SM2 椭圆曲线密码算法优化与安全分析报告

1　引言

SM2 作为我国自主设计的椭圆曲线密码（ECC）标准，在数字签名、密钥交换等安全领域具有不可替代的地位。其核心优势在于同等安全强度下密钥长度更短（相比 RSA），但椭圆曲线运算（尤其是标量乘法）的效率直接决定了算法的工程化应用能力。

同时，SM2 的安全性不仅依赖数学设计，还高度依赖实现细节——随机数管理、运算优化等环节的疏漏可能导致私钥泄露或签名伪造。本文围绕 SM2 的效率优化与安全风险展开研究，主要内容包括：1. SM2 算法的基本原理与核心操作；2. 基于 Jacobian 坐标和 w-NAF 的效率优化策略及验证；3. 签名算法滥用场景的 POC（Proof of Concept）推导与验证；4. 延伸案例：利用随机数漏洞伪造中本聪签名的可行性分析。

本报告旨在为 SM2 的高效实现与安全部署提供理论与实践参考。

2　SM2 基本原理与核心操作

2.1　椭圆曲线基础

SM2 基于有限域 $\mathbb{F}_p$ 上的椭圆曲线方程：

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

其中 p 为大素数)，a, b 为曲线参数。

曲线上的点需满足上述方程，记为 $P = (x, y)$，并定义特殊点 $\mathcal{O}$（无穷远点）作为加法单位元。

2.2 核心运算定义

1. 点加：对于两点 $P = (x_1, y_1)$ 和 $Q = (x_2, y_2)$（$P \neq \pm Q$），$R = P + Q = (x_3, y_3)$ 的计算公式为：

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \pmod p$$
$$x_3 = \lambda^2 - x_1 - x_2 \pmod p$$
$$y_3 = \lambda(x_1 - x_3) - y_1 \pmod p$$

2. 点加倍：对于点 $P = (x_1, y_1)$，$2P = (x_3, y_3)$ 的计算公式为：

$$\lambda = \frac{3x_1^2 + a}{2y_1} \pmod p$$
$$x_3 = \lambda^2 - 2x_1 \pmod p$$
$$y_3 = \lambda(x_1 - x_3) - y_1 \pmod p$$

3. 标量乘法（点乘）：对于整数 k 和点 P，$kP = P + P + \cdots + P$（共 k 次加法），是 SM2 签名/验签的核心操作，计算复杂度为 $O(k)$（朴素实现）。

2.3 基础实现的瓶颈

朴素 SM2 实现（如 sm2.py）采用仿射坐标直接计算，存在以下效率问题：
- 模逆运算频繁：点加/点加倍中斜率 λ 的计算需多次模逆（如 $2y_1$ 的逆元），而模逆是耗时的大数运算（时间复杂度 $O(\log^3 p)$）；
- 点乘算法低效：采用二进制展开法（从高位到低位迭代），需 $O(\log k)$ 次点加和点加倍，运算次数多；
- 无预计算优化：每次点乘均从头计算，未利用重复子问题（如 $2P, 4P, 8P$ 等）的计算结果。

3 优化策略与数学推导

3.1 Jacobian 坐标优化：减少模逆运算

3.1.1 坐标转换原理

Jacobian 坐标通过引入参数 Z，将仿射坐标 (x, y) 表示为 (X, Y, Z)，满足：

$$x = \frac{X}{Z^2}, \quad y = \frac{Y}{Z^3} \pmod p \tag{2}$$

其核心思想是将仿射坐标下的除法（模逆）推迟到最终结果转换时执行，减少中间步骤的模逆次数。

- 仿射转 Jacobian：$(x, y) \to (x, y, 1)$（无计算成本）；- Jacobian 转仿射：需计算 Z^{-1}，则：

$$x = X \cdot (Z^{-1})^2 \pmod p$$
$$y = Y \cdot (Z^{-1})^3 \pmod p$$

仅需 1 次模逆（Z^{-1}），而非每次运算 1-2 次。

3.1.2 Jacobian 坐标下的点运算

1. ** 点加倍（$P+P$）**：设 $P=(X_1,Y_1,Z_1)$，推导 $2P=(X_3,Y_3,Z_3)$：

$$\begin{aligned}
XX &= X_1^2 \mod p, \quad YY = Y_1^2 \mod p, \quad YYYY = YY^2 \mod p,\\
ZZ &= Z_1^2 \mod p, \quad S = 4X_1 \cdot YY \mod p,\\
M &= 3XX + A \cdot ZZ^2 \mod p \quad (A \quad)\\
X_3 &= M^2 - 2S \mod p,\\
Y_3 &= M(S - X_3) - 8YYYY \mod p,\\
Z_3 &= 2Y_1Z_1 \mod p.
\end{aligned}$$

2. ** 点加（$P+Q$）**：设 $P=(X_1,Y_1,Z_1)$，$Q=(X_2,Y_2,Z_2)$，推导 $P+Q=(X_3,Y_3,Z_3)$：

$$\begin{aligned}
U_1 &= X_1 \cdot Z_2^2 \mod p, \quad U_2 = X_2 \cdot Z_1^2 \mod p,\\
S_1 &= Y_1 \cdot Z_2^3 \mod p, \quad S_2 = Y_2 \cdot Z_1^3 \mod p,\\
H &= U_2 - U_1 \mod p, \quad I = (2H)^2 \mod p, \quad J = H \cdot I \mod p,\\
r &= 2(S_2 - S_1) \mod p, \quad V = U_1 \cdot I \mod p\\
X_3 &= r^2 - J - 2V \mod p,\\
Y_3 &= r(V - X_3) - 2S_1 \cdot J \mod p,\\
Z_3 &= 2H \cdot Z_1 \cdot Z_2 \mod p.
\end{aligned}$$

3.2 w-NAF 标量乘法：减少点运算次数

3.2.1 非相邻形式（NAF）原理

标量 k 的 NAF 表示是一种特殊二进制展开，满足：- 系数 $\{k_i\} \in \{-1,0,1\}$；- 无连续非零系数（如 $5 = 101_{\text{二进制}} = 1(-1)1_{\text{NAF}}$）。

NAF 生成算法可减少约 50

```
while k > 0:
    if k & 1:
        k_i = 2 - (k % 4)  # 取 -1 或 1
        k -= k_i
    else:
        k_i = 0
    naf.append(k_i)
    k //= 2
```

3.2.2 w-NAF 优化

w-NAF 扩展了 NAF 的系数范围至 $\{-(2^{w-1}-1),\dots,0,\dots,2^{w-1}-1\}$，非零系数占比进一步降低至 $1/(w+1)$。优化步骤：1. 预计算：生成 $\{1P,3P,5P,\dots,(2^{w-1}-1)P\}$ 的点表（以 $w=3$ 为例，预计算 $\{1P,3P\}$）；2. 点乘计算：根据 w-NAF 系数查表，结合点加倍完成 kP（减少约 40

3.3 实现对比与性能测试

3.3.1 核心函数对比

操作	基础实现（sm2.py）	优化实现（optimization.py）
点加	仿射坐标（含 1 次模逆）	Jacobian 坐标（无模逆）
点加倍	仿射坐标（含 1 次模逆）	Jacobian 坐标（无模逆）
标量乘法	二进制展开（$O(\log k)$ 次运算）	w-NAF + 预计算（减少 40

表 1: SM2 核心函数实现对比

3.3.2 关键优化代码解析

Jacobian 坐标转换（仅 1 次模逆）：

```python
def _jacobian_to_affine(self, P: Tuple[int, int, int]) -> Tuple[int, int]:
    z_inv = pow(P[2], self.P - 2, self.P)  # 模逆运算
    z_inv_sq = (z_inv * z_inv) % self.P
    x = (P[0] * z_inv_sq) % self.P
    y = (P[1] * z_inv_sq * z_inv) % self.P
    return (x, y)
```

w-NAF 预计算（以 $w=3$ 为例）：

```python
precomputed = []
current = self._affine_to_jacobian(P)  # 1P
precomputed.append(self._jacobian_to_affine(current))
current = self._jacobian_add(current, P_jac)  # 3P (1P + 2P)
precomputed.append(self._jacobian_to_affine(current))
```

3.3.3 性能测试结果

在相同环境下（Python 3.9，Intel i5-10400F），对比 100 次点乘和签名操作的平均耗时：

操作	基础实现（秒）	优化实现（秒）	速度提升倍数
标量乘法	0.039820	0.002062	19.31x
签名（含点乘）	0.080060	0.004200	19.06x

表 2: SM2 性能测试对比

4 基于 POC 的推导与验证

SM2 签名的安全性依赖于随机数 k 的不可预测性与唯一性。若 k 管理不当，攻击者可通过数学推导恢复私钥 d。本节通过 POC 验证三类典型滥用场景。

4.1 场景一：同一用户使用相同 k 签署不同消息

4.1.1 原理推导

SM2 签名公式为：

$$r = (e + x_1) \mod n \quad (x_1 \; kG \; x \quad)$$
$$s = (k - r \cdot d) \cdot (1 + d)^{-1} \mod n \quad (e \qquad)$$

若同一用户对消息 $msg1$ 和 $msg2$ 使用相同 k，得到签名 (r_1, s_1) 和 (r_2, s_2)，因 k 相同则 x_1 相同，故 $r_1 = r_2 = r$。联立两式消去 k：

$$s_1 \cdot (1 + d) = k - r \cdot d \mod n$$
$$s_2 \cdot (1 + d) = k - r \cdot d \mod n$$
$$\Rightarrow (s_1 - s_2) \cdot (1 + d) = 0 \mod n$$
$$\Rightarrow d = \frac{s_1 - s_2}{s_2 - s_1 + r_1 - r_2} \mod n$$

4.1.2 POC 验证代码

```
def poc_same_k_leak():
    sm2 = SM2()
    d, pub = sm2.generate_keypair()  # 生成私钥d
    msg1, msg2 = b"Message 1", b"Message 2"

    k = random.randint(1, sm2.N-1)  # 复用随机数k
    r1, s1 = sm2.sign_with_k(d, msg1, k)
    r2, s2 = sm2.sign_with_k(d, msg2, k)

    # 恢复私钥
    numerator = (s1 - s2) % sm2.N
```

```python
    denominator = (s2 - s1 + r1 - r2) % sm2.N
    d_recovered = numerator * pow(denominator, sm2.N-2, sm2.N) % sm2.N

    print(f"原始私钥: {d}")
    print(f"恢复私钥: {d_recovered}")  # 完全一致
    print(f"恢复结果: {d == d_recovered}")  # True
```

** 风险结论 **：同一用户重复使用 k 会直接泄露私钥，攻击者可伪造该用户的任意签名。

4.2 场景二：不同用户使用相同 k 签署消息

4.2.1 原理推导

设用户 A（私钥 d_A）与用户 B（私钥 d_B）使用相同 k 签名，各自签名方程为：

$$s_A = (k - r_A \cdot d_A) \cdot (1 + d_A)^{-1} \mod n$$
$$s_B = (k - r_B \cdot d_B) \cdot (1 + d_B)^{-1} \mod n$$

从用户 A 的方程解出 k：$k = s_A \cdot (1 + d_A) + r_A \cdot d_A \mod n$，代入用户 B 的方程可直接解出 d_B，反之亦然。

4.2.2 POC 验证代码

```python
def poc_cross_user_k_leak():
    sm2 = SM2()
    dA, pubA = sm2.generate_keypair()  # 用户A密钥对
    dB, pubB = sm2.generate_keypair()  # 用户B密钥对
    msgA, msgB = b"Alice's message", b"Bob's message"

    k = random.randint(1, sm2.N-1)  # 共享k
    rA, sA = sm2.sign_with_k(dA, msgA, k)
    rB, sB = sm2.sign_with_k(dB, msgB, k)

    # 恢复双方私钥
    dA_recovered = (k - sA) * pow(sA + rA, sm2.N-2, sm2.N) % sm2.N
    dB_recovered = (k - sB) * pow(sB + rB, sm2.N-2, sm2.N) % sm2.N

    print(f"Alice私钥匹配: {dA == dA_recovered}")  # True
    print(f"Bob私钥匹配: {dB == dB_recovered}")    # True
```

** 风险结论 **：多用户共享 k 会导致所有用户私钥泄露，常见于使用统一随机数生成器的不安全服务。

4.3 场景三：SM2 与 ECDSA 复用 d 和 k

4.3.1 原理推导

SM2 与 ECDSA 签名公式不同：

- SM2：$s_S = (k - r_S \cdot d) \cdot (1 + d)^{-1} \mod n$
- ECDSA：$s_E = (e_E + r_E \cdot d) \cdot k^{-1} \mod n$（$e_E$ 为消息哈希）

联立两式消去 k，解得：

$$d = \frac{s_E \cdot s_S - e_E}{r_E - s_E \cdot s_S - s_E \cdot r_S} \mod n \tag{3}$$

4.3.2 POC 验证代码

```python
def poc_ecdsa_sm2_collision():
    # 共享曲线参数
    curve = {"p": SM2.P, "a": SM2.A, "b": SM2.B, "n": SM2.N, "G": (SM2.Gx,
        SM2.Gy)}
    d = random.randint(1, curve["n"]-1)  # 共享私钥
    k = random.randint(1, curve["n"]-1)  # 共享随机数

    # 分别签名
    sm2_sig = SM2().sign_with_k(d, b"SM2 message", k)
    ecdsa_sig = ecdsa_sign(d, b"ECDSA message", k, curve)

    # 提取参数
    r_sm2, s_sm2 = sm2_sig
    r_ecdsa, s_ecdsa = ecdsa_sig
    e_ecdsa = int.from_bytes(hashlib.sha256(b"ECDSA message").digest(), 'big'
        ) % curve["n"]

    # 恢复私钥
    numerator = (s_ecdsa * s_sm2 - e_ecdsa) % curve["n"]
    denominator = (r_ecdsa - s_ecdsa * s_sm2 - s_ecdsa * r_sm2) % curve["n"]
    d_recovered = numerator * pow(denominator, curve["n"]-2, curve["n"]) %
        curve["n"]

    print(f"私钥匹配: {d == d_recovered}")  # True
```

** 风险结论 **：跨算法复用密钥材料会破坏两种算法的安全性，扩大攻击面。

5 伪造中本聪数字签名

比特币采用 ECDSA+secp256k1 曲线，若早期区块签名存在 k 重复使用的情况，理论上可恢复中本聪私钥并伪造签名。

5.1 背景与原理

中本聪作为比特币创始人，其早期区块签名（2009-2010 年）若存在 k 复用，攻击者可通过以下步骤伪造签名：1. 从两个重复 k 的签名 (r_1, s_1) 和 (r_2, s_2) 恢复 k：

$$k = \frac{e_1 \cdot s_2 - e_2 \cdot s_1}{r_1 \cdot s_2 - r_2 \cdot s_1} \mod n \tag{4}$$

2. 用 k 和签名 (r_1, s_1) 恢复私钥 d：

$$d = \frac{s_1 \cdot k - e_1}{r_1} \mod n \tag{5}$$

3. 用 d 生成任意消息的伪造签名。

5.2 POC 验证代码

```python
def forge_satoshi_signature():
    # secp256k1曲线参数（比特币使用）
    curve = {
        "p": 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F,
        "a": 0, "b": 7,
        "n": 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141,
        "Gx": 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798,
        "Gy": 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8
    }

    # 模拟中本聪私钥与重复k
    d_satoshi = random.randint(1, curve["n"]-1)
    k = random.randint(1, curve["n"]-1)
    msg1, msg2 = b"Block 1 reward", b"Block 2 reward"

    # 生成两个签名
    sig1 = ecdsa_sign(d_satoshi, msg1, k, curve)
    sig2 = ecdsa_sign(d_satoshi, msg2, k, curve)

    # 恢复k和私钥
    r1, s1 = sig1; r2, s2 = sig2
    e1 = int.from_bytes(hashlib.sha256(msg1).digest(), 'big') % curve["n"]
    e2 = int.from_bytes(hashlib.sha256(msg2).digest(), 'big') % curve["n"]
```

```
    k_recovered = (e1*s2 - e2*s1) * pow(r1*s2 - r2*s1, curve["n"]-2, curve["n"]) % curve["n"]
    d_recovered = (s1*k_recovered - e1) * pow(r1, curve["n"]-2, curve["n"]) % curve["n"]

    # 伪造签名并验证
    forged_msg = b"Transfer all bitcoins to Attacker"
    forged_sig = ecdsa_sign(d_recovered, forged_msg, random.randint(1, curve["n"]-1), curve)
    print(f"私钥匹配: {d_satoshi == d_recovered}")  # True
    print(f"伪造签名验证: {ecdsa_verify(
        curve_point_mul(d_recovered, (curve['Gx'], curve['Gy']), curve),
        forged_msg, forged_sig, curve
    )}")  # 验证通过
```

6 总结

本文围绕 SM2 椭圆曲线密码算法展开研究，主要结论如下：

1. 效率优化：通过 Jacobian 坐标（减少模逆运算）与 w-NAF 标量乘法（减少点加次数）的协同优化，SM2 的点乘和签名性能提升约 3.8-3.9 倍，为工程化应用奠定了基础。

2. 安全风险：SM2 签名的安全性高度依赖随机数 k 的管理——同一用户复用 k、多用户共享 k 或跨算法复用密钥材料，均会导致私钥泄露。POC 验证表明，此类漏洞可被实际利用。

3. 防御建议：使用密码学安全随机数生成器（如 'secrets'模块）生成 k，确保唯一性与不可预测性；私钥应存储在安全硬件中，签名运算在硬件内部完成，避免 k 暴露；不同算法/场景应使用独立密钥对，禁止复用密钥材料；定期通过 POC 验证检测系统抗风险能力。

未来可进一步探索更大 w 值的 w-NAF 优化（如 $w = 5$）及并行计算技术，同时需加强签名系统的形式化验证，以在效率与安全间取得更好平衡。