



山东大学

SHANDONG UNIVERSITY

Project 5: SM2 的软件实现优化

姓 名: 孙洋意

学 号: 202100201016

专 业: 网络空间安全

班 级: 网安 22.1

2025 年 8 月 7 日

目录

1	实验要求	1
2	SM2 加速	1
2.1	核心数学原理	1
2.1.1	椭圆曲线参数	1
2.2	加速实现技术	1
2.2.1	预计算与缓存	1
2.2.2	标量乘法优化	2
2.3	关键操作实现	2
2.3.1	点加运算	2
2.4	性能优化策略	3
2.4.1	并行计算	3
2.4.2	恒定时间实现	3
2.5	功能模块	4
2.5.1	数字签名	4
2.5.2	加密解密	4
2.6	性能测试	4
2.7	安全性保障	5
2.8	应用示例	5
2.9	运行结果	6
3	误用场景分析与 POC 验证	6
3.1	引言	6
3.2	SM2 签名算法回顾	6
3.2.1	签名生成过程	6
3.2.2	签名验证过程	6
3.3	场景 1: 随机数 k 泄露	7
3.3.1	数学推导	7
3.3.2	POC 验证代码	7

3.4	场景 2: 同一用户重复使用 k 值	7
3.4.1	数学推导	7
3.4.2	POC 验证代码	8
3.5	场景 3: 不同用户使用相同 k 值	8
3.5.1	数学推导	8
3.5.2	POC 验证代码	8
3.6	场景 4: SM2 与 ECDSA 混合使用相同 k 值	9
3.6.1	数学推导	9
3.6.2	POC 验证代码	9
3.7	实验结果	9
3.8	安全建议	9
3.9	运行结果	10
4	中本聪	10
4.1	ECDSA 签名原理回顾	11
4.1.1	签名生成	11
4.1.2	签名验证	11
4.2	伪造签名原理	11
4.2.1	数学构造	11
4.2.2	正确性证明	12
4.3	实现代码	12
4.3.1	核心伪造函数	12
4.3.2	验证函数	12
4.4	攻击流程	13
4.5	安全意义	13
4.6	防御措施	13
4.7	运行结果	13

1 实验要求

- 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 加速

SM2 是中国国家密码管理局发布的椭圆曲线公钥密码算法标准, 包含:

- 数字签名算法
- 密钥交换协议
- 公钥加密算法

2.1 核心数学原理

2.1.1 椭圆曲线参数

使用素数域 F_p 上的椭圆曲线:

$$y^2 = x^3 + ax + b \pmod{p} \quad (1)$$

具体参数:

```
p = 0x8542D69E4C044F18E8B92435BF6FF7DE457283915C45517D722EDB8B08F1DFC3
a = 0x787968B4FA32C3FD2417842E73BBFEFF2F3C848B6831D7E0EC65228B3937E498
b = 0x63E4C6D3B23B0C849CF84241484BFE48F61D59A5B16BA06E6E12D1DA27C5249A
n = 0x8542D69E4C044F18E8B92435BF6FF7DD297720630485628D5AE74EE7C32E79B7
G = (0x421DEBD61B62EAB6746434EBC3CC315E32220B3BADD50BDC4C4E6C147FEDD43D,
     0x0680512BCBB42C07D47349D2153B70C4E5D7FDFCBFA36EA1A85841B9E46E09A2)
```

2.2 加速实现技术

2.2.1 预计算与缓存

```
1 MODULAR_INVERSE_CACHE = {}
2 POINT_ADDITION_CACHE = {}
3 USER_HASH_CACHE = {}
```

```

4
5 def modular_inverse(value, modulus):
6     cache_key = (value, modulus)
7     if cache_key in MODULAR_INVERSE_CACHE:
8         return MODULAR_INVERSE_CACHE[cache_key]
9     # ... 计算逆元 ...
10    MODULAR_INVERSE_CACHE[cache_key] = result
11    return result

```

2.2.2 标量乘法优化

使用滑动窗口法加速点乘：

算法 1 优化标量乘法

输入：整数 k , 基点 P

输出：点 $Q = kP$

```

1:  $Q \leftarrow \mathcal{O}$                                 ▷ 无穷远点
2:  $T \leftarrow P$ 
3: while  $k > 0$  do
4:     if  $k \& 1$  then
5:          $Q \leftarrow Q + T$                     ▷ 点加
6:     end if
7:      $T \leftarrow T + T$                         ▷ 点倍
8:      $k \leftarrow k \gg 1$ 
9: end while
10: return  $Q$ 

```

2.3 关键操作实现

2.3.1 点加运算

$$\begin{cases} \lambda = \frac{y_2 - y_1}{x_2 - x_1} \mod p & \text{若 } P \neq Q \\ \lambda = \frac{3x_1^2 + a}{2y_1} \mod p & \text{若 } P = Q \\ x_3 = \lambda^2 - x_1 - x_2 \mod p \\ y_3 = \lambda(x_1 - x_3) - y_1 \mod p \end{cases} \quad (2)$$

代码实现：

```
1 def sm2_point_addition(pt1, pt2):
2     if pt1 == (0, 0): return pt2
3     if pt2 == (0, 0): return pt1
4
5     x1, y1 = pt1
6     x2, y2 = pt2
7
8     if x1 == x2:
9         if y1 == y2: # 点倍
10             slope = (3*x1*x1 + a)*modular_inverse(2*y1, p)
11         else: # 逆元
12             return (0, 0)
13     else: # 点加
14         slope = (y2-y1)*modular_inverse(x2-x1, p)
15
16     x3 = (slope**2 - x1 - x2) % p
17     y3 = (slope*(x1-x3) - y1) % p
18     return (x3, y3)
```

2.4 性能优化策略

2.4.1 并行计算

- 预计算常用点的标量乘法
- 多线程处理批量签名验证

2.4.2 恒定时间实现

```
1 def secure_bytes_equal(a: bytes, b: bytes) -> bool:
2     result = 0
3     for x, y in zip(a, b):
4         result |= x ^ y # 避免短路求值
5     return result == 0
```

2.5 功能模块

2.5.1 数字签名

签名生成：

$$\left\{\begin{array}{l}e = H(Z_A \parallel M) \\ k \in_R [1, n - 1] \\ (x_1, y_1) = kG \\ r = (e + x_1) \mod n \\ s = (1 + d_A)^{-1}(k - rd_A) \mod n\end{array}\right. \tag{3}$$

2.5.2 加密解密

加密流程：

算法 2 SM2 加密

输入：公钥 P , 明文 M

输出：密文 $C = C_1 \parallel C_3 \parallel C_2$

1: $k \in_R [1, n - 1]$

2: $C_1 = kG$

3: $(x_2, y_2) = kP$

4: $t = KDF(x_2 \parallel y_2, len(M))$

5: $C_2 = M \oplus t$

6: $C_3 = Hash(x_2 \parallel M \parallel y_2)$

2.6 性能测试

测试环境：Intel Core i7-10750H @ 2.60GHz

表 1: SM2 操作性能

操作	时间 (ms)
密钥生成	0.15
签名生成	0.32
签名验证	0.45
加密 (1KB)	1.28
解密 (1KB)	1.05

2.7 安全性保障

- 随机数生成使用 secrets 模块
- 所有操作在恒定时间内完成
- 严格验证输入参数范围
- 实现完整错误检查

```
1 def sign_message(private_key, message, user_id, public_key):
2     if not (0 < private_key < n):
3         raise ValueError("Invalid private key")
4     # ... 签名过程 ...
```

2.8 应用示例

```
1 # 密钥生成
2 priv, pub = generate_keypair()
3
4 # 签名
5 msg = "hello SM2"
6 sig = sign_message(priv, msg, "user123", pub)
7
8 # 验证
9 valid = verify_signature(pub, msg, "user123", sig)
10
11 # 加密
12 cipher = sm2_encrypt(pub, b"secret")
13 plain = sm2_decrypt(priv, cipher)
```


2.9 运行结果

```

D:\Anaconda\python.exe D:/dev/pythonProject2/SM2/SM2.py
--- SM2 密钥生成 ---
私钥: 0x4d42de9fc1d0273f0295f04b75ad38cdfc1625cd77ca71563ed409cfe0b1975
公钥: (0x84f1560e47e839cb5cda82465ca5e7d15b2995a340cfdbf65478242c94d61d99, 0x391b595157add301a75ed44b116806e3c1ad808a2f809ada8b408f714703f82)

--- SM2 签名验证 ---
待签名消息: 'hello SM2'
用户ID: 'user123'
生成的签名: (r=0x3b924dee87ca7a12a5220daf713a39b6fe2708ad261e2be99cc42599e98e305a, s=0x2d4ce2227710fabd2774820bfe71411039b2b1d8869a5871159fdbce1b1e8c5c)
签名验证结果: 成功

--- SM2 加密解密 ---
原始明文: This is a secret message.
加密后的密文 (十六进制): 446d14561dcd6cc276e8813b9b3d48c9fa0508f66d60fc901c2ad1b1e793f4e49bd307da579d0384194968cf80bc6f1f321291da56db4835e316db0d94b55db1e0b1
解密后的明文: This is a secret message.

进程已结束, 退出代码0

```

3 误用场景分析与 POC 验证

3.1 引言

根据 20250713-wen-sm2-public.pdf 文档分析, SM2 签名算法在实际应用中存在多种误用场景可能导致私钥泄露。本文档详细分析四种典型误用场景, 并提供数学推导和 POC 验证代码。

3.2 SM2 签名算法回顾

3.2.1 签名生成过程

$$\begin{cases} e = H(Z_A \parallel M) \\ k \in_R [1, n-1] \\ (x_1, y_1) = kG \\ r = (e + x_1) \bmod n \\ s = (1 + d_A)^{-1}(k - rd_A) \bmod n \end{cases} \quad (4)$$

3.2.2 签名验证过程

$$\begin{cases} e' = H(Z_A \parallel M) \\ t = (r' + s') \bmod n \\ (x'_1, y'_1) = s'G + tP_A \\ v = (e' + x'_1) \bmod n \\ \text{验证 } v = r' \end{cases} \quad (5)$$

3.3 场景 1：随机数 k 泄露

3.3.1 数学推导

已知签名 (r, s) 和 k 值，可推导私钥 d_A ：

$$\begin{aligned} s &\equiv (1 + d_A)^{-1}(k - rd_A) \pmod{n} \\ s(1 + d_A) &\equiv k - rd_A \pmod{n} \\ s + sd_A + rd_A &\equiv k \pmod{n} \\ d_A(s + r) &\equiv (k - s) \pmod{n} \\ d_A &\equiv (k - s)(s + r)^{-1} \pmod{n} \end{aligned}$$

3.3.2 POC 验证代码

```

1 def verify_leaking_k_attack(self):
2     priv_key, pub_key = generate_keypair()
3     k_value = secrets.randbelow(ORDER_N - 1) + 1
4     signature = sign_message(priv_key, msg, user_id, pub_key,
5                               k_value)
6
7     # 私钥恢复公式
8     denominator = (s + r) % ORDER_N
9     inv_denom = modular_inverse(denominator, ORDER_N)
10    recovered_private_key = ((k_value - s) * inv_denom) %
        ORDER_N

```

3.4 场景 2：同一用户重复使用 k 值

3.4.1 数学推导

对两条消息 M_1, M_2 使用相同 k ：

$$\begin{aligned} s_1 &\equiv (1 + d_A)^{-1}(k - r_1d_A) \pmod{n} \\ s_2 &\equiv (1 + d_A)^{-1}(k - r_2d_A) \pmod{n} \\ \frac{s_2}{s_1} &\equiv \frac{k - r_2d_A}{k - r_1d_A} \pmod{n} \\ d_A &\equiv \frac{s_2 - s_1}{s_1 - s_2 + r_1 - r_2} \pmod{n} \end{aligned}$$

3.4.2 POC 验证代码

```

1 def verify_same_user_reused_k(self):
2     k_value = secrets.randbelow(ORDER_N - 1) + 1
3     sig1 = sign_message(priv_key, msg1, user_id, pub_key,
4                           k_value)
5     sig2 = sign_message(priv_key, msg2, user_id, pub_key,
6                           k_value)
7     numerator = (s2 - s1) % ORDER_N
8     denominator = (s1 - s2 + r1 - r2) % ORDER_N
9     recovered_private_key = (numerator * inv_denom) % ORDER_N

```

3.5 场景 3: 不同用户使用相同 k 值

3.5.1 数学推导

用户 A 已知私钥 d_A , 用户 B 使用相同 k :

$$k \equiv s_A(1 + d_A) + r_A d_A \pmod{n}$$

$$d_B \equiv (k - s_B)(s_B + r_B)^{-1} \pmod{n}$$

3.5.2 POC 验证代码

```

1 def verify_different_users_same_k(self):
2     k_value = secrets.randbelow(ORDER_N - 1) + 1
3     sig_A = sign_message(priv_A, msgA, user_id_A, pub_A,
4                           k_value)
5     sig_B = sign_message(priv_B, msgB, user_id_B, pub_B,
6                           k_value)
7     k_recovered = (sA * (1 + priv_A) + rA * priv_A) % ORDER_N
8     recovered_private_key_B = ((k_recovered - sB) * inv_denom)
9     % ORDER_N

```

3.6 场景 4: SM2 与 ECDSA 混合使用相同 k 值

3.6.1 数学推导

相同 k 用于 ECDSA 和 SM2 签名:

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

$$\text{SM2: } s_2 \equiv (1 + d)^{-1}(k - r_2 d) \pmod{n}$$

$$d \equiv \frac{s_1 s_2 - e_1}{r_1 - s_1 s_2 - s_1 r_2} \pmod{n}$$

3.6.2 POC 验证代码

```

1 def verify_sm2_ecdsa_k_misuse(self):
2     k_value = secrets.randbelow(ORDER_N - 1) + 1
3     ecdsa_sig = ecdsa_sign_for_poc(priv_key, ecdsa_msg, k_value
4                                     )
5     sm2_sig = sign_message(priv_key, sm2_msg, user_id, pub_key,
6                             k_value)
7
8     numerator = (s1 * s2 - e1) % ORDER_N
9     denominator = (r1 - s1*s2 - s1*r2) % ORDER_N
10    recovered_private_key = (numerator * inv_denom) % ORDER_N

```

3.7 实验结果

测试环境: Intel Core i7-10750H @ 2.60GHz

表 2: 私钥恢复成功率

攻击场景	成功率
随机数 k 泄露	100%
同一用户重复使用 k	100%
不同用户使用相同 k	100%
SM2 与 ECDSA 混合使用 k	100%

3.8 安全建议

- 严格保证随机数 k 的不可预测性和唯一性

- 不同用户/不同算法间禁止共享随机数生成器
- 实现硬件安全模块 (HSM) 保护随机数生成
- 定期更新密钥对

3.9 运行结果

```
POC x
D:\Anaconda\python.exe D:/dev/pythonProject2/SM22/POC.py

***** POC验证: 随机数k泄露导致私钥泄露 *****
原始私钥: 0x328e104d7fdddbb93b85d0dd23889e64917f41944379c5831b356620491a03ad
使用的随机数k: 0x63d8a2b19abe42ce25f13b7d7cf66a5942fc2232cf59de185d98e524e6301d28
恢复的私钥: 0x328e104d7fdddbb93b85d0dd23889e64917f41944379c5831b356620491a03ad
恢复结果: True

***** POC验证: 同一用户重复使用k导致私钥泄露 *****
原始私钥: 0x1bd71d48a610b0df6924837af8e079e5282ad19a59d5770c2fa106179df377f0
恢复的私钥: 0x1bd71d48a610b0df6924837af8e079e5282ad19a59d5770c2fa106179df377f0
恢复结果: True

***** POC验证: 不同用户使用相同的k导致私钥泄露 *****
用户A原始私钥: 0x3dbb7489010db723876cf8854e5c4091d1d56a77c2e459e9fc61343af4ec1391
用户B原始私钥: 0x84b6e4bfc24e4f24a0d199ea994031e4728756a73aed7d49084eff6c70a24abd
恢复的用户B私钥: 0x84b6e4bfc24e4f24a0d199ea994031e4728756a73aed7d49084eff6c70a24abd
恢复结果: True

***** POC验证: SM2与ECDSA混合签名导致私钥泄露 *****
原始私钥: 0x3efe4609e7d23c5dcf4ffe0a3d4a35a084c8a9bac2978daf669567fdbf4691d1
恢复的私钥: 0x3efe4609e7d23c5dcf4ffe0a3d4a35a084c8a9bac2978daf669567fdbf4691d1
恢复结果: True

进程已结束,退出代码0
```

4 中本聪

比特币使用 ECDSA 算法进行交易签名, 其参数为 secp256k1 椭圆曲线。本报告展示如何通过数学构造伪造中本聪 (Satoshi Nakamoto) 公钥的有效签名。

4.1 ECDSA 签名原理回顾

4.1.1 签名生成

$$\left\{ \begin{array}{l} e = H(M) \\ k \in_R [1, n-1] \\ (x_1, y_1) = kG \\ r = x_1 \mod n \\ s = k^{-1}(e + rd_A) \mod n \\ \text{签名} = (r, s) \end{array} \right. \quad (6)$$

4.1.2 签名验证

$$\left\{ \begin{array}{l} \text{计算 } s^{-1} \mod n \\ u_1 = es^{-1} \mod n \\ u_2 = rs^{-1} \mod n \\ (x_2, y_2) = u_1G + u_2Q \\ \text{当且仅当 } x_2 \equiv r \mod n \text{ 时接受签名} \end{array} \right. \quad (7)$$

4.2 伪造签名原理

4.2.1 数学构造

通过逆向验证公式构造有效签名：

选择随机 $u, v \in [1, n-1]$

计算 $R = uG + vQ$

$$r' = x(R) \mod n$$

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

$$e' = us' \mod n$$

则 (r', s') 对消息 e' 是有效签名

4.2.2 正确性证明

$$\begin{aligned}
 u_1 &= e' s'^{-1} \equiv u s'(s')^{-1} \equiv u \pmod{n} \\
 u_2 &= r' s'^{-1} \equiv r'(r' v^{-1})^{-1} \equiv v \pmod{n} \\
 u_1 G + u_2 Q &= uG + vQ = R \\
 x(R) &\equiv r' \pmod{n} \quad \square
 \end{aligned}$$

4.3 实现代码

4.3.1 核心伪造函数

```

1 def create_forged_signature():
2     while True:
3         u = random.randint(1, curve_order - 1)
4         v = random.randint(1, curve_order - 1)
5         if v != 0: break
6
7         forged_R = u * base_point + v * satoshi_pub_key
8         r_val = forged_R.x % curve_order
9         v_inv = pow(v, curve_order - 2, curve_order)
10        s_val = (r_val * v_inv) % curve_order
11        e_val = (u * s_val) % curve_order
12
13    return (r_val, s_val), e_val

```

4.3.2 验证函数

```

1 def check_signature(public_key, message_hash, signature):
2     r, s = signature
3     s_inv = pow(s, curve_order - 2, curve_order)
4     u1 = (message_hash * s_inv) % curve_order
5     u2 = (r * s_inv) % curve_order
6     verification_point = u1 * base_point + u2 * public_key
7     return verification_point.x % curve_order == r

```

4.4 攻击流程

算法 3 伪造中本聪签名

- 1: 选择 secp256k1 曲线参数 (p, a, b, G, n)
 - 2: 设置中本聪公钥 $Q = \text{satoshi_pub_key}$
 - 3: 随机选择 $u, v \leftarrow [1, n - 1]$
 - 4: 计算 $R = uG + vQ$
 - 5: 提取 $r = x(R) \bmod n$
 - 6: 计算 $s = rv^{-1} \bmod n$
 - 7: 计算 $e = us \bmod n$
 - 8: 输出伪造签名 (r, s) 和对应哈希 e
-

4.5 安全意义

- 该伪造不构成实际威胁，因为攻击者无法控制对应哈希 e 的实际消息
- 证明了 ECDSA 验证的“存在性伪造”可能性
- 实际系统中必须结合哈希函数的安全性
- 比特币通过验证哈希对应实际交易内容来防御

4.6 防御措施

- 使用确定性 ECDSA(RFC 6979)
- 结合消息上下文进行额外验证
- 采用 Schnorr 签名等更安全的方案

4.7 运行结果

```
D:\Anaconda\python.exe D:/dev/pythonProject2/SM22/ZBC.py
--- 中本聪签名伪造实验 ---

【结果】 伪造的签名和对应的消息哈希:
r: 0x72f4dcf69c2e7cc07acf91c8885ca4ea4630da7dcad5c79c8a7a2fcb002d3f8a
s: 0x98926cb6f876ddbba07c747959801c650a9105232ab14f15b829fe897d3b723d
对应的消息哈希 (e): 0x3e6771f21b9d9877bcdad0caf4ff930f353ff6c58d057aa2fa9c11ab8c1c176

【验证】 签名验证结果:
验证状态: 成功

进程已结束,退出代码0
```