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
23121677 范舒舰

SM4加密解密程序及OFB/CFB模式实现

2025 年 12 月 10 日

一、实验目的

1. 实验目的

- 本次实验的目的是通过动手实现国家商用密码标准SM4算法，深入理解SM4 算法的具体结构。

2. 实验要求

通过S盒、非线性变换 τ 、线性变换L等基本组件的手工实现，完成SM4 算法的实现，严禁直接调用现有的、完整的SM4加密库。使用以下测试实例进行验证：

#实例1:

- **密钥 (Key)**：0123456789ABCDEFFEDCBA9876543210
- **明文 (Plaintext)**：0123456789ABCDEFFEDCBA9876543210

在 OFB和CFB模式下使用SM4算法，分别使用以下测试实例进行验证：

#实例2:

- **密钥 (Key)**：0123456789ABCDEFFEDCBA9876543210
- **IV向量**：1234567890ABCDEFFEDCBA0987654321
- **明文 (Plaintext)**：Hello, OFB mode! (ASCII 编码十六进制表示: 48656C6C6F2C204F4642206D6F646521)

#实例3:

- **密钥 (Key)**：0123456789ABCDEFFEDCBA9876543210
- **IV向量**：1234567890ABCDEFFEDCBA0987654321
- **明文 (Plaintext)**：Testing CFB mode implementation (ASCII 编码十六进制表示： 54657374696E6720434642206D6F646520696D706C656D656 E746174696F6E)

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）
3. 开发工具：Spyder

三、实验内容

1. 算法原理与流程

（1）SM4 基本结构

SM4 是国家密码管理局公布的国产对称分组密码算法，分组长度为 128 比特，密钥长度也为 128 比特。与 AES 不同，SM4 使用类似 Feistel 的结构，以 32 轮轮函数对 128 比特明文进行反复迭代。

将 128 比特明文记为四个 32 位字：

$$X_0, X_1, X_2, X_3 \in \{0,1\}^{32}$$

设轮密钥为 $rk_0, \ldots, rk_{31}$，则 SM4 的轮迭代为：

$$X_{i+4} = X_i \oplus T(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus rk_i), \quad i = 0, \ldots, 31$$

经过 32 轮迭代后，输出按反序组合：

$$Y_0 = X_{35}, \quad Y_1 = X_{34}, \quad Y_2 = X_{33}, \quad Y_3 = X_{32}$$

四个 32 位字 (Y_0, Y_1, Y_2, Y_3) 再拼接为 128 比特密文。

（2）S 盒及非线性变换 τ

SM4 定义了一个 8 比特 S 盒 $S : \{0,1\}^8 \to \{0,1\}^8$，用来实现非线性。对一个 32 位字 A，可拆成 4 个字节：

$$A = (a_0 \| a_1 \| a_2 \| a_3)$$

其中每个 a_i 为 8 比特。非线性变换 τ 定义为对这 4 个字节分别查表：

$$B = \tau(A) = (S(a_0) \| S(a_1) \| S(a_2) \| S(a_3))$$

在程序中，先将 32 位整数拆为 4 个字节索引 S 盒，再拼回 32 位整数。

（3）线性变换 L 与 L'

在 SM4 中，有两种不同用途的线性变换：

- **轮函数线性变换 L：** 对 32 位输入 B，

$$L(B) = B \oplus (B \lll 2) \oplus (B \lll 10) \oplus (B \lll 18) \oplus (B \lll 24)$$

  其中 $\lll$ 在这里表示 32 位循环左移。

- **密钥扩展线性变换 L'：** 对 32 位输入 B，

$$L'(B) = B \oplus (B \lll 13) \oplus (B \lll 23)$$

在实现中，先用整数位运算实现 32 位循环左移函数 `rotl`，再按规范组合出 L 与 L'。

（4）复合变换 T 与 T'

轮函数和密钥扩展中都要先做 S 盒替换再做线性变换，因此定义：

$$T(X) = L(\tau(X)), \qquad T'(X) = L'(\tau(X))$$

其中 T 用于加/解密轮函数，T' 用于密钥扩展。

（5）密钥扩展

SM4 使用 128 比特原始密钥 MK 生成 32 个轮密钥 $rk_0, \ldots, rk_{31}$。过程如下：

1. 将 16 字节密钥分成 4 个 32 位字：

$$MK_0, MK_1, MK_2, MK_3$$

2. 定义系统参数 $FK_0, \ldots, FK_3$，先计算

$$K_0 = MK_0 \oplus FK_0, \quad \ldots, \quad K_3 = MK_3 \oplus FK_3$$

3. 定义 32 个固定轮常量 $CK_0, \ldots, CK_{31}$。之后对 $i = 0, \ldots, 31$：

$$K_{i+4} = K_i \oplus T'(K_{i+1} \oplus K_{i+2} \oplus K_{i+3} \oplus CK_i)$$

4. 将

$$rk_i = K_{i+4}, \quad i = 0, \ldots, 31$$

作为 32 轮轮密钥。

由于 T' 中包含 S 盒和线性变换 L'，轮密钥与原始密钥之间存在复杂的非线性关系。

（6）OFB 模式原理

OFB（Output Feedback）模式将分组密码变为类流密码。其特点是：

- 不直接加密明文块，而是反复用分组密码加密“状态”生成密钥流；
- 明文与密钥流按字节异或得到密文；解密过程与加密完全相同。

设块加密函数为 $E_K(\cdot)$，初始向量为 IV，则：

$$S_0 = IV, \quad S_i = E_K(S_{i-1})$$

$$C_i = P_i \oplus S_i, \qquad P_i = C_i \oplus S_i$$

其中 S_i 就是第 i 个分组的密钥流。由于 OFB 为流模式，适当截断最后一块密钥流即可处理任意长度数据，无需填充。

（7）CFB 模式原理

CFB（Cipher Feedback）模式同样可视为流模式，但其反馈的是上一块**密文**。以 128 比特分组长度为例：

- 加密：令 $S_0 = IV$，对第 i 块：

$$KS_i = E_K(S_i), \quad C_i = P_i \oplus KS_i, \quad S_{i+1} = C_i$$

- 解密：仍令 $S_0 = IV$，对第 i 块：

$$KS_i = E_K(S_i), \quad P_i = C_i \oplus KS_i, \quad S_{i+1} = C_i$$

可看到加解密的核心都是用 E_K 生成密钥流，只是输入数据分别是明文和密文，且状态更新始终使用“当前密文分组”。CFB 同样无需填充，最后一块不足 16 字节时只需截断密钥流对应长度。

2. 关键函数与代码实现

下面选取 SM4 实现中的几个关键函数，并结合代码进行说明。

（1）循环左移与 T/T’ 变换

```
    def rotl(x: int, n: int) -> int:
        n &= 31
        return ((x << n) & 0xFFFFFFFF) | ((x & 0xFFFFFFFF) >> (32 - n))

    def tau(a: int) -> int:
        b0 = SBOX[(a >> 24) & 0xFF]
        b1 = SBOX[(a >> 16) & 0xFF]
        b2 = SBOX[(a >>  8) & 0xFF]
        b3 = SBOX[a & 0xFF]
        return (b0 << 24) | (b1 << 16) | (b2 << 8) | b3

    def L(b: int) -> int:
        return b ^ rotl(b, 2) ^ rotl(b, 10) ^ rotl(b, 18) ^ rotl(b, 24)

    def L_prime(b: int) -> int:
        return b ^ rotl(b, 13) ^ rotl(b, 23)

    def T(x: int) -> int:
        return L(tau(x))

    def T_prime(x: int) -> int:
```

```
        return L_prime(tau(x))

```

上述代码首先实现了 32 位循环左移 `rotl`。随后利用 `tau` 函数实现字节级 S 盒替换，将 32 位整数拆成 4 个字节分别查表，再拼回。

L 和 `L_prime` 对应 SM4 规范中的线性变换 L 与 L'，以不同的位移参数组合异或实现。在此基础上，`T` 与 `T_prime` 将 S 盒替换与线性变换复合，用于轮函数和密钥扩展。

（2）密钥扩展函数

```
    def bytes_to_words(b: bytes) -> list[int]:
        if len(b) != 16:
            raise ValueError("block size must be 16 bytes")
        return [int.from_bytes(b[i*4:(i+1)*4], "big") for i in range(4)]

class SM4Cipher:
    def __init__(self, key: bytes):
        if len(key) != 16:
            raise ValueError("SM4 key length must be 16 bytes")
        self.round_keys = self._key_schedule(key)

    def _key_schedule(self, key: bytes) -> list[int]:
        MK = bytes_to_words(key)
        K = [0] * 36
        for i in range(4):
            K[i] = MK[i] ^ FK[i]

        rk: list[int] = []
        for i in range(32):
            tmp = K[i+1] ^ K[i+2] ^ K[i+3] ^ CK[i]
            K[i+4] = (K[i] ^ T_prime(tmp)) & 0xFFFFFFFF
            rk.append(K[i+4])
        return rk

```

`bytes_to_words` 函数将 16 字节密钥转换为 4 个 32 位大端整数。构造函数中对密钥长度进行检查，然后调用 `_key_schedule` 生成 32 轮轮密钥。

在密钥扩展中，首先按规范用 $MK_i \oplus FK_i$ 初始化 $K_0 \sim K_3$，然后在循环中利用前 3 个 K 值和常量 $CK[i]$ 通过 T' 变换递推得到 K_{i+4}。每次新得到的 K_{i+4} 即为一轮的轮密钥 rk_i。

（3）单块 SM4 加解密

```
    def words_to_bytes(words) -> bytes:
        return b"".join((w & 0xFFFFFFFF).to_bytes(4, "big") for w in
    words)

class SM4Cipher:
        ...
    def encrypt_block(self, block: bytes) -> bytes:
        if len(block) != 16:
            raise ValueError("SM4 block length must be 16 bytes")
        X = [0] * 36
        X[0:4] = bytes_to_words(block)

        for i in range(32):
            tmp = X[i+1] ^ X[i+2] ^ X[i+3] ^ self.round_keys[i]
            X[i+4] = (X[i] ^ T(tmp)) & 0xFFFFFFFF

        Y = [X[35], X[34], X[33], X[32]]
        return words_to_bytes(Y)

    def decrypt_block(self, block: bytes) -> bytes:
        if len(block) != 16:
            raise ValueError("SM4 block length must be 16 bytes")
        rev_keys = list(reversed(self.round_keys))
        X = [0] * 36
        X[0:4] = bytes_to_words(block)

        for i in range(32):
            tmp = X[i+1] ^ X[i+2] ^ X[i+3] ^ rev_keys[i]
            X[i+4] = (X[i] ^ T(tmp)) & 0xFFFFFFFF

        Y = [X[35], X[34], X[33], X[32]]
        return words_to_bytes(Y)

```

加密时，将输入 16 字节分组拆为 X_0, X_1, X_2, X_3，然后按 32 轮迭代公式更新到 X_{35}。最后根据规范将 $X_{35}, X_{34}, X_{33}, X_{32}$ 反序组合输出。

解密只需将轮密钥数组反转，按同样的迭代公式更新 X，即可得到原明文。

（4）OFB 模式封装

```
BLOCK_SIZE = 16

    def xor_bytes(a: bytes, b: bytes) -> bytes:
        return bytes(x ^ y for x, y in zip(a, b))
```

```
    def sm4_ofb_crypt(cipher: SM4Cipher, iv: bytes, data: bytes) -> bytes:
        if len(iv) != BLOCK_SIZE:
            raise ValueError("IV length must be 16 bytes")

        state = iv
        out = bytearray()

        for offset in range(0, len(data), BLOCK_SIZE):
            block = data[offset:offset+BLOCK_SIZE]
            keystream = cipher.encrypt_block(state)
            out_block = xor_bytes(block, keystream[:len(block)])
            out.extend(out_block)
            state = keystream

        return bytes(out)

```

OFB 模式的实现过程为：

- 使用 state 记录当前状态 S_i，初始为 IV；
- 每轮对 state 调用 encrypt_block 得到密钥流块；
- 将当前明文/密文分组与密钥流做按字节异或；
- 将新的状态更新为本轮密钥流，用于下一轮。

由于 OFB 模式下加密和解密完全相同，统一使用一个函数 sm4_ofb_crypt 即可。

（5）CFB 模式封装

```
    def sm4_cfb_encrypt(cipher: SM4Cipher, iv: bytes, data: bytes) -> bytes:
        if len(iv) != BLOCK_SIZE:
            raise ValueError("IV length must be 16 bytes")

        state = iv
        out = bytearray()

        for offset in range(0, len(data), BLOCK_SIZE):
            block = data[offset:offset+BLOCK_SIZE]
            keystream = cipher.encrypt_block(state)
            cipher_block = xor_bytes(block, keystream[:len(block)])
            out.extend(cipher_block)
```

```
            if len(block) == BLOCK_SIZE:
                state = cipher_block
        return bytes(out)

    def sm4_cfb_decrypt(cipher: SM4Cipher, iv: bytes, data: bytes) ->
    bytes:
        if len(iv) != BLOCK_SIZE:
            raise ValueError("IV length must be 16 bytes")

        state = iv
        out = bytearray()

        for offset in range(0, len(data), BLOCK_SIZE):
            block = data[offset:offset+BLOCK_SIZE]
            keystream = cipher.encrypt_block(state)
            plain_block = xor_bytes(block, keystream[:len(block)])
            out.extend(plain_block)
            if len(block) == BLOCK_SIZE:
                state = block
        return bytes(out)

```

CFB 模式状态使用上一块密文：

- **加密**：当前状态 `state` 经过 SM4 加密得到密钥流，与明文块异或生成密文块，同时用密文块更新状态；
- **解密**：当前状态仍然经过 SM4 加密得到密钥流，与密文块异或还原明文块，状态更新使用的是“当前密文块本身”。

最后一块不足 16 字节时，只需截取前若干字节的密钥流参与运算，状态无需再更新。

（6）遇到的问题与解决方法

1. **轮密钥顺序错误导致解密失败**
   原因：最初实现解密函数时，直接重用加密时的轮密钥顺序，导致解密出的结果完全错误。
   解决：解密过程与加密结构相同，只是轮密钥需要倒序使用。最终在构造 `decrypt_block` 时显式对轮密钥数组做 `reversed`。

2. **OFB/CFB 下最后一块长度处理不当**
   原因：一开始默认数据长度一定是 16 字节的整数倍，直接按块处理，导致当测试字符串长度不是 16 的整数倍时出现截断或越界问题。
   解决：将循环按 `range(0, len(data), BLOCK_SIZE)` 遍历，使用

`data[offset:offset+BLOCK_SIZE]` 获取当前块，并在异或时只取 `keystream[:len(block)]`，从而自然支持任意长度数据而无需填充。

3. 结果输出与正确性验证

程序运行时，对实例 1 直接调用 SM4 单块加密函数，将输出的密文与标准向量逐字节比较，结果完全一致，说明核心算法实现正确。

在 OFB/CFB 模式下，对给定明文进行加密后，再使用相同的密钥和 IV 进行解密，得到的明文与原始输入完全相同。实验中将明文的 ASCII 十六进制表示、密文十六进制表示以及解密回的字符串一并输出，便于人工比对。

```
In [1]: runfile('F:/works/cyptowork/4/SM4.py', wdir='F:/works/cyptowork/4')
===== 实例 1: SM4 单块加密 =====
Key       : 0123456789ABCDEFFEDCBA9876543210
Plaintext : 0123456789ABCDEFFEDCBA9876543210
Cipher    : 681EDF34D206965E86B3E94F536E4246
与标准结果是否一致:  True

===== 实例 2: SM4-OFB 模式 =====
Key       : 0123456789ABCDEFFEDCBA9876543210
IV        : 1234567890ABCDEFFEDCBA0987654321
Plaintext : Hello, OFB mode!
Plaintext HEX: 48656C6C6F2C204F4642206D6F646521
Cipher HEX   : 148600F9004C3150D21C334BBA9906A5
Decrypt text : Hello, OFB mode!
解密是否还原原文:  True

===== 实例 3: SM4-CFB 模式 =====
Key       : 0123456789ABCDEFFEDCBA9876543210
IV        : 1234567890ABCDEFFEDCBA0987654321
Plaintext : Testing CFB mode implementation
Plaintext HEX: 54657374696E6720434642206D6F646520696D706C656D656E746174696F6E
Cipher HEX   : 08861FE1060E763FD7185106B89207E195617D98828AA42D68CEEF52E5C331
Decrypt text : Testing CFB mode implementation
解密是否还原原文:  True
```

图 1: SM4 核心及 OFB/CFB 模式运行结果示意

从运行结果可以看出：

- 单块 SM4 加密输出与官方标准测试向量完全一致；
- OFB 模式下加密、解密互为逆操作，能够准确还原明文；
- CFB 模式下同样能够正确恢复原始字符串，证明模式封装逻辑正确。

四、体会与收获

本次实验实现了 SM4 分组密码的核心算法及其 OFB、CFB 两种工作模式，让我对国产密码算法的具体细节有了更加直观的理解。

在实现 SM4 过程中，我加深了对“非线性 S 盒 + 线性变换”的设计思想的理解，体会到通过 S 盒实现混淆，通过多次循环移位和异或实现扩散，可以在简单的位运算基础上实现较强的安全性。

在密钥扩展与轮函数部分，我体会到严格遵从规范的重要性：一个字节序或输出顺序的细微错误就会导致结果完全错误。

在 OFB 和 CFB 模式实现中，我理解了不同分组模式的本质区别：OFB 将分组密码变成纯密钥流生成器，而 CFB 则把上一块密文作为反馈。二者都能自然支持任意长度的数据，而无需填充。