 或 $r + k = n$ ，重新执行步骤 3。6. 计算签名值 s :

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

若 $s = 0$ ，重新执行步骤 3。7. 输出签名: 签名结果为一对整数 (r, s) 。

验签算法

假设验证者收到签名 (r, s) 、消息 M 和签名者的公钥 P_A 。1. 验证 r, s 的合法性: 验证 $r, s \in [1, n-1]$ ，否则验签失败。2. 计算哈希值 e' : 同签名过程，计算 Z_A 和 e' 。

$$e' = \text{SM3}(\text{SM3}(\text{ENTL}_A \parallel ID_A \parallel a \parallel b \parallel x_G \parallel y_G \parallel x_A \parallel y_A) \parallel M)$$

3. 计算 t :

$$t = (r + s) \pmod{n}$$

若 $t = 0$ ，验签失败。4. 计算椭圆曲线点 R' :

$$R' = [s]G + [t]P_A = (x'_1, y'_1)$$

5. 验证等式: 验证等式 $r \equiv (e' + x'_1) \pmod{n}$ 是否成立。若成立，则验签成功。

2.2 实验流程

下面将实现 SM2 的基础功能，包括密钥生成、签名和验证，并进行性能测试。具体代码见 `sm2_base.py` 文件。大致思路为：

- SM2 参数定义

- 定义椭圆曲线参数：素数域 p 、曲线系数 a, b 、基点 G 、阶 n 等
- 使用国家标准 GM/T 0003-2012 推荐的 256 位素数域椭圆曲线

- 核心类方法

- `_point_add`: 实现椭圆曲线点加法运算
 - * 处理无穷远点特殊情况
 - * 计算斜率 λ 并求新点坐标
- `_point_mul`: 实现标量乘法（快速幂算法）
- `generate_keypair`: 生成密钥对 (d, P)
 - * d 为随机私钥， $P = dG$ 为公钥
- `_get_z`: 计算用户标识哈希值（SM3 哈希）
- `_get_e`: 计算消息摘要（组合 z 和消息）
- `sign`: 签名算法
 - * 生成随机数 k
 - * 计算 (r, s) 签名，其中 $r = (e + x_1) \bmod n$, $s = (1 + d)^{-1}(k - rd) \bmod n$
- `verify`: 验证算法
 - * 检查 r, s 范围有效性
 - * 计算 $t = r + s$ 并验证等式 $r \equiv (e + x_1) \bmod n$

- 测试函数

- 测试不同数据大小 (16B-4KB) 下的性能
- 测量密钥生成、签名、验证时间
- 使用随机测试数据验证功能正确性

2.3 实验结果

性能测试结果如图 1 所示：

- 密钥生成时间相对稳定，因为与数据大小无关
- 签名和验证时间会随数据大小而增长，主要因为哈希计算部分

PS D:\2. SDU\01.课程资料\6.大三下\实践\5\script> & S:/MACRO/anaconda/python.exe "c

SM2 功能及性能测试：

数据大小(B)	密钥生成(ms)	签名(ms)	验证(ms)	状态
16	24.061	52.566	49.453	成功
64	27.319	54.556	54.616	成功
256	25.237	52.472	64.650	成功
1024	35.203	63.746	58.521	成功
4096	46.156	78.339	77.483	成功

图 1: SM2 基础算法运行结果

3 SM2 优化算法实现

3.1 实验流程

具体代码见 `sm2_acc.py` 文件。我们将按照以下顺序进行优化，效果递增，实现难度也递增：

1. 算法层面优化
- 优化 `verify` 过程: 使用“Shamir’s Trick”将两次点乘合并为一次。
 - 优化 `_point_mul`: 使用“滑动窗口法”替代简单的“Double-and-Add”。
2. 实现层面优化
- 使用雅可比坐标系 (Jacobian Coordinates): 在点乘过程中完全避免模逆元运算。
3. 缓存优化
- 缓存公钥和 Z 值: 对于重复签名的场景，避免重复计算。

1. 算法层面优化

优化 `verify` 过程: Shamir’s Trick (同时多点乘)

瓶颈分析: `verify` 函数是性能最差的，因为它需要计算两次点乘: $[s]G$ 和 $[t]P$ 。

```
1 p1 = self._point_mul(s, self.G)
2 p2 = self._point_mul(t, P)
```

优化原理: Shamir’s Trick (也称作 Simultaneous Multi-Point Multiplication) 可以将 $sG + tP$ 的计算合并，速度几乎相当于一次点乘，而不是两次。其思想是同时处理 s 和 t 的二进制位。在每一步迭代中，我们根据 s 和 t 当前的比特位是 0 还是 1，来决定加哪个点。

实现: 我们可以创建一个新的函数 `_shamir_mul_add` 来替代 `verify` 中的两次 `_point_mul` 和一次 `_point_add`。

```

1  # 在 SM2 类中添加这个新方法
2  def _shamir_mul_add(self, s: int, P: Tuple[int, int], t: int, Q:
    Tuple[int, int]) -> Tuple[int, int]:
3      """
4      使用 Shamir's Trick 高效计算  $[s]P + [t]Q$ 
5      """
6      # 预计算  $P+Q$ 
7      P_plus_Q = self._point_add(P, Q)
8      R = (0, 0)
9
10     # 从 s 和 t 的最高位开始处理
11     for i in range(max(s.bit_length(), t.bit_length()) - 1, -1, -1):
12         # Double
13         R = self._point_add(R, R)
14
15         # Add based on the bits of s and t
16         s_bit = (s >> i) & 1
17         t_bit = (t >> i) & 1
18
19         if s_bit == 1 and t_bit == 1:
20             R = self._point_add(R, P_plus_Q)
21         elif s_bit == 1:
22             R = self._point_add(R, P)
23         elif t_bit == 1:
24             R = self._point_add(R, Q)
25
26     return R
27
28 # 然后修改 verify 方法
29 def verify(self, P: Tuple[int, int], user_id: bytes, msg: bytes,
    signature: Tuple[int, int]) -> bool:
30     # ... (前略)
31     t = (r + s) % self.n
32     if t == 0: return False
33
34     # === 使用优化后的方法 ===
35     # 计算  $(x1, y1) = [s]G + [t]P$ 

```

```

36     x1, y1 = self._shamir_mul_add(s, self.G, t, P)
37     # =====
38
39     if x1 == 0 and y1 == 0: return False
40     R = (e + x1) % self.n
41     return R == r

```

效果: verify 函数的速度将提升接近一倍。这是最有效且改动相对较小的优化。

优化 _point_mul: 滑动窗口法 (Sliding Window)

瓶颈分析: 单次的点乘 _point_mul 依然是核心。当前的”Double-and-Add” 算法, 平均需要 $1.5 \times \log_2(k)$ 次点运算。

优化原理: 滑动窗口法通过预计算一些点, 然后在主循环中一次处理多个比特 (一个 “窗口”), 从而减少点加法的次数。

实现:

```

1  # 在 SM2 类中, 用这个方法替换原来的 _point_mul
2  def _point_mul(self, k: int, P: Tuple[int, int]) -> Tuple[int, int]:
3      """
4      椭圆曲线点乘 (滑动窗口法)
5      """
6      k = k % self.n
7      if k == 0: return (0, 0)
8
9      # 窗口大小 w, w=4 或 w=5 是一个常见的选择
10     w = 4
11
12     # 1. 预计算: P, [3]P, [5]P, ..., [2^{w-1}]P
13     P2 = self._point_add(P, P)
14     precomputed = [P]
15     for i in range(1, 1 << (w - 1)):
16         precomputed.append(self._point_add(precomputed[i - 1], P2))
17
18     # 2. 主循环
19     R = (0, 0)
20     i = k.bit_length() - 1
21     while i >= 0:
22         if (k >> i) & 1 == 0:
23             R = self._point_add(R, R)
24             i -= 1

```



```

25         else:
26             i_end = max(i - w + 1, -1)
27             window_val = 0
28             for j in range(i, i_end, -1):
29                 if (k >> j) & 1:
30                     window_val = (window_val << 1) | 1
31                 else:
32                     break
33
34             for _ in range(i - j + 1):
35                 R = self._point_add(R, R)
36
37             R = self._point_add(R, precomputed[window_val >> 1])
38             i = j - 1
39
40     return R

```

效果: 提升所有点乘操作的性能, 对 `generate_keypair` 和 `sign` 都有中等程度的加速。

2. 实现层面优化

瓶颈分析: 无论使用哪种点乘算法, 在仿射坐标系 (Affine Coordinates (x, y)) 下, 每一次点加/点倍加都需要一次模逆元 `_mod_inverse`。这是除点乘本身之外最慢的运算。

优化原理: 引入雅可比坐标系, 用 (X, Y, Z) 三个坐标表示一个点, 其中 $x = X/Z^2, y = Y/Z^3$ 。在这种坐标系下, 点加和点倍加不需要任何模逆元运算。我们只在点乘的最后, 将结果从雅可比坐标转换回仿射坐标时, 才需要一次模逆元。

实现: 这需要重写所有点运算函数, 改动较大。

```

1  # 在 SM2 类中添加/修改以下方法
2  def _affine_to_jacobian(self, P):
3      if P == (0, 0): return (1, 1, 0)
4      return (P[0], P[1], 1)
5
6  def _jacobian_to_affine(self, P):
7      X, Y, Z = P
8      if Z == 0: return (0, 0)
9      Z_inv = self._mod_inverse(Z, self.p)
10     Z_inv_sq = (Z_inv * Z_inv) % self.p
11     x = (X * Z_inv_sq) % self.p

```

```

12     y = (Y * Z_inv_sq * Z_inv) % self.p
13     return (x, y)
14
15 def _point_double_jacobian(self, P):
16     X1, Y1, Z1 = P
17     if Z1 == 0: return (1, 1, 0)
18     # ... (此处省略详细公式实现)
19     return X3, Y3, Z3
20
21 def _point_add_jacobian(self, P, Q):
22     # ... (此处省略详细公式实现)
23     return X3, Y3, Z3
24
25 # *** 最后, 重写核心的点乘函数 ***
26 def _point_mul(self, k: int, P: Tuple[int, int]) -> Tuple[int, int]:
27     if k % self.n == 0 or P == (0, 0): return (0, 0)
28     P_jac = self._affine_to_jacobian(P)
29     R_jac = (1, 1, 0) # 无穷远点
30
31     temp = P_jac
32     while k > 0:
33         if k & 1:
34             R_jac = self._point_add_jacobian(R_jac, temp)
35             temp = self._point_double_jacobian(temp)
36         k >>= 1
37
38     return self._jacobian_to_affine(R_jac)

```

效果: 这是一个全局性的巨大提升, 因为它将点乘中 $O(\log k)$ 次模逆元运算降为了 1 次。

3. 缓存优化

瓶颈分析: 在 `sign` 函数中, 每次签名都需要计算 `P = self._point_mul(d, self.G)` 和 `z = self._get_z(user_id, P)`。如果一个签名者 (拥有固定私钥 `d` 和 ID) 需要进行大量签名, 这些计算是重复的。

优化原理: 将签名者抽象为一个对象, 在初始化时就计算并缓存这些值。

```

1 class SM2Signer:

```

```

2     def __init__(self, sm2_instance: SM2, private_key: int, user_id:
      bytes):
3         self.sm2 = sm2_instance
4         self.d = private_key
5         self.user_id = user_id
6
7         # === 预计算和缓存 ===
8         print("Signer初始化: 正在预计算公钥P和Z值...")
9         self.P = self.sm2._point_mul(self.d, self.sm2.G)
10        self.z = self.sm2._get_z(self.user_id, self.P)
11        print("预计算完成。")
12
13    def sign(self, msg: bytes) -> Tuple[int, int]:
14        # 直接使用缓存的z值
15        e = self.sm2._get_e(self.z, msg)
16        # ... (后续签名逻辑不变)

```

效果: 对于一次性签名没有提升, 但对于需要用同一密钥反复签名的服务器、HSM 等场景, 可以省去每次签名时的一次点乘和一次 Z 值计算, 效果显著。

3.2 实验结果

优化后的 SM2 算法性能测试结果如图 2 所示, 对比标准 SM2 算法:

- 密钥生成速度 $\times 10$, 签名速度 $\times 2$, 验证速度 $\times 1.5$ 。
- 在数据量较小时速度提升更为明显。

PS D:\2. SDU\01.课程资料\6.大三下\实践\5\script> & S:/MACRO/anaconda/python.exe

--- 性能测试: 优化版SM2 ---

数据大小(B)	密钥生成(ms)	签名(ms)	验证(ms)	状态
16	2.000	7.002	31.107	成功
64	3.007	8.033	31.178	成功
256	2.556	9.303	34.217	成功
1024	2.550	13.687	38.710	成功
4096	2.005	29.862	52.539	成功

图 2: SM2 加速算法运行结果

4 POC 验证

4.1 实验流程

在 20250713-wen-sm2-public.pdf 文件中提到的 SM2 签名算法的几种典型误用场景，主要围绕着签名过程中一个至关重要的临时随机数 k 。如果 k 的生成、使用或保管不当，将会直接导致签名私钥 d 的泄露，造成严重的安全后果。

下面，我们将对三种最核心的误用情况进行推导和代码验证：

1. 泄露临时随机数 k
2. 重复使用临时随机数 k
3. 在 SM2 和 ECDSA 算法中使用相同的 d 和 k

攻击场景一：泄露临时随机数 k

当攻击者获取到某次签名使用的临时随机数 k 以及该次签名的结果 (r, s) 后，就可以直接计算出签名者的私钥 d 。

原理与推导

根据 SM2 签名算法，签名 (r, s) 的计算公式为：

- $r = (e + x_1) \pmod n$
- $s = ((1 + d)^{-1} \cdot (k - r \cdot d)) \pmod n$

其中， e 是消息摘要， k 是临时随机数， d 是私钥， n 是椭圆曲线的阶。

我们从第二个公式入手，求解 d ：

1. 等式两边同时乘以 $(1 + d)$ ：

$$s \cdot (1 + d) \equiv k - r \cdot d \pmod n$$

2. 展开左侧：

$$s + s \cdot d \equiv k - r \cdot d \pmod n$$

3. 将所有包含 d 的项移到等式左边：

$$s \cdot d + r \cdot d \equiv k - s \pmod n$$

4. 提取公因子 d ：

$$d \cdot (s + r) \equiv k - s \pmod n$$

5. 求解 d ，两边同时乘以 $(s + r)$ 的模逆元：

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

这个推导结果与文档中第 6 页的公式 $d_A = (s + r)^{-1} \cdot (k - s) \bmod n$ 一致。

PoC 验证代码

```

1 def pitfall_1_leaking_k(sm2_instance: 'SM2'):
2     """
3     演示场景1: 泄露临时随机数k导致私钥泄露
4     """
5     print("—— 场景1: 泄露临时随机数 k ——")
6
7     # 1. 用户A生成密钥对
8     d_A, P_A = sm2_instance.generate_keypair()
9     user_id_A = b'user_a@example.com'
10    msg = b'This is a test message for leaking k attack.'
11
12    # 2. 用户A进行签名, 但临时随机数k被泄露
13    k_leaked = secrets.randbelow(sm2_instance.n - 1) + 1
14    r, s = sm2_instance.sign_manual_k(d_A, msg, user_id_A, k_leaked)
15
16    print(f"原始私钥 (d): {hex(d_A)}")
17    print(f"泄露的随机数 (k): {hex(k_leaked)}")
18    print(f"生成的签名 (r, s): ({hex(r)}, {hex(s)})")
19
20    # 3. 攻击者使用泄露的k和签名(r, s)恢复私钥
21    # 根据公式: d = (k - s) * mod_inverse(s + r, n)
22    s_plus_r_inv = sm2_instance._mod_inverse(s + r, sm2_instance.n)
23    d_recovered = ((k_leaked - s) * s_plus_r_inv) % sm2_instance.n
24
25    print(f"恢复的私钥 (d_recovered): {hex(d_recovered)}")
26
27    # 4. 验证私钥是否恢复成功
28    assert d_A == d_recovered
29    print(" 攻击成功: 私钥已成功恢复! \n")

```

攻击场景二: 重复使用临时随机数 k

如果签名者使用相同的私钥 d 和相同的临时随机数 k 对两条不同的消息 $M1$ 和 $M2$ 进行签名, 攻击者只需获取这两次签名 $(r1, s1)$ 和 $(r2, s2)$, 就可以计算出私钥 d 。

原理与推导

我们已知有两次签名：

- 对消息 $M1$ 的签名 $(r1, s1)$: $s_1 \cdot (1 + d) \equiv k - r_1 \cdot d \pmod{n}$
- 对消息 $M2$ 的签名 $(r2, s2)$: $s_2 \cdot (1 + d) \equiv k - r_2 \cdot d \pmod{n}$

由于 k 和 d 在两个方程中是相同的，我们可以建立方程组来消去 k 。将第一个方程减去第二个方程：

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

化简得：

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

展开并求解 d ：

$$s_1 - s_2 + s_1 \cdot d - s_2 \cdot d \equiv r_2 \cdot d - r_1 \cdot d \pmod{n}$$

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

最后得到 d ：

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

这与文档中第 7 页的公式 $d_A = (s_1 - s_2) / (s_1 - s_2 + r_2 - r_1) \pmod{n}$ 是等价的（分子分母同时乘以-1）。

PoC 验证代码

```

1 def pitfall_2_reusing_k(sm2_instance: 'SM2'):
2     """
3     演示场景2: 重复使用临时随机数k导致私钥泄露
4     """
5     print("—— 场景2: 重复使用临时随机数 k ——")
6
7     # 1. 用户A生成密钥对
8     d_A, P_A = sm2_instance.generate_keypair()
9     user_id_A = b'user_a@example.com'
10    msg1 = b'This is the first message.'
11    msg2 = b'This is the second message, signed with the same k.'
12
13    # 2. 用户A使用同一个k对两条不同消息签名
14    k_reused = secrets.randbelow(sm2_instance.n - 1) + 1

```

```

15     r1, s1 = sm2_instance.sign_manual_k(d_A, msg1, user_id_A,
16         k_reused)
17
18     # 确保签名不同 (如果 r1=r2, 则 s1=s2, 攻击失效)
19     if r1 == r2:
20         print("偶然情况: 两次签名完全相同, 无法演示攻击。请重试。")
21         return
22
23     print(f"原始私钥 (d): {hex(d_A)}")
24     print(f"重复使用的k: {hex(k_reused)}")
25     print(f"签名1 (r1, s1): ({hex(r1)}, {hex(s1)})")
26     print(f"签名2 (r2, s2): ({hex(r2)}, {hex(s2)})")
27
28     # 3. 攻击者使用两次签名恢复私钥
29     # 根据公式:  $d = (s1 - s2) * \text{mod\_inverse}(r2 - r1 - s1 + s2, n)$ 
30     # 为避免负数, 计算  $(s1 - s2) \% n$  和  $(s2 - s1 + r2 - r1) \% n$ 
31     numerator = (s1 - s2) % sm2_instance.n
32     denominator = (r2 - r1 - s1 + s2) % sm2_instance.n
33     denominator_inv = sm2_instance._mod_inverse(denominator,
34         sm2_instance.n)
35
36     d_recovered = (numerator * denominator_inv) % sm2_instance.n
37
38     print(f"恢复的私钥 (d_recovered): {hex(d_recovered)}")
39
40     # 4. 验证私钥是否恢复成功
41     assert d_A == d_recovered
42     print(" 攻击成功: 私钥已成功恢复! \n")

```

攻击场景三: 在 SM2 和 ECDSA 中使用相同的 d 和 k

这是一个跨算法的漏洞。如果一个实体在不同的签名体制（如 SM2 和 ECDSA）中使用了相同的椭圆曲线参数、相同的私钥 d 以及一次性的相同的临时随机数 k ，攻击者可以结合这两个签名来恢复私钥 d 。

原理与推导

我们拥有两个不同算法生成的签名：

1. **ECDSA 签名** (r_1, s_1) : 标准 ECDSA 的 s 计算公式为 $s_1 = k^{-1} \cdot (e_1 + r_1 \cdot d) \pmod{n}$ 。其中, e_1 是消息的哈希值, r_1 是 kG 点的 x 坐标。我们可以从中推导出 k :

$$k \equiv s_1^{-1} \cdot (e_1 + r_1 \cdot d) \pmod{n}$$

2. **SM2 签名** (r_2, s_2) : 从场景一的推导中, 我们知道 k 和 d 的关系是:

$$k \equiv s_2 + d \cdot (s_2 + r_2) \pmod{n}$$

现在有了两个关于 k 的表达式, 令它们相等:

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

接下来, 我们解这个关于 d 的一次方程:

$$e_1 + r_1 \cdot d \equiv s_1 \cdot (s_2 + d \cdot (s_2 + r_2)) \pmod{n}$$

$$e_1 + r_1 \cdot d \equiv s_1 \cdot s_2 + d \cdot s_1 \cdot (s_2 + r_2) \pmod{n}$$

$$e_1 - s_1 \cdot s_2 \equiv d \cdot s_1 \cdot (s_2 + r_2) - r_1 \cdot d \pmod{n}$$

$$e_1 - s_1 \cdot s_2 \equiv d \cdot (s_1 \cdot s_2 + s_1 \cdot r_2 - r_1) \pmod{n}$$

最终得到 d 的计算公式:

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

PoC 验证代码

```

1 def pitfall_3_same_d_k_in_ecdsa_sm2(sm2_instance: 'SM2'):
2     """
3     演示场景3: 在SM2和ECDSA中使用相同的私钥d和临时随机数k
4     """
5     print("—— 场景3: SM2与ECDSA共用 d 和 k ——")
6
7     # 1. 用户生成密钥对, 计划在两个算法中使用
8     d, P = sm2_instance.generate_keypair()
9     user_id = b'user_ reusing_keys@example.com'
10    msg = b'A message signed by two different algorithms.'
11

```



```
12     # 2. 用户使用相同的d和k, 分别生成ECDSA和SM2签名
13     k_reused = secrets.randbelow(sm2_instance.n - 1) + 1
14
15     # ECDSA签名
16     r1, s1, e1 = sm2_instance.ecdsa_sign_manual_k(d, msg, k_reused)
17
18     # SM2签名
19     r2, s2 = sm2_instance.sign_manual_k(d, msg, user_id, k_reused)
20
21     print(f"原始私钥 (d): {hex(d)}")
22     print(f"共用的k: {hex(k_reused)}")
23     print(f"ECDSA签名 (r1, s1): ({hex(r1)}, {hex(s1)})")
24     print(f"SM2签名 (r2, s2): ({hex(r2)}, {hex(s2)})")
25
26     # 3. 攻击者结合两个签名恢复私钥
27     # 根据公式  $d = (e1 - s1*s2) * \text{mod\_inverse}(s1*s2 + s1*r2 - r1, n)$ 
28     numerator = (e1 - s1 * s2) % sm2_instance.n
29     denominator = (s1 * s2 + s1 * r2 - r1) % sm2_instance.n
30     denominator_inv = sm2_instance._mod_inverse(denominator,
31                                                  sm2_instance.n)
32
33     d_recovered = (numerator * denominator_inv) % sm2_instance.n
34
35     print(f"恢复的私钥 (d_recovered): {hex(d_recovered)}")
36
37     # 4. 验证
38     assert d == d_recovered
39     print(" 攻击成功: 私钥已成功恢复! \n")
```

4.2 实验结果

攻击结果如图 3 所示, 三种攻击场景下均攻击成功。我们清晰地展示了 SM2 签名算法在几种特定误用场景下的严重脆弱性。这些攻击的核心都在于临时随机数 k 。

```

PS D:\2. SDU\01.课程资料\6.大三下\实践\5\script> & S:\MACRO\anaconda\python.exe "d:/2. SDU/01.课程资料/6.大三下/实践/5/script/sm3_poc.py"
--- 场景1: 泄露临时随机数 k ---
原始私钥 (d): 0xcfa8fc34e7ef229d38fa4452e526d5b6a50001208ce2a840f3b0a8a1ff73f01
泄露的随机数 (k): 0xdd167781e2c8a8da356a2d97405cf8da1f7508b54ebfbf356beb299f7d285989
生成的签名 (r, s): (0x5395bb9c4c157e0b1dee307058bf60b7c46e4cf0cab6b420161e20db49f4ac8a, 0x71c36de58e366b1e33afdcdfcae18abeb8096a9826e013c214a775c9faa11305)
恢复的私钥 (d_recovered): 0xcfa8fc34e7ef229d38fa4452e526d5b6a50001208ce2a840f3b0a8a1ff73f01
✅ 攻击成功: 私钥已成功恢复!

--- 场景2: 重复使用临时随机数 k ---
原始私钥 (d): 0xe94ef93e47ae2c1d2c0e2f16822f0dd5f3264dad2c206ce8b2d3aa5cafb3e47
重复使用的k: 0xf239c8df76abbb1b3069989004aff863f2c03c5fcccdb076239e873f17fd973c8
签名1 (r1, s1): (0x41fc3f51e7b9c1abd5dd32ec75671116ac40f3a03953db71ca08db91030390d5, 0x1f4be4674d60d5194fe9606bcb9ae37bbebd9a8841520e55fbec1d392a0ce631)
签名2 (r2, s2): (0x85ac208442605b86d3448f654cebd7a1f8d918900fb64115e12bb9ed18a80143, 0xde796dcd417f58317a9ce8f08830fb093c481539a28ba2fb1c46ba240b738d9)
恢复的私钥 (d_recovered): 0xe94ef93e47ae2c1d2c0e2f16822f0dd5f3264dad2c206ce8b2d3aa5cafb3e47
✅ 攻击成功: 私钥已成功恢复!

--- 场景3: SM2与ECDSA共用 d 和 k ---
原始私钥 (d): 0x68adfa6da5f5d618e61eda67d13e2474ac4078cc0a8cb2330e0b026c7e15865f
共用的k: 0x922718d8133280387acce69c52cd4ab522f4329de4a70237ed9c5afa8393991
ECDSA签名 (r1, s1): (0xfcbff031bf2c89c501d424a033b6ebf57393897c5dc078bf4c754bb97b6dff, 0x3f1a5c54b3c7557eb0ee15d11761d98755ca01d90b2c02664810ac4e47f47b14)
SM2签名 (r2, s2): (0x586a2e419092575ebb4f58ab1bc856d165e2b8776d7306f9b58f3e3d1b833137, 0x7e8f16aaf977f6cc2f7ea5761c1b5a82e48c3392e8a12d344fbd62dcb4dcddc)
恢复的私钥 (d_recovered): 0x68adfa6da5f5d618e61eda67d13e2474ac4078cc0a8cb2330e0b026c7e15865f
✅ 攻击成功: 私钥已成功恢复!

```

图 3: POC 验证运行结果

核心安全准则:

1. **绝不泄露 k :** k 的保密级别必须等同于私钥 d 。
2. **绝不重复使用 k :** 对于每一次签名,都必须使用一个通过密码学安全伪随机数生成器 (CSPRNG) 产生的、全新的、不可预测的 k 。RFC 6979 提出了一种通过私钥和消息生成确定性 k 的方法,可以有效避免因随机数生成器质量不佳而导致的问题。
3. **避免跨算法重用密钥:** 不应在不同的密码体制中不加考虑地重用相同的密钥对和临时参数,因为不同算法的数学结构可能导致意想不到的关联,从而被攻击者利用。

5 中本聪签名伪造实验

5.1 实验流程

本实验将模拟一个场景: 一个“懒惰”的验证节点,它在验证签名时跳过了对消息本身进行哈希计算的步骤,而是直接使用了签名方提供的哈希值 e 。我们将扮演攻击者,利用这个漏洞,在完全不知道中本聪私钥的情况下,构造一个能通过该节点验证的有效签名。

1. 攻击原理与数学推导

标准的 ECDSA 验证流程。一个签名 (r, s) 是否有效,取决于以下验证方程:

1. 验证者收到消息 M 、签名 (r, s) 和公钥 P 。
2. 验证者自己计算消息哈希: $e = \text{HASH}(M)$ 。
3. 计算 $w = s^{-1} \pmod{n}$ 。
4. 计算 $u_1 = e \cdot w \pmod{n}$ 和 $u_2 = r \cdot w \pmod{n}$ 。

5. 计算曲线上的点 $R' = (x', y') = u_1G + u_2P$ ，其中 G 是基点， n 是曲线的阶。
6. 验证 $r \equiv x' \pmod{n}$ 是否成立。如果成立，签名有效。

漏洞利用点：如果验证者跳过第 2 步，直接接受一个由攻击者提供的 e ，攻击者可以反向构造。我们的目标是让最终的验证方程 $r \equiv x' \pmod{n}$ 成立。我们可以自己定义 u_1 和 u_2 ，然后推导出能满足条件的 r , s , 和 e 。

攻击步骤如下：

1. **选择随机数：**攻击者选择两个任意的、非零的随机整数 u 和 v 。
2. **计算伪造的点 R ：**攻击者使用中本聪的公钥 P 和曲线基点 G ，计算一个新的点 $R = (x, y) = uG + vP$ 。由于 u, v, G, P 都是已知的，这个计算是可行的。
3. **构造签名中的 r ：**攻击者从点 R 的 x 坐标得到 r ： $r = x \pmod{n}$ 。
4. **构造签名中的 s ：**回顾验证方程中的 $u_2 = r \cdot s^{-1} \pmod{n}$ 。在我们的构造中，我们希望 $v = u_2$ 。所以，我们令 $v = r \cdot s^{-1} \pmod{n}$ 。由此可以解出 s ：

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

5. **构造伪造的哈希 e ：**同样，回顾 $u_1 = e \cdot s^{-1} \pmod{n}$ 。我们希望 $u = u_1$ 。所以，我们令 $u = e \cdot s^{-1} \pmod{n}$ 。由此可以解出 e ：

$$e = u \cdot s \pmod{n}$$

至此，攻击者已经凭空构造出了一个伪造的签名 (r, s) 和一个与之对应的伪造哈希 e_{forged} 。

当这个懒惰的验证节点收到任意消息 M_{fake} 、伪造签名 (r, s) 以及攻击者提供的伪造哈希 e_{forged} 时，它的验证流程会变成：

- $w = s^{-1} = (r \cdot v^{-1})^{-1} = r^{-1}v$
- $u_1 = e_{\text{forged}} \cdot w = (u \cdot s) \cdot s^{-1} = u$
- $u_2 = r \cdot w = r \cdot (r^{-1}v) = v$
- 它最终计算的点将是 $R' = u_1G + u_2P = uG + vP$ 。这正是我们一开始构造的 R 点！
- 验证 $r \equiv R'.x \pmod{n}$ 。由于我们就是这么定义的 r ，验证必然通过。

攻击成功。注意，整个过程完全不需要私钥 d 。

5.2 实验结果

完整代码见 `sm2_zbc.py`。模拟实验结果如图 4 所示，可以看出：

- 在存在漏洞的验证器上，我们伪造的签名成功通过了验证。验证器被我们提供的 e_{forged} 所欺骗，错误地确认了一个虚假声明。

- 在正确的验证器上，同一个伪造签名被立即识破并拒绝。因为正确的验证器忽略了外部输入，自行对消息 `fake_message` 计算哈希，得到的真实哈希与我们伪造签名所对应的 `e_forged` 完全不同，导致验证方程不成立。

```
PS D:\2. SDU\01. 课程资料\6. 大三下\实践\5\script> & S:/MACRO/anaconda/python.exe "d:/2. SDU\01. 课程资料/6. 大三下/实践/5/script/sm3_zbc.py"
--- 场景设定 ---
中本聪的公钥 (P): (x=0x48114695616167b5d6474e42335c100574df91d52861fac0861ae1d75baeeadd, y=0x182749ee34207f7f6c4bb5204df3df1e3354033afbc56d77c4dd16fd86412067)
-----

--- 攻击者开始伪造 ---
1. 攻击者选择随机数 u: 0x39bd24f2e0e17a48d0c57de5fb7972d6163451fe9143f733e7185b2f92d5df41
2. 攻击者选择随机数 v: 0x69d5fa7b629a2f64ccd8a7d53c713a2573fa0d6ab4a2a7dc1d702a36d14b2c9d
3. 计算伪造的点 R = uG + vP: (x=0x8eb5a0edfdb06482dd978e793712701181240c45f47c0d7950eb3659b19c8ff6, y=0x218f1df30653888baf9b970da7a35124dcee12fa022aa1aa9e4485423673d2f)
4. 构造伪造签名 (r,s):
   r = R.x mod n = 0x8eb5a0edfdb06482dd978e793712701181240c45f47c0d7950eb3659b19c8ff6
   s = r * v^-1 mod n = 0x254ecae85b8e22d8a8e560f569c0c55646e9c3a74bc26113c23f8e3d45095b12
5. 构造伪造哈希 e: 0x4e85462aa5025ee419cc6ad09b0fd4c530d67a85371e86cc54d3000192f77d54
-----

--- 验证阶段 ---
1. 在【存在漏洞】的验证器上验证:
   提交的消息: 'I am Satoshi Nakamoto and I own all bitcoins.'
   提交的签名: (r=0x8eb5a0edfdb06482dd978e793712701181240c45f47c0d7950eb3659b19c8ff6, s=0x254ecae85b8e22d8a8e560f569c0c55646e9c3a74bc26113c23f8e3d45095b12)
   提交的哈希: 0x4e85462aa5025ee419cc6ad09b0fd4c530d67a85371e86cc54d3000192f77d54
   验证结果: True
   ✅ 攻击成功! 伪造的签名通过了存在漏洞的验证器。
-----
2. 在【正确】的验证器上验证:
   提交的消息: 'I am Satoshi Nakamoto and I own all bitcoins.'
   提交的签名: (r=0x8eb5a0edfdb06482dd978e793712701181240c45f47c0d7950eb3659b19c8ff6, s=0x254ecae85b8e22d8a8e560f569c0c55646e9c3a74bc26113c23f8e3d45095b12)
   验证器内部计算的哈希: 0xdf38ad4881e3964c3bcfa20a6e84e43a3d78e9dc7a78c2cf0c17299858b94dda
   验证结果: False
   ✅ 伪造的签名被正确地拒绝了。
```

图 4: 中本聪签名伪造结果