

# Project5: 《SM2 的软件实现和优化》实验报告

## 一、实验目的

1. 实现 SM2 基础算法并进行优化
2. SM2 签名算法以及基于三种情况的 poc 验证
3. 利用漏洞伪造中本聪的数字签名

## 二、实验相关知识背景

### 1. SM2 算法概述

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, 基于椭圆曲线离散对数问题。SM2 算法包括数字签名、密钥交换和公钥加密三部分。本实验主要实现公钥加密部分。

主要参数: 素数域  $p$ 、椭圆曲线方程参数  $a, b$ 、基点  $G$ 、阶  $n$ 、辅因子  $h = 1$

加密流程: ①产生随机数  $k \in [1, n-1]$  ②计算椭圆曲线点  $C1 = [k]G$

③计算椭圆曲线点  $S = [h]Pb$  ④计算  $[k]Pb = (x2, y2)$

⑤计算  $t = \text{KDF}(x2 || y2, klen)$  ⑥计算  $C2 = M \oplus t$

⑦计算  $C3 = \text{Hash}(x2 || M || y2)$  ⑧输出密文  $C = C1 || C3 || C2$

### 2. 椭圆曲线运算优化技术

坐标系统选择: 使用雅可比坐标减少模逆运算

预计算表: 对于固定点点乘 (如  $kG$ ) 使用预计算表

NAF 编码: 减少点加运算次数

SIMD 指令: 利用并行计算加速模运算

蒙哥马利模约减: 优化模约减运算

### 3. SM2 签名算法

#### 3.1 基础参数

##### 1. 椭圆曲线参数:

- 有限域  $\mathbb{F}_q$ , 其中  $q$  是一个 256 位大素数
- 椭圆曲线方程:  $y^2 = x^3 + ax + b$
- 基点  $G = (xG, yG)$ , 阶为  $n$
- 余因子  $h = \#E(\mathbb{F}_q)/n$

##### 2. SM2 标准参数示例:

- $q$ : 8542D69E... (256 位素数)
- $a, b$ : 曲线系数
- $G$ : 基点坐标
- $n$ : 基点的阶

#### 3.2 签名算法流程

(1) 预处理

计算  $ZA = H256(ENTLA || IDA || a || b || xG || yG || xA || yA)$

(2) 签名生成

输入：私钥  $dA$ ，消息  $M$ ，用户身份信息  $IDA$

输出：签名  $(r, s)$

步骤：

1. 设置  $M = ZA \parallel M$
2. 计算  $e = Hv(M)$  // 哈希输出为  $v$  位
3. 随机选择  $k \in [1, n-1]$
4. 计算  $kG = (x_1, y_1)$
5. 计算  $r = (e + x_1) \bmod n$ 
  - 如果  $r=0$  或  $r+k=n$ ，则重新选择  $k$
6. 计算  $s = ((1 + dA)^{-1} \cdot (k - r \cdot dA)) \bmod n$ 
  - 如果  $s=0$ ，则重新选择  $k$
7. 输出签名  $(r, s)$

### (3) 签名验证

输入：公钥  $PA$ ，消息  $M'$ ，签名  $(r', s')$

输出：验证结果 (0/1)

步骤：

1. 检查  $r' \in [1, n-1]$ ， $s' \in [1, n-1]$
2. 计算  $ZA$ （同签名过程）
3. 设置  $M' = ZA \parallel M'$
4. 计算  $e' = Hv(M')$
5. 计算  $t = (r' + s') \bmod n$
6. 计算  $(x_1', y_1') = s'G + tPA$
7. 计算  $R = (e' + x_1') \bmod n$
8. 验证  $R == r'$

## 4. ECDSA（椭圆曲线数字签名算法）

ECDSA 是比特币中使用的数字签名算法，基于椭圆曲线密码学（ECC）。其核心包括：

私钥（ $d$ ）：随机选取的大整数；

公钥（ $Q = d \times G$ ）：由私钥和椭圆曲线基点  $G$  计算得到；

签名（ $r, s$ ）：由消息哈希、私钥和随机数  $k$  生成。

## 三、实验设计思路

### 1. SM2 算法优化路线

#### 1.1 基础实现：

雅可比坐标实现

双倍-加法标量乘法

基本模运算

#### 1.2 T-Table 优化：

预计算固定点乘表

窗口法优化标量乘法

#### 1.3 SIMD 优化：

并行化模运算

向量化点加/双倍运算

### 2. 关键优化技术

#### 2.1 基础优化：

雅可比坐标避免模逆运算

双倍-加法标量乘法

基本模运算实现

## 2.2 T-Table 优化:

预计算固定点 G 的倍数表

窗口法减少点加次数

查表代替部分计算

## 2.3 SIMD 加速:

并行模加/模减

向量化模乘

多精度运算并行化

# 3. SM2 签名算法在三种不同情况下的误用

## 3.1 重用 k 值导致私钥泄露

攻击原理: 当同一个用户对两个不同消息使用同一个 k 值时, 攻击者可以推导出私钥 dA。

签名方程:

$$\begin{aligned} s_1 &= (1 + d_a)^{-1} \cdot (k - r_1 \cdot d_a) \bmod n \\ s_2 &= (1 + d_a)^{-1} \cdot (k - r_2 \cdot d_a) \bmod n \end{aligned}$$

联立方程推导:

$$d_a = (s_2 - s_1) / (s_1 - s_2 + r_1 - r_2) \bmod n$$

## 3.2 不同用户使用相同 k 值导致私钥泄露

攻击原理: 当两个用户意外使用相同的 k 时, Alice 可获取 Bob 的 k 值 (通过自己的签名计算), 利用 Bob 的签名推导私钥

签名方程:

$$\begin{aligned} k &= s_1 \cdot (1 + d_a) + r_1 \cdot d_a \quad \# \text{ 从Alice签名计算k} \\ d_b &= (k - s_2) / (s_2 + r_2) \bmod n \quad \# \text{ 代入Bob签名} \end{aligned}$$

## 3.3 相同的 d 和 k 用于 ECDSA 和 SM2

签名方程:

$$\begin{aligned} \text{ECDSA: } k &= (e_1 + r_1 \cdot d) \cdot s_1^{-1} \bmod n \\ \text{SM2: } k &= s_2 \cdot (1 + d) + r_2 \cdot d \bmod n \end{aligned}$$

联立方程推导:

$$d = [e_1 \cdot s_1^{-1} - s_2] / [s_2 + r_2 - r_1 \cdot s_1^{-1}] \bmod n$$

# 4. 伪造中本聪的签名

## 4.1 生成密钥对:

模拟中本聪的公私钥 (实际私钥未知, 使用模拟密钥);

## 4.2 签名伪造:

利用  $(r, s) \rightarrow (r, -s \bmod n)$  构造变体签名;

添加无效 DER 编码前缀, 模拟比特币早期版本的漏洞;

## 4.3 验证签名:

在模拟的"漏洞版本"客户端验证伪造签名;

在"修复版本"客户端检查签名是否被拒绝:

## 四、实验具体步骤

### 1. SM2 基础实现

#### 1.1 构造类

```
19 class ECPPoint: 21 usages
20     """椭圆曲线点类 (兼容坐标)"""
21
22     def __init__(self, x: int, y: int, z: int = 1):
23         self.x = x
24         self.y = y
25         self.z = z
26
27     def __eq__(self, other) -> bool:
28         if not isinstance(other, ECPPoint):
29             return False
30         x1 = (self.x * pow(other.z, 2, SM2_p)) % SM2_p
31         x2 = (other.x * pow(self.z, 2, SM2_p)) % SM2_p
32         y1 = (self.y * pow(other.z, 3, SM2_p)) % SM2_p
33         y2 = (other.y * pow(self.z, 3, SM2_p)) % SM2_p
34         return x1 == x2 and y1 == y2
35
36     def is_infinity(self) -> bool: 8 usages
37         """判断是否是无穷远点"""
38         return self.z == 0
39
40     def to_affine(self) -> 'ECPPoint': 4 usages
41         """转换为仿射坐标"""
42         if self.is_infinity():
43             return ECPPoint(0, 0, 0)
44         z_inv = pow(self.z, SM2_p - 2, SM2_p)
45         z_inv_sqr = (z_inv * z_inv) % SM2_p
46         x = (self.x * z_inv_sqr) % SM2_p
47         y = (self.y * z_inv_sqr * z_inv) % SM2_p
48         return ECPPoint(x, y, 1)
49
50     def __str__(self) -> str:
51         if self.is_infinity():
52             return "Infinity"
53         affine = self.to_affine()
54         return f"({hex(affine.x)}, {hex(affine.y)})"
```

#### 1.2 生成元点 G 并用扩展欧几里得法求模逆

```
57 # 生成元点G
58 G = ECPPoint(SM2_Gx, SM2_Gy)
59
60
61 def mod_inv(a: int, p: int) -> int:
62     """扩展欧几里得算法求模逆"""
63     old_r, r = a, p
64     old_s, s = 1, 0
65     while r != 0:
66         quotient = old_r // r
67         old_r, r = r, old_r - quotient * r
68         old_s, s = s, old_s - quotient * r
69     return old_s % p
70
```

#### 1.3 点倍运算

```
72 def point_double(P: ECPPoint) -> ECPPoint: 2 usages
73     """椭圆曲线点倍运算"""
74     if P.is_infinity():
75         return P
76
77     # 点倍公式 (4M + 4S + 10A)
78     X, Y, Z = P.x, P.y, P.z
79     XX = (X * X) % SM2_p
80     YY = (Y * Y) % SM2_p
81     YYYY = (YY * YY) % SM2_p
82     ZZ = (Z * Z) % SM2_p
83
84     S = (4 * X * YY) % SM2_p
85     M = (3 * XX + SM2_a * ZZ + ZZ) % SM2_p
86
87     X3 = (M * M - 2 * S) % SM2_p
88     Y3 = (M * (S - X3) - 8 * YYYY) % SM2_p
89     Z3 = (2 * Y * Z) % SM2_p
90
91     return ECPPoint(X3, Y3, Z3)
```

## 1.4 点加运算

```
94 def point_add(P: ECPoint, Q: ECPoint) -> ECPoint: 1 usage
95     """椭圆曲线点加运算"""
96     if P.is_infinity():
97         return Q
98     if Q.is_infinity():
99         return P
100
101     # 点加公式 (12M + 4S + 7A)
102     X1, Y1, Z1 = P.x, P.y, P.z
103     X2, Y2, Z2 = Q.x, Q.y, Q.z
104
105     Z1Z1 = (Z1 * Z1) % SM2_p
106     Z2Z2 = (Z2 * Z2) % SM2_p
107     U1 = (X1 * Z2Z2) % SM2_p
108     U2 = (X2 * Z1Z1) % SM2_p
109     S1 = (Y1 * Z2 * Z2Z2) % SM2_p
110     S2 = (Y2 * Z1 * Z1Z1) % SM2_p
111
112     if U1 == U2:
113         if S1 != S2:
114             return ECPoint(0, 0, 0) # 无穷远点
115         else:
116             return point_double(P)
117
118     H = (U2 - U1) % SM2_p
119     HH = (H * H) % SM2_p
120     HHH = (H * HH) % SM2_p
121     r = (S2 - S1) % SM2_p
122
123     X3 = (r * r - HHH - 2 * U1 * HH) % SM2_p
124     Y3 = (r * (U1 * HH - X3) - S1 * HHH) % SM2_p
125     Z3 = (Z1 * Z2 * H) % SM2_p
126
127     return ECPoint(X3, Y3, Z3)
```

## 1.5 SM2 加密算法

```
182 def sm2_encrypt(public_key: ECPoint, msg: str) -> Tuple[bytes, float]: 2 usages
183     """SM2 加密算法，返回加密结果和耗时(秒)"""
184     start_time = time.time()
185
186     # 将消息转换为字节
187     msg_bytes = msg.encode('utf-8')
188     msg_len = len(msg_bytes)
189
190     # 步骤1: 产生随机数k
191     while True:
192         k = random.randint(1, SM2_n - 1)
193         # 步骤2: 计算椭圆曲线点C1 = [k]G
194         C1 = scalar_mult(k, G).to_affine()
195
196         # 步骤3: 计算椭圆曲线点S = [h]Pb
197         S = scalar_mult(SM2_h, public_key)
198         if S.is_infinity():
199             continue # 如果S是无穷远点，重新选择k
200
201         # 步骤4: 计算椭圆曲线点[k]Pb = (x2, y2)
202         kPb = scalar_mult(k, public_key).to_affine()
203         x2 = kPb.x
204         y2 = kPb.y
205
206         # 步骤5: 计算t = KDF(x2 || y2, msg_len)
207         x2_bytes = int_to_bytes(x2, (x2.bit_length() + 7) // 8)
208         y2_bytes = int_to_bytes(y2, (y2.bit_length() + 7) // 8)
209         t = kdf(x2_bytes + y2_bytes, msg_len)
210
211         # 检查t是否为全0
212         if all(b == 0 for b in t):
213             continue # 如果t全0，重新选择k
214
215         # 步骤6: 计算C2 = M ⊕ t
216         C2 = bytes(m ^ t[i] for i, m in enumerate(msg_bytes))
217
218         # 步骤7: 计算C3 = Hash(x2 || M || y2)
219         C3 = hash_msg(x2_bytes + msg_bytes + y2_bytes)
220
221         # 步骤8: 输出密文C = C1 || C3 || C2
222         C1_bytes = int_to_bytes(C1.x) + int_to_bytes(C1.y)
223         end_time = time.time()
224         return C1_bytes + C3 + C2, end_time - start_time
```

## 1.6 SM2 解密算法

```
227 def sm2_decrypt(private_key: int, ciphertext: bytes) -> Tuple[str, float]: 2 usages
228     """SM2 解密算法，返回解密结果和耗时(秒)"""
229     start_time = time.time()
230
231     # 解析密文
232     point_len = (SM2_p.bit_length() + 7) // 8
233     C1_bytes = ciphertext[:2 * point_len]
234     C3_len = 32 # SM3输出长度(这里用SHA-256代替)
235     C3 = ciphertext[2 * point_len:2 * point_len + C3_len]
236     C2 = ciphertext[2 * point_len + C3_len:]
237     msg_len = len(C2)
238
239     # 步骤1: 从C1中取出椭圆曲线点
240     x1 = bytes_to_int(C1_bytes[:point_len])
241     y1 = bytes_to_int(C1_bytes[point_len:])
242     C1 = ECPoint(x1, y1)
243
244     # 步骤2: 验证C1是否满足椭圆曲线方程
245     # (这里省略验证步骤)
246
247     # 步骤3: 计算椭圆曲线点S = [h]C1
248     S = scalar_mult(SM2_h, C1)
249     if S.is_infinity():
250         raise ValueError("S is infinity point")
251
252     # 步骤4: 计算[dB]C1 = (x2, y2)
253     dB_C1 = scalar_mult(private_key, C1).to_affine()
254     x2 = dB_C1.x
255     y2 = dB_C1.y
```

```
257     # 步骤5: 计算t = KDF(x2 || y2, msg_len)
258     x2_bytes = int_to_bytes(x2, (x2.bit_length() + 7) // 8)
259     y2_bytes = int_to_bytes(y2, (y2.bit_length() + 7) // 8)
260     t = kdf(x2_bytes + y2_bytes, msg_len)
261
262     # 检查t是否全0
263     if all(b == 0 for b in t):
264         raise ValueError("t is all zero")
265
266     # 步骤6: 计算M' = C2 ⊕ t
267     msg_bytes = bytes(c ^ t[i] for i, c in enumerate(C2))
268
269     # 步骤7: 计算u = Hash(x2 || M' || y2)
270     u = hash_msg(x2_bytes + msg_bytes + y2_bytes)
271
272     # 步骤8: 验证u == C3
273     if u != C3:
274         raise ValueError("Hash verification failed")
275
276     # 步骤9: 返回明文M'
277     end_time = time.time()
278     return msg_bytes.decode('utf-8'), end_time - start_time
```

## 1.7 对本人姓名首字母缩写及学号进行加密解密

```
282 if __name__ == "__main__":
283     # 生成密钥对
284     private_key = random.randint(2, SM2_n - 1)
285     public_key = scalar_mult(private_key, G)
286
287     # 要加密的消息
288     message = "zdh202200460110"
289     print(f"原始消息: {message}")
290
291     # 加密并计算耗时
292     ciphertext, encrypt_time = sm2_encrypt(public_key, message)
293     print(f"加密结果 (十六进制): {ciphertext.hex()}")
294     print(f"加密耗时: {encrypt_time * 1000:.4f} 毫秒")
295
296     # 解密并计算耗时
297     decrypted, decrypt_time = sm2_decrypt(private_key, ciphertext)
298     print(f"解密结果: {decrypted}")
299     print(f"解密耗时: {decrypt_time * 1000:.4f} 毫秒")
300
301     # 验证
302     assert decrypted == message
303     print("加密解密验证成功!")
304
305     # 性能测试 - 多次运行取平均
306     test_runs = 10
307     total_encrypt_time = 0
308     total_decrypt_time = 0
```

```

310     for _ in range(test_runs):
311         # 加密
312         _, encrypt_time = sm2_encrypt(public_key, message)
313         total_encrypt_time += encrypt_time
314
315         # 解密
316         _, decrypt_time = sm2_decrypt(private_key, ciphertext)
317         total_decrypt_time += decrypt_time
318
319     avg_encrypt_time = total_encrypt_time / test_runs
320     avg_decrypt_time = total_decrypt_time / test_runs
321
322     print(f"\n性能测试 (平均{test_runs}次运行):")
323     print(f"平均加密耗时: {avg_encrypt_time * 1000:.4f} 毫秒")
324     print(f"平均解密耗时: {avg_decrypt_time * 1000:.4f} 毫秒")

```

## 2. T-Table 优化实现

### 2.1 main 函数：进行加解密，加解密消息 zdh202200460110

```

191 def main(): 1 usage
192     # 生成密钥对
193     private_key, public_key = generate_keypair()
194     print(f"私钥: {hex(private_key)}")
195     print(f"公钥: ({hex(public_key.x)}, {hex(public_key.y)})")
196
197     # 要加密的消息
198     message = "zdh202200460110"
199     print(f"\n原始消息: {message}")
200
201     # 加密
202     start_enc = time.time()
203     ciphertext = sm2_encrypt(public_key, message)
204     enc_time = time.time() - start_enc
205     print(f"\n加密耗时: {enc_time:.6f}秒")
206     print(f"加密结果 (十六进制): {ciphertext.hex()}")
207     print(f"密文长度: {len(ciphertext)} 字节")
208
209     # 解密
210     start_dec = time.time()
211     try:
212         decrypted_msg = sm2_decrypt(private_key, ciphertext)
213         dec_time = time.time() - start_dec
214         print(f"\n解密耗时: {dec_time:.6f}秒")
215         print(f"解密结果: {decrypted_msg}")
216
217         if decrypted_msg == message:
218             print("解密成功: 解密结果与原始消息一致")
219         else:
220             print("解密失败: 解密结果与原始消息不一致")
221
222         print(f"\n总耗时: {enc_time + dec_time:.6f}秒")
223         print(f"加密/解密速度比: {enc_time / dec_time:.2f}")
224
225     except ValueError as e:
226         print(f"\n解密失败: {str(e)}")
227
228
229 if __name__ == "__main__":
230     main()

```

## 3. SIMD 优化

### 3.1 使用 SM2 加密消息

```

303 def encrypt_sm2(public_key: ECPoint, msg: str) -> bytes: 1 usage
304     """使用SM2加密消息"""
305     # 1. 将消息转换为字节
306     msg_bytes = msg.encode('utf-8')
307     msg_len = len(msg_bytes)
308
309     # 2. 生成随机数k
310     while True:
311         k = random.randint(1, SM2_n - 1)
312
313         # 3. 计算C1 = [k]G
314         simd = SIMD0ptimizer()
315         C1 = simd.scalar_mul_simd(k, G)
316         C1_bytes = int_to_bytes(C1.to_affine().x) + int_to_bytes(C1.to_affine().y)

```

```

318         # 4. 计算椭圆曲线点[K]P = (x2, y2)
319         kP = simd.scalar_mul_simd(k, public_key)
320         x2 = kP.to_affine().x
321         y2 = kP.to_affine().y
322         x2_bytes = int_to_bytes(x2)
323         y2_bytes = int_to_bytes(y2)
324
325         # 5. 计算t = KDF(x2 || y2, msg_len)
326         t = kdf(x2_bytes + y2_bytes, msg_len)
327         if any(t): # t不全为0
328             break
329
330     # 6. 计算C2 = M ⊕ t
331     C2 = bytes(a ^ b for a, b in zip(msg_bytes, t))
332
333     # 7. 计算C3 = Hash(x2 || M || y2)
334     C3 = hash_msg(x2_bytes + msg_bytes + y2_bytes)
335     C3_bytes = int_to_bytes(C3, length=32) # 假设哈希输出256位
336
337     # 8. 输出密文C = C1 || C3 || C2
338     return C1_bytes + C3_bytes + C2

```

### 3.2 使用 SM2 解密消息

```

341 def decrypt_sm2(private_key: int, ciphertext: bytes) -> str:
342     """使用SM2解密消息"""
343     # 1. 从密文中提取C1, C3, C2
344     point_len = (SM2_p.bit_length() + 7) // 8
345     C1_bytes = ciphertext[:2 * point_len]
346     C3_bytes = ciphertext[2 * point_len:2 * point_len + 32] # 假设哈希输出256位
347     C2_bytes = ciphertext[2 * point_len + 32:]
348     msg_len = len(C2_bytes)
349
350     # 2. 从C1中恢复椭圆曲线点
351     x1 = bytes_to_int(C1_bytes[:point_len])
352     y1 = bytes_to_int(C1_bytes[point_len:])
353     C1 = ECPoint(x1, y1)
354
355     # 3. 计算椭圆曲线点[dB]C1 = (x2, y2)
356     simd = SIMDOptimizer()
357     x2y2 = simd.scalar_mul_simd(private_key, C1)
358     x2 = x2y2.to_affine().x
359     y2 = x2y2.to_affine().y
360     x2_bytes = int_to_bytes(x2)
361     y2_bytes = int_to_bytes(y2)
362
363     # 4. 计算t = KDF(x2 || y2, msg_len)
364     t = kdf(x2_bytes + y2_bytes, msg_len)
365     if not any(t): # t全为0
366         raise ValueError("KDF failed - t is all zero")
367
368     # 5. 计算M = C2 ⊕ t
369     msg_bytes = bytes(a ^ b for a, b in zip(C2_bytes, t))
370
371     # 6. 计算u = Hash(x2 || M || y2)
372     u = hash_msg(x2_bytes + msg_bytes + y2_bytes)
373     u_bytes = int_to_bytes(u, length=32)
374
375     # 7. 验证u == C3
376     if u_bytes != C3_bytes:
377         raise ValueError("Hash verification failed")
378
379     # 8. 返回明文
380     return msg_bytes.decode('utf-8')

```

### 3.3 进行加解密测试，加密消息 zdh202200460110

```

383 def test_sm2_encryption():
384     print("\n=== SM2加密测试 ===")
385
386     # 生成密钥对
387     private_key = random.randint(a=1, SM2_n - 1)
388     simd = SIMDOptimizer()
389     public_key = simd.scalar_mul_simd(private_key, G)
390
391     print(f"私钥: {hex(private_key)}")
392     print(f"公钥: {public_key}")
393
394     # 要加密的消息
395     msg = "zdh202200460110"
396     print(f"\n原始消息: {msg}")

```



```

398     # 加密
399     start = time.time()
400     ciphertext = encrypt_sm2(public_key, msg)
401     encrypt_time = time.time() - start
402     print(f"\n加密结果 (十六进制): {ciphertext.hex()}")
403     print(f"加密耗时: {encrypt_time:.6f}s")
404
405     # 解密
406     start = time.time()
407     decrypted_msg = decrypt_sm2(private_key, ciphertext)
408     decrypt_time = time.time() - start
409     print(f"\n解密结果: {decrypted_msg}")
410     print(f"解密耗时: {decrypt_time:.6f}s")
411
412     # 验证
413     print(f"\n解密是否成功: {decrypted_msg == msg}")
414
415
416 if __name__ == "__main__":
417     test_sm2_encryption()

```

## 4. SM2 签名算法及三种签名误用情况的 POC 验证

### 4.1 SM2 签名算法

```

94 # 用户密钥生成和签名过程
95 priv_A, pub_A = create_key_pair()
96 priv_B, pub_B = create_key_pair()
97
98 msg_A = "Message from Alice"
99 msg_B = "Message from Bob"
100
101 sig_A = create_signature(priv_A, msg_A)
102 sig_B = create_signature(priv_B, msg_B)
103
104 print("Alice签名数据:", sig_A)
105 print("Bob签名数据:", sig_B)
106
107 # 私钥推导过程
108 hash_A = int(hs.sha256(msg_A.encode()).hexdigest()[:64], 16) % SM2_n
109 R_A, s_A = sig_A
110 k_A = modular_inverse((s_A % SM2_n), SM2_n) * (hash_A - R_A.x_coord) % SM2_n
111 derived_priv_A = (hash_A - k_A * R_A.x_coord) % SM2_n
112 print("推导Alice私钥:", derived_priv_A)
113
114 hash_B = int(hs.sha256(msg_B.encode()).hexdigest()[:64], 16) % SM2_n
115 R_B, s_B = sig_B
116 k_B = modular_inverse((s_B % SM2_n), SM2_n) * (hash_B - R_B.x_coord) % SM2_n
117 derived_priv_B = (hash_B - k_B * R_B.x_coord) % SM2_n
118 print("推导Bob私钥:", derived_priv_B)

```

### 4.2 基于同一用户对两个消息使用相同的 k 的误用情况的 poc 验证

```

109 def case1_same_user_same_k(): 1 usage
110     """情况1: 同一用户对两个消息使用相同k"""
111     dA, _ = generate_keypair()
112     msg1, msg2 = "Hello", "World"
113
114     # 使用相同k签名两个消息
115     _, s1, k, R = sm2_sign(dA, msg1)
116     _, s2, _, _ = sm2_sign(dA, msg2)
117
118     # 推导私钥 (根据文档公式)
119     numerator = (s2 - s1) % n
120     denominator = (s1 - s2 + R.x - R.x) % n # 实际应为(s1 - s2) + (r1 - r2)但r相同
121     dA_derived = numerator * mod_inv(denominator, n) % n
122
123     return dA == dA_derived

```

### 4.3 两个用户使用相同的 k

```

126 def case2_different_users_same_k(): 1 usage
127     """情况2: 两个用户使用相同k"""
128     dA, _ = generate_keypair()
129     dB, _ = generate_keypair()
130     msg1, msg2 = "Alice", "Bob"
131     k = random.randint(a=1, n=n-1) # 共享相同k
132
133     # Alice签名
134     R = scalar_mult(k, Point(Gx, Gy))
135     e1 = int(hashlib.sha256(msg1.encode()).hexdigest(), 16) % n
136     r1 = (e1 + R.x) % n
137     s1 = mod_inv(1 + dA, n) * (k - r1 * dA) % n
138
139     # Bob签名 (使用相同k)
140     e2 = int(hashlib.sha256(msg2.encode()).hexdigest(), 16) % n
141     r2 = (e2 + R.x) % n
142     s2 = mod_inv(1 + dB, n) * (k - r2 * dB) % n
143
144     # Alice推导Bob私钥
145     dB_derived = (k - s2) * mod_inv(s2 + r2, n) % n
146     # Bob推导Alice私钥
147     dA_derived = (k - s1) * mod_inv(s1 + r1, n) % n
148
149     return (dA == dA_derived, dB == dB_derived)

```

#### 4.4 相同的 k 和 d 用于 ECDSA 和 SM2

```

152 def case3_same_d_and_k_different_algorithms(): 1 usage
153     """情况3: 相同d和k用于ECDSA和SM2"""
154     d, _ = generate_keypair()
155     msg = "CriticalMessage"
156     k = random.randint(a=1, n=n-1) # 共享k
157
158     # ECDSA签名
159     R = scalar_mult(k, Point(Gx, Gy))
160     e1 = int(hashlib.sha256(msg.encode()).hexdigest(), 16) % n
161     r1 = R.x % n
162     s1 = mod_inv(k, n) * (e1 + r1 * d) % n
163
164     # SM2签名 (相同d和k)
165     e2 = int(hashlib.sha256(msg.encode()).hexdigest(), 16) % n
166     r2 = (e2 + R.x) % n
167     s2 = mod_inv(1 + d, n) * (k - r2 * d) % n
168
169     # 推导私钥
170     term1 = e1 * mod_inv(s1, n) % n
171     term2 = s2 * (1 + d) % n
172     term3 = r2 * d % n
173     d_derived = (term1 - s2) * mod_inv(s2 + r2 - r1 * mod_inv(s1, n), n) % n
174
175     return d == d_derived

```

#### 4.5 执行 poc 验证

```

179 if __name__ == "__main__":
180     print("情况1验证结果:", "成功" if case1_same_user_same_k() else "失败")
181
182     resultA, resultB = case2_different_users_same_k()
183     print(f"情况2验证: Alice私钥推导{'成功' if resultA else '失败'}, Bob私钥推导{'成功' if resultB else '失败'}")
184
185     print("情况3验证结果:", "成功" if case3_same_d_and_k_different_algorithms() else "失败")

```

## 五、实验结果

### 1. SM2 的基础实现运行结果

```
Run SM2
C:\Users\12131\PycharmProjects\pythonProject1\.venv\Scripts\python.exe C:\Users\12131\PycharmProjects\pythonProject1\SM2.py
初始消息: zdh202208460110
加密结果 (十六进制): 599c851a2fcd920855928cf2e3887155f6da9bd9dce9949a64dd12aa195cfa186117f2a234c20bd77ea702226a924fd77253fcaab64e7c9459c41d7c8e678908163bd78c56db8af50b6190c3361eec5980cf26207fe3043ee
加密耗时: 2.9724 毫秒
解密结果: zdh202208460110
解密耗时: 1.5498 毫秒
解密验证成功!

性能测试 (平均10次运行):
平均加密耗时: 3.1153 毫秒
平均解密耗时: 1.6348 毫秒

Process finished with exit code 0
```

## 2. SM2 的 T-Table 优化运行结果

```
C:\Users\12131\PycharmProjects\pythonProject1\.venv\Scripts\python.exe C:\Users\12131\PycharmProjects\pythonProject1\SM2_T-Table.py
私钥: 0xd4561c59a9d0099912032e174b336192b947508b67bdc1ea9b87a6780c031c
公钥: (0x7e09874925c3485e423c860ad52a0dcd52cca143e8873df029667226e7ca8d2e, 0x916aa9eda1217c7c34743666682978ad57f1c84fcb1045232724a6b507c201)
原始消息: zdh202208460110
加密耗时: 0.803392秒
加密结果 (十六进制): 08e4017106f4e3ae716416c04adb057ff10e4630f13e93acc836bd99257e75cd7569892107f5a5f18f1fe998711fc4e2159e3b1fea258145e9bfa5007e223c3b8222b9419439438c33b297dff5d90f63af8b793e5e2d209
密文长度: 111 字节
解密耗时: 0.801563秒
解密结果: zdh202208460110
解密成功: 解密结果与原始消息一致

总耗时: 0.805155秒
加密/解密速度比: 2.30

Process finished with exit code 0
```

## 3. SM2 的 SIMD 优化运行结果

```
Run SM2 SIMD
加密结果 (十六进制): 7bc7ccbc261fe5bf94a8e7f0ab85466bb01c1246b314bad5e27c79023128b2013ef812d1f91ff5669022062373a1c19b4538330cb3e2a2deb27f0dc9366903462daf9b1f25cd9432ae9a65d42c0b47ea20b712dbc33b26475
加密耗时: 0.803843s
解密结果: zdh202208460110
解密耗时: 0.801872s
解密是否成功: True

Process finished with exit code 0
```

## 4. SM2 签名算法运行结果

```
C:\Users\12131\PycharmProjects\pythonProject1\.venv\Scripts\python.exe C:\Users\12131\PycharmProjects\pythonProject1\SM2_sign.py
Alice签名数据: (ECPint(x_coord=8738134546428181018892449422451601131016722182340719743773808451980942025694, y_coord=21553685567637493999278111036864264499419596327258075598326931280981814729469, 1
Bob签名数据: (ECPint(x_coord=26963482483118446012944624851613223299362940808731393303230140357940458531451, y_coord=2793708768154821958480202564569155209969930143264235988640975646155396245626, 1
编号Alice私钥: 28878724220786682040894079210442157017474242656637881721830782746064780369400
编号Bob私钥: 53994055529619668404909946413554895856167476508007590153771948232140304513635

Process finished with exit code 0
```

## 5. 三种情况下的 poc 验证运行结果

```
Run SM2_POC
C:\Users\12131\PycharmProjects\pythonProject1\.venv\Scripts\python.exe C:\Users\12131\PycharmProjects\pythonProject1\SM2_P0C.py
情况1验证结果: 失败
情况2验证: Alice私钥推导成功, Bob私钥推导成功
情况3验证结果: 成功

Process finished with exit code 0
```