密码学实验报告

课程名称	密码学实验		成　　绩		教师签名	
实验名称	期末综合实验		实验序号	1	实验日期	2023.12.28
姓　　名	吕岚曦	学　　号	2021302181160			

目录

一、 实验简介

1.1 实验问题

大作业： 以 SM4 作为加密算法开发出文件加密软件系统，软件要求如下：

1. 具有文件加密和解密功能；
2. 具有加解密速度统计功能；
3. 采用密文链接和密文挪用短块处理技术；
4. 具有较好的人机界面

二、 实验背景

2.1 SM4 算法

2006 年我国国家密码管理局公布了无线局域网产品使用的 SM4 密码算法。这是我国第一次公布自己的商用密码算法，意义重大，影响深渊。这一举措标志着我国商用密码管理更加科学化和与国际接轨。这必将促进我国商用密码的科学研究和产业发展。目前我国的国密算法如下：

序号	密码分类		国密算法
1	对称加密	分组加密/块加密	SM1(SCB2)、SM4(SMS4)、SM7
2		序列加密/流加密	ZUC（祖冲之算法）、SSF46
3	非对称/公钥加密	离散对数	SM2、SM9
4	密码杂凑/散列		SM3

图 1: 国密算法

SM4 算法用于保护 WAPI 无线局域网协议中的数据安全，其可抵抗差分分析、线性分析、零相关矩阵线性分析、矩阵分析、不可能差分分析、积分分析和代数分析等传统密码分析的攻击，在安全性上优于 3DES 算法，同时 SM4 算法计算速度快，执行稳定，是我国密码标准的一大突破。

SM4 密码算法是一个分组算法，采用非对称 Feistel 结构，加解密过程以及密钥扩展算法都采用 32 轮迭代结构。SM4 密码的分组长度为 128 位，密钥长度也为 128 位。在密钥扩展阶段，密钥扩展函数将 128 位的密钥拆分成 4 个 32 位的字段，然后和提前定义的系统参数进行多轮异或、S 盒替换等操作，每轮产生一个轮密钥，然后将轮密钥和之前的密钥拼接作为下一个输入，如此迭代 32 次后产生共 32 个轮密钥。对明/密文的处理与密钥类似，采用 32 轮迭代结构，每轮使用一个 32 位的轮密钥进行异或、S 盒替换和移位等操作，产生多轮中间结果作为下一轮的输入。对 32 轮输出的中间结果再进行一次反序变换，就能得到本轮明文分组相应的密文。SM4 密码算法是对合运算，因此解密算法与加密算法相同，只是轮密钥的使用顺序相反，解密轮密钥是加密轮密钥的逆序。

2.2 分组密码工作模式

分组密码可以按不同的模式工作，实际应用的环境不同应采用不同的工作模式。只有这样才能既确保安全，又方便高效。下面简要介绍我们在本实验中使用到的几种工作模式。

2.2.1 电话本模式 ECB

此模式直接利用分组密码对明文的各分组进行加密。设明文 $M = (M_1, M_2, ..., M_n)$，响应的密文 $C = (C_1, C_2, ..., C_n)$ ，其中

$$C_i = E(M_i, K), i = 1, 2, ..., n$$

电话本模式是分组密码的基本工作模式。其缺点如下：

- 要求数据的长度是密码分组长度的整数倍，否则最后一个数据块将是短块，这时需要特殊处理。
- 容易暴露明文的数据模式。

其加解密过程如下：

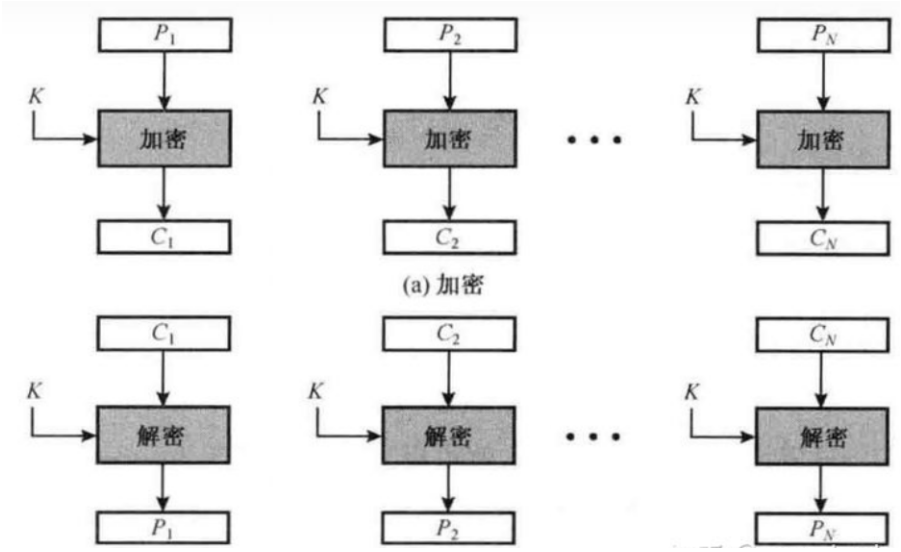

图 2: ECB 加解密过程

2.2.2 密文链接模式 CBC

考虑到 ECB 工作模式的不安全性，推出了链接技术。所谓链接就是使加解密算法的当前输出不仅与当前的输入和密钥相关，而且也和先前的输入和输出相关的一种技术。密码科学家们首先推出了明密文连接模式，在此不做具体描述，其具有加解密错误传播无界的缺点。

由于明密文链接模式具有加解密错误传播无界的特性，而磁盘等文件通常希望错误传播有界，这时可采用密文链接模式，即明文不参与链接，只让密文参与链接。加密算法的输入是明文分组和前一个密文分组的异或，同样均使用相同的密钥进行加密。其中第一个明文加密时，需先与初始向量 IV 异或，再进入加密算法进行加密。如下：

$$C_i = \begin{cases} E(M_i \oplus IV, K), i = 1 \\ E(M_i \oplus C_{i-1}, K), i = 2, ..., n \end{cases}$$

其特点如下：

- 无法处理短块
- 加密错误传播无界，解密错误传播有界

加解密过程如下：

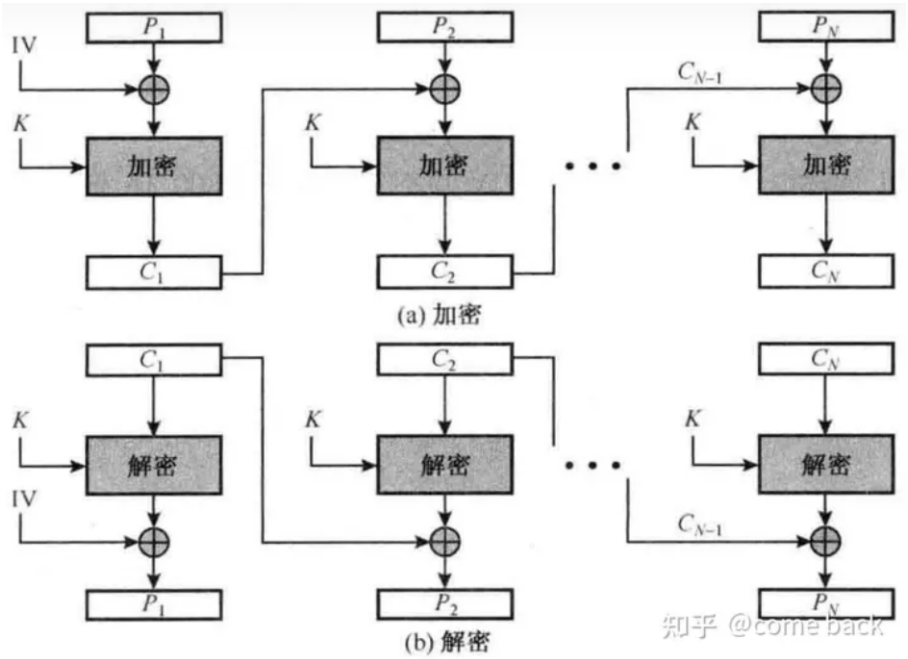

图 3: CBC 加解密过程

2.3 短块处理技术

因为分组密码一次只能对一个固定长度的明文（密文）块进行加（解）密，而对长度小于分组长度的明文（密文）块不能正确进行加（解）密。称长度小于分组长度的数据块为短块。当明文长度大于分组长度而又不是分组长度的整数倍时，短块总是最后一块。无论是网络通信加密还是文件加密，短块是经常遇到的。下面介绍我们使用的两种短块处理方式。

2.3.1 填充法

第一种方式直接利用 0 将短块补充到一个分组的大小。具体方式如下：

$$Pad(X) = \begin{cases} X, \text{当 X 不是短块} \\ X0...0, \text{当 X 是短块} \end{cases}$$

2.3.2 密文挪用技术

在对短块 M_n 加密之前，首先从密文 C_{n-1} 中挪出刚好够填充的位数，填充到 M_n 中去，使 M_n 成为一个标准块。这样 C_{n-1} 却成了短块。然后再对填充后的 M_n 加密，得到密文 C_n。虽然 C_{n-1} 是短块，但 C_n 却是标准块，两者的总位数等于 M_{n-1} 和 M_n 的总位数，没有数据扩张。解密时先对 C_n 解密，还原出明文 M_n 和从 C_{n-1} 中挪用的数据。把从 C_{n-1} 中所挪用的数据再挪回 C_{n-1}，然后再对 C_{n-1} 解密，还原出 M_{n-1}。当明文本身就是一个短块时，用初始向量 Z 代替 C_{n-1}。

解密时 C_{n-1} 中的错误只影响 M_{n-1} 也产生错误，而 C_n 中的错误则将影响 M_n 和 M_{n-1} 都发生错误。密文挪用加密短块的优点是不引起数据扩展，缺点是解密时要先解密 C_n、还原挪用后再解密 C_{n-1}，从而使控制复杂。

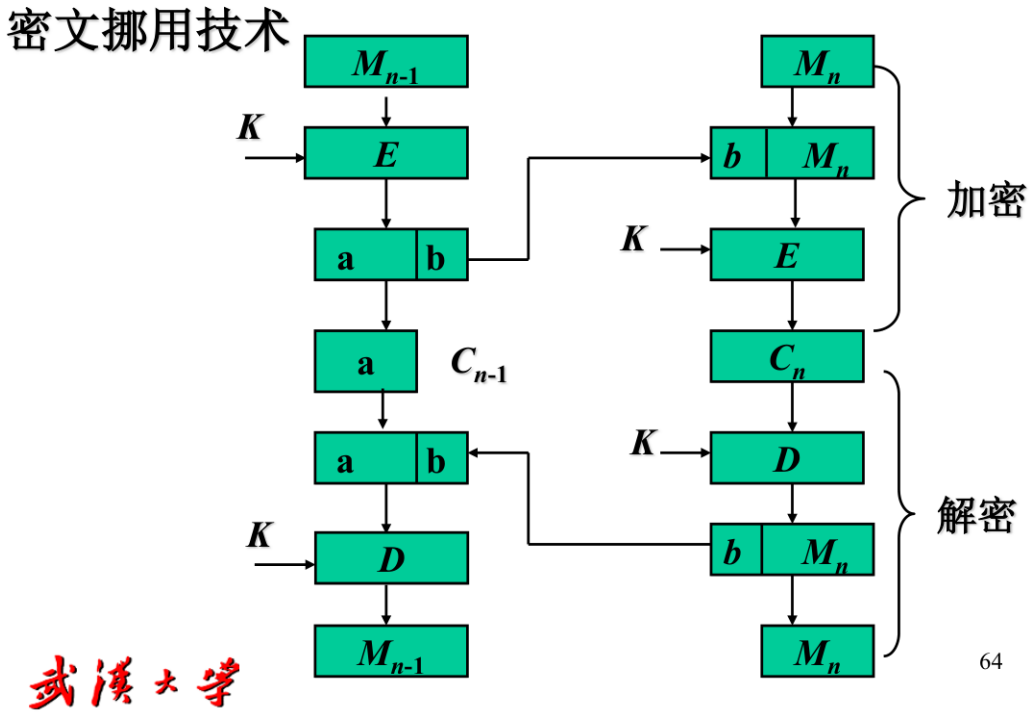

图 4: 密文挪用技术

2.4 文件加密

随着信息技术的发展，云存储、电子邮件、文件通信的使用日益广泛，文件加密变得尤为重要。网络犯罪、黑客技术和数据泄露事件的层出不穷让人们认识到了文件加密的重要性。文件作为存储重要信息的载体如何高效、安全地进行传输，相关密钥如何存储成为了十分重要的问题。

三、 实验具体流程

3.1 SM4 算法实现

SM4 算法包括密钥扩展和加/解密两个部分，在我们的实现中，我们使用了两个类 **sm4_keys** 和 **sm4** 来分别完成两部分的工作，实际上，这两个部分的工作是类似的，都会经过异或、移位、S 盒变换等过程。下面分别给出两个部分的流程图，并根据我们的代码来分别讲解两部分的具体实现方式。

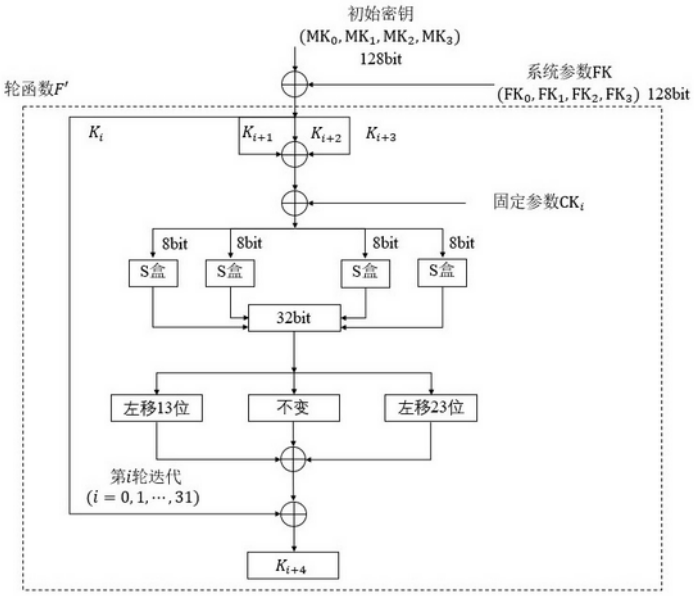

图 5: 密钥扩展

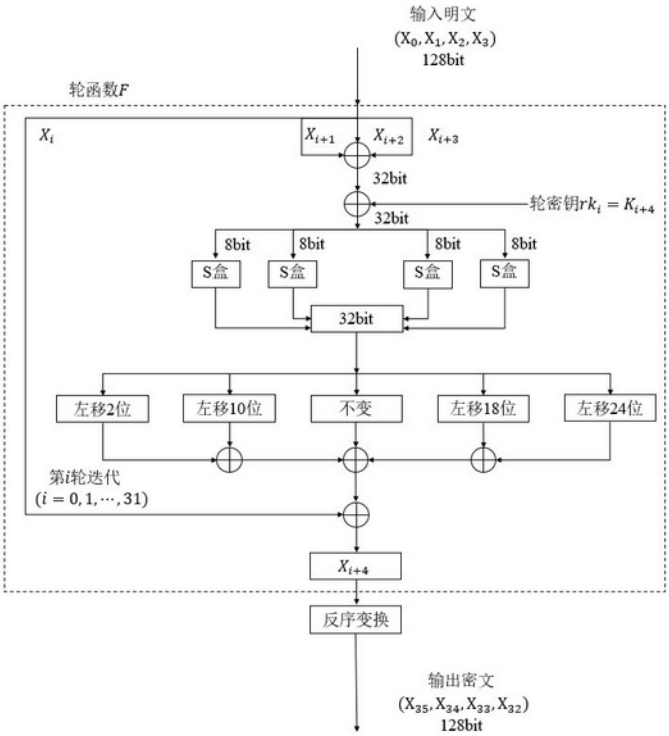

图 6: 加密/解密

3.1.1 密钥扩展

本部分内容在 **sm4_keys** 类中实现。在上文提到，SM4 需要经过 32 轮迭代，且每轮迭代都需要一个 1 字的轮密钥，但是初始密钥只有 4 个字，因此需要密钥扩展算法。我们将输入的 128 位密钥转换为 4 个 32 位字，然后进行 32 轮密钥扩展，用于后续加密算法的实现。

i、密钥初始化 首先需要让原始输入密钥（设为 Key=（$Key_0, Key_1, Key_2, Key_3$））与系统参数 FK_i 异或，得到 K=（K_0, K_1, K_2, K_3）。即

$$(K_0, K_1, K_2, K_3) = (Key_0 \oplus FK_0, Key_1 \oplus FK_1, Key_2 \oplus FK_2, Key_3 \oplus FK_3)$$

其中 FK_i 的取值如下：

```
self.fk = [0xa3b1bac6, 0x56aa3350, 0x677d9197, 0xb27022dc]
```

图 7: 系统参数 FK

ii、迭代生成轮密钥 在上一步中，我们得到了初始化后的密钥 (K_0, K_1, K_2, K_3)，下面我们需要对于这 4 个字进行 32 轮迭代生成 32 个轮密钥。现以前两轮迭代为例，介绍迭代的进行方式。

1. **第一轮迭代**：根据初始化的四个字 (K_0, K_1, K_2, K_3)，计算出第五个字 K_4 的值，并且将 K_4 作为第一轮的轮密钥 rk_0。计算方法如下：$rk_0 = K_4 = K_0 \oplus L(K_1 \oplus K_2 \oplus K_3 \oplus CK_0)$。其中 L 函数以及 CK 的值将在下文给出。

2. **第二轮迭代**：根据上一轮迭代的结果 (K_1, K_2, K_3, K_4)，计算出第六个字 K_5 的值，作为下一轮的轮密钥，计算方法如下：$rk_1 = K_5 = K_1 \oplus L(K_2 \oplus K_3 \oplus K_4 \oplus CK_1)$。以此类推我们可以找出密钥迭代的通式为：

$$rk_i = K_{i+4} = K_i \oplus L(K_{i+1} \oplus K_{i+2} \oplus K_{i+3} \oplus CK_i), i = 0, 1, ..., 31$$

下面给出上面轮密钥迭代计算通式中 CK 的值：

```
self.ck = [
            0x00070e15, 0x1c232a31, 0x383f464d, 0x545b6269,
            0x70777e85, 0x8c939aa1, 0xa8afb6bd, 0xc4cbd2d9,
            0xe0e7eef5, 0xfc030a11, 0x181f262d, 0x343b4249,
            0x50575e65, 0x6c737a81, 0x888f969d, 0xa4abb2b9,
            0xc0c7ced5, 0xdce3eaf1, 0xf8ff060d, 0x141b2229,
            0x30373e45, 0x4c535a61, 0x686f767d, 0x848b9299,
            0xa0a7aeb5, 0xbcc3cad1, 0xd8dfe6ed, 0xf4fb0209,
            0x10171e25, 0x2c333a41, 0x484f565d, 0x646b7279
        ]
```

图 8: 固定参数 CK

下面介绍上述提到的 L 函数。其包含两部分：非线性变换 S 盒代换（函数 S 和函数 S_byte 共同实现），线性变换循环左移（函数 $rotate\_left$）。在非线性变换中，由于我们得到的密钥实际上是 32 位的而 S 盒的输入输出为 8 位，因此我们需要将 32 位的字拆分为四个 8 位的字节。为了实现这个操作，我们利用函数 *num2list*，同理，我们从每个 S 盒中得到的结果也是 8 位，利用 *list2num* 将其转换为 32 位。其具体内容如下：

```
def num2list(number, digit):
    mask = 2 ** digit - 1
    res = []
    for i in range(4):
        res.append((number >> (i * digit)) & mask)
    res = list(reversed(res)) if digit == 32 else res
    return res
```

图 9: 函数 *num2list*

```python
def list2num(num_list, digit):
    Sum = 0
    factor = 1
    for i in range(4):
        Sum += num_list[i] * factor
        factor = factor << digit
    return Sum
```

图 10: 函数 *list2num*

实际上在我们输入密钥时将密钥分为四个字时也用到了这个函数。对于每个 S 盒的 8 位输入，高 4bit 对应 S 盒的行号，低 4bit 对应 S 盒的列号，返回 S 盒中对应位置的数据。

```python
def S(self, word):
    Bytes = num2list(word, 8)
    num_list = [self.S_byte(Bytes[i]) for i in range(4)]
    Sum = list2num(num_list, 8)
    return Sum
def S_byte(self, byte):
    row = byte >> 4
    col = byte & 0xf
    return self.s[row * 16 + col]
```

图 11: 非线性变换实现

对于线性变换，我们将非线性变换函数的返回值作为输入，其值与它循环左移 13 位、循环左移 23 位的结果进行异或，最后得到的即是我们前面提到的函数 L 的返回值。下面是循环左移函数 *rotate_left* 以及函数 L 的实现代码。

```python
# L函数的实现，word为输入的字，dig_list为循环左移位数构成的列表，将每次循环左移后的值进行异或
    def L(self, word, dig_list):
        res = 0
        for digit in dig_list:
            res = res ^ self.rotate_left(word, digit)
        return res
    # 循环左移的实现代码，digit为循环左移的位数
    def rotate_left(self, word, digit):
        high = word >> (32 - digit)
        res = high + (word << digit) & 0xffffffff
        return res
```

图 12: 函数 *rotate_left* 和函数 L

假设当前为第 i 轮，再将我们 L 函数的输出结果与 K_{i-1} 做异或运算，即可得到轮密钥 rk_{i-1}。

```python
# 密钥扩展算法轮函数，每次扩展keys列表
def epoch_func(self, keys, epoch):
    S_input = keys[-3] ^ keys[-2] ^ keys[-1] ^ self.ck[epoch]
    K = keys[-4] ^ self.L(self.S(S_input), [0, 13, 23])
    keys.append(K)
    return keys
# 密钥扩展函数,获取扩展后的密钥
def extend(self):
    keys = []
    for key, fk in zip(self.keys, self.fk):
        keys.append(key ^ fk)
    for i in range(32):
        keys = self.epoch_func(keys, i)
    return keys[4:]
```

图 13: 轮密钥生成

3.1.2 加/解密实现

加/解密的实现在类 **sm4** 中。加/解密算法的实现实际上与密钥扩展类似，也要经过 32 轮迭代，其中每一轮利用上面生成的一个轮密钥，大小为 32 位。加/解密算法的输入的数据分组长度和密钥长度均为 128bit，经过 32 次轮函数 F 后经过反序变换得到密/明文。

首先利用函数 *list2num* 将输入的 128bit 分为四个 32 位字 (X_0, X_1, X_2, X_3)，将其作为轮函数 F 的输入。其中 X_1, X_2, X_3 与轮密钥做异或的结果作为非线性变换的输入。同样，由于输入是一个 32 位

的字，我们要利用函数 *num2list* 将其转换为四个 8 位字节输入。在加/解密中也是利用 S 盒对输入做非线性变换，其实现方式与我们在密钥扩展算法中介绍的一样，高 4bit 作为行号，低 4bit 作为列号。因此我们在此可以直接利用 **sm4_keys** 类中的方法。

接着要进行线性变换，此处与密钥扩展略有不同，密钥扩展时是将非线性的输出结果与其循环左移 13 位和 23 位的值进行异或得到结果。在加/解密算法中是通过将非线性的输出与其循环左移 2 位、10 位、18 位、24 位的结果做异或。在之前 **sm_keys** 类中我们使用的 *rotate_left* 函数中传入的是一个列表。因此我们在此也可以直接利用这个方法。最后这个结果与 X_0 做异或即为一个中间结果 X_4。之后与密钥扩展类似，下一轮的输入为 X_1, X_2, X_3, X_4，依此类推。进行 32 轮迭代，最后的输出结果经历过一次反序变换后即是密文。

由于 SM4 是对合运算，那么解密过程其实是与加密过程相同的，唯一的不同是需要将轮密钥进行逆序输入。下面是我们整个加/解密部分的代码实现。

```
class sm4():
    # 初始化sm4算法，获得轮密钥
    def __init__(self, keys):
        self.Sm4_keys = sm4_keys(keys)
        self.keys = self.Sm4_keys.extend()
    # sm4算法轮函数，使用了sm4_keys算法中的实现方法
    def F(self, messages, epoch):
        S_input = messages[1] ^ messages[2] ^ messages[3] ^ self.keys[epoch]
        M = messages[0] ^ self.Sm4_keys.L(self.Sm4_keys.S(S_input), [0, 2, 10, 18
, 24])
        messages.append(M)
        return messages[1:]
    # sm4加密函数 int → int
    def encryption(self, M):
        messages = num2list(M, 32)
        for i in range(32):
            messages = self.F(messages, i)
        Sum = list2num(messages, 32)
        return Sum
    # sm4解密函数 int → int
    def decryption(self, C):
        messages = num2list(C, 32)
        for i in range(32):
            messages = self.F(messages, 31 - i)
        Sum = list2num(messages, 32)
        return Sum
```

图 14: 加解密实现

3.2 工作模式与短块处理

为了提高我们 SM4 算法的实用性，我们设计了两种工作模式（ECB 模式和 CBC 模式）提供给用户选择，同时每种模式都采用了两种短块处理方式（填充法和密文挪用法）。因此有四种组合方式可供用户选择。

因为我们实现的是一个文件加密，同时 SM4 密码算法的分组大小是 128bit，所以我们先实现了读

写文件函数，但是对于密文挪用法加密的密文在解密时需要特殊处理其读写文件函数有所不同，下文会详细介绍。下面是我们读文件函数 *read_file* 的实现。

```
def read_file(file_path):
    res = []
    with open(file_path, "rb") as f:
        p = f.read()
        for i in range(0, len(p), 16):
            j = min(len(p), i + 16)
            res.append(int.from_bytes(p[i:j], byteorder="little", signed
=False))
    return res, len(p) % 16
```

图 15: *read_file* 函数

我们的实现思路是将文件作为字节串读入，每 16 个字节一组（即 128 位一组），并且提供 *int.from_bytes* 将字节串转为整型加入 res 列表，这将成为我们的输入明/密文。最后返回的内容中的 **len(p)%16** 为我们短块的大小。

下面我们按照几种组合方式来讲解我们工作模式和短块处理的实现。

3.2.1 ECB 模式 + 填充法

这是最简单的一种模式，主要就是对于最后一个分组的处理，当短块存在时，我们利用一个临时变量实现短块的填充。

```
if remainder != 0:
        temp = messages[-1]
        temp = temp.to_bytes(16, byteorder='little')
        messages[-1] = int.from_bytes(temp, byteorder="little",
signed=False)
```

图 16: 填充法

同时由于输入的密钥实际上是字符串，但是我们 SM4 密码算法输入的是一个整型，因此通过函数 *str2num* 将输入的字符串转换成整型处理。

```python
def str2num(s):
    b = s.encode()
    return int.from_bytes(b, byteorder="little", signed=False)
```

图 17: 函数 *str2num*

在上述工作都完成之后我们只用调用之前实现的加密算法即可。下面是 ECB 模式 + 填充法的完整代码实现：

```python
def EN_ECB_PAD(file_path, target_path, keys):
    num_keys = str2num(keys)
    messages, remainder = read_file(file_path)
    if remainder != 0:
        temp = messages[-1]
        temp = temp.to_bytes(16, byteorder='little')
        messages[-1] = int.from_bytes(temp, byteorder="little",
signed=False)
    Sm4 = sm4(num_keys)
    res = []
    for message in messages:
        res.append(Sm4.encryption(message))
    write_file(target_path, res)
```

图 18: ECB+padding 加密

对于解密函数，我们读取的内容是加密过后的文件，那么其一定是没有短块的，那么我们可以直接利用解密算法将其解密。为了使算法更加简洁，我们在此处没有考虑将填充部分的解密结果去除，而是放在了 *write_file* 中处理。下面是我们解密函数的实现：

```python
def DE_ECB_PAD(file_path, target_path, keys):
    num_keys = str2num(keys)
    messages, remainder = read_file(file_path)
    Sm4 = sm4(num_keys)
    res = []
    for message in messages:
        res.append(Sm4.decryption(message))
    write_file(target_path, res, True)
```

图 19: ECB+padding 解密

下面对上文提到的 *write_file* 函数介绍。我们将解密结果以字节串的方式写入文件，函数的第三个参数默认为 False，当使用 padding 时第三个参数设为 True，并将填充的 0 在写入文件之前去除。

```python
def write_file(file_path, res, padding=False):
    with open(file_path, "wb") as f:
        for num in res[:-1]:
            bin_num = num.to_bytes(16, byteorder='little')
            f.write(bin_num)
        num = res[-1]
        bin_num = num.to_bytes(16, byteorder='little')
        if padding:
            bin_num = bin_num.replace(b"\x00", b"")
        f.write(bin_num)
```

图 20: 函数 *write_file*

3.2.2 ECB 模式 + 密文挪用技术

使用密文挪用技术处理短块时，与 padding 方式有所不同。明/密文直接加/解密前 n - 1 个分组，最后一个分组进行特殊处理。首先将短块与分组的差计算出来，取第 n - 1 块密文，截取需要的长度拼接到该组明文的末尾，然后把该块密文剩余部分放回，最后再加密最后一组明文。那么我们现在的最后一组的组成就是：倒数第二组的部分密文 + 最后一组的明文。我们的加密代码实现如下：

```python
def EN_ECB_STE(file_path, target_path, keys):
    num_keys = str2num(keys)
    messages, remainder = read_file(file_path)
    Sm4 = sm4(num_keys)
    res = []
    for message in messages[:-1]:
        res.append(Sm4.encryption(message))
    if remainder == 0:
        res.append(Sm4.encryption(messages[-1]))
    else:
        inv_remainder = 16 - remainder
        temp = res[-1]
        temp_high = temp >> (inv_remainder * 8)
        temp_low = (temp & (2 ** (inv_remainder * 8) - 1))
        res[-1] = temp_high
        temp_low = temp_low << (remainder * 8)
        ls_c = Sm4.encryption(temp_low ^ messages[-1])
        res.append(ls_c)
    write_file_STE(target_path, res, remainder)
```

图 21: ECB+STE 加密

解密方式与加密方式也略有不同，对于利用密文挪用方式生成的密文，正如上文提到的，我们在解密时利用一个新的读文件方式 *read_file_STE*。这种方式先读前 n-2 个分组，对于最后两个分组，先算出多出的短块的大小，并将其放在倒数第二个分组中，最后一个分组仍是一个完整的分组，即在解密时，短块为倒数第二个分组。在解密时，正是因为我们的读取方式，我们先解密前 n-2 个分组，再对最后一个分组解密，得到的实际上是倒数第二个分组的部分密文 + 最后一个分组的明文，并将其解密的结果中高 xB（x=16B-短块大小，即原本倒数第二个分组的部分密文）的内容，通过异或的方式赋给倒数第二个分组。

```python
def read_file_STE(file_path):
    res = []
    with open(file_path, "rb") as f:
        p = f.read()
        i = 0
        while i < len(p) - 32:
            res.append(int.from_bytes(p[i: i + 16], byteorder="little",
signed=False))
            i += 16
        remainder = len(p) % 16
        remainder = 16 if remainder == 0 else remainder
        res.append(int.from_bytes(p[i : i + remainder], byteorder="little",
signed=False))
        res.append(int.from_bytes(p[i + remainder: i + remainder + 16],
byteorder="little", signed=False))
    return res, remainder % 16
```

图 22: *read_file_STE*

下面是我们的解密函数实现：

```python
def DE_ECB_STE(file_path, target_path, keys):
    num_keys = str2num(keys)
    Sm4 = sm4(num_keys)
    messages, remainder = read_file_STE(file_path)
    res = []
    for message in messages[:-2]:
        res.append(Sm4.decryption(message))
    temp = Sm4.decryption(messages[-1])
    inv_remainder = 16 - remainder
    temp_high = temp >> (remainder * 8)
    temp_low = (temp & (2 ** (remainder * 8) - 1))
    temp = messages[-2]
    temp = (temp << (inv_remainder * 8)) ^ temp_high
    temp = Sm4.decryption(temp)
    res.append(temp)
    res.append(temp_low)
    write_file(target_path, res, True)
```

图 23: ECB+STE 解密

同时，对于 STE 方式处理短块得到密文的解密，写文件时也需要利用特殊的函数实现 *write_file_STE*。在写入时直接写入前 n-2 个组，对于最后两个组分别写入一个短块的大小和一个分组的大小。

```python
def write_file_STE(file_path, res, remainder):
    with open(file_path, "wb") as f:
        for num in res[:-2]:
            bin_num = num.to_bytes(16, byteorder='little')
            f.write(bin_num)
        num = res[-2]
        remainder = 16 if remainder == 0 else remainder
        bin_num = num.to_bytes(remainder, byteorder='little')
        f.write(bin_num)
        num = res[-1]
        bin_num = num.to_bytes(16, byteorder='little')
        f.write(bin_num)
```

图 24: *write_file_STE*

3.2.3 CBC 模式 + 填充法

CBC 模式的具体工作方式我们在前面已经介绍过了，具体来说我们要引入一个随机向量 IV，通过 *random.randint* 实现，并且我们每次加密的明文都需要与上一次解密的密文进行异或操作。

```python
def EN_CBC_PAD(file_path, target_path, keys):
    IV = random.randint(0, 2 ** 128 - 1)
    num_keys = str2num(keys)
    messages, remainder = read_file(file_path)
    if remainder != 0:
        temp = messages[-1]
        temp = temp.to_bytes(16, byteorder='little')
        messages[-1] = int.from_bytes(temp, byteorder="little", signed=False)
    Sm4 = sm4(num_keys)
    res = [IV]
    for message in messages:
        res.append(Sm4.encryption(message ^ res[-1]))
    write_file(target_path, res)
```

图 25: CBC+padding 加密

对于 CBC 模式的解密，每一次解密函数的输入都是当前密文与上一次解密所得明文的异或值。与之前一样，我们对于 padding 方式填充的 0 放在 *write_file* 中处理。同时注意到我们使用了 enumerate 方法，我们的 massage 从第二个分组开始，这是因为我们将 IV 放在第一个分组中。

```python
def DE_CBC_PAD(file_path, target_path, keys):
    num_keys = str2num(keys)
    messages, remainder = read_file(file_path)
    Sm4 = sm4(num_keys)
    res = []
    for i, message in enumerate(messages[1:]):
        res.append(Sm4.decryption(message) ^ messages[i])
    write_file(target_path, res, True)
```

图 26: CBC+padding 解密

3.2.4 CBC 模式 + 密文挪用技术

这种工作模式的具体细节几乎是 ii 和 iii 的结合,需要注意的是解密的读写文件需要使用 *read_file_STE* 和 *write_file_STE*。下面直接给出其加密函数和解密函数的实现。

```python
def EN_CBC_STE(file_path, target_path, keys):
    num_keys = str2num(keys)
    messages, remainder = read_file(file_path)
    Sm4 = sm4(num_keys)
    IV = random.randint(0, 2 ** 128 - 1)
    res = [IV]
    for message in messages[:-1]:
        res.append(Sm4.encryption(message ^ res[-1]))
    if remainder == 0:
        res.append(Sm4.encryption(messages[-1] ^ res[-1]))
    else:
        inv_remainder = 16 - remainder
        temp = res[-1]
        temp_high = temp >> (inv_remainder * 8)
        temp_low = (temp & (2 ** (inv_remainder * 8) - 1))
        res[-1] = temp_high
        temp_low = temp_low << (remainder * 8)
        ls_c = Sm4.encryption(temp_low ^ messages[-1])
        res.append(ls_c)
    write_file_STE(target_path, res, remainder)
```

图 27: CBC+STE 加密

```python
def DE_CBC_STE(file_path, target_path, keys):
    num_keys = str2num(keys)
    messages, remainder = read_file_STE(file_path)
    Sm4 = sm4(num_keys)
    res = []
    for i, message in enumerate(messages[1:-2]):
        res.append(Sm4.decryption(message) ^ messages[i])
    temp = Sm4.decryption(messages[-1])
    inv_remainder = 16 - remainder
    temp_high = temp >> (remainder * 8)
    temp_low = (temp & (2 ** (remainder * 8) - 1))
    temp = messages[-2]
    temp = (temp << (inv_remainder * 8)) ^ temp_high
    temp = Sm4.decryption(temp) ^ messages[-3]
    res.append(temp)
    res.append(temp_low)
    write_file(target_path, res, True)
```

图 28: CBC+STE 解密

3.3 用户交互设计

3.3.1 UI 设计

3.3.2 接口设计

四、 实验结果

五、 实验总结

魏彦豪：

赵伯侯：

吕岚曦：