上海大学
Shanghai University

《密码学》实验报告

网络空间安全专业　计算机工程与科学学院
范舒舰 23121677

AES加密解密程序及CBC/CTR模式实现

2025 年 11 月 27 日

一、实验目的与任务

1. 实验目的

- 理解AES算法的不同工作模式。

2. 实验任务

1. 实现两个基于AES的加密/解密系统，一个在 CBC 模式下使用 AES，另一个在CTR模式下使用 AES。
2. 测试用例包含了 AES 密钥和密文（两者都是十六进制），需要恢复出明文并在实验报告中展示结果。

二、实验环境

1. 操作系统：Windows 11；
2. 编程语言：Python 3.11（仅使用标准库）；

三、实验内容

1. 算法原理与流程

（1）AES-128 基本结构

AES-128 是一种 128 位分组、128 位密钥的对称分组密码。明文和密文都被组织为 4×4 的字节状态矩阵（State），按列填充。AES-128 共进行 10 轮轮变换，每一轮使用不同的轮密钥（由原始密钥通过密钥扩展算法生成）。

整体轮结构如下：

- 轮 0：轮密钥加（仅将明文与初始轮密钥异或）；
- 轮 1–9：字节代换 $\rightarrow$ 行移位 $\rightarrow$ 列混淆 $\rightarrow$ 轮密钥加；
- 轮 10：字节代换 $\rightarrow$ 行移位 $\rightarrow$ 轮密钥加（最后一轮不再执行列混淆）。

（2）核心变换简述

- **字节代换**：对状态矩阵中每个字节，利用固定 16×16 S 盒查表替换，实现非线性混淆。字节 `0xXY` 的高 4 位 X 决定行，低 4 位 Y 决定列。
- **行移位**：对状态矩阵每一行进行循环左移：

  - 第 0 行不移动；
  - 第 1 行左移 1 字节；
  - 第 2 行左移 2 字节；
  - 第 3 行左移 3 字节。

  该操作在字节层面打乱了列之间的关系。

- **列混淆**：对状态矩阵的每一列视作 $(a_0, a_1, a_2, a_3)^T$，在有限域 GF(2^8) 上与固定矩阵相乘：

  其中乘法在模多项式 $x^8 + x^4 + x^3 + x + 1$ 的有限域上进行，实现字节线性扩散。

- **轮密钥加**：将状态矩阵与当前轮密钥逐字节异或，是唯一依赖秘密参数的步骤。

（3）密钥扩展（KeyExpansion）

AES-128 需要从 16 字节原始密钥生成 44 个 32 位字（word），每 4 个 word 构成一轮的轮密钥。

- 原始密钥被分为 W_0, W_1, W_2, W_3 四个 word；
- 之后用递推公式生成 $W\_4, \ldots, W\_43$，每当索引是 4 的倍数时对上一 word 进行 RotWord、SubWord 并异或 Rcon 常量；
- SubWord 使用 S 盒逐字节替换，Rcon 为 10 个在 GF(2^8) 上反复乘 2 得到的轮常数。

通过该过程，轮密钥与原始密钥之间形成复杂非线性关系，从而增强安全性。

（4）CBC 模式与 PKCS#7 填充

CBC（Cipher Block Chaining）模式按分组对明文进行加密，并使用前一分组的密文作为“链”：

$$C\_0 = \text{AES\_enc}(P\_0 \oplus IV), \quad C\_i = \text{AES\_enc}(P\_i \oplus C\_i - 1),\ i \geq 1$$

解密时：

$$P\_0 = \text{AES\_dec}(C\_0) \oplus IV, \quad P\_i = \text{AES\_dec}(C\_i) \oplus C\_i - 1$$

由于 AES 是 16 字节分组，本实验使用 PKCS#7 填充：若最后一块缺少 k 字节，则补 k 个值为 k 的字节；若刚好整除，需补 16 个 `0x10`。

（5）CTR 模式与计数器设计

CTR（Counter）模式将分组密码转换为流密码：对每个分组构造唯一的 128 位计数器块，使用 AES 加密得到密钥流块，与明文异或：

$$K\_i = \text{AES\_enc}(\text{nonce}||\text{counter}\_i), \quad C\_i = P\_i \oplus K\_i$$

解密时同样：

$$P\_i = C\_i \oplus K\_i$$

本实验将前 16 字节视为 nonce||初始计数器，按大端整数递增。CTR 模式无需填充，直接支持任意长度数据。

2. 关键函数

（1）密钥扩展函数 key_expansion

```
        def sub_word(w: int) -> int:
        return ((SBOX[(w >> 24) & 0xFF] << 24) |
        (SBOX[(w >> 16) & 0xFF] << 16) |
        (SBOX[(w >>  8) & 0xFF] <<  8) |
        SBOX[w & 0xFF])

        def rot_word(w: int) -> int:
        return ((w << 8) & 0xFFFFFFFF) | ((w >> 24) & 0xFF)

        def key_expansion(key: bytes):
        Nk, Nb, Nr = 4, 4, 10
        w = [0] * (Nb * (Nr + 1))
        for i in range(Nk):
        w[i] = (key[4*i] << 24) | (key[4*i+1] << 16) | \
        (key[4*i+2] << 8) | key[4*i+3]
        for i in range(Nk, Nb * (Nr + 1)):
        temp = w[i - 1]
        if i % Nk == 0:
        temp = sub_word(rot_word(temp)) ^ RCON[i // Nk]
        w[i] = w[i - Nk] ^ temp
        return w

```

上述代码实现了 AES-128 的密钥扩展。

首先将 16 字节密钥打包为 4 个 32 位 word，随后通过循环生成剩余 40 个 word。每当索引是 4 的倍数时，对上一 word 先进行字节循环移位（`rot_word`），再逐字节通过 S 盒替换（`sub_word`），最后异或上对应轮常数 `RCON[i//Nk]`。其余位置采用简单的前 4 个 word 异或方式递推。

（2）单块 AES 加解密：aes_encrypt_block 与 aes_decrypt_block

```
        def add_round_key(state, w, rnd):
        for c in range(4):
        word = w[4 * rnd + c]
        state[0+4*c] ^= (word >> 24) & 0xFF
        state[1+4*c] ^= (word >> 16) & 0xFF
        state[2+4*c] ^= (word >>  8) & 0xFF
        state[3+4*c] ^=  word        & 0xFF

        def aes_encrypt_block(block: bytes, round_keys):
        state = list(block)
        add_round_key(state, round_keys, 0)
        for rnd in range(1, 10):
        sub_bytes(state)
        shift_rows(state)
        mix_columns(state)
        add_round_key(state, round_keys, rnd)
        sub_bytes(state)
        shift_rows(state)
        add_round_key(state, round_keys, 10)
        return bytes(state)

        def aes_decrypt_block(block: bytes, round_keys):
        state = list(block)
        add_round_key(state, round_keys, 10)
        inv_shift_rows(state)
        inv_sub_bytes(state)
        for rnd in range(9, 0, -1):
        add_round_key(state, round_keys, rnd)
        inv_mix_columns(state)
        inv_shift_rows(state)
        inv_sub_bytes(state)
        add_round_key(state, round_keys, 0)
        return bytes(state)

```

在加密函数中，首先将输入 16 字节转为状态数组并与初始轮密钥异或，然后进行 9 轮完整变换（字节代换、行移位、列混淆、轮密钥加），最后一轮仅包含字节代换、行移位与轮密钥加。解密过程则严格按逆序执行：先与最后一轮轮密钥异或，再执行逆行移位、逆字节替换；中间 9 轮中，每轮先做轮密钥加，再执行逆列混淆、逆行移位、逆字节替换；最后再与初始轮密钥异或。

（3）CBC 模式封装：aes_cbc_encrypt 与 aes_cbc_decrypt

```
        def pkcs7_pad(data: bytes, block_size: int = 16) -> bytes:
```

```
        pad_len = block_size - (len(data) % block_size)
        if pad_len == 0:
        pad_len = block_size
        return data + bytes([pad_len]) * pad_len

        def pkcs7_unpad(data: bytes) -> bytes:
        if not data:
        return data
        pad_len = data[-1]
        if pad_len < 1 or pad_len > 16:
        return data
        if data[-pad_len:] != bytes([pad_len]) * pad_len:
        return data
        return data[:-pad_len]

        def aes_cbc_encrypt(plaintext: bytes, key: bytes, iv: bytes) ->
    bytes:
        round_keys = key_expansion(key)
        data = pkcs7_pad(plaintext, 16)
        prev = iv
        ct = b""
        for i in range(0, len(data), 16):
        block = data[i:i+16]
        xored = bytes(a ^ b for a, b in zip(block, prev))
        enc = aes_encrypt_block(xored, round_keys)
        ct += enc
        prev = enc
        return iv + ct

        def aes_cbc_decrypt(ciphertext_with_iv: bytes, key: bytes) ->
    bytes:
        iv = ciphertext_with_iv[:16]
        ct = ciphertext_with_iv[16:]
        round_keys = key_expansion(key)
        prev = iv
        pt = b""
        for i in range(0, len(ct), 16):
        block = ct[i:i+16]
        dec = aes_decrypt_block(block, round_keys)
        pt_block = bytes(a ^ b for a, b in zip(dec, prev))
        pt += pt_block
        prev = block
        return pkcs7_unpad(pt)

```

上述代码首先对明文进行 PKCS#7 填充，然后在每一轮中将明文块与上一轮密文块（首块使用 IV）进行按字节异或，之后送入 AES 单块加密函数。密文最终以 IV||C_0||C_1||... 的形式输出。解密时相反：对每个密文块先进行 AES 单块解密，再与上一块密文（首块用 IV）异或还原明文块，最后整体执行一次去填充。

（4）CTR 模式封装：aes_ctr_crypt 与包装函数

```
    def aes_ctr_crypt(data: bytes, key: bytes, nonce_counter: bytes) -> bytes:
        round_keys = key_expansion(key)
        counter = int.from_bytes(nonce_counter, "big")
        out = b""
        from math import ceil
        num_blocks = ceil(len(data) / 16)
        for i in range(num_blocks):
        counter_block = counter.to_bytes(16, "big")
        keystream = aes_encrypt_block(counter_block, round_keys)
        block = data[16*i:16*(i+1)]
        out += bytes(a ^ b for a, b in zip(block, keystream[:len(block)]))
        counter += 1
        return out

        def aes_ctr_encrypt(plaintext: bytes, key: bytes, nonce_counter: bytes) -> bytes:
        ct = aes_ctr_crypt(plaintext, key, nonce_counter)
        return nonce_counter + ct

        def aes_ctr_decrypt(ciphertext_with_nonce: bytes, key: bytes) -> bytes:
        nonce_counter = ciphertext_with_nonce[:16]
        ct = ciphertext_with_nonce[16:]
        return aes_ctr_crypt(ct, key, nonce_counter)

```

CTR 模式中，nonce_counter 作为 128 位大端整数，每处理一块数据就自增 1，并调用 aes_encrypt_block 对当前计数器块加密生成密钥流。明文和密文均通过与密钥流按字节异或得到，因此加密和解密可以使用统一的 aes_ctr_crypt 函数。由于 CTR 是流模式，不需要填充，可直接处理任意长度数据。

遇到的问题与解决方法

1. **CBC 模式解密后明文尾部出现乱码**

   原因：一开始在每个分组解密后立即尝试去填充，导致 PKCS#7 校验失败。

解决：修改为“先将所有分组解密并拼接成完整明文流，再统一调用一次 `pkcs7_unpad`”。

2. **CTR 模式解密结果与参考实现不一致**

   原因：计数器的字节序最初以小端方式实现。

   解决：统一采用大端字节序 `"big"` 解析和回写计数器，确保 nonce‖counter 的递增方向与 NIST 向量一致；同时固定前 16 字节作为 nonce‖初始计数器，之后每块自增一次。

3. 结果输出与正确性验证

实验中使用了课程给定的 AES-128 密钥和密文，对 CBC 与 CTR 两种模式分别进行解密与再加密。解密得到明文后，使用同一密钥与初始 IV/nonce 重新加密，并逐字节比较回加密密文与原始密文。

```
In [1]: runfile('F:/works/cyptowork/Third/AES.py', wdir='F:/works/cyptowork/Third')
CBC 解密结果:
Basic CBC mode encryption needs padding.
CBC 重新加密后密文 == 原密文 ?  True

CBC 解密结果:
Our implementation uses rand. IV
CBC 重新加密后密文 == 原密文 ?  True

CTR 解密结果:
CTR mode lets you build a stream cipher from a block cipher.
CTR 重新加密后密文 == 原密文 ?  True

CTR 解密结果:
Always avoid the two time pad!
CTR 重新加密后密文 == 原密文 ?  True
```

图 1: 运行结果

通过上述结果可以看出，CBC 与 CTR 两种模式下，解密得到的明文合理，预期正确。

四、体会与收获

本次实验实现了 AES-128 算法及其 CBC、CTR 两种工作模式，通过密钥扩展、轮函数、逆变换和分组模式组合，实现了加密与解密功能。

我学会了如何将 AES 的状态矩阵按列存储，并在字节代换、行移位和列混淆之间保持字节顺序的一致性。

我掌握了在 GF(2^8) 上实现乘法的方法，理解了列混淆中常数矩阵乘法的含义。

本次实验让我真正从实现层面理解了 AES 的内部细节。