DES 算法介绍与实现

20337025 崔璨明
计算机科学与技术（人工智能与大数据）

2023 年 6 月 11 日

目录

1 引言

本报告介绍了数据加密标准（Data Encryption Standard，DES）算法的实现。DES 是一种对称密钥加密算法，被广泛应用于信息安全领域。报告首先提供了 DES 算法的背景和基本原理，然后详细描述了 DES 算法的实现过程，包括密钥生成、初始置换、轮函数、轮密钥生成和最终置换。最后，通过编写代码实现了 DES 算法，并给出了测试样例和性能评估结果。

1.1 DES 算法的背景和历史

DES 由 IBM 的 Horst Feistel 在 1970 年代设计并于 1977 年被美国国家标准与技术研究院（NIST）选定为数据加密标准 [SB88]。DES 算法在过去几十年中一直被广泛应用于保护敏感信息的安全性，直到被更强大的加密算法取代。

DES 的设计目标是提供高度可靠的数据保护，尤其适用于政府和军事领域。在设计 DES 时，考虑到了硬件和软件实现的效率，以及算法的可行性和安全性。DES 使用对称密钥密码体制，即加密和解密使用相同的密钥。

然而，随着计算机处理能力的增强，DES 逐渐变得不够安全。在 1990 年代末，出现了多个针对 DES 的攻击方法，如差分攻击和线性攻击。为了提高数据的安全性，NIST 于 2001 年发布了新的加密标准——高级加密标准（AES），取代了 DES。

尽管 DES 已经被 AES 所取代，但 DES 的设计原理和结构对于理解对称加密算法的基本概念仍然具有重要意义。它奠定了许多后续加密算法的基础，为密码学领域的研究和发展做出了重要贡献。

1.2 DES 算法在信息安全中的重要性

DES 算法在信息安全领域具有重要意义。作为早期的数据加密标准，DES 的设计和应用推动了密码学的发展，为后续加密算法奠定了基础。它能够保护敏感信息的机密性，通过使用密钥对数据进行加密，只有持有正确密钥的人才能解密和访问数据，有效防止未经授权的访问和数据泄露。DES 广泛应用于政府、军事、金融和商业等领域，被许多计算机系统、通信设备和网络协议所采用。虽然现代密码学已经取代了 DES，但理解 DES 算法对于评估和维护现有系统的安全性、以及理解和设计更强大的加密算法仍具有重要意义。

2 DES 算法的原理

在 DES 加密算法中，明文和密文被分为 64 位的块。密钥的长度为 64 位，但密钥的每个第八位都是奇偶校验位，因此实际上密钥的长度为 56 位。DES 加密算法是最常见的分组加密算法，其核心思想是通过数据位的置换和移位过程，在 16 轮迭代加密和最后的逆置换之后得到最终的密文。DES 解密只需按照加密的逆过程进行即可。由于 DES 加密过程的算法是公开的，所以密钥 K 的保密性显得尤为重要，只有发送方和接收方使用相同的密钥进行加密解密才能获取明文数据。DES 加密算法大致包括以下四个步骤：初始置换、生成子密钥、迭代过程和逆置换。整个过程的流程图如图 1 所示。

2.1 初始置换

初始置换是将原始明文经过 IP 置换表处理，即将明文块的位进行换位，其置换表是固定的。置换过程如图 2 所示，图中的置换表第一位 58 表示该位置存放明文中的第 58 位字符。明文中第 1

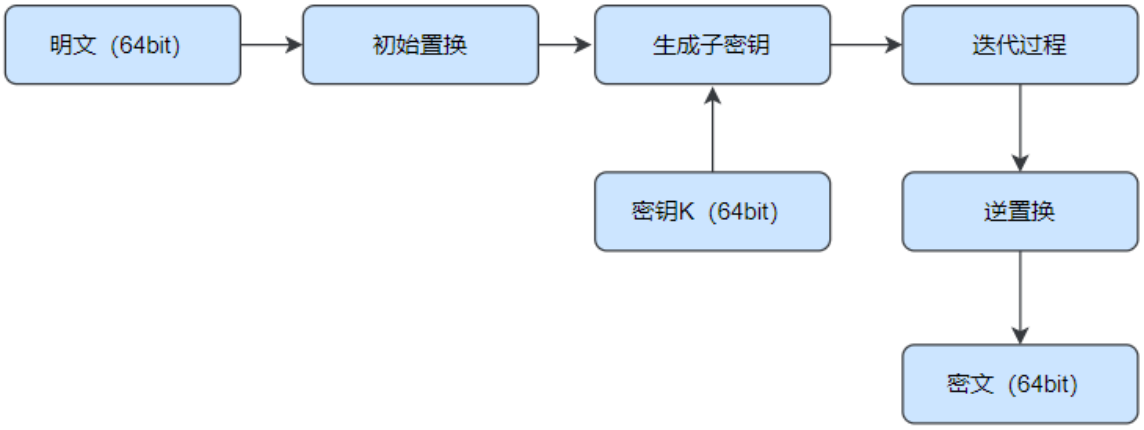


图 1: DES 算法流程图

位字符已经变换到 40 位的位置。

注意初始置换是固定公开的函数，因此没有密码的意义。作用不大，它们的作用在于打乱原来输入 x 的 ASCII 码字划分的关系。经过初始置换之后，64 位明文将分成两组 L 和 R，各 32 位。即经过初始置换，64 位的输入得到了两个 32 位的输出。

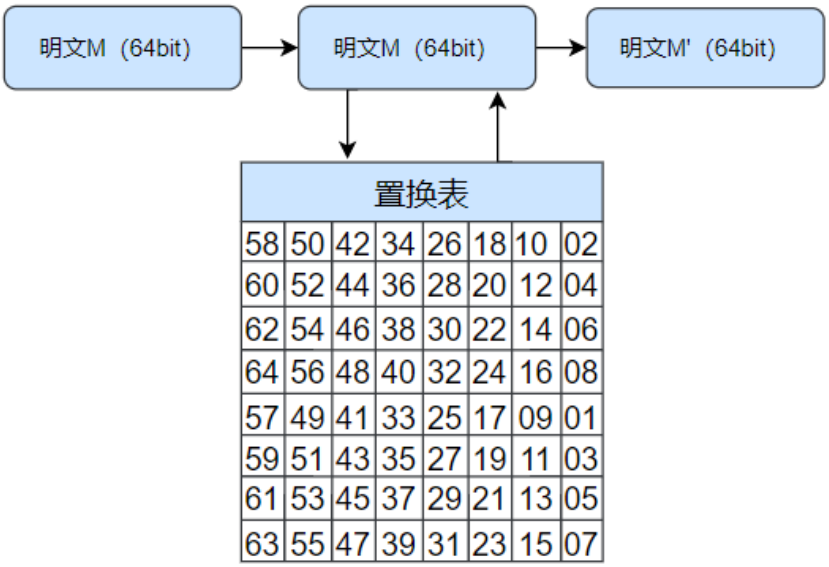


图 2: 初始置换示意图

一个置换的例子如下：

1. 输入 64 位明文数据 M（64 位）：

$$M = 0110001101101111011011010111000001110101011101000110010101110010$$

2. 选择密钥 K（64 位）：

$$K = 0001001100110100010101110111100110011011101111001101111111110001$$

3. 对 M 进行 IP 置换得到 M'：

$$M' = 1111111110111000011101100101011100000000111111110000011010000011$$

4. 将 M' 的前 32 位作为 L_0：

$$L_0 = 11111111101110000111011001010111$$

将 M' 的后 32 位作为 R_0：

$$R_0 = 00000000111111110000011010000011$$

2.2 生成子密钥

DES 加密算法需要执行 16 次迭代，每次迭代的数据长度为 48 位。因此，为了进行加密，需要生成 16 个 48 位的子密钥。DES 算法生成子密钥的过程如下：

1. 初始密钥置换（PC-1）：将 64 位的密钥进行初始置换，通过固定的置换表将密钥的各个位重新排列和选择，生成 56 位的置换后密钥（C_0 和 D_0）。
2. 迭代生成子密钥（PC-2）：根据 DES 算法的规则，在每一轮迭代中，根据前一轮生成的 C_n 和 D_n，通过循环左移和置换操作生成子密钥 K_n。具体步骤如下：
   - 对 Cn 和 Dn 进行循环左移：根据每一轮的左移位数（通过查询循环左移表可得），将 C_n 和 D_n 分别向左循环移动相应的位数，生成 C_{n+1} 和 D_{n+1}。
   - 合并 C_{n+1} 和 D_{n+1}：将 C_{n+1} 和 D_{n+1} 合并成一个 56 位的密钥。
   - 子密钥置换（PC-2）：对合并后的 56 位密钥进行置换操作，通过固定的置换表，选择出 48 位的子密钥 Kn。
3. 重复迭代生成子密钥：根据 DES 算法的规则，重复进行 16 轮的迭代生成子密钥的过程，生成 16 个 48 位的子密钥 K_1-K_{16}。

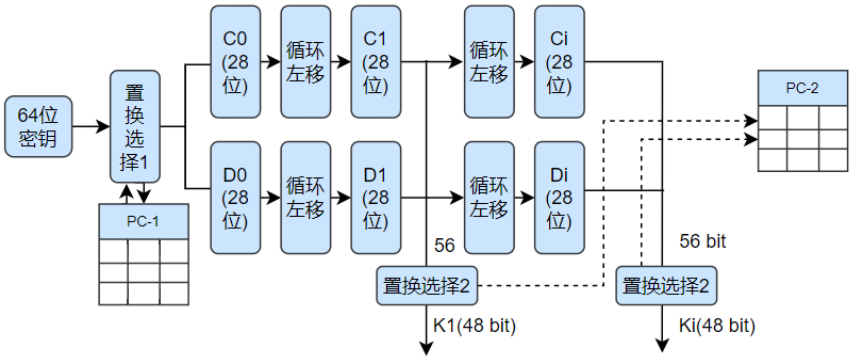


图 3: 子密钥生成流程图

过程图如图 3 所示，最终生成的 16 个子密钥 K_1-K_{16} 将用于 DES 算法的每一轮加密过程中，与明文数据进行异或运算。

2.3 迭代过程

DES 算法总共有 16 轮的迭代加密过程，对于每一轮 i（1 i 16）的迭代加密过程，按照以下步骤进行：

1. 扩展置换（Expansion Permutation, E）：将 Rn-1（前一轮的右半部分）进行扩展，将 32 位的 Rn-1 扩展为 48 位，通过固定的扩展置换表 E，得到扩展后的 Rn-1。
2. 获取子密钥：根据当前轮次 i，从总共的 16 个子密钥中选取对应的子密钥 Kn。
3. 异或运算：将扩展后的 Rn-1 与子密钥 Kn 进行按位异或运算，得到结果为 48 位的数据。
4. S 盒替代（S-Box Substitution）：将异或运算的结果分为 8 个 6 位的块，每个块作为 S 盒的输入。对每个 6 位块，根据对应的 S 盒进行替代操作。S 盒将 6 位输入映射为 4 位输出。这样，共进行 8 次 S 盒替代操作，得到 8 个 4 位的结果，合并得到 32 位的输出。
5. P 置换（Permutation）：对 S 盒替代的输出结果进行固定的置换操作，通过置换表 P，得到 32 位的置换结果。
6. 左右半部分交换：将 Rn-1 的结果与 Ln-1（前一轮的左半部分）进行交换，得到 Rn 和 Ln。

详细叙述一下 S 盒替代和 P 置换。

2.3.1 S 盒的计算

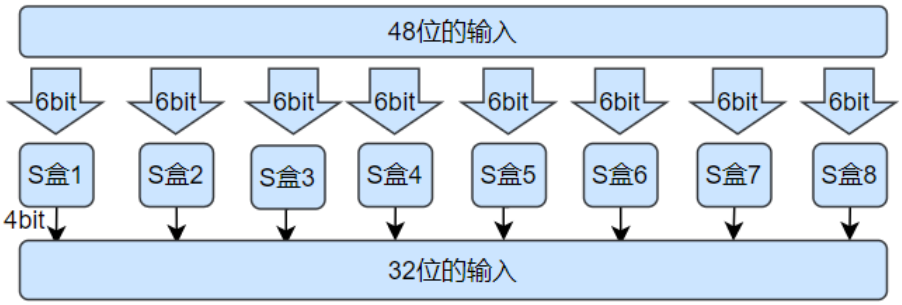


图 4: S 盒替代

S 盒的计算过程如图 4 所示。例如：若 S-盒 1 的输入为 110111，第一位与最后一位构成 11，十进制值为 3，则对应第 3 行，中间 4 位为 1011 对应的十进制值为 11，则对应第 11 列。查找 S-盒 1 表的值为 14，则 S-盒 1 的输出为 1110。8 个 S 盒将输入的 48 位数据输出为 32 位数据。

按照 S-盒的计算过程，设经过异或运算的结果为：

$$100110110001010100010001011111001010010001110100(48bit)$$

通过 S- 盒替换得到的 S 盒输出为

$$10001011110001000110001011101010(32bit)$$

2.3.2 P-盒置换

将 S-盒替代的输出结果作为 P-盒置换的输入。P-盒置换表为：

$$\begin{bmatrix} 32 & 01 & 02 & 03 & 04 & 05 \\ 04 & 05 & 06 & 07 & 08 & 09 \\ 08 & 09 & 10 & 11 & 12 & 13 \\ 12 & 13 & 14 & 15 & 16 & 17 \\ 16 & 17 & 18 & 19 & 20 & 21 \\ 20 & 21 & 22 & 23 & 24 & 25 \\ 24 & 25 & 26 & 27 & 28 & 29 \\ 28 & 29 & 30 & 31 & 32 & 01 \end{bmatrix}$$

将 S 盒输出：

$$10001011110001000110001011101010(32bit)$$

经过 P 盒置换，P 盒置换输出：

$$01001000101111110101010110000001$$

当进行完 16 轮迭代后，得到最终的 R16 和 L16，它们组合在一起形成 64 位的密文块，整体流程如图 5 所示。

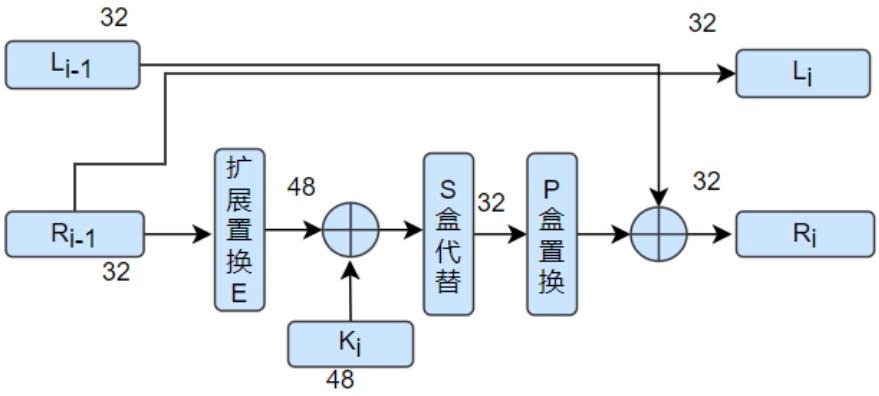


图 5: 迭代加密过程

2.4 逆置换

逆置换过程是初始置换的逆过程，将加密后的 64 位数据重新按照逆置换表进行排列，得到最终的密文。

逆置换的步骤如下：

1. 将加密后的 64 位数据分为左右两个 32 位的数据块，记为 L16 和 R16。
2. 按照固定的逆置换表 IP^{-1}，对 L16 和 R16 进行重排。逆置换表是初始置换表的逆序，即将初始置换表中的位置映射到逆置换表中的位置。
3. 将重排后的 L16 和 R16 合并，得到 64 位的密文数据。

最终，得到的 64 位数据就是 DES 算法的密文。

3 DES 算法的代码实现

3.1 选择编程语言和开发环境

选择了使用 python 进行实现，开发环境为 Windows，并测试了程序的正确性。

3.2 各种表的定义

IP_table：IP 置换表

_IP_table：逆 IP 置换表

S:S 盒

Si(i=1,2...8)：S 盒中的第 i 个盒子

P_table：P 盒

switch1_table：压缩置换表 1，不考虑每字节的第 8 位，将 64 位密钥减至 56 位。然后进行一次密钥置换。

switch2_table：压缩置换表 2，用于将循环左移和右移后的 56bit 密钥压缩为 48bit

extend_table：用于对数据进行扩展置换，将 32bit 数据扩展为 48bit

3.3 实现密钥生成、初始置换、轮函数、轮密钥生成和最终置换的代码

密钥生成函数：

```
# 生成每一轮的key
def build_keys(input_k):
    results = []
    ascii_key = char_to_unicode_ascii(input_k, len(input_k))
    init_key = byte_to_bit(ascii_key, len(ascii_key))
    KEY_0 = [0 for i in range(56)]
    KEY_1 = [0 for i in range(48)]
    # 进行密码压缩置换1，将64位压缩为56
    for i in range(56):
        KEY_0[i] = init_key[switch1_table[i] - 1]
    # 16轮的密码生成
    for i in range(16):
        # 左移的次数
        if (i == 0 or i == 1 or i == 8 or i == 15):      #压缩表左移1
            STEP = 1
        else:
            STEP = 2    #压缩表左移2
        #分两部分，每28bit一部分，循环左移
        for j in range(STEP):
            for k in range(8):
                tmp = KEY_0[k * 7]
                for m in range(7 * k, 7 * k + 6):
                    KEY_0[m] = KEY_0[m + 1]
                KEY_0[k * 7 + 6] = tmp
            tmp = KEY_0[0]
            for k in range(27):
                KEY_0[k] = KEY_0[k + 1]
            KEY_0[27] = tmp
            tmp = KEY_0[28]
            for k in range(28, 55):
                KEY_0[k] = KEY_0[k + 1]
            KEY_0[55] = tmp
```

```
        #56位压缩为48位
        for k in range(48):
            KEY_1[k] = KEY_0[switch2_table[k] - 1]
        results.extend(KEY_1)
    return results
```

DES 算法的执行函数，包含了初始置换、轮函数、轮密钥生成和最终置换的代码：

```
def DES_Aglorithm(text, key, option):
    results = build_keys(key)
    final_text_bit = [0 for i in range(64)]
    final_text_unicode = [0 for i in range(4)]
    if option == 0:  # 为加密
        tmpText = [0 for i in range(64)]  # 将L部分和R部分合并成64位的结果
        extend_res = [0 for i in range(48)]  # R部分的扩展结果
        unicode_text = char_to_unicode_ascii(text, len(text))
        bit_text = unicode_to_bit(unicode_text, len(unicode_text))
        init_switch = [0 for i in range(64)]
        # 初始IP置换
        for i in range(64):
            init_switch[i] = bit_text[IP_table[i] - 1]
        # 64位明文分为左右两部分
        L = [init_switch[i] for i in range(32)]
        R = [init_switch[i] for i in range(32, 64)]
        # 开始进行16轮运算
        for i in range(16):
            tmpR = R  # 用于临时盛放R
            # 将32位扩展为48位
            for j in range(48):
                extend_res[j] = R[extend_table[j] - 1]
            key_i = [results[j] for j in range(i * 48, i * 48 + 48)]
            # 与key值进行异或
            res_of_xor = [0 for j in range(48)]
            for j in range(48):
                if key_i[j] != extend_res[j]:
                    res_of_xor[j] = 1
            Sbox_result = [0 for k in range(32)]
            # S盒替换
            for k in range(8):
                row = res_of_xor[k * 6] * 2 + res_of_xor[k * 6 + 5]
                col = res_of_xor[k * 6 + 1] * 8 + res_of_xor[k * 6 + 2] * 4 + res_of_xor[k * 6 + 3]
                                                                    * 2 + res_of_xor[
                    k * 6 + 4]
                tmp = S[k][row * 16 + col]
                for m in range(4):
                    Sbox_result[k * 4 + m] = (tmp >> m) & 1
            Pbox_result = [0 for k in range(32)]
            # P盒置换
            for k in range(32):
                Pbox_result[k] = Sbox_result[P_table[k] - 1]
            # 与L部分的数据进行异或
            res_of_xor_l = [0 for k in range(32)]
            for k in range(32):
                if L[k] != Pbox_result[k]:
                    res_of_xor_l[k] = 1
            # 将临时保存的R部分值，即tmpR复制给L
            L = tmpR
            R = res_of_xor_l
        # 交换
        L, R = R, L
```

```
        # 合并
        tmpText = L
        tmpText.extend(R)
        # IP逆置换
        for k in range(64):
            final_text_bit[k] = tmpText[_IP_table[k] - 1]
        final_text_unicode = bit_to_byte(final_text_bit, len(final_text_bit))
        res_of_char = unicode_to_char(final_text_unicode, len(final_text_unicode))
        return res_of_char
    else:
        tmpText = [0 for i in range(64)]
        extend_res = [0 for i in range(48)]
        unicode_text = char_to_unicode_ascii(text, len(text))
        bit_text = byte_to_bit(unicode_text, len(unicode_text))
        init_switch = [0 for i in range(64)]
        # 初始IP置换
        for i in range(64):
            init_switch[i] = bit_text[IP_table[i] - 1]
        # 64位明文分为两部分
        L = [init_switch[i] for i in range(32)]
        R = [init_switch[i] for i in range(32, 64)]
        # 16轮的循环
        for i in range(15, -1, -1):
            tmpR = R
            # 将32位扩展为48位
            for j in range(48):
                extend_res[j] = R[extend_table[j] - 1]
            key_i = [results[j] for j in range(i * 48, i * 48 + 48)]
            # 与key值异或
            res_of_xor = [0 for j in range(48)]
            for j in range(48):
                if key_i[j] != extend_res[j]:
                    res_of_xor[j] = 1
            Sbox_result = [0 for k in range(32)]
            # S盒替换
            for k in range(8):
                row = res_of_xor[k * 6] * 2 + res_of_xor[k * 6 + 5]
                col = res_of_xor[k * 6 + 1] * 8 + res_of_xor[k * 6 + 2] * 4 + res_of_xor[k * 6 + 3]
                                                                    * 2 + res_of_xor[
                    k * 6 + 4]
                tmp = S[k][row * 16 + col]
                for m in range(4):
                    Sbox_result[k * 4 + m] = (tmp >> m) & 1
            Pbox_result = [0 for k in range(32)]
            # P盒置换
            for k in range(32):
                Pbox_result[k] = Sbox_result[P_table[k] - 1]
            # 与L部分的数据进行异或
            res_of_xor_l = [0 for k in range(32)]
            for k in range(32):
                if L[k] != Pbox_result[k]:
                    res_of_xor_l[k] = 1
            # 将临时保存的R部分值，即tmpR复制给L
            L = tmpR
            R = res_of_xor_l
        # 交换
        L, R = R, L
        # 合并
        tmpText = L
```

```
        tmpText.extend(R)
        # IP逆置换
        for k in range(64):
            final_text_bit[k] = tmpText[_IP_table[k] - 1]
        final_text_unicode = bit_to_unicode(final_text_bit, len(final_text_bit))
        res_of_char = unicode_to_char(final_text_unicode, len(final_text_unicode))
        return res_of_char
```

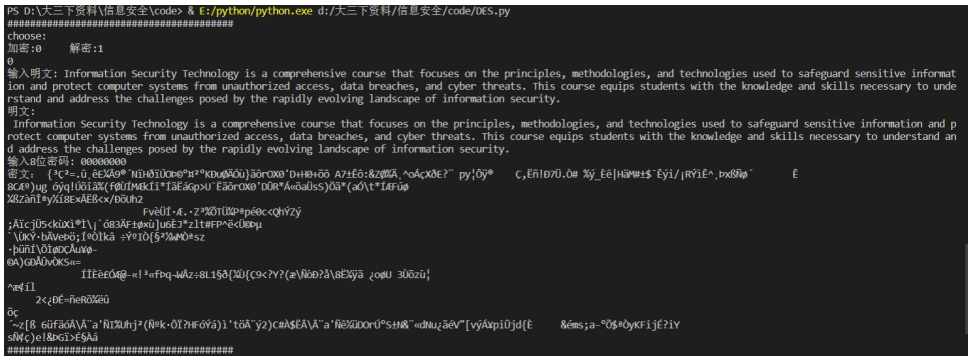
其他代码就不在此赘述，具体内容见 code 文件夹下的 DES.py 文件。

3.4 测试样例和结果分析

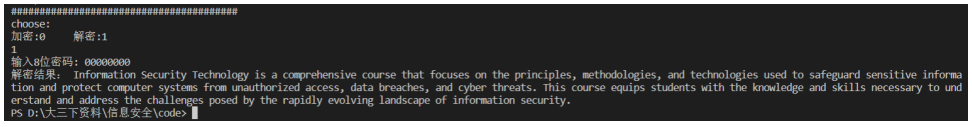
运行指令：python DES.py，生成的密文会保存在同目录下的 cipertext.txt 文件。

用以下明文进行测试：Information Security Technology is a comprehensive course that focuses on the principles, methodologies, and technologies used to safeguard sensitive information and protect computer systems from unauthorized access, data breaches, and cyber threats. This course equips students with the knowledge and skills necessary to understand and address the challenges posed by the rapidly evolving landscape of information security.

密码为 00000000，生成的密文：



解密结果：



可见结果正确，成功实现了 DES 加密解密算法。

4 结论

4.1 优缺点分析

DES 算法的优点：

1. 安全性较高：DES 算法使用 56 位密钥对数据进行加密和解密，对于当时的计算能力而言，提供了相对较高的安全性。
2. 快速加密和解密：DES 算法的加密和解密速度相对较快，适用于对大量数据进行加密和解密的场景。

3. 广泛应用：由于 DES 算法的普及和广泛应用，相关的硬件和软件实现已经得到了广泛支持和优化。

DES 算法缺点：

1. 密钥长度较短：DES 算法的密钥长度只有 56 位，这在当前计算能力的背景下被认为不够安全。通过穷举攻击（brute-force attack）可以在较短的时间内找到正确的密钥。
2. 算法已过时：DES 算法的标准已经公开发布了几十年，并且已经被一些新的、更安全的加密算法所取代。对于更高级的安全需求，像 AES（Advanced Encryption Standard）这样的算法已经成为更常见的选择。

4.2 应用领域

1. 金融行业：DES 算法曾在金融领域广泛应用，用于保护电子支付、交易和敏感的金融数据。
2. 数据库保护：DES 算法可以用于对数据库中的敏感数据进行加密，防止未经授权的访问和泄露。
3. 加密芯片：DES 算法曾被用于硬件加密芯片，以提供设备级别的数据保护和安全。

需要注意的是，由于 DES 算法的密钥长度较短，它已经不再被视为安全的加密算法。更现代和安全的替代方案，如 AES，已被广泛采用，以满足当前和未来的安全需求。

参考文献

[SB88] Miles E Smid and Dennis K Branstad. Data encryption standard: past and future. *Proceedings of the IEEE*, 76(5):550–559, 1988.