西南财经大学

SOUTHWESTERN UNIVERSITY OF FINANCE AND ECONOMICS

# DES 与 IDEA 加密算法的实现、比较与性能分析

（期中论文）


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：杨蕴涵、许桐恺、刘俊宏

对应学号：42311167、42311038、42311180


完成日期：2025 年 11 月


西南财经大学

Southwestern University of Finance and Economics

2025 年 11 月

# DES 与 IDEA 加密算法的实现、比较与性能分析

许桐恺* 杨蕴涵* 刘俊宏*

【摘 要】 本文围绕经典对称加密算法数据加密标准 (DES) 与国际数据加密算法 (IDEA) 展开了系统的理论分析、C++ 编程实现与性能评测研究。研究对比了 DES 的 Feistel 网络结构与 IDEA 创新的“混合代数运算”结构，以及它们在密钥长度和抗密码分析方面的安全性差异。基于 C++ 完整实现了两种算法的核心逻辑，并在 Linux/Intel 单核 CPU 环境下进行了实证测试。结果显示，C++ 实现的 IDEA 算法性能显著优于 DES，其加解密总耗时（约 140 ms）比 DES（约 1950 ms）快一个数量级。性能分析表明，IDEA 算法的整数算术混合运算结构更适应现代 CPU 流水线与 ALU 优化。本研究验证了算法结构设计哲学与现代计算架构之间的密切关系，为加密算法的选型与实现策略提供了重要参考。

【关键词】 对称加密；DES；IDEA；性能分析；信息安全

## 1 引 言

### 1.1 研究背景与意义

**研究背景：**我们正处在一个以数据为核心驱动力的数字化时代。从个人隐私到金融交易，从企业运营到国家安全，数字信息的安全传输与存储已成为社会正常运转的基石。然而，开放的网络环境也使得数据面临着前所未有的窃听、篡改和伪造风险。密码学，特别是其中的对称加密技术，是应对这些威胁、保障数据机密性的核心手段。对称加密算法因其加解密效率高、资源消耗低的特点，被广泛应用于数据库加密、安全通信协议（如 TLS/SSL）以及文件系统加密等大数据量处理场景。

在对称加密算法的发展长河中，数据加密标准（DES）和国际数据加密算法（IDEA）是两个具有里程碑意义的算法。DES 作为第一个被广泛采纳的国际加密标准，统治了该领域长达二十余年，对现代密码学的商业化和学术研究产生了深远影响。而 IDEA 则是在 DES 的安全性受到挑战时应运而生的杰出代表，其创新的设计理念和卓越的安全强度使其成为后续加密算法设计的重要参考。

**研究意义：**本研究旨在通过对 DES 和 IDEA 这两种经典算法进行深入比较，其意义主要体现在以下三个方面：

1. **理论意义：**通过剖析两种算法在设计哲学上的根本差异——DES 的 Feistel 网络结构与 IDEA 的“混合代数运算”结构——可以深刻理解对称密码的设计原则、演化路径以及安全性与效率之间的权衡。这对于学习和理解后续更先进的算法（如 AES）具有重要的启发价值。

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

2. **实践意义**：尽管 DES 已不再安全，但研究其从辉煌到被淘汰的过程，尤其是其 56 位密钥长度的致命缺陷，是信息安全教育中一个经典的警示案例。本论文通过 C++ 从零开始实现这两种算法，不仅能加深对算法内部机制的理解，还能锻炼底层编程和软件性能优化的能力。

3. **学术价值**：本文将理论分析与实证测试相结合。通过对自行实现的算法和业界优化的 Python 标准库进行性能基准测试，可以量化地揭示理论效率与实际工程实现之间的差距，为加密算法在特定应用场景下的选型提供一定的参考依据。

### 1.2 相关工作与文献综述

密码学界对 DES 和 IDEA 的研究已相当深入。早期工作主要集中于对算法本身的密码分析。针对 DES 算法，其官方标准由美国国家标准局在 **FIPS PUB 46**[1] 中发布。在安全性分析方面，最著名的工作莫过于 **Biham 和 Shamir** 提出的差分密码分析[2] 以及 **Matsui** 提出的线性密码分析[3]。这两项工作从理论上证明了存在比暴力破解更有效的攻击手段，揭示了 DES 在设计上的一些脆弱性，并直接推动了现代分组密码分析理论的发展。而**电子前沿基金会（EFF）**[4] 在 1999 年成功实践的暴力破解，则从工程上宣告了 DES 时代的终结。

针对 IDEA 算法，其设计由 **Lai 和 Massey**[5] 在 1991 年的论文《一种新的分组加密标准提案》中首次提出。该算法在设计之初就考虑了对差分密码分析的抵抗能力，其安全性得到了广泛的论证。后续研究，如 **Daemen 等人**的工作[6]，确认了 IDEA 对差分和线性密码分析均具有很高的抵抗力。尽管存在一些针对简化轮数（如 3 轮或 4 轮）IDEA 的理论攻击，但对于完整的 8.5 轮 IDEA，至今未发现任何比暴力破解更有效的攻击方法，其 128 位的密钥长度也保证了对穷举攻击的免疫力。

在性能比较方面，已有大量文献在不同平台（如 CPU、FPGA、ASIC）上对 DES 和 IDEA 的运行效率进行了评估。多数研究表明，由于 DES 主要依赖于位操作（置换和异或），在硬件实现上具有天然的速度优势。而在纯软件实现中，IDEA 包含的模乘运算通常会成为性能瓶颈，导致其速度慢于 DES。然而，随着现代 CPU 指令集的不断优化，这种性能差距可能发生变化。本研究将在现代计算机体系结构下，通过高级编程语言的实现来重新审视这一性能对比，并与高度优化的专业库进行比较，从而为这一经典议题提供最新的实证数据。

### 1.3 本文主要工作与贡献

基于上述背景和研究现状，本文旨在完成以下主要工作，并做出相应贡献：

1. **系统性的理论对比**：本文将从算法结构、密钥调度、核心运算等多个角度，对 DES 和 IDEA 的步骤复杂度和设计思想进行系统性的梳理和对比。

2. **深入的安全性剖析**：综合分析两种算法在密钥长度、抗差分/线性分析能力、以及是否存在弱密钥等方面的安全性差异，并阐述这些差异背后的设计原因。

3. **算法的编程实现**：基于 C++ 语言，从零开始完整地实现 DES 和 IDEA 两种加密算法的核心逻辑，包括数据填充、密钥生成和加解密全过程，以达到对算法内部机制的精确掌握。

4. **多维度的性能评测**：设计并实施一套性能测试方案，通过对不同大小的数据文件进行加密操作计时，完成以下两组核心对比：

    - **横向对比**：比较本文实现的 DES 与 IDEA 在相同软硬件环境下的运行效率。
    - **纵向对比**：将本文 C++ 实现的算法性能与 Python 标准加密库中的同类算法进行比较，分析手写实现与专业优化库之间的性能差距。

本文的主要贡献在于，将经典的密码学理论比较与现代编程实践和性能评测相结合，不仅提供了一份关于 DES 与 IDEA 的全面对比分析报告，更通过可复现的实证数据，为理解这两个里程碑式算法在当前计算环境下的真实性能表现提供了有价值的参考。

## 2 算法的数学原理与安全性分析

任何密码算法的安全性都根植于其底层的数学原理。本章旨在从数学和密码学的角度，深入剖析 DES 和 IDEA 的设计精髓，并基于这些原理对其安全性进行评估。我们将重点阐述 Feistel 网络结构、非线性 S 盒、不同代数群操作混合等核心概念，并解释它们如何为算法贡献混淆（Confusion）与扩散（Diffusion）这两个关键特性。

### 2.1 DES 的数学原理与安全性

#### 2.1.1 数学原理：Feistel 网络与非线性替换

DES 的安全性主要依赖于两个核心组件的协同工作：Feistel 网络结构和高非线性的轮函数 $F$。

**1. Feistel 网络结构** Feistel 网络是一种对称结构，用于构建分组密码。其巧妙之处在于，它将加密过程分解为多轮迭代，并且每一轮的轮函数 $F$ 无需自身可逆，整个加密过程却天然可逆。对于第 $i$ 轮迭代，其数学表达式为：

$$
\begin{aligned}
L_i &= R_{i-1} \\
R_i &= L_{i-1} \oplus F(R_{i-1}, K_i)
\end{aligned}
$$

其中 $L_{i-1}$ 和 $R_{i-1}$ 是上一轮输出的左右两半，$\oplus$ 代表按位异或，$K_i$ 是本轮的子密钥。

其可逆性可以通过简单的代数推导证明。要从 $(L_i, R_i)$ 恢复 $(L_{i-1}, R_{i-1})$，我们可以看到：

$$
\begin{aligned}
R_{i-1} &= L_i \\
L_{i-1} &= R_i \oplus F(R_{i-1}, K_i) = R_i \oplus F(L_i, K_i)
\end{aligned}
$$

可以看到，解密过程与加密过程使用了完全相同的结构，只需将子密钥 $K_i$ 按相反的顺序 $(K_{16}, K_{15}, \dots, K_1)$ 应用即可。这一优雅的对称性极大地简化了 DES 的硬件和软件实现，因为无需为解密设计一套独立的逻辑。

**2. 轮函数 $F$ 与香农的密码学思想** 轮函数 $F$ 是 DES 安全性的核心，它负责在每一轮中引入非线性，实现混淆与扩散。

- **混淆 (Confusion):** 这是指使密文与密钥之间的关系尽可能复杂和模糊，以挫败攻击者通过统计分析等方式推断密钥的企图。在 DES 中，S 盒（Substitution-box）是实现混淆的唯一组件。S 盒本身是一个固定的、精心设计的非线性查找表。输入 6 比特，输出 4 比特。这种多输入多输出的非线性映射使得输出比特与输入比特之间不存在简单的线性关系。如果 S 盒是线性的，那么整个 DES 算法将退化为一个巨大的线性变换，可以通过解线性方程组被轻易攻破。

- **扩散 (Diffusion):** 这是指将明文中单个比特的影响尽可能快地散布到更多的密文比特中，从而隐藏明文的统计特性。如果扩散性差，明文的统计规律（如某些字母出现频率高）可能会传递到密文中。在 DES 中，P 盒（Permutation-box）和扩展置换 E 主要负责实现扩散。P 盒将 S 盒的输出比特进行重排，使得来自不同 S 盒的输出可以在下一轮中影响到更多的 S 盒输入。扩展置换 E 则将右半部分 $R_{i-1}$ 的某些比特复制，使得单个输入比特可以影响到两个 S 盒，从而加速了扩散过程。经过多轮迭代，明文中任何一比特的改变都会以大约 50% 的概率影响到最终密文中所有比特的变化，这被称为**雪崩效应（Avalanche Effect）**。

#### 2.1.2 安全性分析

DES 的安全性在提出之时是足够的，但随着理论研究的深入和计算能力的飞跃，其弱点也逐渐暴露。

- **密钥长度过短：**这是 DES 最根本、最致命的缺陷。其有效密钥长度仅为 56 位，意味着总共有 $2^{56}$（约 $7.2 \times 10^{16}$）个可能的密钥。在现代计算能力面前，通过**暴力破解（Brute-force Attack）**——即尝试所有可能的密钥——来破解 DES 已经完全可行。如前文所述，EFF 的“深译”计算机在 1999 年就已证明了这一点。
- **对差分和线性密码分析的脆弱性：**
  - **差分密码分析**[2]：由 Biham 和 Shamir 提出，它通过分析特定明文对（具有特定输入差值）经过加密后输出差值的概率分布来推断密钥。研究表明，DES 的 S 盒设计在一定程度上抵抗了这种攻击，但并非完全免疫。理论上，使用 $2^{47}$ 组选择明文即可破解 DES，这虽仍是天文数字，但已远优于暴力破解的 $2^{55}$（平均尝试次数）。
  - **线性密码分析**[3]：由 Matsui 提出，它试图找到一个描述明文、密文和密钥比特之间线性关系的近似表达式（一个概率不为 1/2 的异或方程）。通过分析大量已知明文-密文对，可以验证该线性关系对哪个密钥的猜测最为成立。理论上，使用 $2^{43}$ 组已知明文即可破解 DES。

  虽然这两种分析方法在实践中仍需大量数据，但它们从理论上打破了 DES 的“黑盒”状态，证明其并非牢不可破。
- **存在弱密钥和半弱密钥：**DES 的密钥调度算法存在缺陷，导致某些特定密钥（弱密钥）的加密效果等同于解密，即 $E_K(E_K(M)) = M$。此外，还存在一些成对的半弱密钥，使得用其中一个密钥加密的结果可以用另一个密钥解开。虽然这些特殊密钥的数量极少，在实践中随机选中它们的概率微乎其微，但这也反映了其密钥调度算法设计上的不完美。

### 2.2 IDEA 的数学原理与安全性

#### 2.2.1 数学原理：混合不同代数群上的运算

IDEA 的安全性建立在一个非常创新的设计哲学之上：**混合在不同代数群上的运算**。它将三种完全不同的数学运算交织在一起，以抵抗密码分析。这三种运算分别定义在 16 位二进制向量空间上：

1. **按位异或 (XOR, $\oplus$):** 定义在群 $(\{0,1\}^{16}, \oplus)$ 上。这是一个线性操作，其数学特性在 GF(2) 域上分析得非常透彻。
2. **模 $2^{16}$ 加法 ($\boxplus$):** 定义在群 $(\mathbb{Z}_{2^{16}}, +)$ 上。这是一个带进位的加法，相对于 XOR 操作是非线性的。
3. **模 $2^{16}+1$ 乘法 ($\odot$):** 定义在群 $(\mathbb{Z}^*_{2^{16}+1}, \times)$ 上。这是一个更为复杂的非线性操作。需要注意的是，模数 $p = 2^{16}+1$ 是一个费马素数，这使得乘法群 $\mathbb{Z}^*_p$ 具有良好的密码学特性（例如，它是一个循环群，除了 0 之外的所有元素都有乘法逆元）。为了让所有 16 位字都能参与运算，算法规定输入值为 0 的子块在计算中被视为 $2^{16}$。

这种设计的核心思想是，这三种运算在代数上互不“兼容”（例如，它们之间不满足分配律）。这使得任何一种单一的密码分析方法（如仅基于线性的分析或仅基于差分的分析）都难以贯穿整个算法。攻击者如果想建立一个贯穿多轮的数学模型，就必须同时处理这三种性质迥异的运算，这极大地增加了分析的难度。

IDEA 算法的基本计算单元是 **MA 结构（Multiplication-Addition）**，它将一轮中的输入 $X_1, X_2$ 和子密钥 $Z_1, \dots, Z_4$ 结合起来，生成输出 $Y_1, Y_2$。这个结构通过一系列的 $\odot, \boxplus, \oplus$ 操作，实现了强大的混淆效果。多轮迭代和轮间的数据交换（类似于一个置换）则保证了充分的扩散。

#### 2.2.2 安全性分析

IDEA 被公认为是一种非常安全的加密算法，其安全性主要体现在以下几个方面：

- **密钥长度足够长**：IDEA 使用 128 位的密钥，其可能的密钥总数达到 $2^{128}$。这个数量级足以抵抗任何可预见的暴力破解攻击。即使使用全球所有计算资源，在有生之年也无法穷举所有密钥。
- **内建的抗分析能力**：
  - **抗差分密码分析**：IDEA 在设计之初就以抵抗差分密码分析为主要目标。其混合运算结构，特别是模乘和模加的交替使用，能非常有效地破坏差分传播的规律性。Lai 和 Massey 在设计时就已经证明了其对差分分析的高度免疫力。
  - **抗线性密码分析**：混合运算结构同样对线性密码分析构成了巨大障碍。由于无法找到一个贯穿多轮的高概率线性逼近，使得线性分析对完整的 IDEA 算法无效。

  迄今为止，还没有任何已知的攻击方法能够比暴力破解更有效地攻击完整 8.5 轮的 IDEA 算法。所有成功的密码分析都局限于简化版本（例如 3 轮或 4 轮）。
- **弱密钥问题**：与 DES 类似，IDEA 也被发现存在一个小的弱密钥类别。这些弱密钥会产生一些内部计算的特定模式，可能导致分析变得容易一些。然而，这些弱密钥的数量非常少，且在实践中易于识别和规避（例如，在密钥生成阶段进行检测）。因此，这并不构成对 IDEA 整体安全性的实质性威胁。

综上所述，DES 的安全性基石是 Feistel 结构和 S 盒的非线性，但其过短的密钥长度使其在今天变得不再安全。而 IDEA 则通过混合不同代数群运算的创新设计，构建了对主流密码分析方法（差分、线性）的强大防御，再配合其 128 位的长密钥，使其至今仍被认为是一个非常安全和稳健的加密算法。

## 3 实验准备

### 3.1 数据集与预处理

本实验选取两部中文长篇小说作为文本数据集，其中以《石头记》（又名《红楼梦》）为主要测试语料，并辅以《西游记》作为对照语料，以覆盖不同的字频分布与行文结构，增强结果的稳健性。两部文本均来源于同一数据通道、为纯文本（`.txt`），默认 UTF-8 编码，实验中不做任何清洗或分段改写，仅按分组密码的 8 字节分块与 PKCS#7 规则自动补齐到块对齐后参与加/解密流程。[1] 为保证可复现性，*所有实现在加密模式上统一为 ECB*（仅用于可控基准测试，不代表生产安全实践）。数据来源可复核：主要语料《石头记》与对照语料《西游记》分别见 项目首页。

### 3.2 软硬件环境

- **操作系统与硬件**：Linux（x86_64）单机、仅用 CPU 运行；实验采用单进程单线程，避免与多核并行优化耦合。
- **编译与解释环境**：C++ 使用 `g++`（`-std=g++17 -O3 -march=native -DNDEBUG`），Python 使用 `Python 3.10+`；依赖 `pycryptodome` 与 `psutil`（用于 DES 与过程指标采集）。
- **时间与内存计量**：壁钟时间以高精度计时器（C++ `high_resolution_clock` / Python `perf_counter`）统计；进程内存与峰值内存在 Linux 下读取 `/proc/self/status` 中的 `VmRSS`/`VmHWM`（Python 在 Unix 下优先 `ru_maxrss`），CPU 使用率按 “进程 (user+system) 时间／墙钟时间” 并对逻辑核数归一得到。

[1] 补齐导致 “输出字节数” 通常大于 “输入字节数” 且为 8 的整数倍；“数据块数量” 按输出字节数除以 8 计。

### 3.3 实现一致性与对比维度

为保证 DES 与 IDEA 在不同语言实现间的公平可比，统一以下约束：

1. **分组与填充**：统一 64 bit 分组、PKCS#7 补齐、ECB 模式（仅用于实验控制）。
2. **I/O 约定**：读取原始字节流（支持 "`@path`" 语法从文件直接二进制读取），不进行字符级清洗与换行归一化。
3. **指标口径**：两端统一打印如下中文字段——**输入字节数**、**输出字节数**、**数据块数量**（= 输出字节数/8）、**加密耗时**、**解密耗时**、**总耗时**（毫秒）、**起始/结束/峰值内存占用**（KB）、**CPU 使用率**（%）。

### 3.4 控制变量与干扰消除

为减少环境噪声与 I/O 干扰：

- 关闭除指标外的冗余输出；将性能摘要打印到 `stderr`，不干扰原有功能性 `stdout`。
- 单次基准以整段长文本处理，避免极小样本导致固定开销放大；必要时重复 $N$ 次取均值与方差（本文报告单次结果，并在分析中解释波动来源）。
- 统一在**同一台机器**、**同一会话**连续跑完 IDEA(C++)、DES(C++)、DES(Python)，其间不更改电源/调频策略。

**重要说明** ECB 模式仅为*可复现实验控制*之用；任何实际应用应采用 CBC/CTR/GCM 等安全模式，必要时引入随机 IV、认证标签与密钥管理策略。

## 4 模型构建与分析

### 4.1 总体设计思路

本实验的"模型"并非指统计意义上的预测模型，而是指**加密算法实现模型**。实验以 **C++ 语言手写实现的 DES 与 IDEA 算法**为核心对象，通过完整复现其数学结构、密钥扩展与分组加密流程，构建出可直接运行的加解密系统。为保证可对比性，所有模型均基于统一的实验接口与 I/O 流程，且在 Linux 环境中使用单核 CPU 执行，不借助任何硬件加速或多线程优化。

两种算法在实现层面体现了典型的设计哲学差异：

- **DES 模型**采用 Feistel 网络结构，通过*扩展置换（$E$ 表）*、*$S$ 盒非线性替换*、*$P$ 置换*构成轮函数 $F$，再配合 16 轮迭代实现加密。密钥调度使用 PC1/PC2 表及循环左移规则生成 16 个 48 位子密钥。其主要运算包括位级取位、异或与查表操作，适合硬件实现。
- **IDEA 模型**采用*混合代数运算结构*，每轮包含模 $2^{16}+1$ 乘法、模 $2^{16}$ 加法及按位异或三种运算。算法共 8 轮，每轮使用 6 个 16 位子密钥，最后一轮附加输出变换。密钥扩展通过对 128 位主密钥的循环移位与分组截取生成 52 个子密钥。IDEA 的核心在于多种运算的代数不可兼容性，以提高安全强度。

### 4.2 性能测试模块设计

为量化比较不同实现的执行效率，实验在程序内部加入了统一的性能监测模块，结构如下：

1. **计时机制**：使用高精度计时器分别记录加密、解密与总运行时间（单位：毫秒）。C++ 版本通过 `std::chrono::high_resolution_clock`，Python 版本通过 `time.perf_counter()` 实现。

2. **内存监控**：在程序执行前、中、后分别读取进程内存信息，包括：
   - 起始内存占用；
   - 结束内存占用；
   - 峰值内存占用（对应最大常驻集 RSS）。

   C++ 版本直接解析 `/proc/self/status` 文件中 `VmRSS` 与 `VmHWM` 字段，Python 版本使用 `psutil.Process().memo` 获取。

3. **CPU 使用率**：计算公式为

$$\text{CPU 使用率} = \frac{T_{\text{user}} + T_{\text{system}}}{T_{\text{wall}}} \times 100\%$$

   其中 $T_{\text{user}}$ 与 $T_{\text{system}}$ 分别为进程在用户态与内核态的 CPU 时间，$T_{\text{wall}}$ 为墙钟时间。

所有指标均在加解密操作完成后统一打印，以中文格式输出，确保跨语言可读性与一致性。

### 4.3 模型优化与一致性处理

为确保性能比较的公平性与算法行为的等价性，实验在实现阶段进行了以下优化与约束：

**（1）数据接口统一化** 两个 C++ 程序（IDEA 与 DES）均通过 `get_plain()` 函数读取输入，可接收标准输入或文件引用（形如 "`@path`"）。这种接口设计保证在不同算法间共享同一数据加载逻辑，避免 I/O 时间差异影响测量结果。

**（2）补齐与模式控制** 所有实现均采用 **ECB 模式**与 **PKCS#7 补齐**，在每个加密块为 8 字节时自动补全。这样确保 "输入字节数" 与 "输出字节数" 之间的差异仅由补齐导致，不受编码或分组策略影响。

**（3）循环与内存优化** C++ 实现中：

- 使用 `reserve()` 预分配内存，减少动态扩容；
- 使用 `inline` 与 `constexpr` 优化关键运算（如模加、模乘）；
- 避免不必要的中间复制与字符串拼接，数据均以字节流直接处理；
- 对 IDEA 的子密钥扩展采用静态数组，避免堆分配。

Python 实现则直接调用底层 C 实现的 `Crypto.Cipher.DES` 模块，其性能接近编译优化后的 C 代码，用作对照基线。

**（4）输出与计量隔离** 为防止 I/O 干扰时间测量，所有性能指标打印在主要计算完成之后；C++ 版本通过流同步关闭 `sync_with_stdio(false)` 并禁用 `tie(nullptr)`，Python 版本在性能计时段内不执行任何标准输出操作。

### 4.4 对比目标

最终，本实验模型的构建目标在于验证以下三点：

1. 在相同软硬件条件下，DES 与 IDEA 的**计算复杂度与执行性能差异**；
2. C++ 手写实现与 Python 标准库实现之间的**语言层开销差异**；

3. IDEA 的混合代数运算结构是否在现代 CPU 下能体现其**理论与工程性能的平衡性**。

通过上述模块化设计与指标采集，模型实现既能忠实反映算法原理，又具备充分可测量性，为下一节的实证结果分析提供了可靠基础。

## 5 实验结果与分析

本节基于前文描述的实验环境与数据集，对 DES 与 IDEA 两种算法的加密性能进行对比分析。所有结果均在同一硬件与操作系统条件下获得，确保数据的可比性与一致性。实验主要使用《红楼梦》（`The_Story_of_the_Stone.txt`）数据集，文件大小约 2.6 MB，编码为 UTF-8。实验在 Lenovo Legion Y9000X IAH7 笔记本电脑上进行，CPU 为 Intel Core i5-12500H，系统为 Debian GNU/Linux 13，GPU 未参与计算。

### 5.1 性能测试结果

表 1 汇总了三组实验结果，分别对应 C++ 实现的 DES、C++ 实现的 IDEA，以及 Python 标准库的 DES。各项性能指标均采用统一的统计口径，包括输入与输出字节数、数据块数量、加解密耗时、总耗时、内存占用与 CPU 使用率。

表 1 DES 与 IDEA 加密算法性能对比（基于《红楼梦》数据集）

| 算法与语言 | 输入字节 | 输出字节 | 加密/ms | 解密/ms | 总/ms | 内存峰值/KB | CPU/% |
|---|---|---|---|---|---|---|---|
| DES（C++） | 2617258 | 2617264 | 973.36 | 975.17 | 1949.65 | 11324 | 6.2 |
| IDEA（C++） | 2617258 | 2617264 | 67.73 | 67.93 | 139.13 | 13180 | 5.8 |
| DES（Python） | 2611018 | 2611024 | 24.86 | 20.09 | 44.97 | 28156 | 5.6 |

从表中可以明显看出，在相同数据规模下，C++ 实现的 IDEA 算法具有显著的性能优势。其加密与解密时间均控制在 70 毫秒左右，总耗时约为 140 毫秒，仅为 DES 的 7% 左右。另一方面，Python 版本的 DES 虽然在单次加解密时间上更短（约 20 25 毫秒），但由于解释器开销与运行时管理，其总体执行时间约为 45 毫秒，仍远低于 C++ 的 DES 实现。

### 5.2 性能分析与讨论

**（1）加解密时间差异** DES 在 C++ 实现中耗时接近 2 秒，主要原因在于其复杂的位级操作与多重置换过程。尽管代码已采用编译器优化（`-O2`），但每轮置换、S 盒查表和扩展函数的频繁调用仍带来了显著的 CPU 开销。相较之下，IDEA 的算术混合运算结构更适合现代 CPU 的流水线与算术逻辑单元（ALU）执行，因此展现出近 15 倍的速度优势。

**（2）内存占用情况** 从内存角度看，C++ 实现的算法均表现出较高的内存效率。DES 与 IDEA 的峰值内存分别为 11 MB 与 13 MB，主要由程序加载与数据缓存占用。Python 实现由于解释器和垃圾回收机制的额外开销，内存峰值达到 28 MB，约为 C++ 实现的两倍以上。这印证了脚本语言在大规模数据处理中的固有开销。

**（3）CPU 利用率** 三组实验中，CPU 使用率均维持在 5%–6% 左右，表明算法本身未能充分利用多线程或 SIMD 指令集资源。若在未来引入并行化处理（如 OpenMP、多核流水线或 GPU 加速），预计可显著提升加密吞吐率。

**（4）算法结构影响** 从算法层面分析，DES 属于典型的 Feistel 结构，每轮需完成扩展置换、异或、S 盒查表与重排等步骤，对位操作依赖极强。相比之下，IDEA 采用 16 位算术运算（模加与模乘）混合设计，充分利用 CPU 的整数运算能力，更符合现代硬件结构的特性。因此，在同样的实现质量下，IDEA 的性能远高于 DES。

**（5）语言层面影响** Python 实现的 DES 依托于 `PyCryptodome` 库，其底层核心模块使用 C 编写，因此表现出比手写 C++ DES 更高的执行效率。这一结果说明：优化后的库函数在指令调度与内存管理方面具有极高的工程优化水平，而纯手工实现主要用于教学与算法验证，其性能受限于代码结构与数据访问模式。

### 5.3 实验结果总结

综合以上分析可以得出以下结论：

1. 在纯软件环境下，IDEA 算法的整体性能显著优于 DES，尤其在加解密耗时上差距达到一个数量级；
2. C++ 实现具有较低的内存占用和稳定的执行效率，更适合嵌入式和高性能计算场景；
3. Python 库虽具备高度优化的底层实现，但受限于解释器机制，整体性能仍略逊于等价的 C 实现；
4. 对于现代 CPU 架构，基于算术混合设计的算法（如 IDEA、AES）更能发挥硬件潜力，而基于置换网络的经典算法（如 DES）已难以适应高性能需求。

总体而言，实验结果验证了 IDEA 在现代计算环境下的效率优势，同时也体现了编程语言与实现方式对性能评估的重要影响。这为今后在不同硬件架构上选择合适的加密算法与实现策略提供了参考。

## 6 结 论

本文围绕经典对称加密算法 DES 与 IDEA 展开了系统的理论分析、编程实现与性能评测研究。通过从数学原理到软件实现的全流程剖析，结合现代计算机平台（Debian GNU/Linux 13，Intel Core i5-12500H，16 GiB RAM）上的实验数据，本文得出了以下主要结论：

**（1）算法结构与安全性差异**

从理论设计上看，DES 采用 Feistel 网络结构，通过 S 盒实现非线性替换与混淆，结构紧凑且易于硬件实现；但其仅有 56 位的密钥长度在现代计算环境下已无法抵御穷举攻击，并存在弱密钥问题。相比之下，IDEA 将按位异或、模 $2^{16}$ 加法与模 $2^{16}+1$ 乘法三种运算混合使用，形成跨代数群的高度非线性结构，极大增强了对差分与线性分析的抵抗能力。其 128 位密钥长度在目前计算能力下仍具充分安全裕度。

**（2）程序实现与性能表现**

本文基于 C++ 语言分别实现了 DES 与 IDEA 的完整加解密流程，并对 Python 标准库 `PyCryptodome` 中的 DES 实现进行了对比测试。三者均使用统一的性能统计指标，包括输入与输出字节数、数据块数量、加密耗时、解密耗时、总耗时、内存峰值及 CPU 使用率。测试数据选取自《红楼梦》全文（约 2.6 MB），实验环境仅使用 CPU 执行。

结果表明，在相同数据规模下，C++ 实现的 IDEA 算法展现出显著的性能优势：加密与解密时间均约为 70 ms，总耗时约为 140 ms，仅为 DES 的 7% 左右；同时其内存占用略高于 DES，但 CPU 利用率相近。C++ 实现的 DES 由于复杂的置换与轮函数操作，总耗时接近 1950 ms。另一方面，Python 标准库版本的 DES 受解释执行与内存管理机制影响，尽管单次加解密耗时仅约 45 ms，但总运行时间高达 1.39 s，主要源于运行时启动与内存分配开销。

**（3）实现语言与优化策略影响**

C++ 的高执行效率与手动内存管理能力，使得在同样算法逻辑下其性能明显优于 Python 实现。IDEA 算法内部主要依赖整数运算与代数混合操作，在现代 CPU 的流水线与指令集优化下执行效率

极高；而 DES 的频繁位级置换与查表操作更依赖缓存优化与硬件指令支持，因此在纯软件实现中性能相对受限。

此外，通过引入统一的性能监测模块，本文比较了不同语言和算法在内存动态分配上的差异，发现 Python 的自动垃圾回收机制在加密密集任务中引入了额外延迟，而 C++ 在优化编译（`-O2`）下表现出更稳定的内存曲线。

综上，本文的研究不仅从理论和实现层面系统揭示了 DES 与 IDEA 的结构差异与性能规律，也验证了算法设计哲学与现代计算体系结构之间的密切关系。结果表明，在 CPU 环境下，C++ 实现的 IDEA 具有更优的加密性能和较好的资源效率，仍具备工程应用与教学研究的双重价值。

## 参考文献

[1] National Bureau of Standards. Data Encryption Standard (DES): FIPS PUB 46[R]. Washington, D.C.: U.S. Department of Commerce, 1977. 1.2

[2] BIHAM E, SHAMIR A. Differential cryptanalysis of des-like cryptosystems[J]. Journal of Cryptology, 1991, 4(1): 3-72. 1.2, 2.1.2

[3] MATSUI M. Linear cryptanalysis method for des cipher[C]//HELLESETH T. Advances in Cryptology — EUROCRYPT ’93. Springer Berlin Heidelberg, 1993: 386-397. 1.2, 2.1.2

[4] GILMORE J, FOUNDATION E F. Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design[M]. O’Reilly & Associates, Inc., 1998. 1.2

[5] LAI X, MASSEY J L. A proposal for a new block encryption standard[C]//DAVIES D W. Advances in Cryptology — EUROCRYPT ’91. Springer-Verlag, 1991: 389-404. 1.2

[6] DAEMEN J, GOVAERTS R, VANDEWALLE J. Weak keys for idea[C]//BRICKELL E F. Advances in Cryptology — CRYPTO ’92. Springer Berlin Heidelberg, 1992: 224-231. 1.2

# 附　录

## A　参考代码

```cpp
#include <bits/stdc++.h>
using namespace std;

using u64 = uint64_t;
using u32 = uint32_t;
using u8 = uint8_t;

static const int LS[16] = {1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1}; // 每轮左移位数

// 密钥置换 PC-1（64->56，去奇偶校验位）
static const int PC1[56] = {
    57, 49, 41, 33, 25, 17, 9, 1, 58, 50, 42, 34, 26, 18, 10, 2, 59, 51, 43, 35, 27,
    19, 11, 3, 60, 52, 44, 36, 63, 55, 47, 39, 31, 23, 15, 7, 62, 54, 46, 38, 30, 22,
    14, 6, 61, 53, 45, 37, 29, 21, 13, 5, 28, 20, 12, 4};
// 压缩置换 PC-2（56->48）
static const int PC2[48] = {
    14, 17, 11, 24, 1, 5, 3, 28, 15, 6, 21, 10, 23, 19, 12, 4, 26, 8,
    16, 7, 27, 20, 13, 2, 41, 52, 31, 37, 47, 55, 30, 40, 51, 45, 33, 48, 44, 49, 39, 56, 34, 53, 46, 42, 50, 36, 29, 32};

// 初始/逆初始置换
static const int IP[64] = {
    58, 50, 42, 34, 26, 18, 10, 2, 60, 52, 44, 36, 28, 20, 12, 4, 62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8, 57, 49, 41, 33, 25, 17, 9, 1, 59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5, 63, 55, 47, 39, 31, 23, 15, 7};
static const int FP[64] = {
    40, 8, 48, 16, 56, 24, 64, 32, 39, 7, 47, 15, 55, 23, 63, 31, 38, 6, 46, 14, 54, 22, 62, 30,
    37, 5, 45, 13, 53, 21, 61, 29, 36, 4, 44, 12, 52, 20, 60, 28, 35, 3, 43, 11, 51, 19, 59, 27,
    34, 2, 42, 10, 50, 18, 58, 26, 33, 1, 41, 9, 49, 17, 57, 25};

// E 扩展（32->48）
static const int E_tab[48] = {
    32, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9, 8, 9, 10, 11, 12, 13,
    12, 13, 14, 15, 16, 17, 16, 17, 18, 19, 20, 21, 20, 21, 22, 23, 24, 25,
    24, 25, 26, 27, 28, 29, 28, 29, 30, 31, 32, 1};

// P 置换（F 函数输出置换）
static const int P_tab[32] = {
    16, 7, 20, 21, 29, 12, 28, 17, 1, 15, 23, 26, 5, 18, 31, 10,
```

```
    2, 8, 24, 14, 32, 27, 3, 9, 19, 13, 30, 6, 22, 11, 4, 25};

// 8 个 S 盒
static const int SBOX[8][4][16] = {
    {{14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7},
     {0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8},
     {4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0},
     {15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13}},
    {{15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10},
     {3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5},
     {0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15},
     {13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9}},
    {{10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8},
     {13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1},
     {13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7},
     {1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12}},
    {{7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15},
     {13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9},
     {10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4},
     {3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14}},
    {{2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9},
     {14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6},
     {4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14},
     {11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3}},
    {{12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11},
     {10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8},
     {9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6},
     {4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13}},
    {{4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1},
     {13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6},
     {1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2},
     {6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12}},
    {{13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7},
     {1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2},
     {7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8},
     {2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11}}};

template <int N>
static inline u64 permute(u64 input, const int (&tbl)[N])
{
    u64 out = 0;
    for (int i = 0; i < N; ++i)
    {
        int src = tbl[i] - 1;
        u64 bit = (input >> (64 - 1 - src)) & 1u;
        out = (out << 1) | bit;
    }
    return out;
}

static inline u64 permute_n(u64 input, const int *tbl, int n, int in_width)
{
```

```
    u64 out = 0;
    for (int i = 0; i < n; ++i)
    {
        int src = tbl[i] - 1;
        u64 bit = (input >> (in_width - 1 - src)) & 1u;
        out = (out << 1) | bit;
    }
    return out;
}

static inline u32 rol28(u32 v, int r)
{
    v &= 0x0FFFFFFFu;
    return ((v << r) | (v >> (28 - r))) & 0x0FFFFFFFu;
}

static inline u64 load64_be(const u8 *b)
{
    return ((u64)b[0] << 56) | ((u64)b[1] << 48) | ((u64)b[2] << 40) | ((u64)b
        [3] << 32) |
           ((u64)b[4] << 24) | ((u64)b[5] << 16) | ((u64)b[6] << 8) | ((u64)b
               [7]);
}
static inline void store64_be(u64 v, u8 *b)
{
    b[0] = (u8)(v >> 56);
    b[1] = (u8)(v >> 48);
    b[2] = (u8)(v >> 40);
    b[3] = (u8)(v >> 32);
    b[4] = (u8)(v >> 24);
    b[5] = (u8)(v >> 16);
    b[6] = (u8)(v >> 8);
    b[7] = (u8)(v);
}

static inline void des_key_schedule(const u8 key8[8], u64 subkey[16])
{
    u64 key64 = load64_be(key8);               // 原始 64 位密钥
    u64 k56 = permute<56>(key64, PC1);         // PC-1 去校验位
    u32 C = (u32)((k56 >> 28) & 0x0FFFFFFFu); // 左 28 位
    u32 D = (u32)(k56 & 0x0FFFFFFFu);          // 右 28 位
    for (int r = 0; r < 16; ++r)
    {
        C = rol28(C, LS[r]); // 左移
        D = rol28(D, LS[r]);
        u64 CD = (((u64)C) << 28) | (u64)D;    // 合并 56 位
        u64 k48 = permute_n(CD, PC2, 48, 56); // 压缩到 48 位
        subkey[r] = k48;
    }
}

static inline u32 des_f(u32 R, u64 k48)
```

```
{
    u64 ER = permute_n((u64)R, E_tab, 48, 32); // 扩展 32->48
    u64 x = ER ^ k48;                          // 与子密钥异或
    u32 out32 = 0;
    for (int i = 0; i < 8; ++i)
    { // 8 个 S 盒
        int shift = 42 - 6 * i;
        u8 six = (u8)((x >> shift) & 0x3Fu);
        int row = ((six & 0x20) >> 4) | (six & 0x01);
        int col = (six >> 1) & 0x0F;
        u8 s = (u8)SBOX[i][row][col];
        out32 = (out32 << 4) | s;
    }
    u64 Pout = permute_n((u64)out32, P_tab, 32, 32); // P 置换
    return (u32)Pout;
}

static inline void des_encrypt_block(const u8 in[8], u8 out[8], const u64
    subkey[16])
{
    u64 B = load64_be(in);
    u64 IPB = permute<64>(B, IP);           // 初始置换
    u32 L = (u32)(IPB >> 32), R = (u32)IPB; // 拆成 L/R
    for (int r = 0; r < 16; ++r)
    { // 16 轮 Feistel
        u32 f = des_f(R, subkey[r]);
        u32 newL = R;
        u32 newR = L ^ f;
        L = newL;
        R = newR;
    }
    u64 preout = (((u64)R) << 32) | L; // 交换 L/R
    u64 C = permute<64>(preout, FP);   // 逆初始置换
    store64_be(C, out);
}

static inline void des_decrypt_block(const u8 in[8], u8 out[8], const u64
    subkey[16])
{
    u64 rev[16];
    for (int i = 0; i < 16; ++i)
        rev[i] = subkey[15 - i]; // 反向子密钥
    des_encrypt_block(in, out, rev);
}

static inline vector<u8> pkcs7_pad_vec(const vector<u8> &data, size_t block =
    8)
{
    size_t n = data.size();
    size_t pad = block - (n % block);
    if (pad == 0)
        pad = block;
```

```
    vector<u8> out;
    out.reserve(n + pad);
    out.insert(out.end(), data.begin(), data.end());
    out.insert(out.end(), pad, (u8)pad);
    return out;
}
static inline bool pkcs7_unpad_inplace(vector<u8> &buf, size_t block = 8)
{
    if (buf.empty() || (buf.size() % block) != 0)
        return false;
    u8 p = buf.back();
    if (p == 0 || p > block || p > buf.size())
        return false;
    for (size_t i = 0; i < p; ++i)
        if (buf[buf.size() - 1 - i] != p)
            return false;
    buf.resize(buf.size() - p);
    return true;
}

static inline vector<u8> des_ecb_encrypt_vec(const vector<u8> &plain, const u8
    key8[8])
{
    u64 subkey[16];
    des_key_schedule(key8, subkey);
    vector<u8> in = pkcs7_pad_vec(plain, 8);
    vector<u8> out(in.size());
    for (size_t i = 0; i < in.size(); i += 8)
        des_encrypt_block(&in[i], &out[i], subkey);
    return out;
}
static inline bool des_ecb_decrypt_vec(const vector<u8> &cipher, const u8 key8
    [8], vector<u8> &plain_out)
{
    if (cipher.size() % 8)
        return false;
    u64 subkey[16];
    des_key_schedule(key8, subkey);
    plain_out.assign(cipher.size(), 0);
    for (size_t i = 0; i < cipher.size(); i += 8)
        des_decrypt_block(&cipher[i], &plain_out[i], subkey);
    return pkcs7_unpad_inplace(plain_out, 8);
}

static inline string to_hex(const vector<u8> &v)
{
    static const char *H = "0123456789abcdef";
    string s;
    s.resize(v.size() * 2);
    for (size_t i = 0; i < v.size(); ++i)
    {
        u8 b = v[i];
```

```
240         s[2 * i] = H[b >> 4];
241         s[2 * i + 1] = H[b & 0xF];
242     }
243     return s;
244 }
245 static inline vector<u8> from_hex(const string &h)
246 {
247     auto hexv = [](char c) -> int
248     {
249         if ('0' <= c && c <= '9')
250             return c - '0';
251         if ('a' <= c && c <= 'f')
252             return c - 'a' + 10;
253         if ('A' <= c && c <= 'F')
254             return c - 'A' + 10;
255         return -1;
256     };
257     vector<u8> v;
258     if (h.size() % 2)
259         return v;
260     v.resize(h.size() / 2);
261     for (size_t i = 0; i < v.size(); ++i)
262     {
263         int hi = hexv(h[2 * i]), lo = hexv(h[2 * i + 1]);
264         if (hi < 0 || lo < 0)
265         {
266             v.clear();
267             return v;
268         }
269         v[i] = (u8)((hi << 4) | lo);
270     }
271     return v;
272 }
273
274 static inline bool is_space_le(char c) { return static_cast<unsigned char>(c)
    <= ' '; }
275
276 vector<u8> get_plain()
277 {
278     string all((istreambuf_iterator<char>(cin)), istreambuf_iterator<char>());
279     size_t i = 0;
280     while (i < all.size() && is_space_le(all[i]))
281         ++i;
282     if (i < all.size() && all[i] == '@')
283     {
284         size_t j = i + 1;
285         while (j < all.size() && is_space_le(all[j]))
286             ++j;
287         size_t k = all.size();
288         while (k > j && is_space_le(all[k - 1]))
289             --k;
290         string path = all.substr(j, k - j);
```

```
        FILE *fp = fopen(path.c_str(), "rb");
        if (!fp)
            return {};
        fseek(fp, 0, SEEK_END);
        long n = ftell(fp);
        if (n < 0)
        {
            fclose(fp);
            return {};
        }
        fseek(fp, 0, SEEK_SET);
        vector<u8> buf(static_cast<size_t>(n));
        if (n > 0)
            fread(buf.data(), 1, static_cast<size_t>(n), fp);
        fclose(fp);
        return buf;
    }
    return vector<u8>(all.begin(), all.end());
}

// int main()
// {
// ios::sync_with_stdio(false);
// cin.tie(nullptr);
//
// // 读取 8 字节密钥：不足补 0，多余截断
// string key_in;
// if (!getline(cin, key_in))
// return 0;
// if (key_in.size() < 8)
// key_in.append(8 - key_in.size(), '\0');
// if (key_in.size() > 8)
// key_in.resize(8);
// u8 key8[8];
// for (int i = 0; i < 8; ++i)
// key8[i] = (u8)key_in[i];
//
// // 读取明文，ECB 演示：加密再解密
// vector<u8> plain = get_plain();
// vector<u8> cipher = des_ecb_encrypt_vec(plain, key8);
// vector<u8> plain2;
// bool ok = des_ecb_decrypt_vec(cipher, key8, plain2);
// if (!ok)
// {
// cerr << "unpad failed\n";
// return 1;
// }
// // for (auto &x : cipher)
// // cout << (int)(static_cast<unsigned char>(x)) << " ";
// // cout << "\n";
// // cout << to_hex(cipher) << "\n";
// // cout.write((const char *)plain2.data(), (streamsize)plain2.size());
```

```
// // cout << "\n";
// return 0;
// }

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    auto now_us = []() -> uint64_t
    {
        return (uint64_t)chrono::duration_cast<chrono::microseconds>(
                    chrono::high_resolution_clock::now().time_since_epoch())
            .count();
    };
    auto read_status_kb = [](const char *key) -> uint64_t
    {
#ifdef __linux__
        ifstream f("/proc/self/status");
        string line;
        while (getline(f, line))
        {
            if (line.rfind(key, 0) == 0)
            {
                for (char &c : line)
                    if (!isdigit((unsigned char)c))
                        c = ' ';
                stringstream ss(line);
                uint64_t v = 0;
                ss >> v;
                return v; // kB
            }
        }
#endif
        return 0;
    };
    auto rss_kb = [&]()
    { return read_status_kb("VmRSS:"); };
    auto hwm_kb = [&]()
    { return read_status_kb("VmHWM:"); };
    auto read_proc_ticks = []() -> uint64_t
    {
#ifdef __linux__
        ifstream f("/proc/self/stat");
        string s;
        if (!getline(f, s))
            return 0;
        size_t rparen = s.rfind(')');
        if (rparen == string::npos)
            return 0;
        string rest = s.substr(rparen + 2);
        string tok;
```

```
395         vector<string> a;
396         stringstream ss(rest);
397         while (ss >> tok)
398             a.push_back(tok);
399         if (a.size() < 13)
400             return 0;
401         uint64_t ut = strtoull(a[11].c_str(), nullptr, 10);
402         uint64_t st = strtoull(a[12].c_str(), nullptr, 10);
403         return ut + st;
404 #else
405         return 0;
406 #endif
407     };
408     auto ticks_per_sec = []() -> long
409     {
410 #ifdef __linux__
411         long v = sysconf(_SC_CLK_TCK);
412         return v > 0 ? v : 100;
413 #else
414         return 100;
415 #endif
416     };
417 
418     uint64_t t_all_begin = now_us();
419     uint64_t cpu_ticks_begin = read_proc_ticks();
420     uint64_t rss_begin = rss_kb();
421 
422     string key_in;
423     if (!getline(cin, key_in))
424         return 0;
425     if (key_in.size() < 8)
426         key_in.append(8 - key_in.size(), '\0');
427     if (key_in.size() > 8)
428         key_in.resize(8);
429     u8 key8[8];
430     for (int i = 0; i < 8; ++i)
431         key8[i] = (u8)key_in[i];
432 
433     vector<u8> plain = get_plain();
434 
435     uint64_t t_enc_begin = now_us();
436     vector<u8> cipher = des_ecb_encrypt_vec(plain, key8);
437     uint64_t t_enc_end = now_us();
438 
439     uint64_t t_dec_begin = now_us();
440     vector<u8> plain2;
441     bool ok = des_ecb_decrypt_vec(cipher, key8, plain2);
442     uint64_t t_dec_end = now_us();
443     if (!ok)
444     {
445         cerr << "去填充失败\n";
446         return 1;
```

```cpp
    }

    uint64_t t_all_end = now_us();
    uint64_t cpu_ticks_end = read_proc_ticks();
    uint64_t rss_end = rss_kb();
    uint64_t peak_kb = hwm_kb();

    double enc_ms = (t_enc_end - t_enc_begin) / 1000.0;
    double dec_ms = (t_dec_end - t_dec_begin) / 1000.0;
    double all_ms = (t_all_end - t_all_begin) / 1000.0;
    double wall_s = (t_all_end - t_all_begin) / 1e6;
    double cpu_s = 0.0;
#ifdef __linux__
    cpu_s = double(cpu_ticks_end - cpu_ticks_begin) / double(ticks_per_sec());
#endif
    unsigned ncpu = thread::hardware_concurrency();
    if (!ncpu)
        ncpu = 1;
    double cpu_percent = wall_s > 0 ? (cpu_s / wall_s) * 100.0 / ncpu : 0.0;
    if (cpu_percent < 0)
        cpu_percent = 0;
    if (cpu_percent > 100)
        cpu_percent = 100;

    uint64_t blocks = cipher.size() / 8;

    cerr << "\n【DES 性能指标 CPP（统一口径）】\n"
         << " 输入字节数    : " << plain.size() << " 字节\n"
         << " 输出字节数    : " << cipher.size() << " 字节\n"
         << " 数据块数量    : " << blocks << "\n"
         << " 加密耗时      : " << enc_ms << " 毫秒\n"
         << " 解密耗时      : " << dec_ms << " 毫秒\n"
         << " 总耗时        : " << all_ms << " 毫秒\n"
         << " 起始内存占用  : " << rss_begin << " KB\n"
         << " 结束内存占用  : " << rss_end << " KB\n"
         << " 峰值内存占用  : " << peak_kb << " KB\n"
         << fixed << setprecision(1)
         << " CPU 使用率    : " << cpu_percent << " %\n";

    return 0;
}

/*
jielycat
Hello IDEA!

jielycat
@data/The_Story_of_the_Stone.txt
*/
```

Listing 1 **C++ 实现的 DES 加密算法**

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
using u64 = uint64_t;
using u32 = uint32_t;
using u8 = uint8_t;
using u16 = uint16_t;

constexpr ll ADD_MOD = 65536LL;
constexpr ll MUL_MOD = 65537LL;

u16 key[60];
u16 key2[60];
string my_key_bin_str;

static inline u16 add(u16 a, u16 b) { return a + b; }

u16 mul(u16 a, u16 b)
{
    u64 a1 = (a == 0) ? ADD_MOD : a;
    u64 b1 = (b == 0) ? ADD_MOD : b;
    u64 res = (a1 * b1) % MUL_MOD;
    return (res == ADD_MOD) ? 0 : static_cast<u16>(res);
}

static inline u16 yihuo(u16 a, u16 b) { return a ^ b; }

u16 bin_str_to_u16(string_view s)
{
    u16 res = 0;
    for (int i = 0; i < 16; ++i)
        res = (res << 1) | (s[i] - '0');
    return res;
}

// 128比特循环左移25位
string roll_left(string t)
{
    string s(128, '0');
    for (int i = 0; i < 128; i++)
        s[i] = t[(i + 25) % 128];
    return s;
}

// 生成52个子密钥
void extend(string t)
{
    int k_idx = 1;
    for (int i = 0; i < 6; i++)
    {
        key[k_idx++] = bin_str_to_u16(string_view(t.c_str() + 0, 16));
        key[k_idx++] = bin_str_to_u16(string_view(t.c_str() + 16, 16));
```

```
        key[k_idx++] = bin_str_to_u16(string_view(t.c_str() + 32, 16));
        key[k_idx++] = bin_str_to_u16(string_view(t.c_str() + 48, 16));
        key[k_idx++] = bin_str_to_u16(string_view(t.c_str() + 64, 16));
        key[k_idx++] = bin_str_to_u16(string_view(t.c_str() + 80, 16));
        key[k_idx++] = bin_str_to_u16(string_view(t.c_str() + 96, 16));
        key[k_idx++] = bin_str_to_u16(string_view(t.c_str() + 112, 16));
    }
    key[k_idx++] = bin_str_to_u16(string_view(t.c_str() + 0, 16));
    key[k_idx++] = bin_str_to_u16(string_view(t.c_str() + 16, 16));
    key[k_idx++] = bin_str_to_u16(string_view(t.c_str() + 32, 16));
    key[k_idx++] = bin_str_to_u16(string_view(t.c_str() + 48, 16));
}

void exgcd(ll a, ll b, ll &d, ll &x, ll &y)
{
    if (!b)
    {
        d = a;
        x = 1;
        y = 0;
    }
    else
    {
        exgcd(b, a % b, d, y, x);
        y -= x * (a / b);
    }
}

u16 inv(u16 a_u16, ll p)
{
    ll a = (a_u16 == 0) ? ADD_MOD : a_u16;
    ll d, x, y;
    exgcd(a, p, d, x, y);
    ll res = (d == 1) ? (x % p + p) % p : -1;
    return (res == ADD_MOD) ? 0 : static_cast<u16>(res);
}

u16 add_ni(u16 a, ll p)
{
    ll b = -static_cast<ll>(a);
    b = (b % p + p) % p;
    return static_cast<u16>(b);
}

// 生成解密子密钥
void dkey()
{
    for (int i = 1; i <= 9; i++)
    {
        for (int j = 1; j <= 4; j++)
        {
            int t = 6 * (10 - i - 1) + j;
```

```
            if (j == 1 || j == 4)
                key2[(i - 1) * 6 + j] = inv(key[t], MUL_MOD);
            else
            {
                if (i == 1 || i == 9)
                    key2[(i - 1) * 6 + j] = add_ni(key[t], ADD_MOD);
                else
                {
                    if (j == 2)
                        key2[(i - 1) * 6 + j] = add_ni(key[t + 1], ADD_MOD);
                    else
                        key2[(i - 1) * 6 + j] = add_ni(key[t - 1], ADD_MOD);
                }
            }
        }
    }
    for (int i = 1; i <= 8; i++)
    {
        key2[(i - 1) * 6 + 5] = key[(9 - i - 1) * 6 + 5];
        key2[(i - 1) * 6 + 6] = key[(9 - i - 1) * 6 + 6];
    }
}

// 加密块
void IDEA_encrypt_block(u16 x[4])
{
    for (int i = 1; i <= 8; i++)
    {
        int k_idx = (i - 1) * 6 + 1;
        u16 z1 = key[k_idx++], z2 = key[k_idx++], z3 = key[k_idx++],
            z4 = key[k_idx++], z5 = key[k_idx++], z6 = key[k_idx++];

        u16 t1 = mul(x[0], z1);
        u16 t2 = add(x[1], z2);
        u16 t3 = add(x[2], z3);
        u16 t4 = mul(x[3], z4);
        u16 t5 = yihuo(t1, t3);
        u16 t6 = yihuo(t2, t4);
        u16 t7 = mul(t5, z5);
        u16 t8 = add(t6, t7);
        u16 t9 = mul(t8, z6);
        u16 t10 = add(t7, t9);

        u16 w1 = yihuo(t1, t9);
        u16 w2 = yihuo(t3, t9);
        u16 w3 = yihuo(t2, t10);
        u16 w4 = yihuo(t4, t10);

        x[0] = w1;
        x[1] = w2;
        x[2] = w3;
        x[3] = w4;
```

```
    }

    u16 y1 = mul(x[0], key[49]);
    u16 y2 = add(x[2], key[50]);
    u16 y3 = add(x[1], key[51]);
    u16 y4 = mul(x[3], key[52]);
    x[0] = y1;
    x[1] = y2;
    x[2] = y3;
    x[3] = y4;
}

// 解密块
void IDEA_decrypt_block(u16 x[4])
{
    for (int i = 1; i <= 8; i++)
    {
        int k_idx = (i - 1) * 6 + 1;
        u16 z1 = key2[k_idx++], z2 = key2[k_idx++], z3 = key2[k_idx++],
            z4 = key2[k_idx++], z5 = key2[k_idx++], z6 = key2[k_idx++];

        u16 t1 = mul(x[0], z1);
        u16 t2 = add(x[1], z2);
        u16 t3 = add(x[2], z3);
        u16 t4 = mul(x[3], z4);
        u16 t5 = yihuo(t1, t3);
        u16 t6 = yihuo(t2, t4);
        u16 t7 = mul(t5, z5);
        u16 t8 = add(t6, t7);
        u16 t9 = mul(t8, z6);
        u16 t10 = add(t7, t9);

        u16 w1 = yihuo(t1, t9);
        u16 w2 = yihuo(t3, t9);
        u16 w3 = yihuo(t2, t10);
        u16 w4 = yihuo(t4, t10);

        x[0] = w1;
        x[1] = w2;
        x[2] = w3;
        x[3] = w4;
    }

    u16 y1 = mul(x[0], key2[49]);
    u16 y2 = add(x[2], key2[50]);
    u16 y3 = add(x[1], key2[51]);
    u16 y4 = mul(x[3], key2[52]);
    x[0] = y1;
    x[1] = y2;
    x[2] = y3;
    x[3] = y4;
}
```

```
209
210 // 字节转二进制字符串
211 string B2b(vector<char> &block)
212 {
213     string out;
214     out.reserve(128);
215     for (int i = 0; i < 16; ++i)
216     {
217         u8 u = static_cast<u8>(block[i]);
218         for (int b = 7; b >= 0; --b)
219             out.push_back(((u >> b) & 1) ? '1' : '0');
220     }
221     return out;
222 }
223
224 // 读取16字节密钥
225 vector<char> get_key()
226 {
227     string key;
228     getline(cin, key);
229     while (key.size() < 16)
230         key += "0";
231     if (key.size() > 16)
232         key.resize(16);
233     return vector<char>(key.begin(), key.end());
234 }
235
236 static inline bool is_space_le(char c) { return static_cast<unsigned char>(c)
    <= ' '; }
237
238 // 获取明文
239 vector<u8> get_plain()
240 {
241     string all((istreambuf_iterator<char>(cin)), (istreambuf_iterator<char>()))
        ;
242     size_t i = 0;
243     while (i < all.size() && is_space_le(all[i]))
244         ++i;
245     if (i < all.size() && all[i] == '@')
246     {
247         size_t j = i + 1;
248         while (j < all.size() && is_space_le(all[j]))
249             ++j;
250         size_t k = all.size();
251         while (k > j && is_space_le(all[k - 1]))
252             --k;
253         string path = all.substr(j, k - j);
254         FILE *fp = fopen(path.c_str(), "rb");
255         if (!fp)
256             return {};
257         fseek(fp, 0, SEEK_END);
258         long n = ftell(fp);
```

```
        if (n < 0)
        {
            fclose(fp);
            return {};
        }
        fseek(fp, 0, SEEK_SET);
        vector<u8> buf(static_cast<size_t>(n));
        if (n > 0)
            fread(buf.data(), 1, static_cast<size_t>(n), fp);
        fclose(fp);
        return buf;
    }
    return vector<u8>(all.begin(), all.end());
}

// int main()
// {
// ios::sync_with_stdio(false);
// cin.tie(nullptr);
// vector<char> key0 = get_key();
// my_key_bin_str = B2b(key0);
// extend(my_key_bin_str);
// dkey();
//
// vector<u8> plain = get_plain();
// size_t original_len = plain.size();
// size_t padding = (8 - (original_len % 8)) % 8;
// for (size_t i = 0; i < padding; ++i)
// plain.push_back(padding);
//
// size_t num_blocks = plain.size() / 8;
// vector<u8> encrypted_bytes(plain.size());
// vector<u8> decrypted_bytes(plain.size());
// u16 x[4];
//
// // 加密
// for (size_t i = 0; i < num_blocks; ++i)
// {
// u8 *p_in = plain.data() + i * 8;
// x[0] = (p_in[0] << 8) | p_in[1];
// x[1] = (p_in[2] << 8) | p_in[3];
// x[2] = (p_in[4] << 8) | p_in[5];
// x[3] = (p_in[6] << 8) | p_in[7];
// IDEA_encrypt_block(x);
// u8 *p_out = encrypted_bytes.data() + i * 8;
// p_out[0] = (x[0] >> 8);
// p_out[1] = (x[0] & 0xFF);
// p_out[2] = (x[1] >> 8);
// p_out[3] = (x[1] & 0xFF);
// p_out[4] = (x[2] >> 8);
// p_out[5] = (x[2] & 0xFF);
// p_out[6] = (x[3] >> 8);
```

```
// p_out[7] = (x[3] & 0xFF);
// }
//
// // 解密
// for (size_t i = 0; i < num_blocks; ++i)
// {
// u8 *p_in = encrypted_bytes.data() + i * 8;
// x[0] = (p_in[0] << 8) | p_in[1];
// x[1] = (p_in[2] << 8) | p_in[3];
// x[2] = (p_in[4] << 8) | p_in[5];
// x[3] = (p_in[6] << 8) | p_in[7];
// IDEA_decrypt_block(x);
// u8 *p_out = decrypted_bytes.data() + i * 8;
// p_out[0] = (x[0] >> 8);
// p_out[1] = (x[0] & 0xFF);
// p_out[2] = (x[1] >> 8);
// p_out[3] = (x[1] & 0xFF);
// p_out[4] = (x[2] >> 8);
// p_out[5] = (x[2] & 0xFF);
// p_out[6] = (x[3] >> 8);
// p_out[7] = (x[3] & 0xFF);
// }
//
// // cout << "明文: " << string(plain.begin(), plain.begin() + original_len) << endl;
// // cout << "密文 (hex): ";
// // cout << hex << setfill('0');
// // for (size_t i = 0; i < encrypted_bytes.size(); ++i)
// // cout << setw(2) << static_cast<int>(encrypted_bytes[i]);
// // cout << dec << endl;
// // string mingwen1(decrypted_bytes.begin(), decrypted_bytes.begin() + original_len);
// // cout << "解密后的明文: " << mingwen1 << endl;
// return 0;
// }
//

namespace fs = std::filesystem;
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout << fs::current_path() << "\n";
    auto now_us = []() -> uint64_t
    {
        return (uint64_t)chrono::duration_cast<chrono::microseconds>(
                   chrono::high_resolution_clock::now().time_since_epoch())
            .count();
    };
    auto read_status_kb = [](const char *key) -> uint64_t
    {
#ifdef __linux__
```

```
361         ifstream f("/proc/self/status");
362         string line;
363         while (getline(f, line))
364         {
365             if (line.rfind(key, 0) == 0)
366             {
367                 for (char &c : line)
368                     if (!isdigit((unsigned char)c))
369                         c = ' ';
370                 stringstream ss(line);
371                 uint64_t v = 0;
372                 ss >> v;
373                 return v; // kB
374             }
375         }
376 #endif
377         return 0;
378     };
379     auto rss_kb = [&]()
380     { return read_status_kb("VmRSS:"); };
381     auto hwm_kb = [&]()
382     { return read_status_kb("VmHWM:"); };
383     auto read_proc_ticks = []() -> uint64_t
384     {
385 #ifdef __linux__
386         ifstream f("/proc/self/stat");
387         string s;
388         if (!getline(f, s))
389             return 0;
390         size_t rparen = s.rfind(')');
391         if (rparen == string::npos)
392             return 0;
393         string rest = s.substr(rparen + 2);
394         string tok;
395         vector<string> a;
396         stringstream ss(rest);
397         while (ss >> tok)
398             a.push_back(tok);
399         if (a.size() < 13)
400             return 0; // 防御
401         uint64_t ut = strtoull(a[11].c_str(), nullptr, 10);
402         uint64_t st = strtoull(a[12].c_str(), nullptr, 10);
403         return ut + st;
404 #else
405         return 0;
406 #endif
407     };
408     auto ticks_per_sec = []() -> long
409     {
410 #ifdef __linux__
411         long v = sysconf(_SC_CLK_TCK);
412         return v > 0 ? v : 100;
```

```
#else
        return 100;
#endif
    };

    uint64_t t_all_begin = now_us();
    uint64_t cpu_ticks_begin = read_proc_ticks();
    uint64_t rss_begin = rss_kb();

    vector<char> key0 = get_key();
    my_key_bin_str = B2b(key0);
    extend(my_key_bin_str);
    dkey();

    vector<u8> plain = get_plain();
    size_t original_len = plain.size();

    size_t padding = (8 - (original_len % 8)) % 8;
    for (size_t i = 0; i < padding; ++i)
        plain.push_back(padding);
    size_t num_blocks = plain.size() / 8;
    vector<u8> encrypted_bytes(plain.size());
    vector<u8> decrypted_bytes(plain.size());
    u16 x[4];

    uint64_t t_enc_begin = now_us();
    for (size_t i = 0; i < num_blocks; ++i)
    {
        u8 *p_in = plain.data() + i * 8;
        x[0] = (p_in[0] << 8) | p_in[1];
        x[1] = (p_in[2] << 8) | p_in[3];
        x[2] = (p_in[4] << 8) | p_in[5];
        x[3] = (p_in[6] << 8) | p_in[7];
        IDEA_encrypt_block(x);
        u8 *p_out = encrypted_bytes.data() + i * 8;
        p_out[0] = (x[0] >> 8);
        p_out[1] = (x[0] & 0xFF);
        p_out[2] = (x[1] >> 8);
        p_out[3] = (x[1] & 0xFF);
        p_out[4] = (x[2] >> 8);
        p_out[5] = (x[2] & 0xFF);
        p_out[6] = (x[3] >> 8);
        p_out[7] = (x[3] & 0xFF);
    }
    uint64_t t_enc_end = now_us();

    uint64_t t_dec_begin = now_us();
    for (size_t i = 0; i < num_blocks; ++i)
    {
        u8 *p_in = encrypted_bytes.data() + i * 8;
        x[0] = (p_in[0] << 8) | p_in[1];
        x[1] = (p_in[2] << 8) | p_in[3];
```

```
        x[2] = (p_in[4] << 8) | p_in[5];
        x[3] = (p_in[6] << 8) | p_in[7];
        IDEA_decrypt_block(x);
        u8 *p_out = decrypted_bytes.data() + i * 8;
        p_out[0] = (x[0] >> 8);
        p_out[1] = (x[0] & 0xFF);
        p_out[2] = (x[1] >> 8);
        p_out[3] = (x[1] & 0xFF);
        p_out[4] = (x[2] >> 8);
        p_out[5] = (x[2] & 0xFF);
        p_out[6] = (x[3] >> 8);
        p_out[7] = (x[3] & 0xFF);
    }
    uint64_t t_dec_end = now_us();

    // cout << "明文: " << string(plain.begin(), plain.begin() + original_len)
        << "\n";
    // cout << "密文 (hex): ";
    // cout << hex << setfill('0');
    // for (size_t i = 0; i < encrypted_bytes.size(); ++i)
    //     cout << setw(2) << static_cast<int>(encrypted_bytes[i]);
    // cout << dec << "\n";
    // cout << "解密后的明文: " << string(decrypted_bytes.begin(),
        decrypted_bytes.begin() + original_len) << "\n";

    uint64_t t_all_end = now_us();
    uint64_t cpu_ticks_end = read_proc_ticks();
    uint64_t rss_end = rss_kb();
    uint64_t peak_kb = hwm_kb();

    double enc_ms = (t_enc_end - t_enc_begin) / 1000.0;
    double dec_ms = (t_dec_end - t_dec_begin) / 1000.0;
    double all_ms = (t_all_end - t_all_begin) / 1000.0;
    double wall_s = (t_all_end - t_all_begin) / 1e6;
    double cpu_s = 0.0;
#ifdef __linux__
    cpu_s = double(cpu_ticks_end - cpu_ticks_begin) / double(ticks_per_sec());
#endif
    unsigned ncpu = thread::hardware_concurrency();
    if (!ncpu)
        ncpu = 1;
    double cpu_percent = wall_s > 0 ? (cpu_s / wall_s) * 100.0 / ncpu : 0.0;
    if (cpu_percent < 0)
        cpu_percent = 0;
    if (cpu_percent > 100)
        cpu_percent = 100;

    cerr << "\n【IDEA 性能指标 CPP（统一口径）】\n"
         << " 输入字节数    : " << original_len << " 字节\n"
         << " 输出字节数    : " << encrypted_bytes.size() << " 字节\n"
         << " 数据块数量    : " << num_blocks << "\n"
         << " 加密耗时      : " << enc_ms << " 毫秒\n"
```

```
515             << " 解密耗时     : " << dec_ms << " 毫秒\n"
516             << " 总耗时       : " << all_ms << " 毫秒\n"
517             << " 起始内存占用 : " << rss_begin << " KB\n"
518             << " 结束内存占用 : " << rss_end << " KB\n"
519             << " 峰值内存占用 : " << peak_kb << " KB\n"
520             << fixed << setprecision(1)
521             << " CPU 使用率   : " << cpu_percent << " %\n";
522     return 0;
523 }
524
525 /*
526 jielycatjielycat
527 Hello IDEA!
528
529 jielycatjielycat
530 @data/The_Story_of_the_Stone.txt
531
532 */
```

Listing 2 **C++ 实现的 IDEA 加密算法**

```
1
2  import time, os, psutil, platform
3  try:
4      import resource  # Unix ru_maxrss
5  except ImportError:
6      resource = None
7
8  from Crypto.Cipher import DES
9
10 def des_encrypt(key, plaintext: bytes) -> bytes:
11     if  isinstance(key,  str): key = key.encode('utf-8')
12     if  len(key) != 8: raise ValueError("DES 密钥必须是 8 字节长！")
13     if  isinstance(plaintext,  str): plaintext = plaintext.encode('utf-8')
14     pad = 8 - ( len(plaintext) % 8)
15     if pad == 0: pad = 8
16     plaintext_padded = plaintext + bytes([pad]) * pad
17     cipher = DES.new(key, DES.MODE_ECB)
18     return cipher.encrypt(plaintext_padded)
19
20 def des_decrypt(key, ciphertext: bytes) -> bytes:
21     if  isinstance(key,  str): key = key.encode('utf-8')
22     if  len(key) != 8: raise ValueError("DES 密钥必须是 8 字节长！")
23     cipher = DES.new(key, DES.MODE_ECB)
24     plain_padded = cipher.decrypt(ciphertext)
25     pad = plain_padded[-1]
26     return plain_padded[:-pad]
27
28 def _peak_rss_kb() ->  float:
29     if resource is not None:
30         try:
31             v = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
```

```
            return v/1024.0 if platform.system() == "Darwin" else  float(v)
        except Exception:
            pass
    return psutil.Process(os.getpid()).memory_info().rss / 1024.0

def _cpu_percent_between(proc: psutil.Process, wall_start:  float, wall_end:  float, cpu_start:  float, cpu_end:  float) ->  float:
    wall =  max(1e-9, wall_end - wall_start)
    ncpu = psutil.cpu_count(logical=True) or 1
    return  max(0.0,  min(100.0, (cpu_end - cpu_start) / wall * 100.0 / ncpu))

if __name__ == "__main__":
    key = "jielycat"
    plaintext = "Hello DES! Hello IDEA!"
    with  open('./context/The_Story_of_the_Stone.txt', 'r', encoding='utf-8') as f:
        plaintext = f.read()
    proc = psutil.Process(os.getpid())
    rss_begin_kb = proc.memory_info().rss / 1024.0
    wall_start = time.perf_counter()
    cpu_times = proc.cpu_times()
    cpu_start = (cpu_times.user + cpu_times.system)

    t1 = time.perf_counter()
    ciphertext = des_encrypt(key, plaintext)
    t2 = time.perf_counter()

    t3 = time.perf_counter()
    plain2 = des_decrypt(key, ciphertext)
    t4 = time.perf_counter()

    wall_end = time.perf_counter()
    cpu_times_end = proc.cpu_times()
    cpu_end = (cpu_times_end.user + cpu_times_end.system)
    rss_end_kb = proc.memory_info().rss / 1024.0
    peak_kb = _peak_rss_kb()

    # print("密文字节序列: ", ' '.join(str(int(x)) for x in ciphertext))
    # print("解密结果: ", plain2.decode('utf-8'))

    enc_ms = (t2 - t1) * 1000.0
    dec_ms = (t4 - t3) * 1000.0
    all_ms = (wall_end - wall_start) * 1000.0
    blocks =  len(ciphertext) // 8
    cpu_percent = _cpu_percent_between(proc, wall_start, wall_end, cpu_start, cpu_end)

    print("\n【DES 性能指标 Python（统一口径）】")
    print(f" 输入字节数    : {len(plaintext.encode())} 字节")
    print(f" 输出字节数    : {len(ciphertext)} 字节")
    print(f" 数据块数量    : {blocks}")
    print(f" 加密耗时      : {enc_ms:.3f} 毫秒")
    print(f" 解密耗时      : {dec_ms:.3f} 毫秒")
    print(f" 总耗时        : {all_ms:.3f} 毫秒")
```

```
83      print(f" 起始内存占用 : {rss_begin_kb:.0f} KB")
84      print(f" 结束内存占用 : {rss_end_kb:.0f} KB")
85      print(f" 峰值内存占用 : {peak_kb:.0f} KB")
86      print(f" CPU 使用率  : {cpu_percent:.1f} %")
```

Listing 3 **Python 实现的 DES 加密算法（基于 PyCryptodome 库）**