SM2数字签名算法流程（简化描述）：

1. 签名：

- 输入：消息M，私钥dA，用户标识IDA（一般取固定值如"1234567812345678"的十六进制）

- 计算ZA = H256(ENTL\_A || IDA || a || b || xG || yG || xA || yA)，其中xA,yA是公钥，H256为SM3哈希算法（输出256位）。

- 设置M\_ = ZA || M，并计算e = Hv(M\_)，这里Hv也是SM3，但输出作为整数处理。

- 生成随机数k（在[1, n-1]范围内）

- 计算椭圆曲线点(x1, y1) = [k]G

- 计算r = (e + x1) mod n，若r=0或r+k=n则重新选择k

- 计算s = ((1 + dA)^(-1) \* (k - r \* dA)) mod n，若s=0则重新选择k

- 输出签名(r, s)

2. 验签：

- 输入：消息M，签名(r, s)，公钥PA

- 检查r, s是否在[1, n-1]内

- 计算ZA（同上）

- 计算e（同上）

- 计算t = (r + s) mod n，若t=0则失败

- 计算椭圆曲线点(x1, y1) = [s]G + [t]PA

- 计算R = (e + x1) mod n，检查R是否等于r

由于需要自己实现SM3哈希算法，这会使项目变得复杂。为了简化，我们可以使用现有的sm3实现（如`gmssl`库）或者自己实现一个简单的sm3（但这里我们为了重点放在SM2上，先使用现有库）。

我们选择使用`gmssl`库，它提供了SM2、SM3、SM4等算法的实现。如果没有安装，可以使用`pip install gmssl`。

但是，为了完全理解，我们也可以考虑自己实现基础部分，包括椭圆曲线运算和SM3。由于时间关系，我们这里先使用`gmssl`完成基础实现，然后自己实现关键部分以进行改进尝试。

### b) 签名算法误用的POC验证

根据文档"20250713-wen-sm2-public.pdf"（由于我们没有实际文档，这里假设文档中提到的误用是常见的两种：1. 随机数k重用；2. 签名过程中s的计算公式误用为s = (k - r \* dA) \* (1 + dA)^(-1) mod n，而实际上应该是s = ((1 + dA)^(-1) \* (k - r \* dA)) mod n，但公式本身已经如此，所以可能是其他误用。实际上，常见的误用是随机数k的重用和泄露，以及签名过程中未检查r和s是否为0）。

假设文档中提到的误用是随机数k的重用（与DSA中类似，会导致私钥泄露）。我们将对此进行POC验证。

### c) 伪造中本聪的数字签名

中本聪在比特币中使用的签名算法是ECDSA（secp256k1曲线）。要求伪造其签名，实际上是对ECDSA签名算法进行攻击。常见的攻击包括：利用随机数k重用、利用随机数k的可预测性（如使用线性同余生成器）、利用签名过程中未检查消息哈希值是否在范围内等。但中本聪的私钥未知，且我们假设其实现是安全的，所以通常无法伪造。但如果我们能够获取到两个使用相同k的签名（这在比特币区块链上可能发生），则可以恢复私钥。

因此，我们可能需要在比特币区块链上寻找是否有重复k的签名交易。

输出结果：