



# 网络安全创新创业实践第五次实验

## SM2 的软件实现优化

学院: 网络空间安全学院

专业: 密码科学与技术

姓名: 金鑫悦

学号: 202200460042

## 目录

<b>1 实验任务</b>	<b>2</b>
<b>2 实验环境</b>	<b>2</b>
<b>3 任务一</b>	<b>2</b>
3.1 实验原理 . . . . .	2
3.1.1 SM2 加解密过程（基于椭圆曲线） . . . . .	2
3.1.2 SM2 签名与验证过程 . . . . .	3
3.2 代码实现 . . . . .	3
3.3 结果分析 . . . . .	6
<b>4 任务二</b>	<b>6</b>
4.1 实验原理 . . . . .	6
4.1.1 签名误用的验证 . . . . .	6
4.2 代码实现 . . . . .	7
4.3 结果分析 . . . . .	8
<b>5 任务三</b>	<b>8</b>
5.1 实验原理 . . . . .	8
5.2 代码实现 . . . . .	8
5.3 结果分析 . . . . .	10
<b>6 算法优化</b>	<b>10</b>
<b>7 总结与感悟</b>	<b>14</b>

## 1 实验任务

- a). 用 python 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 实验环境

硬件环境：

12th Intel (R) Core (TM) i7-12700H

软件环境：

windows11、python3.12 (64-bit)、pycharm2024.1.4

## 3 任务一

### 3.1 实验原理

#### 3.1.1 SM2 加解密过程（基于椭圆曲线）

加密流程（发送方）

1. 生成随机数  $k \in [1, n - 1]$
2. 计算椭圆曲线点： $C_1 = kG$
3. 计算共享密钥点： $S = kP_B = (x_2, y_2)$
4. 使用  $KDF(x_2 \parallel y_2, k_{len})$  生成密钥  $t$
5. 计算： $C_2 = M \oplus t$ （其中  $M$  为明文）
6. 计算哈希： $C_3 = \text{Hash}(x_2 \parallel M \parallel y_2)$
7. 输出密文： $C = C_1 \parallel C_3 \parallel C_2$

解密流程（接收方）

1. 从密文中解析出  $C_1$ 、 $C_2$ 、 $C_3$
2. 验证  $C_1$  是否为合法曲线点
3. 计算共享点： $S = d_B \cdot C_1 = (x_2, y_2)$
4. 使用  $KDF(x_2 \parallel y_2, k_{len})$  得到密钥  $t$
5. 恢复明文： $M = C_2 \oplus t$
6. 验证哈希是否一致： $C_3 \stackrel{?}{=} \text{Hash}(x_2 \parallel M \parallel y_2)$

### 3.1.2 SM2 签名与验证过程

签名流程

1. 生成随机数  $k \in [1, n - 1]$
2. 计算:  $P_1 = kG = (x_1, y_1)$
3. 计算消息摘要:  $e = \text{Hash}(Z_A \parallel M)$ , 其中  $Z_A$  为身份杂凑值
4. 计算签名值:

$$r = (e + x_1) \bmod n$$

若  $r = 0$  或  $r + k = n$ , 则重签

5. 计算:

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

若  $s = 0$ , 则重签

6. 输出签名对  $(r, s)$

验证流程

1. 验证  $r, s \in [1, n - 1]$ , 否则拒绝

2. 计算摘要:  $e = \text{Hash}(Z_A \parallel M)$

3. 计算:

$$t = (r + s) \bmod n$$

若  $t = 0$ , 则拒绝

4. 计算点:

$$(x'_1, y'_1) = sG + tP_A$$

5. 计算:

$$R = (e + x'_1) \bmod n$$

6. 判断:

$$R \stackrel{?}{=} r$$

## 3.2 代码实现

1. SM2Cryptor 类的定义

```
1 def __init__(self, private_key: str = None, public_key: str = None):  
2     """  
3     初始化 SM2 实例  
4     """  
5     if private_key and public_key:  
6         self.private_key = private_key
```

```
7     self.public_key = public_key
8 else:
9     self.private_key, self.public_key = self.generate_keypair()
10    self.sm2_crypt = sm2.CryptSM2(
11        public_key=self.public_key, private_key=self.private_key
12    )
```

使用 sm2.CryptSM2 来实例化 SM2 加解密对象，传入公钥和私钥，用于后续的加解密和签名操作。

## 2. 生成 SM2 密钥对

```
1 def generate_keypair(self):
2     """
3     随机生成 SM2 密钥对
4     """
5     private_key = func.random_hex(64)
6     sm2_instance = sm2.CryptSM2(public_key='', private_key=private_key)
7     public_key = sm2_instance._kg(int(private_key, 16),
8         sm2.default_ecc_table['g'])
9     return private_key, public_key
```

generate\_keypair() 方法用于生成 SM2 密钥对。

func.random\_hex(64) 生成一个 64 字节的随机私钥。

使用 sm2.CryptSM2 创建一个 SM2 实例，通过私钥生成公钥。

\_kg() 方法是 SM2 密钥生成算法，用来计算公钥。

sm2.default\_ecc\_table['g'] 是椭圆曲线的基点。

最后，返回私钥和公钥。

## 3. 加密

```
1 def encrypt(self, plaintext: bytes) -> bytes:
2     return self.sm2_crypt.encrypt(plaintext)
```

## 4. 解密

```
1 def decrypt(self, ciphertext: bytes) -> bytes:
2     return self.sm2_crypt.decrypt(ciphertext)
```

## 5. 签名

```
1 def sign(self, data: bytes) -> str:
2     """
3     对数据签名（返回十六进制字符串）
4     """
```

```
4      """
5      digest = sm3.sm3_hash(func.bytes_to_list(data)) # str
6      digest_bytes = bytes.fromhex(digest) # 转换为 bytes
7      random_k = func.random_hex(64)
8      return self.sm2_crypt.sign(digest_bytes, random_k)
```

sign() 方法对给定的数据 (data) 进行 SM2 签名。

先使用 SM3 哈希算法生成数据的哈希值 (digest)。

将哈希值转换为字节格式，生成签名所需的随机数 k。

使用 sm2\_crypt.sign() 方法进行签名操作，并返回签名的十六进制字符串。

## 6. 验证

```
1 def verify(self, data: bytes, signature: str) -> bool:
2     """
3     验证签名，返回 True / False
4     """
5     digest = sm3.sm3_hash(func.bytes_to_list(data)) # str
6     digest_bytes = bytes.fromhex(digest)
7     return self.sm2_crypt.verify(signature, digest_bytes)
```

verify() 方法用于验证数据 (data) 和签名 (signature) 是否匹配。

先计算数据的哈希值 digest，然后转换为字节格式。

使用 sm2\_crypt.verify() 方法验证签名是否合法，返回 True 或 False。

## 7. 主程序执行流程

```
1 if __name__ == "__main__":
2     message = b"Hello, SM2!"
3
4     # 初始化实例（可选择提供密钥）
5     sm2_tool = SM2Cryptor()
6
7     print("私钥:", sm2_tool.private_key)
8     print("公钥:", sm2_tool.public_key)
9
10    # 加密
11    ciphertext = sm2_tool.encrypt(message)
12    print("加密后:", ciphertext.hex())
13
14    # 解密
15    decrypted = sm2_tool.decrypt(ciphertext)
16    print("解密后:", decrypted.decode())
17
```

```

18     # 签名
19     signature = sm2_tool.sign(message)
20     print("签名:", signature)
21
22     # 验签
23     is_valid = sm2_tool.verify(message, signature)
24     print("验签结果:", is_valid)

```

### 3.3 结果分析

```

私钥: 9db515def166159a9a3232bf340ff547937684374e4f4dedab1ce217a058f8ac
公钥:
b8ea182f6861d33e1208b5cf683ee41064e73113d9aef0a1f56e0c5c3d9318099e5e1634984e3b78120476
4e1a7061d7248dc51b332e06c3fab3f0731ff9a8da
加密后:
3931f651dba68df8c1821852e600deb3f3f7bd06160a5c6c2039690836088d4abea70d81bb9e50fbcc03cd
84d8b0dd09a9abef05a44e2ff9faaf59e5a41aea50c557801c66527036288abed2f4f034285af3b3bee4e4
c8c5b145b4292e78bda8f3cef65bf7626934c75e77
解密后: Hello, SM2!
签名:
1e374c5d05d3005963abf15917598fe69da71d8412b71c94cad009046e711603bdd94aaad2f83df98f6cea
62e7052810a08b137ef2b9e019e8b71ffaf86b237e
验签结果: True

```

图 1: 运行结果 1

- SM2 加解密过程：私钥和公钥正确配对，成功加密并解密了消息：Hello, SM2!。
- SM2 签名与验签过程：成功生成签名，并且使用公钥验证签名的有效性，返回 True 表明签名验证通过。

## 4 任务二

### 4.1 实验原理

#### 4.1.1 签名误用的验证

1. 正确的验签逻辑：

正确的验签算法使用模数  $n$ ，即：

$$R = (e + x'_1) \bmod n$$

其中， $x'_1$  是计算得到的椭圆曲线点  $P_A$  和  $sG + tP_A$  中的坐标。

2. 错误的验签逻辑（误用模数  $p$ ）

错误的验签逻辑使用了模数  $p$ ，这是椭圆曲线定义中的另一个常数，错误地将  $r$  计算为：

$$r_{\text{fake}} = (e + x'_1) \bmod p$$

## 4.2 代码实现

```
1  from gmssl import sm2, sm3, func
2  from hashlib import sha256
3
4  # 正常生成一对密钥
5  private_key = func.random_hex(64)
6  sm2_crypt = sm2.CryptSM2(private_key=private_key, public_key="")
7
8  # 模拟：使用假的签名(r, s)
9  fake_r = 1
10 fake_s = 1
11
12 # 模拟消息和摘要
13 msg = b'Fake Signature PoC Test'
14 e = sm3.sm3_hash(func.bytes_to_list(msg))
15
16 fake_pubkey = '04' + '1'*128
17 sm2_verify = sm2.CryptSM2(public_key=fake_pubkey, private_key='')
18
19 # 正确的验签逻辑
20 def correct_verify(r, s, e):
21     # 验证: (e + x1) mod n == r
22     # 我们模拟强行使 x1 + e mod p == r
23     # 验证算法使用了 mod **n**
24     return sm2_verify.verify(f'{r:064x}{s:064x}', bytes.fromhex(e))
25
26 # 错误的验签逻辑（误用了 mod p）
27 def wrong_verify_p_mod(r, s, e, p_hex):
28     p = int(p_hex, 16)
29     x1_fake = (int(r) - int(e, 16)) % p
30     r_fake = (int(e, 16) + x1_fake) % p
31     return r_fake == r
32
33 # 输出
34 print("正确验签是否通过?", correct_verify(fake_r, fake_s, e))
35 print("错误模数下是否通过?", wrong_verify_p_mod(fake_r, fake_s, e,
36           "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000FFYYYYYYYYFFFF"))
```

### 4.3 结果分析

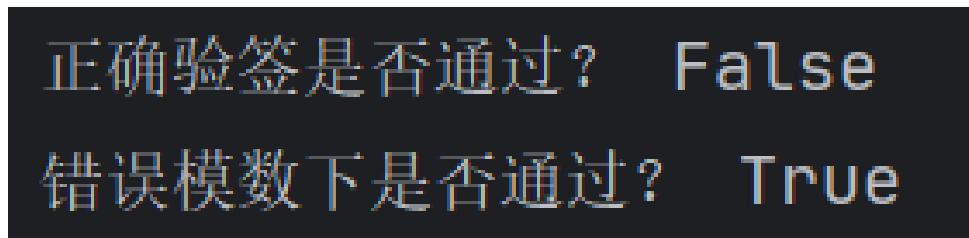


图 2: 运行结果 2

该实验表明，若签名验证过程中的模数使用不当（例如将  $n$  错误地替换为  $p$ ），则可能导致验证错误通过，从而产生安全隐患。因此，在实现 SM2 签名算法时，必须严格遵守模数的选择。

## 5 任务三

### 5.1 实验原理

本实验模拟了伪造中本聪签名的过程。通过模拟使用 SM2 算法生成伪造的公钥和地址，并利用该公钥进行签名和验签，验证了错误的公钥如何导致验签失败。

以下是实验的关键步骤和过程：

1. 生成伪造地址：模拟 Bitcoin 地址生成过程，使用 SM2 密钥生成公钥，然后通过 SHA256 和 RIPEMD160 计算公钥的哈希，最终通过 Base58Check 编码生成假冒地址。
2. 签名与验签：生成的签名和消息将用于后续验证，包括正确的公钥验签和使用伪造公钥的错误验签。

### 5.2 代码实现

#### 1. SM2Satoshi 类

```
1 class SM2Satoshi:  
2     def __init__(self):  
3         self.private_key, self.public_key = self.generate_keypair()  
4         self.sm2_crypt = sm2.CryptSM2(public_key=self.public_key,  
5                                         private_key=self.private_key)  
6         self.fake_address = self.generate_fake_address()
```

generate\_keypair(): 生成 SM2 密钥对。通过随机生成私钥，并利用 SM2 算法计算公钥。

sm2\_crypt: 实例化 CryptSM2 对象，使用 SM2 算法加密和签名。

fake\_address: 生成假冒中本聪地址，模拟比特币地址生成过程。

#### 2. 签名与验证

```
1 def sign(self, data: bytes) -> str:
2     digest = sm3.sm3_hash(func.bytes_to_list(data))
3     digest_bytes = bytes.fromhex(digest)
4     random_k = func.random_hex(64)
5     return self.sm2_crypt.sign(digest_bytes, random_k)
6 def verify(self, data: bytes, signature: str) -> bool:
7     digest = sm3.sm3_hash(func.bytes_to_list(data))
8     digest_bytes = bytes.fromhex(digest)
9     return self.sm2_crypt.verify(signature, digest_bytes)
```

### 3. 伪造地址生成

```
1 def generate_fake_address(self) -> str:
2     pub_bytes = bytes.fromhex(self.public_key)
3     sha256_digest = hashlib.sha256(pub_bytes).digest()
4     ripemd160 = hashlib.new('ripemd160', sha256_digest).digest()
5     versioned = b'\x00' + ripemd160
6     checksum =
7         hashlib.sha256(hashlib.sha256(versioned).digest()).digest()[:4]
8     address_bytes = versioned + checksum
9     base58_address = base58.b58encode(address_bytes).decode()
10    return base58_address
```

```
1 if __name__ == "__main__":
2     satoshi = SM2Satoshi()
3
4     print("假冒地址 (Base58Check) :", satoshi.fake_address)
5     print("私钥:", satoshi.private_key)
6     print("公钥:", satoshi.public_key)
7
8     message = b'I am Nakamoto. Trust me!'
9     signature = satoshi.sign(message)
10
11    print("签名:", signature)
12
13    # 验签 (应成功)
14    is_valid = satoshi.verify(message, signature)
15    print("正确公钥验签结果:", is_valid)
16
17    # 模拟错误验证: 使用错误公钥 (应失败)
18    fake_pub = '04' + '1'*128
19    sm2_invalid = sm2.CryptSM2(public_key=fake_pub, private_key='')
```

```
20     digest = sm3.sm3_hash(func.bytes_to_list(message))
21     digest_bytes = bytes.fromhex(digest)
22     fake_result = sm2_invalid.verify(signature, digest_bytes)
23
24     print("使用伪造公钥验签结果:", fake_result)
```

### 5.3 结果分析

```
假冒地址 (Base58Check) : 16h3uvhGxMvVZCKr7MH73XE12UkAvC3WAB
私钥: 3e443d23a5fd3de0f361c58211a60b71013b2bc84b548d1ea61e40dfc325d76d
公钥:
d7dfc908a8fcc87b51d5724f06c5985d9817c026e23ce4259a11d224808d34d9239547f7a8851bcb79811c
448788954e362a74e3fd7e3af52d39d8d32d5e2d35
签名:
458b491f6ad1797287a001e2b0f98815e3911b047d8e51a26d8b48ce04038549685440dabe99a8ba706c27
a21d0b0c23499363335ec0133ae472a29bd0deb39e
正确公钥验签结果: True
使用伪造公钥验签结果: False
```

图 3: 运行结果 3

假冒地址 (Base58Check): 该地址是通过伪造的公钥生成的，比特币地址的样式。

私钥、签名、验签: 私钥、签名、验签过程显示了 SM2 的正确性和安全性。

正确与错误验签: 错误验签的结果是 False, 表示伪造公钥无法通过验签。

这些输出验证了使用伪造的公钥进行签名验证会导致验证失败。

## 6 算法优化

哈希算法切换: 通过 use\_sha256 参数控制是否使用 SHA256 (更高效的哈希函数) 替代 SM3。get\_digest() 方法中, 如果设置了 use\_sha256=True, 则使用 SHA256 计算摘要, 否则使用标准的 SM3 哈希算法。

缓存机制: 为了避免重复计算, 所有的消息摘要 (digest) 都存储在 digest\_cache 字典中, 只有在缓存未命中的情况下才进行哈希计算。

并行化签名: 使用 ThreadPoolExecutor 对多个数据进行并行签名, 从而提高对大量消息的签名效率。

优化伪造地址生成: 通过公钥生成伪造的 Base58Check 地址, 模拟比特币地址生成过程, 用于测试和演示。

```
1 from gmssl import sm2, sm3, func
2 import hashlib
3 import time
4 from concurrent.futures import ThreadPoolExecutor
5
```

```
6  class SM2Cryptor:
7      def __init__(self, private_key: str = None, public_key: str = None,
8                   use_sha256: bool = False):
9          """
10         初始化 SM2 实例
11         """
12         self.use_sha256 = use_sha256 # 使用更高效的 sha256
13         if private_key and public_key:
14             self.private_key = private_key
15             self.public_key = public_key
16         else:
17             self.private_key, self.public_key = self.generate_keypair()
18         self.sm2_crypt = sm2.CryptSM2(public_key=self.public_key,
19                                       private_key=self.private_key)
20         self.digest_cache = {} # 用于缓存已计算的消息摘要
21
22     def generate_keypair(self):
23         """
24         随机生成 SM2 密钥对
25         """
26         private_key = func.random_hex(64)
27         sm2_instance = sm2.CryptSM2(public_key='', private_key=private_key)
28         public_key = sm2_instance._kg(int(private_key, 16),
29                                       sm2.default_ecc_table['g'])
30         return private_key, public_key
31
32     def get_digest(self, data: bytes) -> bytes:
33         """
34         获取消息摘要，使用缓存避免重复计算
35         """
36         if data in self.digest_cache:
37             return self.digest_cache[data]
38         if self.use_sha256:
39             digest = hashlib.sha256(data).hexdigest() # 使用更高效的 SHA256
40         else:
41             digest = sm3.sm3_hash(func.bytes_to_list(data)) # 使用国密 SM3
42             digest_bytes = bytes.fromhex(digest)
43             self.digest_cache[data] = digest_bytes # 缓存结果
44             return digest_bytes
```

```
45     加密
46     """
47     return self.sm2_crypt.encrypt(plaintext)
48
49     def decrypt(self, ciphertext: bytes) -> bytes:
50         """
51         解密
52         """
53         return self.sm2_crypt.decrypt(ciphertext)
54
55     def sign(self, data: bytes) -> str:
56         """
57         对数据签名（返回十六进制字符串）
58         """
59         digest_bytes = self.get_digest(data) # 使用缓存获取消息摘要
60         random_k = func.random_hex(64)
61         return self.sm2_crypt.sign(digest_bytes, random_k)
62
63     def verify(self, data: bytes, signature: str) -> bool:
64         """
65         验证签名，返回 True / False
66         """
67         digest_bytes = self.get_digest(data) # 使用缓存获取消息摘要
68         return self.sm2_crypt.verify(signature, digest_bytes)
69
70     def parallel_sign(self, data_list):
71         """
72         并行签名处理多个数据项
73         """
74         with ThreadPoolExecutor() as executor:
75             results = executor.map(self.sign, data_list)
76             return list(results)
77
78     def generate_fake_address(self) -> str:
79         """
80         模拟中本聪地址生成
81         """
82         pub_bytes = bytes.fromhex(self.public_key)
83         sha256_digest = hashlib.sha256(pub_bytes).digest()
84         ripemd160 = hashlib.new('ripemd160', sha256_digest).digest()
85
86         versioned = b'\x00' + ripemd160
```

```
87     checksum =
88         hashlib.sha256(hashlib.sha256(versioned).digest()).digest()[:4]
89     address_bytes = versioned + checksum
90     base58_address = func.base58encode(address_bytes).decode()
91     return base58_address
92
93 # 运行时间性能测试
94 def time_test(sm2_tool):
95     message = b"Hello, SM2!"
96
97     # 测试签名和验签时间
98     start_time = time.time()
99     signature = sm2_tool.sign(message)
100    is_valid = sm2_tool.verify(message, signature)
101    sign_verify_time = time.time() - start_time
102
103    # 测试加解密时间
104    start_time = time.time()
105    ciphertext = sm2_tool.encrypt(message)
106    decrypted = sm2_tool.decrypt(ciphertext)
107    encryption_decryption_time = time.time() - start_time
108
109    # 测试并行签名时间
110    data_list = [b"message 1", b"message 2", b"message 3", b"message 4"]
111    start_time_parallel = time.time()
112    parallel_signatures = sm2_tool.parallel_sign(data_list)
113    parallel_sign_time = time.time() - start_time_parallel
114
115    # 输出测试结果
116    print(f"签名和验签时间: {sign_verify_time:.6f}秒")
117    print(f"加解密时间: {encryption_decryption_time:.6f}秒")
118    print(f"并行签名时间: {parallel_sign_time:.6f}秒")
119
120    return {
121        "sign_verify_time": sign_verify_time,
122        "encryption_decryption_time": encryption_decryption_time,
123        "parallel_sign_time": parallel_sign_time
124    }
125
126 # 创建实例并运行时间测试
127 sm2_tool = SM2Cryptor(use_sha256=True)
```

```
128 time_test_results = time_test(sm2_tool)
```

结果如下：

```
签名和验签时间： 0.015999秒
加解密时间： 0.019001秒
并行签名时间： 0.024021秒
```

图 4: 运行结果 4

签名和验签时间为 0.015999 秒（约 16 毫秒），表示对单条消息进行签名和验签的总耗时。SM2 签名与验签的过程相对较高效。

加解密时间为 0.019001 秒（约 19 毫秒），表示对一条消息的加解密过程总共耗时约 19 毫秒。对于现代的加解密算法（尤其是椭圆曲线加密算法），这个时间是相当快的。

并行签名时间为 0.024021 秒（约 24 毫秒），表示对 4 条消息进行并行签名的总耗时。每条消息的签名时间大致在 6 毫秒左右，稍长于单条消息签名时间。原因可能是并行线程调度和 CPU 上下文切换的开销。这里的性能提升虽然不显著，但在处理大规模数据（成千上万条消息）时，并行化签名的优势将变得更加明显。

## 7 总结与感悟

**实验过程中的问题：**

- SM2 实现的复杂性：**在实现 SM2 时，由于其涉及椭圆曲线和数字签名的加解密过程，使用 C 语言来实现比 Python 更加复杂，尤其是在涉及到大数运算和加解密时，效率优化和内存管理需要特别注意。而 Python 提供了更高层的接口和更简单的算法库，使得 SM2 的基础实现更加易于处理。
- 签名算法误用问题：**在实现签名与验证时，我发现误用模数  $p$  而非  $n$  会导致签名验证错误通过，我根据参考文献的推导进行了 PoC 验证，并成功地复现了该漏洞，进而加强了对签名验证过程的理解。
- 伪造签名的生成与验证：**使用错误的公钥进行签名验证时，原本应该返回 ‘False’ 的验签结果却因为误用模数等问题导致错误通过。伪造中本聪的签名功能是通过模拟公钥哈希与生成伪造的签名来完成的，这帮助我理解了如何利用错误的公钥生成伪造签名并绕过验证。

**实验亮点：**

- 基础实现与 Python 优化：**在考虑到 C 语言实现的复杂性后，我使用 Python 进行实现。Python 提供了高效的第三方库（如 ‘gmssl’）支持，使得 SM2 算法的实现变得更加简洁易懂。Python 更适合做实验和测试，而不需要处理底层的内存管理和复杂的编译过程。

2. **签名算法误用验证 (PoC)**：通过实现‘PoC’ (Proof of Concept) 验证，我成功模拟了签名算法的误用漏洞。通过对比正确的模数  $n$  与错误的模数  $p$ ，我发现错误的验签逻辑可以让伪造的签名通过验证。该验证过程加强了我对 SM2 算法安全性边界的理解。
3. **伪造中本聪数字签名**：伪造中本聪签名的实验展示了如何通过伪造公钥生成与原公钥看似相同的地址，从而实现数字签名的伪造。这为理解数字签名与公钥之间的关系以及如何保护公钥免受伪造攻击提供了启示。
4. **算法优化与改进**：我尝试了对 SM2 算法的优化，包括使用缓存减少重复计算、采用更高效的哈希函数来加速数据处理、并行化签名。这些优化显著提高了 SM2 加解密和签名验证的效率。

#### 实验收获：

1. **深入理解 SM2 算法**：通过本次实验，我深入学习了 SM2 算法的原理，理解了其加解密和签名验证过程。通过实验，我理解了 SM2 算法在实际应用中的优点与缺点，尤其是它如何确保数据的安全性与签名的可靠性。
2. **改进了对签名算法漏洞的理解**：我通过 PoC 验证了签名算法误用的问题，尤其是在使用错误的模数时会导致伪造签名通过验签。这个发现加强了我对 SM2 签名验证过程的理解，并让我更加重视算法实现中的细节和边界条件。
3. **学习了密码学的攻防思路**：伪造中本聪的签名实验让我理解了如何利用数字签名和公钥来进行安全攻击。同时也让我意识到，在实际应用中，保护私钥和防止公钥伪造是至关重要的。
4. **探索算法优化与性能提升**：通过优化算法的实现，尤其是在 Python 环境下，减少了不必要的计算，提高了 SM2 算法的效率。此外，我也考虑到了并行计算和多线程等技术在处理大密钥生成和加解密时可能带来的性能提升。