武汉大学国家网络安全学院

密 码 学 实 验 报 告

学　　号　2021302181156

姓　　名　赵伯侯

实验名称　分组密码工作模式

指导教师　何琨

一、 实验名称: 分组密码工作模式

二、 实验目的及要求:

2.1 实验目的

(1) 掌握分组密码的基本概念

(2) 掌握 DES、AES、SMS4 密码算法

(3) 了解分组密码 DES、AES、SMS4 的安全性

(4) 掌握分组密码常用工作模式及其特点

(5) 熟悉分组密码的应用

2.2 实验要求

(1) 掌握分组密码的 ECB、CBC、OFB、CFB、CTR 等常用工作模式

(2) 掌握分组密码的短块加密技术

(3) 熟悉分组密码各工作模式的（数据掩盖、错误传播、效率等）特点

(4) 利用分组密码工作模式和短块处理技术实现任意长度输入的加密与解密

三、 实验设备环境及要求:

Windows 操作系统，python 高级语言开发环境

四、 实验内容与步骤:

4.1 分组密码的常用工作模式

4.1.1 电码本模式 ECB

电码本模式是利用分组密码对明文的各个分组进行加密，

其加密公式为:$C_i = E(M_i, K), i = 1, 2, \ldots, n$

解密公式为：$M_i = D(C_i, K), i = 1, 2, \ldots, n$

该模式的加解密流程如下图所示

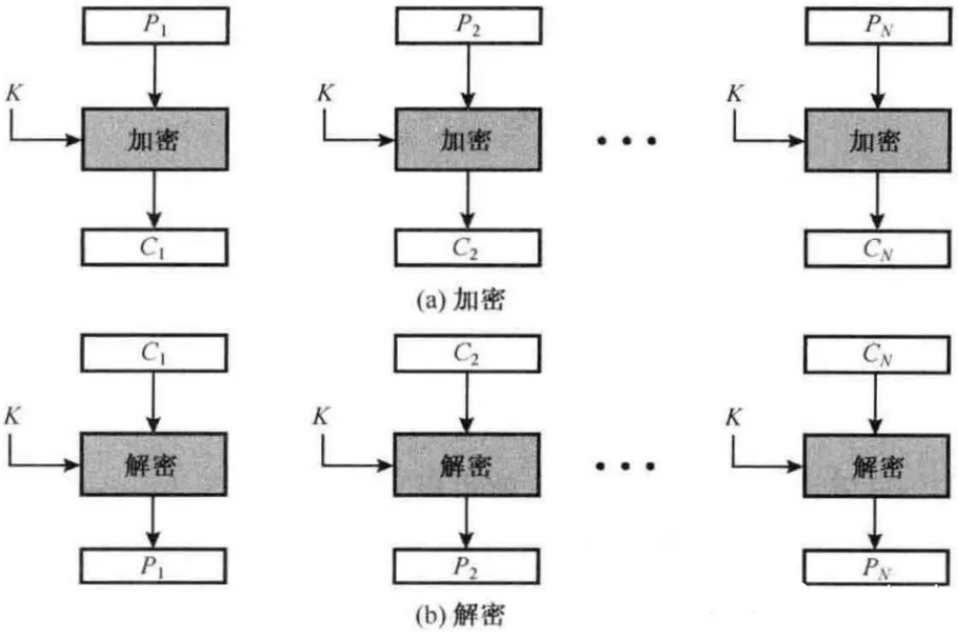

图 1: ECB 加解密流程

在本次实验中的 ECB 实现代码如下所示

```
1 from codes.cryptography.lab_3.AES_encode import AES
2 from codes.cryptography.lab_3.AES_encode import MATH
3 import os
4
5
6 def divide(msgs):
```

```
    """分割函数将输入每32个分割成一个列表"""
    part_msgs = []
    msg = ""
    for index in range(len(msgs)):
        msg += msgs[index]
        if (index + 1) % 32 == 0:
            part_msgs.append(msg)
            msg = ""
    if len(msg) > 0:
        print("出现短块，报错")
    return part_msgs


def EBC(key, msg):
    messages = divide(msg)
    secrets = []
    for message in messages:
        secret = AES.encode(key, message)
        secrets.append(secret)
    secrets_str = "".join(secrets)
    return secrets_str


def enEBC(key, secret_str):
    secrets = divide(secret_str)
    Ms = []
    for secret in secrets:
        M = AES.decode(key, secret)
        Ms.append(M)
```

```
36      message_str = " ".join(Ms)
37      return message_str
38
39
40  if __name__ == '__main__':
41      # 设置工作目录为脚本所在目录
42      script_dir = os.path.dirname(__file__)
43      os.chdir("E:\\Python_code\\codes\\cryptography\\lab_3\\work_style\\ECB")
44
45      key = AES.read_file("key.txt")
46      msg = AES.read_file("message.txt")
47
48      secret_str = EBC(key, msg)
49      print("加密前的明文为:" + msg)
50      print("加密后的密文为: " + secret_str)
51
52      # 解密
53      M = enEBC(key, secret_str)
54      print("解密后的明文为: " + M)
55      # print(len(msg))
```

代码 1: ECB

4.1.2 密文链接模式 CBC

密文分组链接模式（Cipher Block Chaining，CBC）中，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，需先与初始向量异或，再进入加密算法进行加密。

CBC 模式的加密公式为：

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

CBC 模式解密公式为：

$$M_i = \begin{cases} D(C_i, K) \oplus Z, i = 1 \\ D(C_i, K) \oplus C_{i-1}, i = 2, \ldots, n \end{cases}$$

加解密过程流程图为：

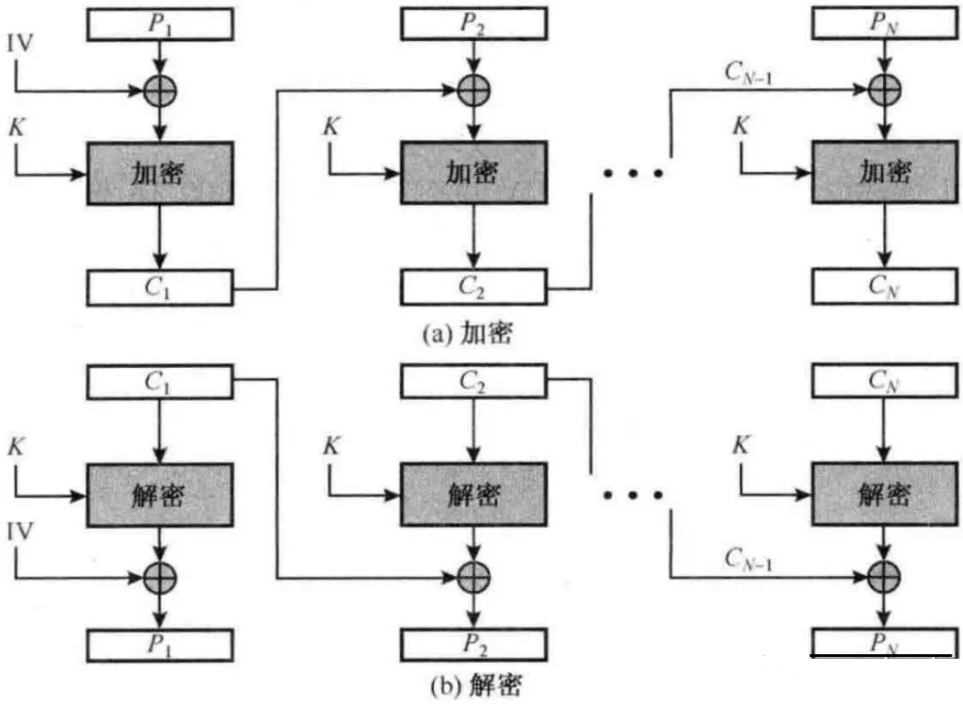

图 2: CBC 加解密流程

在本次实验中的 CBC 实现代码如下所示

```
from codes.cryptography.lab_3.AES_encode import AES
from codes.cryptography.lab_3.AES_encode import MATH
import random
import os



def divide(msgs):
    """分割函数将输入每32个分割成一个列表"""
    part_msgs = []
    msg = ""
    for index in range(len(msgs)):
        msg += msgs[index]
        if (index + 1) % 32 == 0:
            part_msgs.append(msg)
            msg = ""
    if len(msg) > 0:
        print("出现短块，报错")
    return part_msgs



def CBC(key, msg, Z):
    """加密算法"""
    Ms = divide(msg)
    Cs = []

    E_in = MATH.xor(Ms[0], Z)
    Cs.append(AES.encode(key, E_in))

```

```
29      for i in range(1, len(Ms)):
30          E_in = MATH.xor(Ms[i], Cs[i - 1])
31          Cs.append(AES.encode(key, E_in))
32
33      secrets_str = "".join(Cs)
34
35      return secrets_str
36
37
38  def enCBC(key, secret_str, Z):
39      """解密算法"""
40      secrets = divide(secret_str)
41      Ms = []
42
43      D_out = AES.decode(key, secrets[0])
44      Ms.append(MATH.xor(D_out, Z))
45
46      for i in range(1, len(secrets)):
47          D_out = AES.decode(key, secrets[i])
48          Ms.append(MATH.xor(D_out, secrets[i - 1]))
49
50      message_str = " ".join(Ms)
51      return message_str
52
53
54  if __name__ == '__main__':
55      # 设置工作目录为脚本所在目录
56      script_dir = os.path.dirname(__file__)
57      os.chdir("E:\\Python_code\\codes\\cryptography\\lab_3\\work_style\\CBC")
```

```
58
59      key = AES.read_file("key.txt")
60      msg = AES.read_file("message.txt")
61      characters = '0123456789abcdef'
62
63      # 加密
64      Z = ''.join(random.choice(characters) for _ in range(32))
65      secret_str = CBC(key, msg, Z)
66      print("加密前的明文为:" + msg)
67      print("CBC加密后的密文为:" + secret_str)
68
69      # 解密
70      M = enCBC(key, secret_str, Z)
71      print("解密后的明文为: " + M)
72      # print(len(msg))
```

代码 2: CBC

4.1.3 输出反馈模式 OFB

输出反馈模式（Output Feedback，OFB）也是一种类似于流密码的工作模式。在这种模式中，明文分组同样没有进入到加密算法中，加密算法只是用来计算密钥流的。直接将加密算法的输出反馈到下一分组加密算法的输入中。

其加解密算法流程图为：

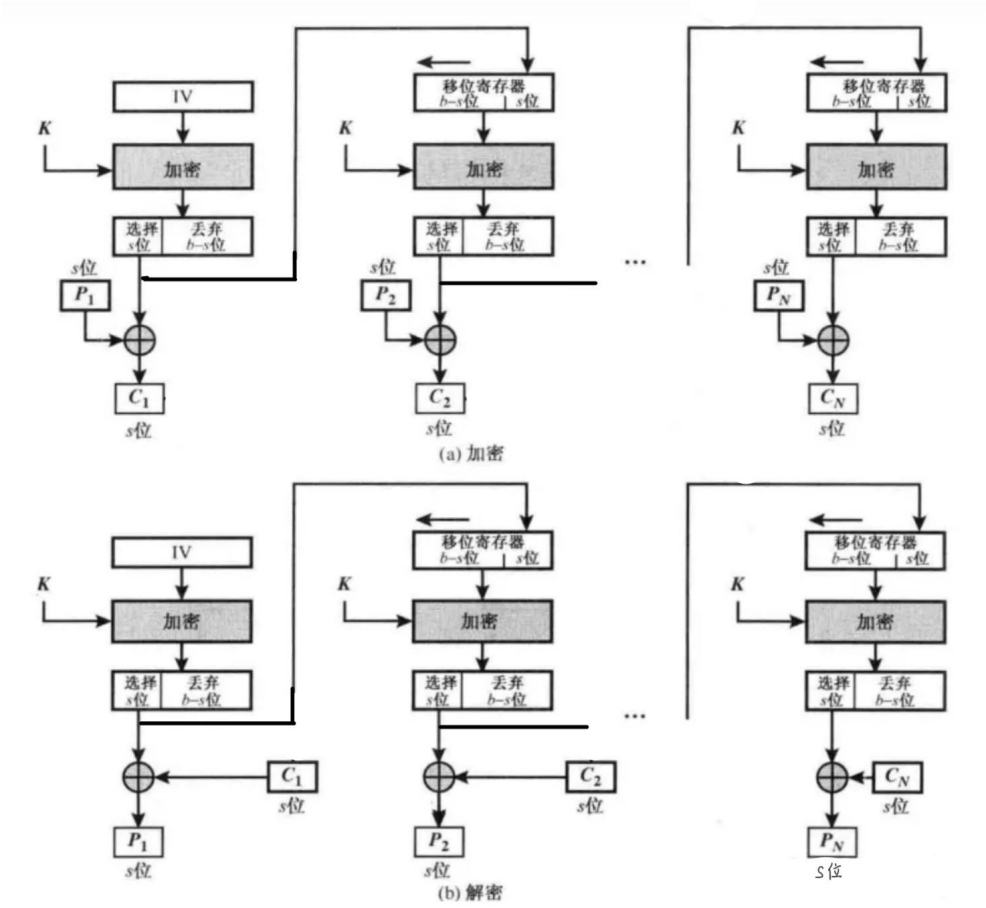

图 3: OFB 加解密流程

在本次实验中的 OFB 实现代码如下所示

```
1 from codes.cryptography.lab_3.AES_encode import AES
2 from codes.cryptography.lab_3.AES_encode import MATH
3 import random
4 import os
5
```

```
6
7  def divide(msgs):
8      """分割函数将输入每16个分割成一个列表"""
9      part_msgs = []
10     msg = ""
11     for index in range(len(msgs)):
12         msg += msgs[index]
13         if (index + 1) % 16 == 0:
14             part_msgs.append(msg)
15             msg = ""
16     if len(msg) > 0:
17         part_msgs.append(msg)
18     return part_msgs
19
20
21 def OFB(key, msg, I):
22     messages = divide(msg)
23     secrets = []
24     E_in = I
25     for message in messages:
26         length = len(message)
27         E_out = AES.encode(key, E_in)
28         E_out_ = E_out[0:length]
29         secret = MATH.xor(E_out_, message)
30         secrets.append(secret)
31         E_in = E_out
32     secrets_str = "".join(secrets)
33     return secrets_str
34
```

```
35
36 def enOFB(key, secret_str, I):
37     secrets = divide(secret_str)
38     Ms = []
39     D_in = I
40     for secret in secrets:
41         length = len(secret)
42         D_out = AES.encode(key, D_in)
43         D_out_ = D_out[0:length]
44         M = MATH.xor(D_out_, secret)
45         Ms.append(M)
46         D_in = D_out
47     message_str = " ".join(Ms)
48     return message_str
49
50
51 if __name__ == '__main__':
52     # 设置工作目录为脚本所在目录
53     script_dir = os.path.dirname(__file__)
54     os.chdir("E:\\Python_code\\codes\\cryptography\\lab_3\\work_style\\OFB")
55
56     key = AES.read_file("key.txt")
57     msg = AES.read_file("message.txt")
58     characters = '0123456789abcdef'
59
60     # 加密
61     I = ''.join(random.choice(characters) for _ in range(32))
62     secret_str = OFB(key, msg, I)
63     print("加密前的明文为:" + msg)
```

```
64      print("正确加密后的密文：" + secret_str)
65
66      # 解密
67      M = enOFB(key, secret_str, I)
68      print("解密后的明文为：" + M)
69
70      # print(len(msg))
```

代码 3: OFB

4.1.4 密文反馈模式 CFB

在 CFB（Cipher Feedback，CFB）模式下，明文本身并没有进入加密算法中进行加密，而是与加密函数的输出进行了异或，得到了密文。其加解密算法流程图为：

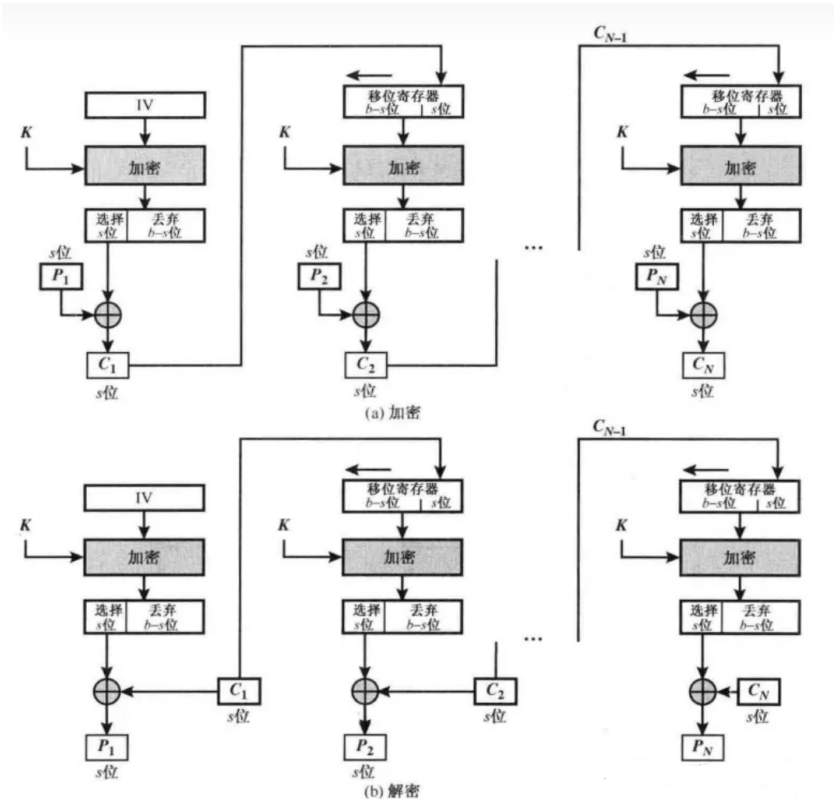

图 4: CFB 加解密流程

在本次实验中的 CFB 的实现代码如下所示

```
from codes.cryptography.lab_3.AES_encode import AES
from codes.cryptography.lab_3.AES_encode import MATH
import random
import os



def divide(msgs):
    """分割函数将输入每16个分割成一个列表"""
    part_msgs = []
    msg = ""
    for index in range(len(msgs)):
        msg += msgs[index]
        if (index + 1) % 16 == 0:
            part_msgs.append(msg)
            msg = ""
    if len(msg) > 0:
        part_msgs.append(msg)
    return part_msgs



def CFB(key, msg, I):
    messages = divide(msg)
    secrets = []
    E_in = I
    for message in messages:
        length = len(message)
        E_out = AES.encode(key, E_in)
        E_out_ = E_out[:length]
```

```
        secret = MATH.xor(E_out_, message)
        secrets.append(secret)
        E_in = (E_in[len(I) - length:] + secret)
    secrets_str = "".join(secrets)
    return secrets_str


def enCFB(key, secret_str, I):
    secrets = divide(secret_str)
    Ms = []
    D_in = I
    for secret in secrets:
        length = len(secret)
        D_out = AES.encode(key, D_in)
        D_out_ = D_out[:length]
        M = MATH.xor(D_out_, secret)
        Ms.append(M)
        D_in = D_in[len(I) - length:] + secret
    message_str = " ".join(Ms)
    return message_str


if __name__ == '__main__':
    # 设置工作目录为脚本所在目录
    script_dir = os.path.dirname(__file__)
    os.chdir("E:\\Python_code\\codes\\cryptography\\lab_3\\work_style\\CFB")

    key = AES.read_file("key.txt")
    msg = AES.read_file("message.txt")
```

```
    characters = '0123456789abcdef'

    # 加密
    I = ''.join(random.choice(characters) for _ in range(32))
    secret_str = CFB(key, msg, I)
    print("加密前的明文为:" + msg)
    print("正确加密后的密文: " + secret_str)

    # 解密
    M = enCFB(key, secret_str, I)
    print("解密后的明文为: " + M)

    # print(len(msg))
```

代码 4: CFB

4.1.5 X CBC 模式

XCBC 模式可以处理任意长度的数据，在处理最后一个明文块时，若其不是标准块则需要将其末尾填充字符串‘1’和若干个‘0’并且在加密时需要先将填充后的明文块与 C_{i-1} 和 K_3 进行异或操作后再进行加密，若最后一个明文块没有进行填充则需要将其与 C_{i-1} 和 K_2 进行异或操作后再进行加密

XCBC 的加密算法公式为：

$$C_i = \begin{cases} E(M_i \oplus Z, K_1), i = 1 \\ E(M_i \oplus C_{i-1}, K_1), i = 2, 3, \ldots n-1 \\ E(M_n \oplus C_{n-1} \oplus K_2, K_1), i = n, andlen(M_n) = 32 \\ E(Pad(M_n) \oplus C_{n-1} \oplus K_3, K_1), i = n, andlen(M_n) < 32 \end{cases}$$

XCBC 的解密算法公式为：

$$M_i = \begin{cases} D(C_i, K_1) \oplus Z, i = 1 \\ D(C_i, K_1) \oplus C_{i-1}, i = 2, 3, \ldots n-1 \\ D(C_n, K_1) \oplus C_{n-1} \oplus K_2, i = n, andlen(M_n) = 32 \\ dePad(D(C_n, K_1) \oplus C_{n-1} \oplus K_3), i = n, andlen(M_n) < 32 \end{cases}$$

XCBC 加密算法的流程图如下图所示

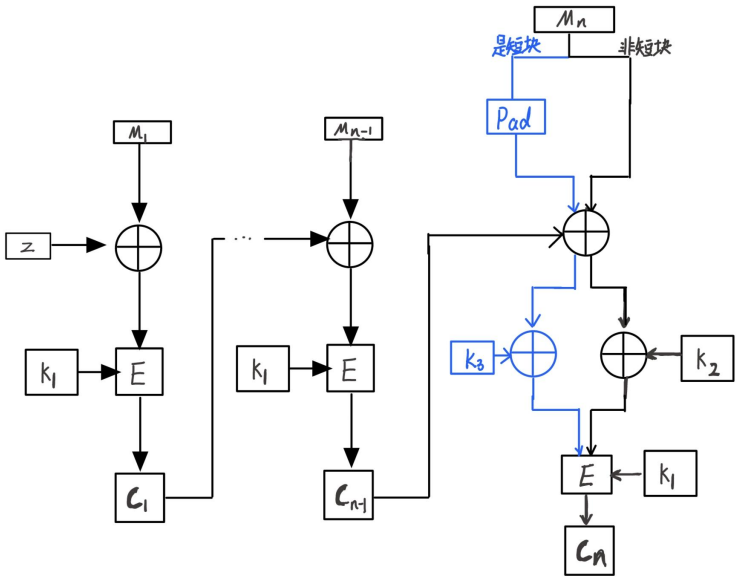

图 5: XCBC 加密流程

XCBC 解密算法的流程图与加密算法相似，在此不做赘述。

在本次实验中 XCBC 的实现代码如下所示

```
from codes.cryptography.lab_3.AES_encode import AES
from codes.cryptography.lab_3.AES_encode import MATH
import random
import os


def divide(msgs):
    """分割函数将输入每32个分割成一个列表"""
    part_msgs = []
```

```
    msg = ""
    for index in range(len(msgs)):
        msg += msgs[index]
        if (index + 1) % 32 == 0:
            part_msgs.append(msg)
            msg = ""
    if len(msg) > 0:
        part_msgs.append(msg)
    return part_msgs


def Pad(msg):
    pad = 1
    msg = msg + '1'
    while len(msg) < 32:
        pad += 1
        msg = msg + '0'
    return pad, msg


def dePad(msg, pad):
    msg = msg[:-pad]
    return msg


def XCBC(key, msg, Z):
    Ms = divide(msg)
    Cs = []
    E_in = MATH.xor(Ms[0], Z)
```

```
39      Cs.append(AES.encode(key[0], E_in))
40      pad = 0
41      for i in range(1, len(Ms) - 1):
42          E_in = MATH.xor(Ms[i], Cs[i - 1])
43          Cs.append(AES.encode(key[0], E_in))
44      if len(Ms[-1]) < 32:
45          pad, M_p = Pad(Ms[-1])
46          E_in = MATH.xor(M_p, Cs[-1])
47          E_in = MATH.xor(E_in, key[2])
48          Cs.append(AES.encode(key[0], E_in))
49      else:
50          E_in = MATH.xor(Ms[-1], Cs[-1])
51          E_in = MATH.xor(E_in, key[1])
52          Cs.append(AES.encode(key[0], E_in))
53
54      secrets_str = "".join(Cs)
55
56      return pad, secrets_str
57
58
59  def enXCBC(key, secret_str, Z, pad):
60      secrets = divide(secret_str)
61      Ms = []
62      D_out = AES.decode(key[0], secrets[0])
63      Ms.append(MATH.xor(D_out, Z))
64      for i in range(1, len(secrets) - 1):
65          D_out = AES.decode(key[0], secrets[i])
66          Ms.append(MATH.xor(D_out, secrets[i - 1]))
67      if pad > 0:
```

```
68          D_out = AES.decode(key[0], secrets[-1])
69          M_P = MATH.xor(D_out, key[2])
70          M_P = MATH.xor(M_P, secrets[-2])
71          M = dePad(M_P, pad)
72          Ms.append(M)
73      else:
74          D_out = AES.decode(key[0], secrets[-1])
75          M = MATH.xor(D_out, key[1])
76          M = MATH.xor(M, secrets[-2])
77          Ms.append(M)
78
79      message_str = " ".join(Ms)
80      return message_str
81
82
83  if __name__ == '__main__':
84      # 设置工作目录为脚本所在目录
85      script_dir = os.path.dirname(__file__)
86      os.chdir("E:\\Python_code\\codes\\cryptography\\lab_3\\work_style\\XCBC")
87
88      keys = AES.read_file("keys.txt")
89      keys = keys.split()
90      msg = AES.read_file("message.txt")
91      characters = '0123456789abcdef'
92
93      # 加密
94      Z = ''.join(random.choice(characters) for _ in range(32))
95      pad, secret_str = XCBC(keys, msg, Z)
96      print("加密前的明文为:" + msg)
```

```
97      print("XCBC加密后的密文为:" + secret_str)
98
99      # 解密
100     M = enXCBC(keys, secret_str, Z, pad)
101     print("解密后的明文为: " + M)
102     # print(len(msg))
```

代码 5: XCBC

4.1.6 计数器模式 CTR

计数器模式（counter，CTR）也是一种类似于流密码的模式。加密算法只是用来产生密钥流与明文分组异或。区别在于每一次将计数器的值进行加一再进行加密操作。CTR 模式的加密公式为：

$$\begin{cases} O_i = E(T_i, K), i = 1, 2, \ldots, n \\ C_i = M_i \oplus O_i, i = 1, 2, \ldots, n-1 \\ C_n = M_n \oplus MSB_n(O_n) \end{cases}$$

CTR 模式的解密公式为：

$$\begin{cases} O_i = E(T_i, K), i = 1, 2, \ldots, n \\ M_i = C_i \oplus O_i, i = 1, 2, \ldots, n-1 \\ M_n = C_n \oplus MSB_n(O_n) \end{cases}$$

CTR 模式的加密解密算法流程图如下图所示

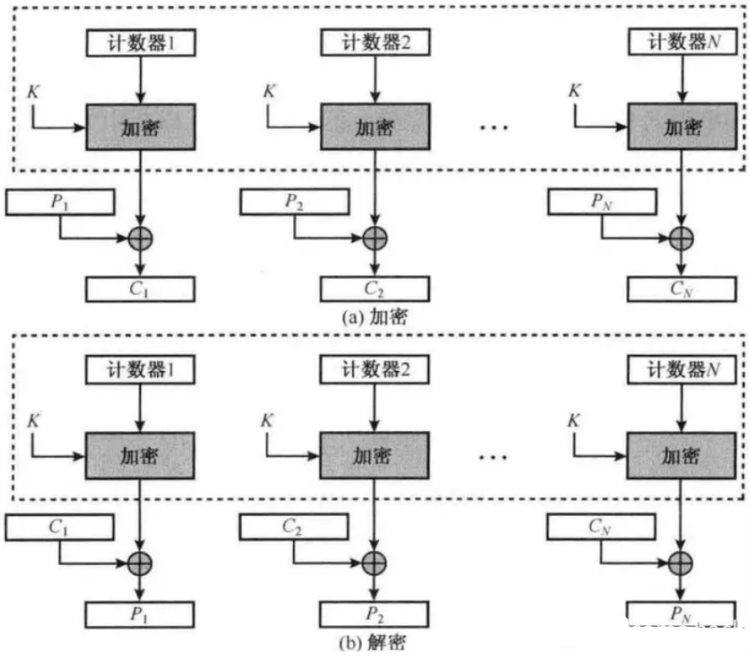

图 6: CTR 加解密流程

在本次实验中 CTR 的实现代码如下所示

```
from codes.cryptography.lab_3.AES_encode import AES
from codes.cryptography.lab_3.AES_encode import MATH
import random
import os



def divide(msgs):
    """分割函数将输入每16个分割成一个列表"""
    part_msgs = []
    msg = ""
    for index in range(len(msgs)):
        msg += msgs[index]
        if (index + 1) % 32 == 0:
            part_msgs.append(msg)
```

```
            msg = ""
    if len(msg) > 0:
        part_msgs.append(msg)
    return part_msgs


def CTR(key, msg, T):
    messages = divide(msg)
    secrets = []
    E_in = T
    for message in messages:
        length = len(message)
        E_out = AES.encode(key, E_in)
        E_out_ = E_out[:length]
        secret = MATH.xor(E_out_, message)
        secrets.append(secret)
        E_in = MATH.xor(E_in, '1')
    secrets_str = "".join(secrets)
    return secrets_str


def enCTR(key, secret_str, T):
    secrets = divide(secret_str)
    Ms = []
    D_in = T
    for secret in secrets:
        length = len(secret)
        D_out = AES.encode(key, D_in)
        D_out_ = D_out[:length]
```

```
44          M = MATH.xor(D_out_, secret)
45          Ms.append(M)
46          D_in = MATH.xor(D_in, '1')
47      message_str = " ".join(Ms)
48      return message_str
49
50
51  if __name__ == '__main__':
52      # 设置工作目录为脚本所在目录
53      script_dir = os.path.dirname(__file__)
54      os.chdir("E:\\Python_code\\codes\\cryptography\\lab_3\\work_style\\CTR")
55
56      key = AES.read_file("key.txt")
57      msg = AES.read_file("message.txt")
58      characters = '0123456789abcdef'
59      # 加密
60      T = ''.join(random.choice(characters) for _ in range(32))
61      secret_str = CTR(key, msg, T)
62      print("加密前的明文为:" + msg)
63      print("加密后的密文: " + secret_str)
64
65      # 解密
66      M = enCTR(key, secret_str, T)
67      print("解密后的明文为: " + M)
68      # print(len(msg))
```

代码 6: CTR

4.1.7 明密文链接模式 MCBC

明密文链接模式是在 CBC 的基础上将交付给下一个加密迭代的 C_{n-1} 与明文异或之后再进行之后的操作其加密公式为：

$$C_i = \begin{cases} E(M_i \oplus Z, K), i = 1 \\ E(M_i \oplus M_{i-1} \oplus C_{i-1}, K), i = 2, 3, \ldots, n \end{cases}$$

其解密公式为：

$$M_i = \begin{cases} D(C_i, K) \oplus Z, i = 1 \\ D(C_i, K) \oplus M_{i-1} \oplus C_{i-1}, i = 2, 3, \ldots, n \end{cases}$$

MCBC 算法流程图如下图所示

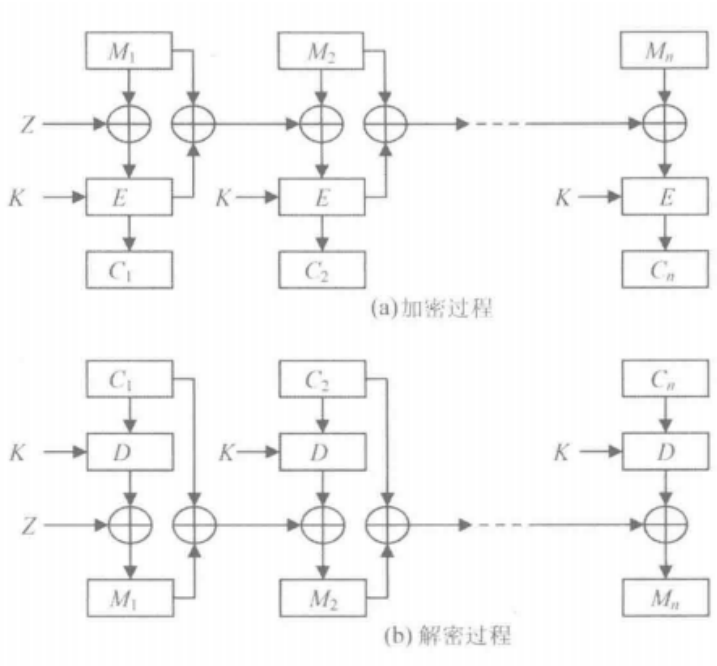

图 7: MCBC 加解密流程

在本次实验中 MCBC 的实现代码如下所示

```
1 from codes.cryptography.lab_3.AES_encode import AES
2 from codes.cryptography.lab_3.AES_encode import MATH
3 import random
4 import os
5
```

```
6
7  def divide(msgs):
8      """分割函数将输入每32个分割成一个列表"""
9      part_msgs = []
10     msg = ""
11     for index in range(len(msgs)):
12         msg += msgs[index]
13         if (index + 1) % 32 == 0:
14             part_msgs.append(msg)
15             msg = ""
16     if len(msg) > 0:
17         print("出现短块，报错")
18     return part_msgs
19
20
21 def MCBC(key, msg, Z):
22     Ms = divide(msg)
23     Cs = []
24     E_in = MATH.xor(Ms[0], Z)
25     Cs.append(AES.encode(key, E_in))
26     for i in range(1, len(Ms)):
27         E_M = MATH.xor(Cs[i - 1], Ms[i - 1])
28         E_in = MATH.xor(Ms[i], E_M)
29
30         Cs.append(AES.encode(key, E_in))
31
32     secrets_str = "".join(Cs)
33
34     return secrets_str
```

```
35
36
37  def enMCBC(key, secret_str, Z):
38      secrets = divide(secret_str)
39      Ms = []
40      D_out = AES.decode(key, secrets[0])
41      Ms.append(MATH.xor(D_out, Z))
42      for i in range(1, len(secrets)):
43          D_M = MATH.xor(secrets[i - 1], Ms[i - 1])
44          D_out = AES.decode(key, secrets[i])
45          Ms.append(MATH.xor(D_out, D_M))
46      message_str = " ".join(Ms)
47      return message_str
48
49
50  if __name__ == '__main__':
51      # 设置工作目录为脚本所在目录
52      script_dir = os.path.dirname(__file__)
53      os.chdir("E:\\Python_code\\codes\\cryptography\\lab_3\\work_style\\MCBC")
54
55      key = AES.read_file("key.txt")
56      msg = AES.read_file("message.txt")
57      characters = '0123456789abcdef'
58
59      # 加密
60      Z = ''.join(random.choice(characters) for _ in range(32))
61      secret_str = MCBC(key, msg, Z)
62      print("加密前的明文为:" + msg)
63      print("MCBC加密后的密文为:" + secret_str)
```

```
64
65      # 解密
66      M = enMCBC(key, secret_str, Z)
67      print("解密后的明文为：" + M)
68
69      # print(len(msg))
```

代码 7: MCBC

4.2 分组密码的短块处理技术

4.2.1 填充法

在本次实验中 XCBC 工作模式使用填充法来处理短块。主要原理是将短块补充成标准快所需要的长度，补充的字符为‘1’以及若干个‘0’在解密时需要将该补充的字符进行消除。实现这一步骤的代码如下所示

```
1  def Pad(msg):
2      pad = 1
3      msg = msg + '1'
4      while len(msg) < 32:
5          pad += 1
6          msg = msg + '0'
7      return pad, msg
8
9
10 def dePad(msg, pad):
11     msg = msg[:-pad]
12     return msg
```

代码 8: 填充法实现代码

4.2.2 序列密码加密法

序列密码加密法是混合使用分组密码和序列密码两种技术的方案，其中对于标准块用分组密码加密，而对于短块用序列密码加密

序列密码的加密公式为：$C_n = M_n \oplus MSB_u(E(C_{n-1}, K))$

序列密码的加解密流程图如下图所示

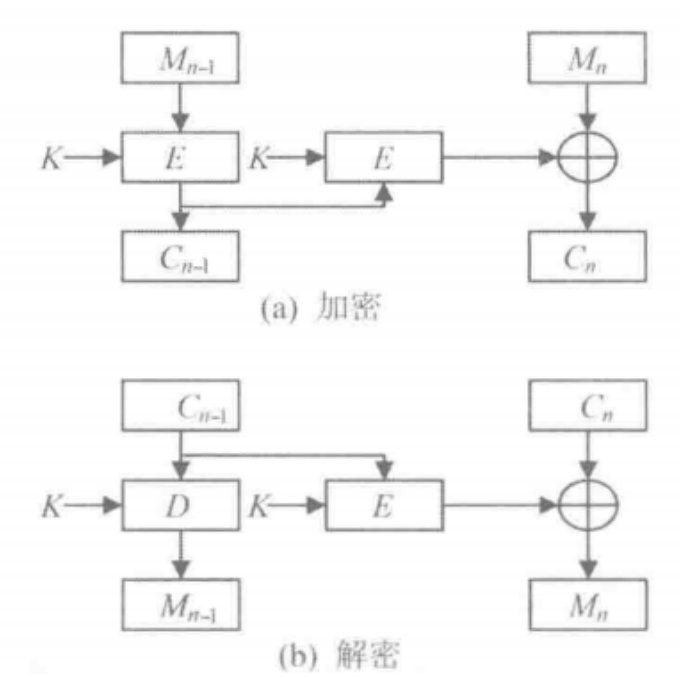

图 8: 序列密码加密法

在本次实验中实现序列密码加解密的代码如下所示

```
from codes.cryptography.lab_3.AES_encode import AES
from codes.cryptography.lab_3.AES_encode import MATH



def divide(msgs):
    """分割函数将输入每32个分割成一个列表"""
    part_msgs = []
    msg = ""
```

```
    for index in range(len(msgs)):
        msg += msgs[index]
        if (index + 1) % 32 == 0:
            part_msgs.append(msg)
            msg = ""
    if len(msg) > 0:
        part_msgs.append(msg)
    return part_msgs


def xulie(key, msg):
    messages = divide(msg)
    secrets = []
    for message in messages:
        if len(message) < 32:
            X_in = AES.encode(key, secrets[-1])
            X_in = X_in[:len(message)]
            X_out=MATH.xor(X_in, message)
            secrets.append(X_out)
        else:
            secret = AES.encode(key, message)
            secrets.append(secret)

    secrets_str = "".join(secrets)
    return secrets_str


def enxulie(key, secret_str):
    secrets = divide(secret_str)
```

```
38      Ms = []
39      for secret in secrets:
40          if len(secret) < 32:
41              X_in = AES.encode(key, secrets[-2])
42              X_in = X_in[:len(secret)]
43              X_out=MATH.xor(X_in, secret)
44              Ms.append(X_out)
45          else:
46              M = AES.decode(key, secret)
47              Ms.append(M)
48
49      message_str = "".join(Ms)
50      return message_str
51
52
53  if __name__ == '__main__':
54      # 加密
55      key = AES.read_file("key.txt")
56      msg = AES.read_file("message.txt")
57      secret_str = xulie(key, msg)
58      print("加密后的密文:")
59      print(secret_str)
60
61      # 解密
62      M = enxulie(key, secret_str)
63      print("解密后的明文为:")
64      print(M)
```

代码 9: 序列密码加解密

4.2.3 密文挪用技术

密文挪用技术中，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，然后将其填充到 M_n 中去使得 M_n 成为一个标准块，这样 C_{n-1} 却成为了短块，然后再对填充后的 M_n 进行加密操作，得到密文 C_n。在解密时先对 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}

密文挪用技术算法流程图如下图所示

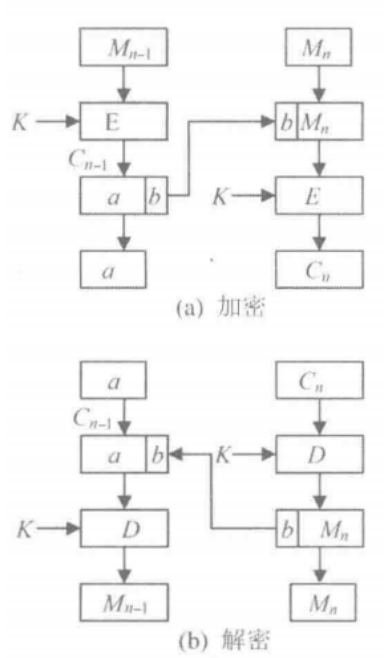

图 9: 密文挪用技术

在本次实验中的实现代码如下所示

```
1  from codes.cryptography.lab_3.AES_encode import AES
2
3
4  def divide(msgs):
5      """分割函数将输入每32个分割成一个列表"""
6      part_msgs = []
```

```
    msg = ""
    for index in range(len(msgs)):
        msg += msgs[index]
        if (index + 1) % 32 == 0:
            part_msgs.append(msg)
            msg = ""
    if len(msg) > 0:
        part_msgs.append(msg)
    return part_msgs


def divide_c(msgs):
    part_c = []
    part = ""
    final_part = msgs[len(msgs) - 32:]
    msgs = msgs[:len(msgs) - 32]
    for index in range(len(msgs)):
        part += msgs[index]
        if (index + 1) % 32 == 0:
            part_c.append(part)
            part = ""
    if len(part) > 0:
        part_c.append(part)
    part_c.append(final_part)

    return part_c


def nuoyong(key, msg):
```

```
    messages = divide(msg)
    secrets = []
    length = 0
    for message in messages:
        if len(message) < 32:
            length = len(message)
            a = secrets[-1][:length]
            b = secrets[-1][length:]
            secrets[-1] = a
            b_M = b + message
            E_out = AES.encode(key, b_M)
            secrets.append(E_out)

        else:
            secret = AES.encode(key, message)
            secrets.append(secret)

    secrets_str = "".join(secrets)
    return length, secrets_str


def ennuoyong(key, secret_str, length):
    secrets = divide_c(secret_str)
    Ms = []
    for secret in secrets:
        if len(secret) < 32:
            Dn_out = AES.decode(key, secrets[-1])
            Mn = Dn_out[32-length:]
            b = Dn_out[:32-length]
```

```
65              D_in = secret + b
66              M = AES.decode(key, D_in)
67              Ms.append(M)
68              Ms.append(Mn)
69              break
70
71          else:
72              M = AES.decode(key, secret)
73              Ms.append(M)
74
75      message_str = "".join(Ms)
76      return message_str
77
78
79  if __name__ == '__main__':
80      # 加密
81      key = AES.read_file("key.txt")
82      msg = AES.read_file("message.txt")
83      length, secret_str = nuoyong(key, msg)
84      print("加密后的密文:")
85      print(secret_str)
86
87      # 解密
88      M = ennuoyong(key, secret_str, length)
89      print("解密后的明文为:")
90      print(M)
```

代码 10: 密文挪用

4.3 各工作模式的特点比较

4.3.1 各工作模式是否能够掩盖明文中的数据模式

对于每一种工作模式都选定明文为‘0001000101a198afda78173486153566 0001000101a198afda78173486153566aaa’将加密两次得到的结果 C1,C2 进行比较

（1）ECB

运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\ECB\ECB.py
两次加密使用的明文为:0001000101a198afda781734861535660001000101a198afda78173486153566aaa
第一次加密后的密文：
6cdd596b8f5642cbd23b47981a65422a6cdd596b8f5642cbd23b47981a65422a5b48f25d66a396ba556a37a49fb477c0
第二次加密后的密文：
6cdd596b8f5642cbd23b47981a65422a6cdd596b8f5642cbd23b47981a65422a5b48f25d66a396ba556a37a49fb477c0

Process finished with exit code 0
```

图 10: ECB 相同明文加密两次结果

观察结果发现，两次加密得到的结果相同，所以 ECB 不能够掩盖明文中的数据模式

（2）MCBC

运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\MCBC\MCBC.py
第一次MCBC加密后的密文为:
71bc2264999fa2f7fd3d4e96da8c1068a381e44c5499bb2cc20be36cb74e4983124908ff7d16e97ae6fb3456fd843e72
第二次MCBC加密后的密文为:
9d3c8fa0e763dd8c1dfc24b08c1cd10d404a0e24cc8072aa7bf2f6afd87853d001bbc574783501c459d60026242d36f4

Process finished with exit code 0
```

图 11: MCBC 相同明文加密两次结果

通过观察发现，两次加密得到的结果不相同，所以 MCBC 能够掩盖明文中的数据模式

（3）CBC

运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\CBC\CBC.py
加密过程使用的明文为:0001000101a198afda78173486153566000100010 1a198afda78173486153566aaa
第1次CBC加密后的密文为:
99416114f58c43a41bf233bf55d12fdbca0a3f662e47921360ea96abba575fe898c7939fba27426c2d64fdb60fb91cb4
第2次CBC加密后的密文为:
1a40995cc58dda6982290091d82785d40b1805d6f811f9043e513637953a8571fe934fdff3483a61737f397e02f8eeef

Process finished with exit code 0
```

图 12: CBC 相同明文加密两次结果

通过观察发现，两次加密得到的结果不相同，所以 CBC 能够掩盖明文中的数据模式

（4）OFB

运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\OFB\OFB.py
第1次加密后的密文：
186cf1ae8953eb28ee5539ad06264b0621fc050936e81a2b1f8fb0ce182a7bae5dd
第2次加密后的密文：
80d5145844689 70fd12bca95e0040022332c339145a2f152fe04c024c842b59012e

Process finished with exit code 0
```

图 13: OFB 相同明文加密两次结果

通过观察发现，两次加密得到的结果不相同，所以 OFB 能够掩盖明文中的数据模式

（5）CFB

运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\CFB\CFB.py
第一次加密后的密文：
3a040a435bcb17a647564d4dfb87bdec952c35b049dcd91e8231bfce906e22bd144
第二次加密后的密文，
f695f5290b0339743b1de68ce2f83b1a7ef07b598924841fbec3f4d89e8bc7ae189

Process finished with exit code 0
```

图 14: CFB 相同明文加密两次结果

通过观察发现，两次加密得到的结果不相同，所以 CFB 能够掩盖明文中的数

据模式

（6）XCBC

运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\XCBC\XCBC.py
第一次XCBC加密后的密文为：
fc5b5e8dc445286a7f08e3ea756aaf539463f400ccd1b59bfd85ca6bf92b79d8ae4559144559a0bf24e88559662f19f1
第二次XCBC加密后的密文为：
3407de3955cb6ec1998a36c914c0fc0d2ef29535786a18981d9bb156a0710a7228e2a14937d9161235fd55d1dbbd0a57

Process finished with exit code 0
```

图 15: XCBC 相同明文加密两次结果

通过观察发现，两次加密得到的结果不相同，所以 XCBC 能够掩盖明文中的数据模式

（7）CTR

运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\CTR\CTR.py
第一次加密后的密文：
363137e905cb61aa6de2f4bcd49ca31eb8c58ae8370c108e8a56937033bd62509c9
第二次加密后的密文：
aad3736bfcb3ef83e87014af47371db2cd969a1aa6e459dce6b5198500dcf9f0007

Process finished with exit code 0
```

图 16: CTR 相同明文加密两次结果

通过观察发现，两次加密得到的结果不相同，所以 CTR 能够掩盖明文中的数据模式

4.3.2 各工作模式是否具有加密错误传播无界特性

选择第一次加密的明文为 0001000101a198afda78173486153566000010001 01a198afda78173486153566aaa 第二次加密的明文将第一位的 0 改为 a 然后进行加密，观察两次加密得出的密文进行判断

（1）ECB

运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\ECB\ECB.py
正确加密后的密文：
6cdd596b8f5642cbd23b47981a65422a 6cdd596b8f5642cbd23b47981a65422a 5b48f25d66a396ba556a37a49fb477c0
修改第一位后加密后的密文：
c3fdaa10dd999809e085e5a06166b717 6cdd596b8f5642cbd23b47981a65422a 5b48f25d66a396ba556a37a49fb477c0

Process finished with exit code 0
```

图 17: ECB 修改一位明文两次加密结果

通过观察发现，第一个分组的结果全体出现错误，但是影响范围只涉及到第一个分组，其他分组并未受到干扰，所以 ECB 工作模式具有加密错误传播有界的特性

（2）MCBC

运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\MCBC\MCBC.py
MCBC加密后的密文为:
665741826a3e7f76637e137eb6c18c35 535d2f058f45c226315bfa669f2ba9a8 faeb51a9d05acab9dcdc6681f9e581f7
MCBC加密后的密文为:
8fb4ab548027d1e6a5dcdee313736564 53f5f132991ac48d39d62f73683b2e3c e88082124c87c9b909079aafb31aa0b1

Process finished with exit code 0
```

图 18: MCBC 修改一位明文两次加密结果

通过观察发现，所有分组的加密结果均出现了错误，由此可见 MCBC 具有加密传播的无界性

（3）CBC

运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\CBC\CBC.py
CBC加密后的密文为:
10dbd946b1f05a9bebde61e5bc952062 859e7348be7cc83d2e50ccbb738693ac babcfac93ed79962352e48320dcbca5b
CBC加密后的密文为:
137e88ad72243853dee7083ccbf40f95 e37c23ab814379a060bab386776d4ade 8fb4bba94e19cc35f0d4dfd92eee894b

Process finished with exit code 0
```

图 19: CBC 修改一位明文两次加密结果

通过观察发现，所有分组的加密结果均出现了错误，由此可见 CBC 具有加密传播的无界性

（4）OFB

运行结果如下图所示

图 20: OFB 修改一位明文两次加密结果

通过观察发现，第一个分组的结果全体出现错误，但是影响范围只涉及到第一个分组，其他分组并未受到干扰，所以 OFB 工作模式具有加密错误传播有界的特性

（5）CFB

运行结果如下图所示

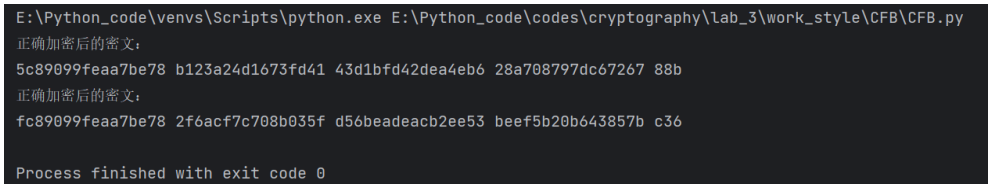

图 21: CFB 修改一位明文两次加密结果

通过观察发现，所有分组的加密结果均出现了错误，由此可见 CFB 具有加密传播的无界性

（6）XCBC

运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\XCBC\XCBC.py
XCBC加密后的密文为:
f9ccbdfeede329fa1ebefcb9a54c34b3 ec12c1978cc9ddb8f60d14f74ab85210 e32aef1e8b18086b91de4a717b8393a5
XCBC加密后的密文为:
2181b49e0c00f5d1b3689613e344b7ec 896fd86083039725eff3a90d5483db61 53881ecc45ca7c47726aebd140d62ed2

Process finished with exit code 0
```

图 22: XCBC 修改一位明文两次加密结果

通过观察发现，所有分组的加密结果均出现了错误，由此可见 XCBC 具有加密传播的无界性

（7）CTR

运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\CTR\CTR.py
加密后的密文：
6aa82107f7f5d25dedbb83b21848d10d 24f6b6432dfe48a323aa82178d6c98d7 c00
加密后的密文：
caa82107f7f5d25dedbb83b21848d10d 24f6b6432dfe48a323aa82178d6c98d7 c00

Process finished with exit code 0
```

图 23: CTR 修改一位明文两次加密结果

通过观察发现，第一个分组的结果全体出现错误，但是影响范围只涉及到第一个分组，其他分组并未受到干扰，所以 CTR 工作模式具有加密错误传播有界的特性

4.3.3 各工作模式是否具有解密错误传播无界特性

将每一次实验中加密后得到的密文的首位与‘1’进行异或操作，即对密文的第一个分块进行修改，查看修改后的解密结果与正常解密结果之间的差异

（1）ECB

运行后的结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\ECB\ECB.py
正确加密后的密文：
6cdd596b8f5642cbd23b47981a65422a6cdd596b8f5642cbd23b47981a65422a5b48f25d66a396ba556a37a49fb477c0
解密后的明文为：
0001000101a198afda78173486153566 0001000101a198afda78173486153566 aaa00000000000000000000000000000
修改密文解密后的明文为：
16f55b7b320c00883f8c5160843a0064 0001000101a198afda78173486153566 aaa00000000000000000000000000000

Process finished with exit code 0
```

图 24: ECB 修改一位密文后两次加密结果

通过观察发现，在将第一块中的密文修改之后进行解密操作后得到的明文仅有第一个分组中出现错误，其余分组中解密出的明文与正常解密的明文相同，所以 ECB 具有解密错误传播有界的特性

（2）MCBC

运行后的结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\MCBC\MCBC.py
MCBC加密后的密文为:
b36f4622f92a0ae6510ccb28beb290f15f54f0cfec8facf941d96dbc1a7e524e08910c8430d630b380add9943d3199f9
解密后的明文为:
0001000101a198afda78173486153566 0001000101a198afda78173486153566 aaa00000000000000000000000000000
修改密文解密后的明文为:
8a8da2f3a7e27fbffb7b3db1e026459d 9a8da2f3a7e27fbffb7b3db1e026459d 302ca2f2a643e71021032a85663370fb

Process finished with exit code 0
```

图 25: MCBC 修改一位密文后两次加密结果

通过观察发现，在将第一块中的密文修改之后进行解密操作后得到的明文中的所有分组均出现了错误，所以 MCBC 具有解密错误传播无界的特性

（3）CBC

运行后的结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\CBC\CBC.py
CBC加密后的密文为:
93a2ef388d399733e8b3bb750bc83018535ce751e96a8eb153135f557a032da2f9b43f337fef01b3fb97c087260f3997
解密后的明文为:
0001000101a198afda78173486153566 0001000101a198afda78173486153566 aaa00000000000000000000000000000
修改密文解密后的明文为:
8cfb4e8cefad7272363ab282654e3d4b 1001000101a198afda78173486153566 aaa00000000000000000000000000000

Process finished with exit code 0
```

图 26: CBC 修改一位密文后两次加密结果

通过观察发现，在将第一块中的密文修改之后进行解密操作后得到的明文仅有第一个分组中出现错误以及第二个分组的首项出现错误，其余分组中解密出的明文与正常解密的明文相同，所以 CBC 具有解密错误传播有界的特性

（4）OFB

运行后的结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\OFB\OFB.py
正确加密后的密文:
62eaaf9f888a43f572123c5d7d412b8fc908baf8f1d5f77745219c1da969c1c67d8
解密后的明文为:
0001000101a198af da78173486153566 0001000101a198af da78173486153566 aaa
修改密文解密后的明文为:
1001000101a198af da78173486153566 0001000101a198af da78173486153566 aaa

Process finished with exit code 0
```

图 27: OFB 修改一位密文后两次加密结果

通过观察发现，在将第一块中的密文修改之后进行解密操作后得到的明文仅有第一个分组中出现错误，其余分组中解密出的明文与正常解密的明文相同，所以 OFB 具有解密错误传播有界的特性

（5）CFB

运行后的结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\CFB\CFB.py
正确加密后的密文：
413482f0e30033e8d0fae7ab2c850f3f7d7ce818784d74a7e6c3312f06dbeb421bf
解密后的明文为：
0001000101a198af da78173486153566 0001000101a198af da78173486153566 aaa
修改密文解密后的明文为：
1001000101a198af de14f2f10b6e4957 512ddff4788a0df4 da78173486153566 aaa

Process finished with exit code 0
```

图 28: CFB 修改一位密文后两次加密结果

通过观察发现，在将第一块中的密文修改之后进行解密操作后得到的明文中的所有分组均出现了错误，所以 CFB 具有解密错误传播无界的特性

（6）XCBC

运行后的结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\XCBC\XCBC.py
XCBC加密后的密文为:
f35aa0154f8df8554ff36d817f67174d76f51813e25b663d38a3fa87fc067b490d0fb15221ef64b183b238243847bec0
解密后的明文为：
0001000101a198afda78173486153566 0001000101a198afda78173486153566 aaa
修改密文解密后的明文为：
7b3e1f401352c94cb5a5c7ee313e31e6 1001000101a198afda78173486153566 aaa

Process finished with exit code 0
```

图 29: XCBC 修改一位密文后两次加密结果

通过观察发现，在将第一块中的密文修改之后进行解密操作后得到的明文仅有第一个分组中出现错误以及第二个分组的首项出现错误，其余分组中解密出的明文与正常解密的明文相同，所以 CBC 具有解密错误传播有界的特性

（7）CTR

运行后的结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\CTR\CTR.py
加密后的密文：
4bac8943f3d22fb41792a9e92983c56be28d75a51f362e8198af708f5d80a67fe10
解密后的明文为：
0001000101a198afda78173486153566 0001000101a198afda78173486153566 aaa
修改密文解密后的明文为：
1001000101a198afda78173486153566 0001000101a198afda78173486153566 aaa
```

图 30: CTR 修改一位密文后两次加密结果

通过观察发现，在将第一块中的密文修改之后进行解密操作后得到的明文仅有第一个分组中出现错误，其余分组中解密出的明文与正常解密的明文相同，所以CTR 具有解密错误传播有界的特性

4.3.4 不同的工作模式对于输入消息长度的要求

在 7 中不同的工作模式中只有 ECB、CBC、明密文链接 CBC 模式要求输入的数据长度是密码分组长度的整数倍，其他工作模式均不对数据的长度做要求。

4.3.5 不同工作模式能否处理短块

在 7 中不同的工作模式中 ECB、CBC、明密文链接 CBC 模式不具备天生的处理短块的能力，其余的工作模式均能够直接处理短块，无需额外的短块处理技术

4.3.6 不同工作模式是否改变消息长度

（1）ECB

ECB 加密的运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\ECB\ECB.py
加密前的明文为:
0001000101a198afda78173486153566001000101a198afda78173486153566
加密后的密文为:
6cdd596b8f5642cbd23b47981a65422a6cdd596b8f5642cbd23b47981a65422a

Process finished with exit code 0
```

图 31: ECB 加密结果

通过观察可以发现 ECB 加密后的密文与加密前的明文位数相同，由此可见ECB 工作模式不会改变信息的长度

（2）MCBC

MCBC 加密的运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\MCBC\MCBC.py
加密前的明文为:
0001000101a198afda78173486153566000100101a198afda78173486153566
MCBC加密后的密文为:
b354a9837595c14a76cb7a0d96e3631a92cfa40d953a74f470483d905d9448d5

Process finished with exit code 0
```

图 32: MCBC 加密结果

通过观察可以发现 MCBC 加密后的密文与加密前的明文位数相同，但是因为输出的值还要包括初始向量 IV 所以密文的长度要长于明文，由此可见 MCBC 工作模式会改变信息的长度

（3）CBC

CBC 加密的运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\CBC\CBC.py
加密前的明文为:
0001000101a198afda78173486153566000100101a198afda78173486153566
CBC加密后的密文为:
33750e33ad1c7ea7790f3ff916ed0e93af2eb9e914df0ef72bebc96c18f104d6

Process finished with exit code 0
```

图 33: CBC 加密结果

通过观察可以发现 CBC 加密后的密文与加密前的明文位数相同，但是因为输出的值还要包括初始向量 IV 所以密文的长度要长于明文，由此可见 CBC 工作模式会改变信息的长度

（4）OFB

OFB 加密的运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\OFB\OFB.py
加密前的明文为:
0001000101a198afda78173486153566000100101a198afda78173486153566aaa
正确加密后的密文：
2efd0551bda1ff43985dba1800d187ce863837a66ad06a2c6790782e9ee70724d9d

Process finished with exit code 0
```

图 34: OFB 加密结果

通过观察可以发现 OFB 加密后的密文与加密前的明文位数相同，但是因为输出的值还要包括初始向量 IV 所以密文的长度要长于明文，由此可见 OFB 工作模式会改变信息的长度

（5）CFB

CFB 加密的运行结果如下图所示

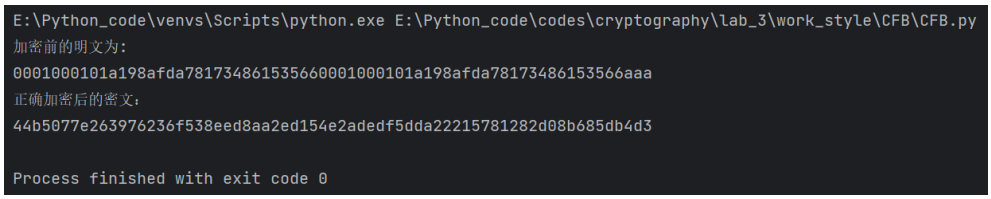

图 35: CFB 加密结果

通过观察可以发现 CFB 加密后的密文与加密前的明文位数相同，但是因为输出的值还要包括初始向量 IV 所以密文的长度要长于明文，由此可见 CFB 工作模式会改变信息的长度

（6）XCBC

XCBC 加密的运行结果如下图所示

图 36: XCBC 加密结果

通过观察可以发现 XCBC 加密后的密文与加密前的明文位数不同，加密后的密文位数增加，由此可见 XCBC 工作模式会改变信息的长度

（7）CTR

CTR 加密的运行结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\CTR\CTR.py
加密前的明文为:
0001000101a198afda78173486153566001000101a198afda78173486153566aaa
加密后的密文:
962efe00009433f37881994070b2fecb1288e2371a8f8a2ec63750e42b4322d93c8

Process finished with exit code 0
```

图 37: CTR 加密结果

通过观察可以发现 CTR 加密后的密文与加密前的明文位数相同，但是因为输出的值还要包括初始向量 IV 所以密文的长度要长于明文，由此可见 CTR 工作模式会改变信息的长度

4.3.7 不同的工作模式的执行效率

将每种工作模式程序中的提示输出注释掉之后统计每个工作模式的工作时间，并用工作时间除加密的比特数即可得知程序运行的效率

在本次实验中使用如下程序统计每一种工作模式下的运行速率并输出。程序代码如下所示

```
1  import subprocess
2  import time
3
4
5  def measure_runtime(script_path):
6      """计算程序运行速度"""
7      start_time = time.time()  # 开始时间
8
9      # 运行外部Python脚本
10     result = subprocess.run(["python", script_path], capture_output=True, text=
           True)
11
```

```
    end_time = time.time()  # 结束时间
    runtime = end_time - start_time  # 计算运行时间

    byte_number = float(result.stdout)
    run_speed = runtime / byte_number
    return run_speed


# 调用函数，传入您想测量的Python脚本的路径
if __name__ == '__main__':
    speed_ECB = measure_runtime("ECB/ECB.py")
    print("ECB的每字节加密速度为：  " + str(speed_ECB)+" s/byte")

    speed_MCBC = measure_runtime("MCBC/MCBC.py")
    print("MCBC的每字节加密速度为：" + str(speed_MCBC)+" s/byte")

    speed_CBC = measure_runtime("CBC/CBC.py")
    print("CBC的每字节加密速度为：  " + str(speed_CBC) + " s/byte")

    speed_OFB = measure_runtime("OFB/OFB.py")
    print("OFB的每字节加密速度为：  " + str(speed_OFB) + " s/byte")

    speed_CFB = measure_runtime("CFB/CFB.py")
    print("CFB的每字节加密速度为：  " + str(speed_CFB) + " s/byte")

    speed_XCBC = measure_runtime("XCBC/XCBC.py")
    print("XCBC的每字节加密速度为：" + str(speed_XCBC) + " s/byte")

    speed_CTR = measure_runtime("CTR/CTR.py")
```

```
41      print("CTR的每字节加密速度为：  " + str(speed_CTR) + " s/byte")
```

代码 11: 统计运行效率

程序运行的结果即各个工作模式的工作效率如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\cryptography\lab_3\work_style\caculate_speed.py
ECB的每字节加密速度为：  0.0008282214403152466 s/byte
MCBC的每字节加密速度为：0.0008483231067657471 s/byte
CBC的每字节加密速度为：  0.0008676126599311829 s/byte
OFB的每字节加密速度为：  0.0011121372678386632 s/byte
CFB的每字节加密速度为：  0.0010512515680113836 s/byte
XCBC的每字节加密速度为：0.0011135962472033146 s/byte
CTR的每字节加密速度为：  0.000783884703223385 s/byte

Process finished with exit code 0
```

图 38: 所有工作模式运行效率

4.4 短块处理技术的比较

4.4.1 短块数据扩张

（1）短块填充法：

填充法会导致数据扩张，因为它添加了额外的字节。数据块的最终大小将等于加密算法所需的固定块大小，即使原始数据较短。

（2）序列密码加密法

序列密码加密通常不会导致数据扩张。它生成的密文长度与原始数据长度相同，因为它是逐字节或逐位加密的。

（3）密文挪用技术

密文挪用技术通常不会导致数据扩张。它通过重新组织最后两个块的内容来确保输出数据与输入数据具有相同的长度，从而避免了填充带来的数据扩张。

4.4.2 安全性

（1）短块填充法

这种填充方法本身并不增加明显的安全风险，因为添加的填充数据是固定和可预测的，不包含有关原始数据的敏感信息。然而，如果攻击者能够控制某些明文

块并分析响应的密文变化，他们可能能够得出一些关于填充过程或加密算法的信息。但这种方法的安全性主要依赖于底层加密算法的强度。

（2）序列密码加密法

这种方法的安全性较高，因为它依赖于已加密的数据块，该数据块应该是难以预测和无法控制的。由于每个块的加密依赖于前一个块的密文，这为加密过程提供了额外的复杂性。在选择明文攻击中，即使攻击者可以选择明文块，他们也无法直接控制或预测用于加密的“密钥流”，这使得攻击变得困难。

（3）密文挪用技术

密文挪用技术不会增加额外的安全风险，因为它只是对最后两个块进行重新安排，而不是改变加密算法本身。它的目的是避免填充带来的潜在风险，同时保持数据长度不变。在选择明文攻击的情境下，这种方法并不提供额外的保护，但也不会降低现有的加密算法的安全性。

五、 实验结果与数据处理

表 1: 各工作模式特点

工作模式	ECB	明密文链接	CBC	OFB	CFB	XCBC	CTR
能否掩盖数据模式	否	是	是	是	是	是	是
加密错误传播	有界	无界	无界	有界	无界	无界	有界
解密错误传播	有界	无界	有界	有界	无界	有界	有界
是否改变消息长度	否	是	是	是	是	是	是
能否处理短块	否	否	否	是	是	是	是
执行速度（ms/byte）	0.828	0.848	0.868	1.112	1.051	1.114	0.784

表 2: 各短块处理技术特点

短块处理方式	填充	序列密码加密	密文挪用
短块数据扩张	会	否	否
实现难度	低	中	高
安全性	依赖加密算法	中	高

六、 分析与讨论

在这次的实验中，我深入探索了分组密码的各种工作模式，包括 ECB（电子密码本模式）、CBC（密码块链接模式）、OFB（输出反馈模式）、CFB（密码反馈模式）和 CTR（计数器模式），以及分组密码的短块加密技术。通过这个实验，我不仅提高了自己的理论知识，还获得了宝贵的实践经验，尤其是在处理任意长度输入数据的加密和解密方面。

在实验中，我特别注意了短块加密技术的应用。通过使用填充法、序列密码加密法和密文挪用技术，我成功地对不满一个完整块的数据进行了有效处理。这些技术的实际应用不仅增强了我的编程技能，也加深了我对于这些概念的理解。

利用这些工作模式和技术，我成功实现了对任意长度输入的加密和解密。这个过程不仅考验了我的理论知识，也锻炼了我的编程能力。我学会了如何根据不同的应用场景选择最合适的加密模式，并理解了在安全性和效率之间取得平衡的重要性。

七、 教师评语