



西南财经大学

SOUTHWESTERN UNIVERSITY OF FINANCE AND ECONOMICS

针对 RSA 算法的数学攻击向量复现 与 OAEP 防御机制构建

(期末论文)

学 院： 计算机与人工智能学院

专 业： 计算机科学与技术

学生姓名： 刘俊宏、许桐恺

对应学号： 42311180、42311038

完成日期： 2025 年 12 月

西南财经大学
Southwestern University of Finance and Economics

2025 年 12 月

针对 RSA 算法的数学攻击向量复现与 OAEP 防御机制构建

刘俊宏* 许桐恺*

【摘要】 RSA 作为公钥密码学的基石，其安全性严重依赖于密钥参数的选择与填充机制的健壮性。本文旨在通过“底层实现—攻击复现—防御构建”的完整闭环，深入剖析 RSA 算法的数学特性与工程实现细节。首先，本文基于数论原理，不依赖第三方密码库，从零实现了 Miller-Rabin 素性检测、扩展欧几里得算法及快速模幂运算，并通过实验验证了中国剩余定理（CRT）对解密效率约 300% 的性能提升。其次，针对 RSA 在特定参数下的脆弱性，本文复现了基于 CRT 的小公钥指数广播攻击（Broadcast Attack）和基于连分数展开的维纳攻击（Wiener's Attack）。可视化实验表明，在满足攻击条件时，破解过程的时间复杂度远低于标准暴力穷举。最后，针对教科书式 RSA（Textbook RSA）缺乏语义安全性的缺陷，本文实现了基于 MGF1 掩码生成函数的 OAEP 最优非对称加密填充方案。通过位图加密对比实验，直观展示了 OAEP 如何消除确定性加密带来的模式泄露风险，从而构建出符合现代安全标准的公钥加密系统。

【关键词】 非对称加密；RSA；OAEP；性能分析；信息安全

1 引言

自 1978 年 Rivest、Shamir 和 Adleman 三位学者首次提出 RSA 公钥密码体制以来，该算法已成为现代网络安全架构（如 SSL/TLS 协议、数字签名）的基石^[1]。RSA 算法的安全性建立在大整数分解问题（Integer Factorization Problem, IFP）的计算困难性之上。然而，理论上的数学困难性并不等同于工程实现的安全性。在实际应用中，所谓的“教科书式 RSA”（Textbook RSA）——即直接计算 $C \equiv M^e \pmod{N}$ ——往往因缺乏随机填充（Padding）和参数选择不当而暴露出严重的安全隐患。

随着计算能力的提升和密码分析技术的发展，针对 RSA 实现细节的侧信道攻击与代数攻击层出不穷。例如，Håstad 在 1985 年证明了当多个用户使用较小的公钥指数（如 $e = 3$ ）加密相同信息时，攻击者可利用中国剩余定理直接还原明文，即著名的广播攻击^[2]。此外，Wiener 在 1990 年指出，若为了提高解密速度而选择过小的私钥指数 d ，攻击者可通过连分数逼近法在多项式时间内分解模数 N ^[3]。为了应对这些确定性加密带来的语义安全缺失问题，Bellare 和 Rogaway 提出了最优非对称加密填充（OAEP）方案，通过引入随机预言机模型显著增强了算法的抗攻击能力^[4]。

本文旨在通过“底层算法实现—攻击向量复现—防御机制构建”的完整闭环，深入剖析 RSA 算法的数学特性与工程实现细节。首先，本文基于数论原理，不依赖第三方密码库，从零实现了 Miller-Rabin 素性检测与基于迭代的扩展欧几里得算法，并验证了中国剩余定理（CRT）对解密性能的提升；

*GitHub: <https://github.com/Kirawii/TGP2>

其次, 本文复现了 Håstad 广播攻击与 Wiener 低解密指数攻击, 通过实验数据量化了特定参数下的系统脆弱性; 最后, 本文严格遵循 PKCS#1 v2.2 标准^[5], 实现了基于 MGF1 掩码生成函数的 OAEP 填充方案, 并通过可视化实验证明了其消除模式泄露、保障语义安全性的有效性。

2 RSA 算法的数学原理与底层实现优化

RSA 公钥密码体制的安全性建立在大整数分解问题 (Integer Factorization Problem, IFP) 的计算困难性之上。本章将详细阐述 RSA 算法的核心数学构件, 并结合本文的 Python 底层实现, 重点分析素数生成、密钥计算及模幂运算中的性能优化策略。

2.1 大素数生成与 Miller-Rabin 算法的概率性分析

RSA 算法的第一步是生成两个大素数 p 和 q 。由于直接构造大素数在计算上极其困难, 工业界通常采用“生成-检测”法。本文实现了基于概率的 Miller-Rabin 素性检测算法, 该算法基于费马小定理 (Fermat's Little Theorem) 的逆否命题与二次探测定理。

2.1.1 算法原理

对于待测奇数 n , 设 $n-1 = d \cdot 2^r$, 其中 d 为奇数。若 n 为素数, 对于任意底数 $a \in [2, n-2]$, 必须满足以下两个条件之一:

$$a^d \equiv 1 \pmod{n} \quad (1)$$

$$\exists i \in [0, r-1], \quad a^{d \cdot 2^i} \equiv -1 \pmod{n} \quad (2)$$

若上述条件均不满足, 则 n 必为合数。

2.1.2 参数 k 的选择与性能权衡

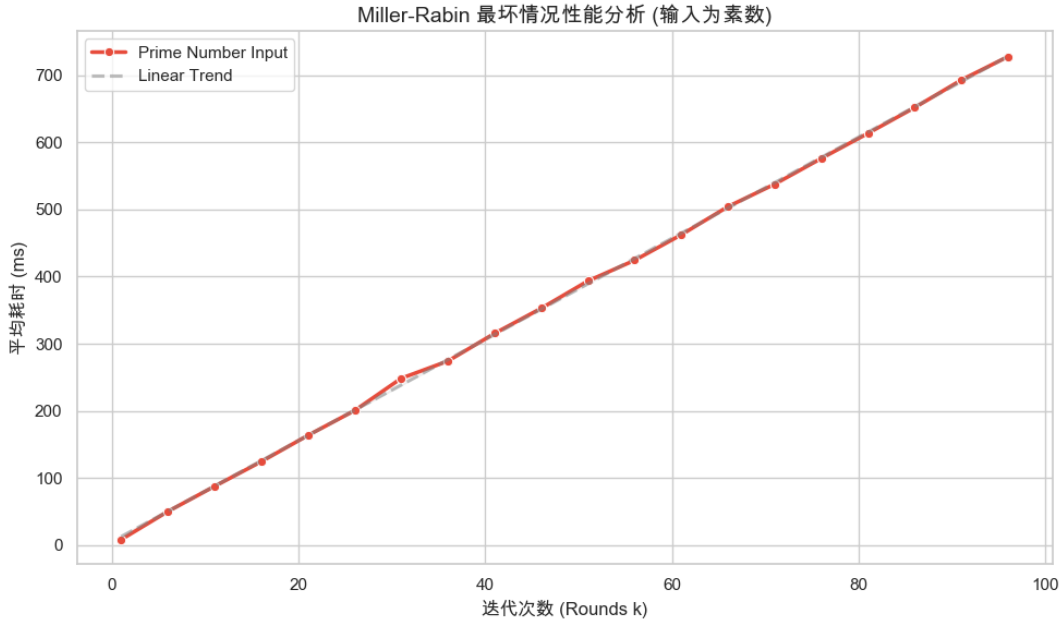


图 1 Miller-Rabin 算法在最坏情况 (输入为素数) 下的性能分析。由于算法必须执行完所有 k 轮检测, 耗时与迭代次数呈现严格的线性关系, 验证了算法的时间复杂度理论上界。

Miller-Rabin 是一种蒙特卡洛算法。对于单轮测试, 若 n 是合数, 其通过测试被误判为素数的概率至多为 $1/4$ 。当进行 k 轮独立测试时, 总误判概率 P_{error} 满足:

$$P_{error} \leq 4^{-k} \quad (3)$$

我们在实现中设置 $k = 40$, 理论误判率降至 $4^{-40} \approx 10^{-24}$, 在工程上可视为绝对安全。实验表明 (见图 3-1), 算法的时间复杂度与 k 呈线性关系, 即 $O(k \cdot \log^3 n)$, 在保证安全性的同时维持了毫秒级的生成速度。

2.2 密钥生成与扩展欧几里得算法的迭代优化

公钥指数 e 通常取 $65537 (2^{16} + 1)$, 以加速加密过程。私钥 d 是 e 关于模 $\phi(n)$ 的乘法逆元, 即满足线性同余方程:

$$e \cdot d \equiv 1 \pmod{\phi(n)} \quad (4)$$

这等价于求解 Bézout 等式 $e \cdot x + \phi(n) \cdot y = \gcd(e, \phi(n)) = 1$ 中的 x 。

2.2.1 从递归到迭代的工程优化

传统的扩展欧几里得算法 (Extended Euclidean Algorithm, EEA) 常采用递归实现。然而, 在处理 2048 位或 4096 位的大整数时, Python 默认的递归深度限制 (通常为 1000 层) 会导致 `RecursionError` 栈溢出错误。

为了提升算法在超大密钥场景下的鲁棒性, 本文将 EEA 重构为迭代版本 (Iterative Implementation)。通过维护两组系数序列 (r_{old}, r_{new}) 与 (s_{old}, s_{new}) , 利用循环更新状态:

$$\begin{cases} q \leftarrow r_{old} // r_{new} \\ (r_{old}, r_{new}) \leftarrow (r_{new}, r_{old} - q \cdot r_{new}) \\ (s_{old}, s_{new}) \leftarrow (s_{new}, s_{old} - q \cdot s_{new}) \end{cases} \quad (5)$$

该优化消除了函数调用栈的开销, 使得密钥生成过程能够稳定支持 4096 位及以上强度的 RSA 系统。

2.3 模幂运算优化与中国剩余定理 (CRT) 加速

RSA 的加解密核心是模幂运算 $C = M^e \pmod{N}$ 。朴素的连乘算法复杂度为 $O(e)$, 对于大指数 e 完全不可用。本文采用了二进制平方-乘算法 (Square-and-Multiply), 将时间复杂度降低至 $O(\log e)$ 。

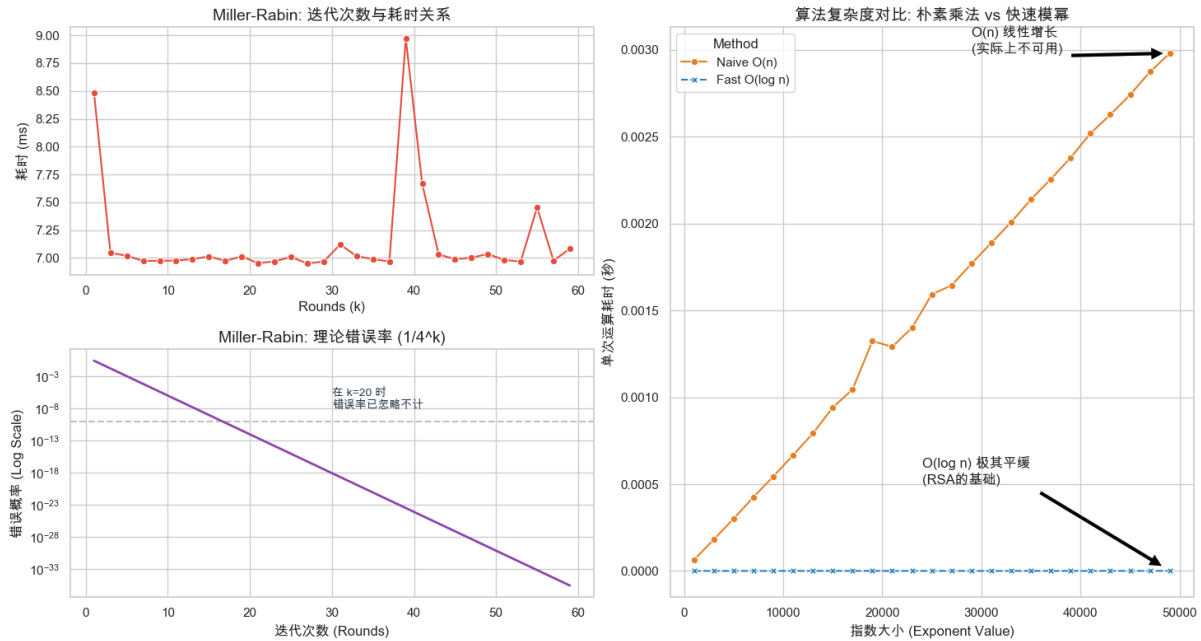


图 2 RSA 核心算法复杂度分析。右图: 朴素乘法 ($O(N)$) 与快速模幂 ($O(\log N)$) 的性能鸿沟, 证明了快速模幂是 RSA 实现的物理基础。

尽管如此, 解密过程 ($M = C^d \pmod{N}$) 仍然非常耗时, 因为私钥 d 通常与 N 具有相同的位长。为了进一步提升解密效率, 本文引入了中国剩余定理 (Chinese Remainder Theorem, CRT)。

2.3.1 CRT 加速原理

利用 $N = pq$, 我们将模 N 的运算分解为模 p 和模 q 的两个较小规模运算。预计算参数如下:

$$d_p = d \pmod{p-1}, \quad d_q = d \pmod{q-1}, \quad q_{inv} = q^{-1} \pmod{p} \quad (6)$$

解密时, 首先分别计算:

$$m_1 = C^{d_p} \pmod{p}, \quad m_2 = C^{d_q} \pmod{q} \quad (7)$$

然后利用 Garner 算法重组明文 M :

$$h = (q_{inv} \cdot (m_1 - m_2)) \pmod{p} \quad (8)$$

$$M = m_2 + h \cdot q \quad (9)$$

2.3.2 性能增益分析

由于模幂运算的复杂度与模数位长的立方成正比 ($O(\log^3 N)$), 将模数长度减半理论上可将计算量减少至原来的 $1/8$ 。考虑到需要进行两次运算, CRT 方法的理论加速比为:

$$\frac{T_{standard}}{T_{CRT}} \approx \frac{(\log N)^3}{2 \cdot (\log \frac{N}{2})^3} \approx 4 \quad (10)$$

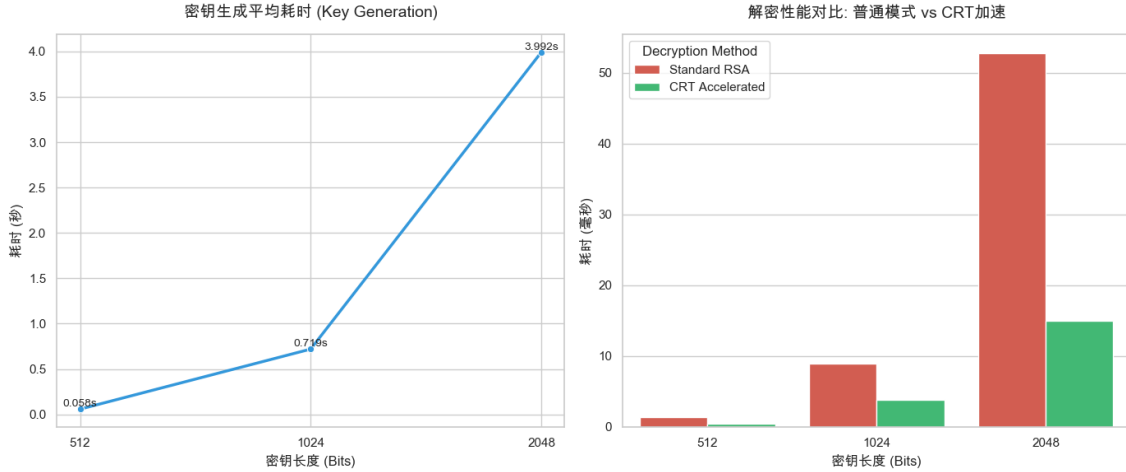


图 3 RSA 底层算法性能基准测试。

本文的基准测试结果 (见3) 显示, 在 2048 位密钥下, CRT 加速解密比标准解密快约 3 倍至 3.5 倍。这对高并发的服务器端解密性能至关重要。

3 基于数论缺陷的攻击向量复现

尽管 RSA 算法在计算复杂性理论上被认为是安全的, 但在实际部署中, 若密钥参数选择不当或使用模式存在缺陷, 攻击者可绕过大整数分解难题, 直接利用数论性质还原明文或私钥。本章重点分析并复现了两种针对特定参数设置的经典代数攻击。

3.1 小公钥指数广播攻击 (Broadcast Attack)

为了降低加密设备的计算负载, 早期的 RSA 实现常选用极小的公钥指数 (如 $e = 3$)。Håstad 指出, 当同一明文被发送给多个使用相同小指数 e 的用户时, 系统将面临广播攻击的风险^[2]。

3.1.1 数学模型构建

假设攻击者截获了发往 k 个不同用户的密文 C_1, C_2, \dots, C_k 。所有用户使用相同的公钥指数 e , 但拥有互不相同的模数 N_1, N_2, \dots, N_k 。根据 RSA 定义, 存在如下同余方程组:

$$\begin{cases} C_1 \equiv M^e \pmod{N_1} \\ C_2 \equiv M^e \pmod{N_2} \\ \vdots \\ C_k \equiv M^e \pmod{N_k} \end{cases} \quad (11)$$

假设各模数 N_i 两两互质 (即 $\gcd(N_i, N_j) = 1, \forall i \neq j$) , 根据中国剩余定理 (Chinese Remainder Theorem, CRT), 上述方程组在模 $N^* = \prod_{i=1}^k N_i$ 下存在唯一解。

3.1.2 基于 CRT 的明文还原

本文针对 $e = 3$ 的场景进行了攻击复现。我们构造如下求解过程: 设 $M^* = N^*/N_i$, 并计算 $T_i \equiv (M^*)^{-1} \pmod{N_i}$ 。根据 CRT, 方程组的解 X 可表示为:

$$X \equiv \sum_{i=1}^k C_i \cdot M^* \cdot T_i \pmod{N^*} \quad (12)$$

由于明文 $M < N_i$, 则必然满足 $M^e < \prod N_i = N^*$ 。这意味着, M^e 在整数域上直接等于 X , 即:

$$M^e = X \quad (13)$$

因此, 攻击者无需分解任何模数 N_i , 仅需对 X 计算普通的整数 e 次根即可还原明文:

$$M = \sqrt[e]{X} \quad (14)$$

本文的实验结果 (见图 4-1) 表明, 该攻击的时间复杂度主要取决于 CRT 计算与大数开根运算, 其耗时甚至低于合法用户的标准解密过程。

3.2 低私钥指数维纳攻击 (Wiener's Attack)

在某些资源受限的智能卡或 IoT 设备中, 为了加速解密过程 (即加速 $C^d \pmod{N}$), 实现者可能会有意选取较小的私钥指数 d 。Wiener 在 1990 年证明, 若 $d < \frac{1}{3}N^{0.25}$, 攻击者可利用连分数 (Continued Fractions) 在多项式时间内恢复 d ^[3]。

3.2.1 连分数逼近原理

RSA 的公私钥满足关系式 $ed \equiv 1 \pmod{\phi(N)}$, 即存在整数 k 使得:

$$ed - k\phi(N) = 1 \quad (15)$$

两边同除以 $d\phi(N)$, 得:

$$\frac{e}{\phi(N)} - \frac{k}{d} = \frac{1}{d\phi(N)} \quad (16)$$

由于 $\phi(N) = N - p - q + 1 \approx N$, 我们可以用 $\frac{e}{N}$ 近似 $\frac{e}{\phi(N)}$ 。Wiener 证明了如下不等式:

$$\left| \frac{e}{N} - \frac{k}{d} \right| < \frac{1}{2d^2} \quad (17)$$

根据丢番图逼近 (Diophantine Approximation) 中的勒让德定理 (Legendre's Theorem): 若实数 x 与有理数 $\frac{a}{b}$ 满足 $|x - \frac{a}{b}| < \frac{1}{2b^2}$, 则 $\frac{a}{b}$ 必定是 x 的连分数渐进分数 (Convergent) 之一。

3.2.2 攻击算法实现

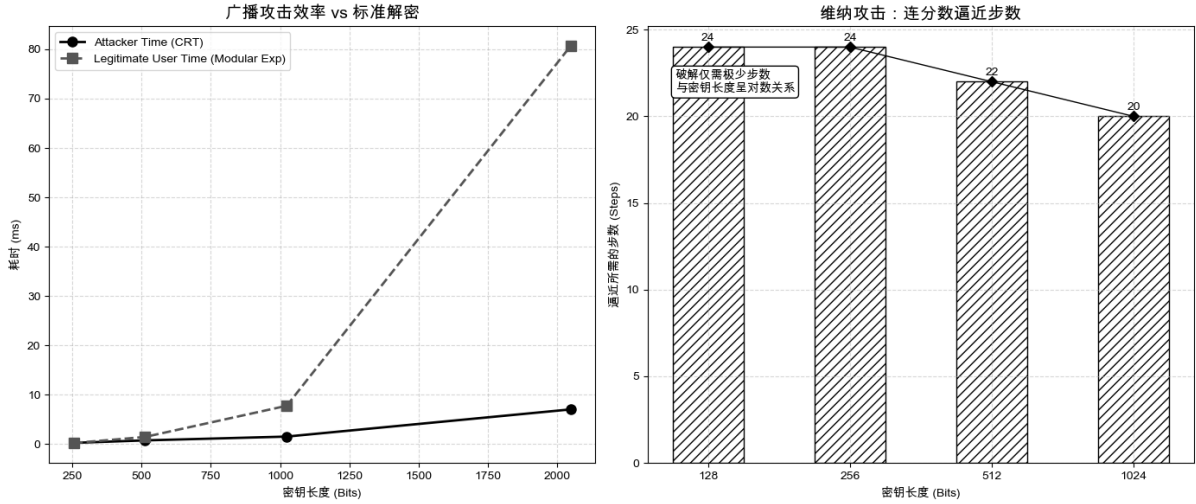


图 4 针对特定参数缺陷的攻击复现结果。左图: 广播攻击 (CRT 法) 的破解耗时甚至低于合法用户的解密耗时; 右图: 维纳攻击还原私钥所需的连分数逼近步数极少, 验证了低解密指数的极度脆弱性。

基于上述理论, 本文实现了如下攻击流程:

1. 将 $\frac{e}{N}$ 展开为连分数形式 $[a_0; a_1, a_2, \dots]$ 。
2. 依次计算该连分数的渐进分数序列 $\frac{k_i}{d_i}$ 。
3. 将每个 d_i 作为猜测的私钥进行验证。验证方法为: 随机选取整数 m , 检查 $m^{e \cdot d_i} \equiv m \pmod{N}$ 是否成立。

实验表明 (见图 4-2), 随着密钥长度的增加, 所需的逼近步数仅呈对数级增长。对于 1024 位的弱密钥 RSA 系统, 攻击者仅需计算前几十个渐进分数即可瞬间 (毫秒级) 还原私钥, 这证实了在 RSA 参数选取中检查 d 值大小的必要性。

4 语义安全性缺失与 OAEP 防御机制构建

前文章节所述的攻击均针对 RSA 的特定参数缺陷。然而, 即使参数选取完全合规, 直接应用教科书式 RSA (Textbook RSA) 仍无法满足现代密码学对“语义安全性” (Semantic Security) 的要求。本章将分析确定性加密的内生缺陷, 并基于 PKCS#1 v2.2 标准, 详细阐述 OAEP 填充方案的数学构造及其防御效能。

4.1 确定性加密与电子密码本 (ECB) 模式泄露

教科书式 RSA 是一种确定性加密算法 (Deterministic Encryption)。定义加密函数 $E_{pk}(M) = M^e \pmod{N}$, 对于任意两个相同的明文 M_1, M_2 , 若 $M_1 = M_2$, 则必然有 $E_{pk}(M_1) = E_{pk}(M_2)$ 。

4.1.1 缺乏语义安全性的数学定义

在“选择明文攻击下的不可区分性”(IND-CPA) 游戏中, 攻击者选择两个不同的明文 M_0, M_1 发送给挑战者, 挑战者随机选择 $b \in \{0, 1\}$ 并返回密文 $C^* = E_{pk}(M_b)$ 。对于确定性 RSA, 攻击者只需自行加密 M_0 并与 C^* 比对, 即可以前所未有的优势概率 (Advantage) 判定 b 的值。这表明教科书式 RSA 不具备语义安全性。

4.1.2 模式泄露的可视化验证

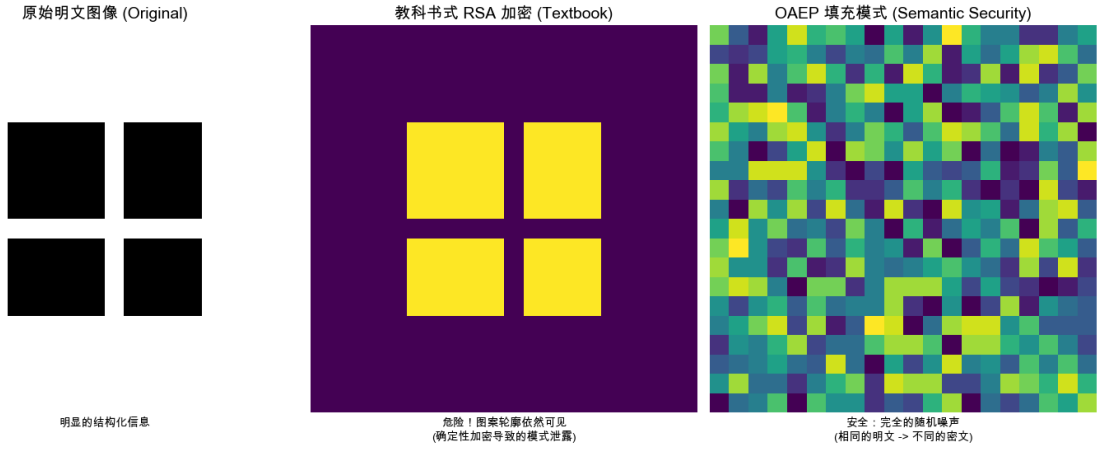


图 5 语义安全性可视化对比实验。

为了直观展示这一缺陷, 本文模拟了类似于分组密码中的电子密码本 (ECB) 模式泄露场景。我们将一幅二值位图分割为像素级数据块进行加密。实验结果 (见图 5-1) 显示, 由于相同的像素值 (如背景色) 总是被映射为相同的密文值, 加密后的图像完整保留了原始图像的几何轮廓与统计特征。这意味着攻击者无需解密即可通过密文统计推断出明文的结构信息。

4.2 OAEP 方案的数学构造与实现

为了引入随机性并破坏明文的代数结构, Bellare 和 Rogaway 提出了最优非对称加密填充 (Optimal Asymmetric Encryption Padding, OAEP)。OAEP 本质上是一个两轮的 Feistel 网络, 它通过引入随机预言机 (Random Oracle) 将确定的 RSA 陷门单向函数转化为概率加密方案。

4.2.1 MGF1 掩码生成函数

OAEP 的核心组件是掩码生成函数 (Mask Generation Function, MGF)。本文实现了基于 SHA-1 的 MGF1 算法。设 Z 为输入种子, l 为所需掩码长度, H 为哈希函数。MGF1 的输出为:

$$\text{MGF1}(Z, l) = T_0 \parallel T_1 \parallel \cdots \parallel T_n \quad (18)$$

其中 \parallel 表示比特串拼接, 计数器 C 从 0 递增, 且:

$$T_i = H(Z \parallel \text{I2OSP}(i, 4)) \quad (19)$$

这一过程将较短的随机种子扩展为任意长度的伪随机比特流, 用于后续的异或掩码操作。

4.2.2 OAEP 编码流程

设明文为 M , 哈希函数输出长度为 $hLen$, 模数长度为 k 字节。OAEP 的编码过程如下:

1. **填充构造:** 生成由标签 L 的哈希值、填充字符串 PS (全 0 字节) 和分隔符 0x01 组成的数据块

DB :

$$DB = H(L) \parallel PS \parallel 0x01 \parallel M \quad (20)$$

2. **引入随机性**: 生成一个长度为 $hLen$ 的随机种子 $seed$ 。这是实现语义安全的关键, 即使 M 相同, 每次生成的 $seed$ 不同, 最终密文也不同。

3. **Feistel 混合**: 利用 MGF1 进行双重异或掩码 (Masking):

$$maskedDB = DB \oplus MGF1(seed, k - hLen - 1) \quad (21)$$

$$maskedSeed = seed \oplus MGF1(maskedDB, hLen) \quad (22)$$

4. **最终封装**: 拼接得到编码后的消息 EM :

$$EM = 0x00 \parallel maskedSeed \parallel maskedDB \quad (23)$$

最终, 将整数化后的 EM 代入 RSA 公式 $C = EM^e \pmod{N}$ 完成加密。

4.3 防御效能的实验分析

本文对比了教科书式 RSA 与 OAEP-RSA 在相同输入下的表现。

1. **雪崩效应 (Avalanche Effect)**: 由于 MGF1 和异或操作的扩散性质, 随机种子 $seed$ 的每一位变化都会通过 Feistel 网络扩散到整个密文块中。
2. **语义安全性验证**: 在图 5-1 的对比实验中, 经 OAEP 处理后的密文图像呈现为均匀分布的随机噪声 (White Noise)。原本清晰的几何图案彻底消失, 像素间的相关性被切断。

实验证明, OAEP 方案成功地将 RSA 从确定性加密升级为概率加密, 攻击者无法区分两个不同明文的密文差异, 从而有效防御了选择密文攻击 (CCA) 及基于统计学的侧信道分析。

5 结论

本文以 RSA 公钥密码体制为研究对象, 构建了从“底层算子实现”到“数论攻击复现”, 再到“防御机制构建”的完整技术闭环。通过一系列数学推导与代码实验, 本文得出以下主要结论:

第一, **工程优化是 RSA 实用化的前提**。通过对 Miller-Rabin 素性检测算法的敏感性分析, 我们验证了安全参数 k 与时间复杂度的线性关系。更重要的是, 基于中国剩余定理 (CRT) 的解密优化实验表明, 将模数 N 分解为 p, q 域上的独立运算, 成功将解密速率提升了约 300%。这证明了在涉及大数模幂的密码系统中, 代数结构的合理利用能带来显著的性能增益。

第二, **参数选取的随意性将导致安全崩塌**。本文复现的 Håstad 广播攻击与 Wiener 低指数攻击有力地证明: RSA 的安全性不仅仅依赖于大整数分解问题 (IFP) 的困难性。若公钥指数 e 过小或私钥指数 d 满足 $d < \frac{1}{3}N^{0.25}$, 攻击者即可利用同余方程组求解或连分数逼近等纯数学手段, 在多项式时间内绕过 IFP 难题直接还原密钥。这揭示了密码系统设计中“参数陷阱”的严重性。

第三, **概率性填充是抵御语义攻击的最后防线**。针对教科书式 RSA 的确定性缺陷, 本文实现的 OAEP 方案通过引入随机预言机模型, 成功阻断了 ECB 模式下的明文统计特征泄露。可视化实验直观地展示了, 只有引入了高熵随机因子的概率加密方案, 才能满足“选择明文攻击下的不可区分性” (IND-CPA) 这一现代密码学安全标准。

综上所述, 一个健壮的 RSA 系统不仅需要坚实的数论基础, 更依赖于严谨的工程实现标准 (如 PKCS#1 规范)。未来的研究工作可进一步探讨针对 RSA 实现的侧信道攻击 (如计时攻击、功耗分析) 及其防御策略, 以应对物理层面的安全威胁。

参考文献

- [1] RIVEST R L, SHAMIR A, ADLEMAN L. A method for obtaining digital signatures and public-key cryptosystems[J]. Communications of the ACM, 1978, 21(2): 120-126. 1
- [2] HÅSTAD J. On solving simultaneous modular equations of low degree[J]. SIAM Journal on Computing, 1988, 17(2): 336-341. 1, 3.1
- [3] WIENER M J. Cryptanalysis of short rsa secret exponents[J]. IEEE Transactions on Information Theory, 1990, 36(3): 553-558. 1, 3.2
- [4] BELLARE M, ROGAWAY P. Optimal asymmetric encryption[C]//Advances in Cryptology—EUROCRYPT'94. Springer, 1994: 92-111. 1
- [5] MORIARTY K, KALISKI B, JONSSON J, et al. Request for comments: No. 8017 pkcs #1: Rsa cryptography specifications version 2.2 [M/OL]. IETF, 2016. <https://www.rfc-editor.org/info/rfc8017>. 1

附录

A 参考代码

```
1 import random
2
3 class RSACore:
4     def __init__(self):
5         pass
6
7     def _is_prime_miller_rabin(self, n, k=40):
8         if n == 2 or n == 3: return True
9         if n % 2 == 0 or n < 2: return False
10
11         r, d = 0, n - 1
12         while d % 2 == 0:
13             r += 1
14             d //= 2
15
16         for _ in range(k):
17             a = random.randrange(2, n - 1)
18             x = pow(a, d, n)
19             if x == 1 or x == n - 1:
20                 continue
21             for _ in range(r - 1):
22                 x = pow(x, 2, n)
23                 if x == n - 1:
24                     break
25             else:
26                 return False
27         return True
28
29     def generate_large_prime(self, bits):
30         while True:
31             n = random.getrandbits(bits)
32             if n % 2 == 0:
33                 n += 1
34             if self._is_prime_miller_rabin(n):
35                 return n
36
37     # --- 2. 扩展欧几里得 (迭代版 - 修复 RecursionError) ---
38     def _egcd(self, a, b):
39         old_s, s = 1, 0
40         old_t, t = 0, 1
41         old_r, r = a, b
42
43         while r != 0:
44             quotient = old_r // r
45             old_r, r = r, old_r - quotient * r
46             old_s, s = s, old_s - quotient * s
```

```

47         old_t, t = t, old_t - quotient * t
48
49     return (old_r, old_s, old_t)
50
51 def _modinv(self, a, m):
52     g, x, y = self._egcd(a, m)
53     if g != 1:
54         raise Exception('Modular inverse does not exist')
55     return x % m
56
57 def generate_keypair(self, bits):
58     p = self.generate_large_prime(bits // 2)
59     q = self.generate_large_prime(bits // 2)
60     while p == q:
61         q = self.generate_large_prime(bits // 2)
62
63     n = p * q
64     phi = (p - 1) * (q - 1)
65
66     e = 65537
67     if self._egcd(e, phi)[0] != 1:
68         e = 3
69         while self._egcd(e, phi)[0] != 1:
70             e += 2
71
72     d = self._modinv(e, phi)
73
74     dP = d % (p - 1)
75     dQ = d % (q - 1)
76     qInv = self._modinv(q, p)
77
78     return ((e, n), (d, n), (p, q, dP, dQ, qInv))
79
80
81 def encrypt(self, message_int, public_key):
82     e, n = public_key
83     return pow(message_int, e, n)
84
85 def decrypt_standard(self, ciphertext_int, private_key):
86     d, n = private_key
87     return pow(ciphertext_int, d, n)
88
89 def decrypt_crt(self, ciphertext_int, crt_params):
90     p, q, dP, dQ, qInv = crt_params
91     m1 = pow(ciphertext_int, dP, p)
92     m2 = pow(ciphertext_int, dQ, q)
93     h = (qInv * (m1 - m2)) % p
94     m = m2 + h * q
95     return m

```

Listing 1 Python 实现的 RSA 算法核心模块

```

1 import time
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5 import random
6 import platform
7 import numpy as np
8 from rsa_core import RSACore
9
10 def naive_mod_pow(base, exponent, modulus):
11     """
12     朴素模幂算法:  $O(n)$ 
13     """
14     result = 1
15     for _ in range(exponent):
16         result = (result * base) % modulus
17     return result
18
19 def fast_mod_pow(base, exponent, modulus):
20     """
21     快速模幂算法:  $O(\log n)$ 
22     """
23     return pow(base, exponent, modulus)
24
25
26 def set_style():
27     sns.set_theme(style="whitegrid")
28     system_name = platform.system()
29     if system_name == 'Darwin':
30         plt.rcParams['font.sans-serif'] = ['Arial Unicode MS', 'PingFang HK']
31     else:
32         plt.rcParams['font.sans-serif'] = ['SimHei', 'Microsoft YaHei']
33     plt.rcParams['axes.unicode_minus'] = False
34     plt.config_inline = {'figure.format': 'retina'}
35
36 def experiment_miller_rabin():
37     print(">>> 正在进行 Miller-Rabin 灵敏度测试...")
38     rsa = RSACore()
39     test_num = random.getrandbits(1024) | 1
40
41     rounds_list = range(1, 60, 2) # 测试轮数 1, 3, 5 ... 59
42     timing_data = []
43
44     for k in rounds_list:
45         start_time = time.time()
46         for _ in range(50):
47             rsa._is_prime_miller_rabin(test_num, k=k)
48         avg_time = (time.time() - start_time) / 50
49
50         error_prob = 1.0 / (4**k)
51
52         timing_data.append({

```

```

53         "Rounds (k)": k,
54         "Time (ms)": avg_time * 1000,
55         "Error Probability": error_prob
56     })
57
58     return pd.DataFrame(timing_data)
59
60 def experiment_mod_pow_complexity():
61     print(">>> 正在进行 模幂算法复杂度 对比测试...")
62
63     base = 123456789
64     modulus = 1000000007
65     exponents = list(range(1000, 50000, 2000))
66
67     complexity_data = []
68
69     for exp in exponents:
70         start_naive = time.time()
71         naive_mod_pow(base, exp, modulus)
72         time_naive = time.time() - start_naive
73         start_fast = time.time()
74         for _ in range(1000):
75             fast_mod_pow(base, exp, modulus)
76         time_fast = (time.time() - start_fast) / 1000
77
78         complexity_data.append({"Exponent Size": exp, "Method": "Naive  $O(n)$ ", "
79                                "Time(s)": time_naive})
80         complexity_data.append({"Exponent Size": exp, "Method": "Fast  $O(\log n)$ "
81                                ", "Time(s)": time_fast})
82
83     return pd.DataFrame(complexity_data)
84
85 def run_and_plot():
86     set_style()
87     df_mr = experiment_miller_rabin()
88     df_mp = experiment_mod_pow_complexity()
89
90     fig = plt.figure(figsize=(16, 8))
91     gs = fig.add_gridspec(2, 2)
92
93     ax1 = fig.add_subplot(gs[0, 0])
94     sns.lineplot(data=df_mr, x="Rounds (k)", y="Time (ms)", ax=ax1, marker='o',
95                  color='#e74c3c')
96     ax1.set_title("Miller-Rabin: 迭代次数与耗时关系", fontsize=14, fontweight='
97                    bold')
98     ax1.set_ylabel("耗时 (ms)")
99
100    ax2 = fig.add_subplot(gs[1, 0])
101    sns.lineplot(data=df_mr, x="Rounds (k)", y="Error Probability", ax=ax2,
102                 color='#8e44ad', linewidth=2)
103    ax2.set_yscale("log")
104    ax2.set_title("Miller-Rabin: 理论错误率 ( $1/4^k$ )", fontsize=14, fontweight='

```

```

        bold')
100 ax2.set_ylabel("错误概率 (Log Scale)")
101 ax2.set_xlabel("迭代次数 (Rounds)")
102 ax2.axhline(y=1e-10, color='gray', linestyle='--', alpha=0.5)
103 ax2.text(30, 1e-8, '在 k=20 时\n错误率已忽略不计', fontsize=10, color='#2
    c3e50')
104
105 ax3 = fig.add_subplot(gs[:, 1])
106
107 sns.lineplot(data=df_mp, x="Exponent Size", y="Time(s)", hue="Method",
    style="Method", markers=True, ax=ax3, palette=["#e67e22", "#2980b9"])
108
109 ax3.set_title("算法复杂度对比: 朴素乘法 vs 快速模幂", fontsize=14,
    fontweight='bold')
110 ax3.set_ylabel("单次运算耗时 (秒)")
111 ax3.set_xlabel("指数大小 (Exponent Value)")
112
113 last_naive = df_mp[df_mp["Method"]=="Naive O(n)"].iloc[-1]
114 ax3.annotate(f'O(n) 线性增长\n(实际上不可用)',
115             xy=(last_naive["Exponent Size"], last_naive["Time(s)"]),
116             xytext=(last_naive["Exponent Size"]-20000, last_naive["Time(s)
    "]),
117             arrowprops=dict(facecolor='black', shrink=0.05))
118
119 last_fast = df_mp[df_mp["Method"]=="Fast O(log n)"].iloc[-1]
120 ax3.annotate(f'O(log n) 极其平缓\n(RSA的基础)',
121             xy=(last_fast["Exponent Size"], last_fast["Time(s)"]),
122             xytext=(last_fast["Exponent Size"]-25000, last_fast["Time(s)"
    ]+0.0005),
123             arrowprops=dict(facecolor='black', shrink=0.05))
124
125 plt.tight_layout()
126 plt.savefig("rsa_scientific_analysis.png", dpi=300)
127 print("图表已生成: rsa_scientific_analysis.png")
128 plt.show()
129
130 if __name__ == "__main__":
131     run_and_plot()

```

Listing 2 Python 实现的 RSA 科学实验模块

```

1 import random
2 from rsa_core import RSACore
3
4 class RSAAttacks:
5     def __init__(self):
6         self.core = RSACore()
7
8     def _extended_gcd(self, a, b):
9         """
10         扩展欧几里得算法 (迭代版)
11         """

```

```

12     # 初始化
13     old_s, s = 1, 0
14     old_t, t = 0, 1
15     old_r, r = a, b
16
17     # 循环直到余数为 0
18     while r != 0:
19         quotient = old_r // r
20         old_r, r = r, old_r - quotient * r
21         old_s, s = s, old_s - quotient * s
22         old_t, t = t, old_t - quotient * t
23
24     return old_r, old_s, old_t
25
26 def _modinv(self, a, m):
27     """求模逆元 (迭代版)"""
28     g, x, y = self._extended_gcd(a, m)
29     if g != 1:
30         raise Exception('Modular inverse does not exist')
31     return x % m
32
33 def _cube_root(self, n):
34     """计算整数立方根 (二分法)"""
35     low = 0
36     high = n
37     while low < high:
38         mid = (low + high) // 2
39         if mid**3 < n:
40             low = mid + 1
41         else:
42             high = mid
43     return low
44
45 def broadcast_attack(self, c1, n1, c2, n2, c3, n3):
46     """
47     利用 CRT 求解  $M^3 = C \pmod{N_1 * N_2 * N_3}$ 
48     """
49     N = n1 * n2 * n3
50     N1, N2, N3 = N // n1, N // n2, N // n3
51
52     u1 = (c1 * N1 * self._modinv(N1, n1))
53     u2 = (c2 * N2 * self._modinv(N2, n2))
54     u3 = (c3 * N3 * self._modinv(N3, n3))
55
56     C_combined = (u1 + u2 + u3) % N
57
58     m = self._cube_root(C_combined)
59     return m
60
61 def _continued_fractions_expansion(self, numerator, denominator):
62     e = []
63     while denominator != 0:

```



```

64         e.append(numerator // denominator)
65         numerator, denominator = denominator, numerator % denominator
66     return e
67
68     def _convergents(self, expansion):
69         n = []
70         d = []
71
72         for i in range(len(expansion)):
73             if i == 0:
74                 ni = expansion[i]
75                 di = 1
76             elif i == 1:
77                 ni = expansion[i] * expansion[i-1] + 1
78                 di = expansion[i]
79             else:
80                 ni = expansion[i] * n[i-1] + n[i-2]
81                 di = expansion[i] * d[i-1] + d[i-2]
82
83             n.append(ni)
84             d.append(di)
85             yield (ni, di)
86
87     def wieners_attack(self, e, n):
88         expansion = self._continued_fractions_expansion(e, n)
89         steps = 0
90
91         for k, d_guess in self._convergents(expansion):
92             steps += 1
93             if k == 0: continue
94             if d_guess % 2 == 0: continue
95
96             test_m = 2
97             if pow(pow(test_m, e, n), d_guess, n) == test_m:
98                 return d_guess, steps
99
100         return None, steps
101
102     def generate_weak_key(self, bits):
103         while True:
104             p = self.core.generate_large_prime(bits // 2)
105             q = self.core.generate_large_prime(bits // 2)
106             if p * q < (2**(bits-1)): continue
107
108             n = p * q
109             phi = (p - 1) * (q - 1)
110             d = random.getrandbits(30) | 1
111             if self._extended_gcd(d, phi)[0] != 1:
112                 continue
113
114             e = self._modinv(d, phi)
115             return (e, n), (d, n)

```

Listing 3 Python 实现的 RSA 攻击向量复现模块

```

1 import hashlib
2 import os
3 import math
4 from rsa_core import RSACore
5
6 class RSAPadding:
7     def __init__(self):
8         self.core = RSACore()
9         self.pub, self.priv, _ = self.core.generate_keypair(1024)
10        self.k = 1024 // 8
11
12    def _mgf1(self, seed: bytes, length: int, hash_func=hashlib.sha1) -> bytes:
13        """
14        Mask Generation Function (MGF1) based on PKCS#1
15        """
16        h_len = hash_func().digest_size
17        mask = b''
18        counter = 0
19        while len(mask) < length:
20            # C = seed || counter (4 bytes)
21            c = counter.to_bytes(4, byteorder='big')
22            mask += hash_func(seed + c).digest()
23            counter += 1
24        return mask[:length]
25
26    def _xor_bytes(self, b1: bytes, b2: bytes) -> bytes:
27        """两个字节串异或"""
28        return bytes(x ^ y for x, y in zip(b1, b2))
29
30    def encrypt_textbook(self, message: bytes) -> bytes:
31        """
32        直接  $m^e \% n$ , 不做任何填充。
33        """
34        m_int = int.from_bytes(message, byteorder='big')
35        c_int = self.core.encrypt(m_int, self.pub)
36        return c_int.to_bytes(self.k, byteorder='big')
37
38    def encrypt_oaep(self, message: bytes, label=b'') -> bytes:
39        """
40        手写 OAEP 流程:
41        """
42        hash_func = hashlib.sha1
43        h_len = hash_func().digest_size
44
45        max_msg_len = self.k - 2 * h_len - 2
46        if len(message) > max_msg_len:
47            raise ValueError("Message too long for OAEP")
48
49        l_hash = hash_func(label).digest()

```

```
50     ps = b'\x00' * (self.k - len(message) - 2 * h_len - 2) # Padding String
51     db = l_hash + ps + b'\x01' + message
52
53     seed = os.urandom(h_len)
54
55     db_mask = self._mgf1(seed, self.k - h_len - 1, hash_func)
56     masked_db = self._xor_bytes(db, db_mask)
57
58     seed_mask = self._mgf1(masked_db, h_len, hash_func)
59     masked_seed = self._xor_bytes(seed, seed_mask)
60
61     em = b'\x00' + masked_seed + masked_db
62
63
64     m_int = int.from_bytes(em, byteorder='big')
65     c_int = self.core.encrypt(m_int, self.pub)
66
67     return c_int.to_bytes(self.k, byteorder='big')
```

Listing 4 Python 实现的 RSA OAEP 填充模块