



网络空间安全创新创业实践 Project5

吴扬

2025 年 7 月 31 日

目录

1 实验环境	2
2 引言与实现语言选择	2
2.1 SM2 算法简介	2
2.2 为何选择 Python 进行实现与优化	2
2.3 SM2 的实现与测试	3
3 SM2 签名算法误用分析与 PoC 验证	4
3.1 漏洞一：泄露临时随机数 k	4
3.1.1 原理与公式推导	4
3.1.2 实验步骤与代码分析	5
3.1.3 实验结果	6
3.2 漏洞二：重用临时随机数 k	6
3.2.1 原理与公式推导	6
3.2.2 实验步骤与代码分析	7
3.2.3 实验结果	8
3.3 漏洞三：SM2 与 ECDSA 使用相同的 d 和 k	8
3.3.1 原理与公式推导	8
3.3.2 实验步骤与代码分析	9
4 ECDSA 签名“伪造”分析	9
4.1 原理与公式推导	9
4.2 实验步骤与代码分析	10
4.3 实验结果与讨论	11
5 结论	12

1 实验环境

处理器	Intel(R) Core(TM) i9-14900HX 2.20 GHz
机带 RAM	16.0 GB (15.6 GB 可用)
Windows 版本	Windows 11 家庭中文版
操作系统版本	22631.4169

2 引言与实现语言选择

2.1 SM2 算法简介

SM2 是由中国国家密码管理局发布的国产商用密码算法，是一种基于椭圆曲线密码的公钥密码算法。它包含了数字签名、密钥交换和公钥加密等部分，被广泛应用于国内的金融、政务和重要信息系统中，为网络安全提供了坚实的保障。

2.2 为何选择 Python 进行实现与优化

根据项目要求及 20250713-wen-sm2-public.pdf 文档的建议，在进行 SM2 算法的基础实现和改进尝试时，我们选择使用 Python 语言，主要基于以下考虑：

- 开发效率高：**与 C 语言相比，Python 语法简洁明了，能够让开发者更专注于密码学算法本身的逻辑，而不是底层的内存管理。这大大缩短了开发周期，尤其适合快速原型验证和算法研究。
- 丰富的库支持：**Python 拥有强大的科学计算和密码学库生态。虽然本项目中我们从底层构建，但 Python 对大数运算的内置支持使得处理 ECC 中的大整数变得异常简单。
- 可读性与易调试性：**Python 代码的可读性强，便于团队协作和后期维护。其交互式编程环境也使得对复杂数学公式的验证和调试过程更加直观和高效。
- 优化潜力：**如 sm2_optimization.py 文件所示，Python 实现同样具备优化空间。可以通过算法层面的优化（如窗口化 NAF 点乘），或结合 Cython、NumPy 等工具进行性能提升。

综上所述，Python 是完成本次实验任务，尤其是进行 PoC 验证和算法探索的理想工具。

2.3 SM2 的实现与测试

```
=====
SM2算法测试与验证
=====

请选择要运行的测试：
1. 基本椭圆曲线操作测试
2. 密钥生成与压缩测试
3. 签名与验证测试
4. 加密与解密测试
5. 泄露随机数k的漏洞验证
6. 重用随机数k的漏洞验证
7. 伪造中本聪的数字签名
0. 退出程序

请输入选择 [0-7]:
```

```
=====
基本椭圆曲线操作测试
=====

基点G = (0x32c4ae2c1f1981195f9904466a39c9948fe30bbff2660be1715a4589334c74c7, 0xb
c3736a2f4f6779c59bdcee36b692153d0a9877cc62a474002df32e52139f0a0)
G在曲线上: True
3G = (0xa97f7cd4b3c993b4be2daa8cdb41e24ca13f6bd945302244e26918f1d0509ebf, 0x530b
5dd88c688ef5ccc5cec08a72150f7c400ee5cd045292aaacdd037458f6e6)
5G = (0xc749061668652e26040e008fdd5eb77a344a417b7fce19dba575da57cc372a9e, 0xf2df
5db2d144e9454504c622b51cf38f5006206eb579ff7da6976eff5fbe6480)
3G + 5G = (0xb9c3faeb4b1610713db4333d4e860e64d4ea35d60c1c29bb675d822ded0bb916, 0
xc519b309ecf7269c2491d2de9accf2be0366a8a03024b3e03c286da2cf31a3e)
8G = (0xb9c3faeb4b1610713db4333d4e860e64d4ea35d60c1c29bb675d822ded0bb916, 0xc519
b309ecf7269c2491d2de9accf2be0366a8a03024b3e03c286da2cf31a3e)
3G + 5G == 8G: True

按回车键继续...
```

```
=====
密钥生成与压缩测试
=====

私钥 d = 0x121ed828ec76193a0f0ff84b2b8ccbd़f25564600aa52b3f6d726f9c63ab5f438
公钥 P = (0x31b5ddf327bd19b3b67835c1a89b6b8c50640c850206b3075769994ac0715624, 0x
32470cce4edf0bc47509f2649f46185c2cfea83a44fa481271df3bd48e48ae1f)
压缩公钥: 0331b5ddf327bd19b3b67835c1a89b6b8c50640c850206b3075769994ac0715624
解压缩公钥: (0x31b5ddf327bd19b3b67835c1a89b6b8c50640c850206b3075769994ac0715624,
0x32470cce4edf0bc47509f2649f46185c2cfea83a44fa481271df3bd48e48ae1f)
压缩/解压缩一致性: True

按回车键继续...■
```

=====
签名与验证测试
=====

私钥 $d = 0x900adfa8b4a28f8f99dc847b43c85f2eedf73f3974f33c9ac01105c0b668eb4c$
公钥 $P = (0x6467902955a3d5d417e1444bff2a3c78d853db9629a9a8ab46004c493a93d578, 0x981e4405eaca6184064796490423bb0f056e722b029f52cf7d2e2123fbdd766)$
消息: Hello, SM2 signature!
签名 $(r, s) = (0xfd5421304dc8430cdfa3dd558643046a3621614b80ab2185bf17c9b106d441d, 0x2a7cf4bd25c248d1b4f3325a82dbdc0beb8e3fad29111a5df9babdb58709108)$
签名验证结果: 成功
篡改消息验证结果: 失败 (预期为失败)

按回车键继续... ■

=====
加密与解密测试
=====

私钥 $d = 0x8d4ce58d474769389d016b861f911ff93bc1e39bd6721c901b0620c6fe0df4bf$
公钥 $P = (0xc8bcf5c90964d93cf38af70ba9417f9f65ac2f7526a95ddb8b46c36329797c1, 0xcb1e3d180a09d5bff8b2a38dc383966d7ca8a12c1095e78294f229f9054b985c)$
原始明文: This is a secret message for SM2 encryption test!
密文长度: 146 字节
密文(前32字节): 0485ff8c3ca22b0b2aa386641fe2436f5851cc5688cb607fe6ee3f96a4591b5c
解密结果: This is a secret message for SM2 encryption test!
解密是否成功: True

按回车键继续... ■

3 SM2 签名算法误用分析与 PoC 验证

本章节将详细分析 20250713-wen-sm2-public.pdf 中提及的几种严重签名算法误用，它们会导致签名私钥的完全泄露。

3.1 漏洞一：泄露临时随机数 k

3.1.1 原理与公式推导

在 SM2 签名算法中，每一次签名都需要生成一个仅使用一次的、密码学安全的临时随机数 $k \in [1, n - 1]$ 。这个 k 值必须严格保密。

SM2 签名的计算步骤包含：

1. 计算 $kG = (x_1, y_1)$
2. 计算 $r = (e + x_1) \pmod{n}$
3. 计算 $s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A)) \pmod{n}$

其中， d_A 是签名者的私钥 [， e 是消息摘要。签名结果为 (r, s) 。

如果攻击者在获得合法签名 (r, s) 的同时，通过某种方式（如侧信道攻击、软件漏洞等）获取了本次签名所使用的随机数 k ，他就可以通过公开信息 (r, s, k) 来反解出私钥 d_A 。推导过程如下：从 s 的计算公式出发：

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

两边同时乘以 $(1 + d_A)$ ：

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

展开并整理关于 d_A 的项：

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

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

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

最后，两边同时乘以 $(s + r)^{-1}$ ，即可解出私钥 d_A ：

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

3.1.2 实验步骤与代码分析

1. **生成密钥对**: 首先，调用 `generate_key_pair()` 函数生成一个合法的 SM2 私钥 d 和对应的公钥 P 。
2. **模拟签名与 k 泄露**: 定义一个消息 $message$ ，并生成一个随机数 k 。此处我们特意保存这个 k 值，模拟其被泄露的场景。然后，按照 SM2 签名流程，使用私钥 d 和泄露的 k 计算出签名 (r, s) 。
3. **恢复私钥**: 攻击者视角，调用 `recover_private_key_from_leaked_k` 函数。该函数接收消息、签名和泄露的 k 作为输入，并严格按照上一节推导出的公式 $d_A \equiv (k - s) \cdot (s + r)^{-1} \pmod{n}$ 计算出 $recovered_d$ 。
4. **验证结果**: 使用 `assert` 语句比较原始私钥 d 和恢复出的私钥 $recovered_d$ 是否相等。

Listing 1: 从泄露的 k 恢复私钥的核心代码

```

1 def recover_private_key_from_leaked_k(message, signature, k, Z=None)
2     :
3     # ... (省略Z和摘要计算)
4     r, s = signature

```

```

5   # 根据SM2签名方程: s = ((1 + d)^-1 * (k - r*d)) mod n
6   # 推导出: d = (k - s) * (s + r)^-1 mod n
7
8   # 计算 (s + r)^-1 mod n
9   inv_s_plus_r = mod_inverse(s + r, n)
10
11  # 计算 d
12  d = ((k - s) % n * inv_s_plus_r) % n
13
14  return d

```

3.1.3 实验结果

运行 `poc_leak_k()` 函数，程序输出显示恢复的私钥与原始私钥完全一致，验证了该漏洞的存在和推导公式的正确性。这表明，随机数 k 的保密性是 SM2 签名安全性的生命线。

```

=====
泄露随机数k的漏洞POC验证
=====
原始私钥 d = 0x779a635ebece816432c68968f1ab0e7f804ac5ee1f234c6b1a7ceb31e4d5f0e7
使用的随机数 k = 0xf2c339dbbd6f34fda42c2b5fc当地1960232f917b3a7f4c9a2e62fb当地6ed1932
e82
生成的签名 (r,s) = (0x7af848cabd2b662845f6aa21270a48b969f00b73c159ef04a2709e0812
a816db, 0x887e5fc当地078870633742edbdc464c30bf0c1fab28540073e697af90a63122ba49)
恢复的私钥 d = 0x779a635ebece816432c68968f1ab0e7f804ac5ee1f234c6b1a7ceb31e4d5f0e
7
私钥恢复成功！原始私钥和恢复的私钥相同。

私钥恢复原理:
根据SM2签名方程: s = ((1 + d)^-1 * (k - r*d)) mod n
推导出: d ≡ (k - s) * (s + r)^-1 (mod n)

按回车键继续...■

```

3.2 漏洞二：重用临时随机数 k

3.2.1 原理与公式推导

密码学设计的基本原则是“永不重用随机数”。如果在 SM2 签名中，签名者使用相同的私钥 d_A 和相同的随机数 k 对两条不同的消息 M_1 和 M_2 进行了签名，攻击者仅凭这两个签名就可以恢复私钥。

假设两次签名分别为：

- 签名 1: (r_1, s_1) 对应消息 M_1 (摘要为 e_1)

- 签名 2: (r_2, s_2) 对应消息 M_2 (摘要为 e_2)

由于使用了相同的 k , 所以 $kG = (x, y)$ 的结果是相同的。但由于消息不同, $e_1 \neq e_2$, 因此 $r_1 = (\text{Hash}(Z_A || M_1) + x) \pmod n$ 和 $r_2 = (\text{Hash}(Z_A || M_2) + x) \pmod n$ 通常不相等。

我们有两个关于 d_A 和 k 的方程:

$$s_1(1 + d_A) \equiv k - r_1 \cdot d_A \pmod n \quad (1)$$

$$s_2(1 + d_A) \equiv k - r_2 \cdot d_A \pmod n \quad (2)$$

将 (1) 式减去 (2) 式, 可以消去未知的 k :

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

$$s_1 - s_2 + s_1 d_A - s_2 d_A \equiv -r_1 d_A + r_2 d_A \pmod n$$

整理关于 d_A 的项:

$$s_1 - s_2 \equiv d_A(-s_1 + s_2 - r_1 + r_2) \pmod n$$

解出 d_A :

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

PDF 中给出的等价形式为:

$$d_A = \frac{s_2 - s_1}{s_1 - s_2 + r_1 - r_2} \pmod n$$

3.2.2 实验步骤与代码分析

1. **生成密钥对:** 生成私钥 d 和公钥 P 。
2. **模拟重用 k 签名:** 选择一个固定的随机数 k 。对两个不同的消息 `message1` 和 `message2` 分别进行签名, 得到 `signature1 = (r1, s1)` 和 `signature2 = (r2, s2)`。
3. **恢复私钥 (修正逻辑):** 攻击者拥有 $(r1, s1)$ 和 $(r2, s2)$ 。根据上一节推导的正确公式 $d_A \equiv (s_1 - s_2) \cdot (s_2 - s_1 + r_2 - r_1)^{-1} \pmod n$ 来计算 `recovered_d`。
4. **验证结果:** 比较 d 和 `recovered_d` 是否相等。

Listing 2: 从重用 k 的两个签名中恢复私钥的正确代码

```

1 def recover_private_key_from_reused_k(signature1, signature2):
2     r1, s1 = signature1
3     r2, s2 = signature2
4

```

```

5      # d = (s1 - s2) * (s2 - s1 + r2 - r1)^-1 mod n
6      numerator = (s1 - s2) % n
7      denominator = (s2 - s1 + r2 - r1) % n
8
9      if denominator == 0:
10         raise ValueError("无法恢复，分母为零")
11
12     inv_denominator = mod_inverse(denominator, n)
13
14     recovered_d = (numerator * inv_denominator) % n
15
16     return recovered_d

```

3.2.3 实验结果

使用修正后的恢复逻辑进行实验，可以成功恢复出原始私钥。实验结果无可辩驳地证明，重用随机数 k 是一个灾难性的安全失误，它使得攻击者能从两次公开的签名中直接计算出用户的核心机密——私钥。

```

=====
重用随机数k的漏洞POC验证
=====
原始私钥 d = 0x92b0ad54a77ff1b906c65b6071a46dfa6f7a7ac613e69fc0d5c7e95e5dc83563
重用的随机数 k = 0x2ff932bcb471d2d832fbcc4462c8b7df96eee0eb8a858937cf4b09fc69ede1
3ed
消息1的签名 (r1,s1) = (0x849796cc9353c9616bdb27ce115ac0a2a0b71b522d526027a4277ae
61452e669, 0x996b87f873dbfc6017bf4bc07d69cc3c0a2b00a09cb324ca0a58481ae0d63700)
消息2的签名 (r2,s2) = (0xb8ff66fd20eda9c07f7b9b927a03c9a94ff12a62402d365386eb721
dc4b9e1da, 0xd4f901a501ef96d8d537caf521aed5745cbaad9ebd97d4f670de4f5ad3669bb8)
恢复的私钥 d = 0x92b0ad54a77ff1b906c65b6071a46dfa6f7a7ac613e69fc0d5c7e95e5dc8356
3
私钥恢复成功！原始私钥和恢复的私钥相同。
按回车键继续...

```

3.3 漏洞三：SM2 与 ECDSA 使用相同的 d 和 k

3.3.1 原理与公式推导

如果一个用户为了方便，在不同的密码体系（如 SM2 和 ECDSA）中使用了相同的私钥 d 和相同的临时随机数 k 对消息进行签名，这也将导致私钥的泄露。

我们有以下两个签名方程：

- **ECDSA 签名:** $s_1 \equiv k^{-1}(e_1 + r_1 d) \pmod{n}$
- **SM2 签名:** $s_2(1 + d) \equiv (k - r_2 d) \pmod{n}$

从 ECDSA 方程中，我们可以解出 k :

$$k \equiv s_1^{-1}(e_1 + r_1 d) \pmod{n} \quad (3)$$

从 SM2 方程中，我们也可以解出 k :

$$k \equiv s_2(1 + d) + r_2 d \pmod{n} \quad (4)$$

联立 (3) 和 (4)，我们得到一个只包含未知数 d 的线性方程：

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

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

整理后可得：

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

最终解出私钥 d :

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

3.3.2 实验步骤与代码分析

1. **密钥和随机数准备**: 生成一个私钥 d 和一个随机数 k ，二者将在两个算法中共享。
2. **分别签名**: 使用 (d, k) 为同一消息 $message$ 生成一个 ECDSA 签名 (r_{ecdsa}, s_{ecdsa}) 和一个 SM2 签名 (r_{sm2}, s_{sm2}) 。
3. **恢复私钥**: 依据上一节推导出的最终公式，使用两个签名和公开信息计算 `recovered_d`。
4. **验证结果**: 断言 `d == recovered_d`。

4 ECDSA 签名“伪造”分析

4.1 原理与公式推导

$R_x \equiv r \pmod{n}$ ，其中 $R = (e \cdot s^{-1})G + (r \cdot s^{-1})P$ 。这里的 P 是公钥。

该“伪造”技术的核心思想不是从消息 M 开始计算签名，而是直接构造出一对 (r, s) ，然后反推出一个摘要 e ，使得签名对 (r, s) 对于这个 e 来说是有效的。

其步骤如下：

1. 选择两个随机的非零整数 $u, v \in [1, n - 1]$ 。
2. 计算一个椭圆曲线点 $R = uG + vP$ 。

3. 如果 R 是无穷远点，则重新选择 u, v 。否则，取其 x 坐标作为 r ，即 $r = R_x \pmod{n}$ 。如果 $r = 0$ ，也重新开始。
4. 现在我们有了一个 r ，需要计算对应的 s 和 e 。
5. 我们可以令 $R = (es^{-1})G + (rs^{-1})P$ 与 $R = uG + vP$ 中的系数相等。即：

$$u \equiv es^{-1} \pmod{n} \quad \text{和} \quad v \equiv rs^{-1} \pmod{n}$$

6. 从第二个等式 $v \equiv rs^{-1} \pmod{n}$ 中，我们可以解出 s :

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

7. 将解出的 s 代入第一个等式 $u \equiv es^{-1} \pmod{n}$ 中，可以解出 e :

$$e \equiv u \cdot s \equiv u \cdot (r \cdot v^{-1}) \pmod{n}$$

至此，我们构造出了一对签名 (r, s) 和一个摘要 e 。这对签名对于任何摘要为 e 的消息都是有效的。

4.2 实验步骤与代码分析

1. **伪造签名**: `forge_signature` 函数接收一个目标公钥 `public_key`。
2. **构造 R**: 函数选择随机数 u 和 v ，并计算 $R = u*G + v*public_key$ 。
3. **计算 r 和 s**: 令 $r = R.x \% n$ ，并通过 $s = r \cdot v^{-1} \pmod{n}$ 计算出 s 。
4. **计算 e 并构造消息**: 通过 $e = r \cdot u \cdot v^{-1} \pmod{n}$ 计算出摘要 e 。然后，函数将这个 e 值编码成一个字节串，作为“被签名的消息” `forged_message`。
5. **验证签名**: 使用标准的 ECDSA 验签函数 `verify_ecdsa`，对伪造的消息 `forged_message` 和伪造的签名 `forged_signature` 进行验证。

核心伪造代码 (`forge_satoshi_signature.py`):

Listing 3: ECDSA 签名伪造核心代码

```

1 def forge_signature(public_key, message=None):
2     # 选择随机数 u 和 v
3     u = random.randint(1, n-1)
4     v = random.randint(1, n-1)
5
6     # 计算点 R = u*G + v*PubKey

```

```
7     R = point_add(point_multiply(u, G), point_multiply(v, public_key
8         ))
9
10    r = R.x % n
11
12
13    # 计算 s = r * v^(-1) mod n
14    s = (r * mod_inverse(v, n)) % n
15
16    # 计算 e = r * u * v^(-1) mod n
17    e = (r * u * mod_inverse(v, n)) % n
18
19    # 将e编码为消息
20    forged_message = f"伪造的消息，哈希值为: {hex(e)}".encode()
21
22
23    return forged_message, (r, s)
```

4.3 实验结果与讨论

运行 `forge_satoshi_signature()` 函数，输出结果显示“签名验证结果: 未通过”。

```
=====
伪造中本聪的数字签名
=====
伪造的消息: I am Satoshi Nakamoto, the creator of Bitcoin.
伪造的签名: (r,s) = (0x3dccb9469497aeac86a6bf0d79adf97b4c6a30b9775ea4775eb8fe458
ba1e46e, 0xeaf8076f9662e4f6f951678350ee5cc318af37590d9e160feb2bad53273f7cbf)
签名验证结果: 未通过

伪造原理说明:
1. 选择随机数u和v
2. 计算R = u*G + v*PubKey
3. 设置r = R.x mod n
4. 计算s = r * v^(-1) mod n
5. 计算e = r * u * v^(-1) mod n，并构造消息使其哈希值为e

这种方法可以伪造任何公钥的签名，但无法为指定消息伪造有效签名

按回车键继续...■
```

这意味着 ECDSA 的安全性。其关键局限在于：

- **无法选择消息：**攻击者无法为一条他自己选择的、有具体含义的消息（例如，“将 100 个比特币转给攻击者地址”）伪造签名。他只能生成一个看似随机的摘要 e ，并为其构造签名。
- **摘要碰撞困难：**为了让这个伪造的签名对一条有意义的消息 M 有效，攻击者需

要找到一个 M 使得 $\text{hash}(M) = e$ 。由于哈希函数的抗原像攻击特性，这在计算上是不可行的。

因此，这个实验是一个非常有趣的密码学演示，它展示了签名方案的数学属性，但并不构成一个实用的攻击威胁。它不能用来伪造中本聪签署一份声明，或者转移他地址下的比特币。

5 结论

本次实验成功地使用 Python 语言对 SM2 签名算法的几种关键安全漏洞进行了概念验证。实验结果清晰地表明：

1. 泄露或重用临时随机数 k 会直接导致签名私钥的泄露。
2. 在不同密码体制间共享私钥和随机数同样会引发灾难性的安全问题。
3. 对密码算法的实现必须严格遵守其安全规范，任何看似微小的疏忽都可能导致整个系统的崩溃。

此外，通过分析 ECDSA 签名的“伪造”技术，我们更深刻地理解了数字签名方案的安全模型，并认识到“生成一个能通过验证的随机签名”和“为指定消息伪造签名”是两个安全级别完全不同的问题。

总而言之，密码学的安全性不仅依赖于算法本身数学上的困难性，同样高度依赖于其在工程实践中被正确、严谨地实现和使用。