

中国矿业大学计算机学院

17 级本科生课程报告

课程名称 密码学课程设计

报告时间 2019.12.30

学生姓名 袁孝健

学 号 06172151

专 业 信息安全

任课教师 谢林

密码学课程设计评分标准

编号	实验教学目标	考查方式与考查点	应得分	得分
1	目标 1: 能够对一个信息系统（或算法）进行基本的安全性分析。	实验演示，实验报告，实验代码。 能够针对每个算法，经过分析并实现该算法的加解密，最后验证所实现算法正确性，并进行基本安全分析。	60	
2	目标 2: 结合所编写的古典密码, 对称密码算法, Hash 函数等算法进行有效性、合理性等方面的算法分析。	实验报告。 能对编写的算法进行诸如有效性、合理性等方面的算法分析。	20	
3	目标 3: 能够根据已有的开发工具, 选择一种恰当工具开发编写算法。能够根据一个工程问题, 分析其安全隐患, 并给出加密方案。	实验报告。 能给出解释所选择开发工具的原因, 并能够根据问题给出解决方案和实现算法。	20	
			100	

评阅人:

摘 要

本次密码学课程设计，总共分为古典密码、流密码、分组密码、Hash 函数、公钥密码、综合实验六个部分。我依次对各部分代表性的密码算法进行了编程实现，并验证了正确性、分析比较了性能效率、阐述了可用性与安全性，并通过查阅资料对攻击方式也进行了一定的拓展。最后，通过综合实验——基于 C/S 架构的文件加密传输系统，对前面所进行实验内容进行了总结与应用，从而使得整个密码学课程设计更加的具有实用性与连贯性。

关键词：古典密码; 流密码; 分组密码; RSA; Hash 函数

Abstract

The course design of cryptography, total is divided into the classical password, stream cipher, block cipher, Hash function and public key cryptography, comprehensive experiment six parts. I programmed the representative cryptographic algorithms of each part in turn, verified the correctness, analyzed and compared the performance efficiency, elaborated the availability and security, and expanded the attack methods by consulting the data. Finally, through the comprehensive experiment -- the file encryption and transmission system based on C/S architecture, the previous experiment content is summarized and applied, so as to make the whole course design more practical and coherent.

Keywords: Classical Cipher; Stream Cipher; Block Cipher; RSA; Hash function

目 录

前言.....	8
第 1 章 古典密码	10
1.1 凯撒密码.....	10
1.1.1 算法原理.....	10
1.1.2 代码实现.....	10
1.1.3 正确性验证与性能分析.....	11
1.1.4 安全性与可用性分析.....	12
1.1.5 破解或攻击方式分析.....	12
1.2 维吉尼亚密码.....	13
1.2.1 算法原理.....	13
1.2.2 代码实现.....	13
1.2.3 正确性验证与性能分析.....	14
1.2.4 安全性与可用性分析.....	14
1.2.5 破解或攻击方式分析.....	15
1.3 实验总结.....	17
1.3.1 存在的问题.....	17
1.3.2 实验感想.....	17
第 2 章 流密码	18
2.1 线性反馈移位寄存器（LFSR）	18
2.1.1 LFSR 的原理分析.....	18
2.1.2 LFSR 的代码实现.....	18
2.1.3 LFSR 的周期分析.....	19
2.1.4 m 序列密码的破译	20
2.2 RC4 算法	21
2.2.1 算法原理.....	21
2.2.2 代码实现.....	21
2.2.3 正确性验证与性能分析.....	22
2.2.4 安全性与可用性分析.....	23
2.2.5 破解或攻击方式分析.....	23
2.3 实验总结.....	24
2.3.1 存在的问题.....	24

2.3.2 实验感想	24
第 3 章 分组密码	25
3.1 数据加密标准 (DES)	25
3.1.1 原理分析	25
3.1.2 代码实现	26
3.1.3 正确性验证与性能分析	30
3.1.4 安全性与可用性分析	31
3.1.5 破解或攻击方式分析	32
3.2 分组密码的工作模式	32
3.2.1 电子密码本模式 (ECB)	32
3.2.1.1 原理分析	32
3.2.1.2 DES-ECB 代码实现	33
3.2.1.3 正确性验证	33
3.2.1.4 可用性与安全性分析	34
3.2.2 密码分组链接模式 (CBC)	35
3.2.2.1 原理分析	35
3.2.2.2 DES-CBC 代码实现	35
3.2.2.3 正确性验证	36
3.2.2.4 可用性与安全性分析	37
3.2.3 密码反馈模式 (CFB)	38
3.2.3.1 原理分析	38
3.2.3.2 DES-CFB 代码实现	38
3.2.3.3 正确性验证	39
3.2.3.4 可用性与安全性分析	40
3.2.4 输出反馈模式 (OFB)	41
3.2.4.1 原理分析	41
3.2.4.2 DES-OFB 代码实现	41
3.2.4.3 正确性验证	42
3.2.4.4 可用性与安全性分析	43
3.2.5 计数器模式 (CTR)	44
3.2.5.1 原理分析	44
3.2.5.2 DES-CTR 代码实现	44
3.2.5.3 正确性验证	45

3.2.2.4 可用性与安全性分析	46
3.2.6 各工作模式效率分析与对比	47
3.4 实验总结.....	49
3.4.1 存在的问题	49
3.4.2 实验感想	49
第 4 章 HASH 函数.....	51
4.1 MD5 算法	51
4.1.1 原理分析	51
4.1.2 代码实现.....	51
4.1.3 正确性验证与性能分析	54
4.1.4 可用性与安全性分析.....	58
4.1.5 破解或攻击方式分析.....	58
4.1.5.1 学术界成果.....	58
4.1.5.2 生日攻击	59
4.2 实验总结.....	59
4.2.1 存在的问题	59
4.2.2 实验感想.....	59
第 5 章 公钥密码	60
5.1 RSA 公钥加密体制	60
5.1.1 原理分析	60
5.1.1.1 RSA 算法.....	60
5.1.1.2 Miller-Rabin 素性检验	60
5.1.2 代码实现.....	61
5.1.3 正确性验证	63
5.1.4 性能分析	64
5.1.4.1 快速幂算法与内置函数效率对比	64
5.1.4.2 Miller-Rabin 素性检验迭代轮数分析	66
5.1.5 可用性与安全性分析.....	67
5.1.6 破解或攻击方式分析.....	67
5.1.6.1 低加密指数攻击	67
5.1.6.2 低加密指数广播攻击	68
5.1.6.3 共模攻击	68
5.2 基于 RSA 的数字签名.....	69

5.2.1 原理分析	69
5.2.2 代码实现	69
5.2.3 正确性验证	70
5.2.4 可用性与安全性分析	71
5.3 实验总结	71
5.3.1 存在的问题	71
5.3.2 实验感想	71
第 6 章 综合实验——基于 C/S 架构的文件加密传输系统	73
6.1 开发工具选择	73
6.1.1 开发语言选择	73
6.1.2 开发 IDE 选择	73
6.2 设计目的与要求	73
6.3 概要设计	74
6.3.1 系统功能描述	74
6.3.2 系统功能流程图	75
6.4 系统代码实现	76
6.5 运行结果及分析	79
6.5.1 基本功能运行	79
6.5.2 加密传输分析	80
6.6 实验总结	81
6.6.1 遇到的问题及解决	81
6.6.2 改进方案	82
6.6.3 实验感想	82
附录	82

前 言

此次的密码学课程设计中，我的实验内容可大致总结如下，过程中所有的源代码均已上传至本人 Github（见附录）。

1. 古典密码

- 编写了凯撒密码的加解密代码，并验证其正确性
- 对凯撒密码的攻击破解方式进行了分析，并成功进行了唯密文破解
- 分析了凯撒密码的安全性、可用性与时间复杂度
- 编写了维吉尼亚密码的加解密代码，并验证其正确性
- 对维吉尼亚密码的唯密文破解进行分析，并用代码成功实现
- 分析了维吉尼亚密码的安全性、可用性与时间复杂度

2. 流密码

- 实现了线性反馈移位寄存器生成伪随机密钥流的代码
- 验证了本原多项式对 LFSR 最大周期的影响及其正确性
- 对 m 序列密码的破译进行了一定的分析
- 编写了 RC4 算法的加解密代码，并验证其正确性
- 分析了 RC4 算法的安全性、可用性
- 通过查阅资料，对 RC4 算法的现代破解方式进行一定的阐述

3. 分组密码

- 对数据加密标准（DES）进行了代码上的实现，并验证其正确性
- 将 DES 与 RC4 进行了效率上的对比，并简单分析
- 分析了在现代环境中，DES 的可用性与安全性
- 简单介绍了对 DES 的两种攻击方式
- 分别实现了 DES 在 ECB、CBC、CFB、OFB、CTR 五种工作模式下的加解密过程，并验证其正确性
- 对 DES 在五种工作模式下的可用性与安全性进行了分析
- 对比了五种工作模式下 DES 加解密的效率，并给出总结与分析
- 对比了 pyDes 官方库与自己实现的代码在效率上的差异，并分析原因

4. Hash 函数

- 编写了 MD5 算法的过程，并验证其正确性
- 对 MD5 算法的可用性与安全性进行了分析
- 与官方库的 MD5 算法进行了效率上的对比，并分析了原因
- 查阅资料，对 MD5 的攻击方式进行了一定的总结与阐述

5. 公钥密码

- 编写了 RSA 算法的加密解密过程，并验证其正确性
- 对 RSA 公钥加密体制的可用性与安全性进行了分析
- 比较了快速幂算法与内置函数的效率
- 对 Miller-Rabin 素性检验的迭代轮数选择进行了分析
- 对 RSA 的低加密指数攻击、低加密指数广播攻击、共模攻击进行了分析，并给出攻击脚本代码
- 编写了基于 RSA 与 md5 的数字签名体制流程，并对验证其正确性
- 对 RSA 数字签名体制的可用性与安全性进行了分析

6. 综合实验

- 实现了基于 C/S 架构的文件加密传输系统
- 对系统的设计目标、设计流程进行了阐述与分析
- 对系统功能的实现进行了展示
- 通过 Wireshark 对系统的文件加密传输流程进行了分析
- 总结了实验中遇到的问题以及改进方案

第 1 章 古典密码

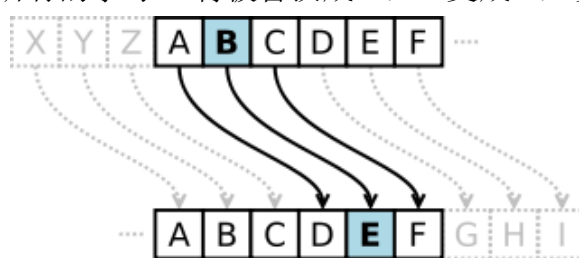
实验内容：

- 编写了凯撒密码的加解密代码，并验证其正确性
- 对凯撒密码的攻击破解方式进行了分析，并成功进行了唯密文破解
- 分析了凯撒密码的安全性、可用性与时间复杂度
- 编写了维吉尼亚密码的加解密代码，并验证其正确性
- 对维吉尼亚密码的唯密文破解进行分析，并用代码成功实现
- 分析了维吉尼亚密码的安全性、可用性与时间复杂度

1.1 凯撒密码

1.1.1 算法原理

凯撒密码是一种最简单且最广为人知的加密技术，它是一种替换加密的技术，明文中的所有字母都在字母表上向后（或向前）按照一个固定数目进行偏移后被替换成密文。例如，当偏移量是 3 的时候，所有的字母 A 将被替换成 D，B 变成 E，以此类推。



算法实现：对字母进行编号，大写字母 A~Z 或小写字母 a~z 分别对应数字 0~25。

加密算法： $c_i = (m_i + k) \bmod 26$

解密算法： $m_i = (c_i - k) \bmod 26$

解密算法就是加密算法的逆运算，在具体的实现的时候，因为明文中可能会存在特殊字符，所以要对特殊字符单独处理，而为了保持密文解密后的大小写，也可以对大写和小写字母分别处理。

1.1.2 代码实现

(1) 加密函数

```
def Encrypt(m, k):
    """
    凯撒密码加密
    :param m: 明文
    :param k: 密钥(移位量)
    :return: 加密的结果
    """
    k %= 26
    c = ''
    mlen = len(m)
    for i in range(mlen):
        if m[i].isupper():
            c += chr((ord(m[i]) - ord('A') + k) % 26 + ord('A'))
        elif m[i].islower():
            c += chr((ord(m[i]) - ord('a') + k) % 26 + ord('a'))
        else:
            c += m[i]
    return c
```

(2) 解密函数

```
def Decrypt(c, k):
    """
    凯撒密码解密
    :param c: 密文
    :param k: 密钥
    :return: 解密的结果
    """
    k %= 26
    m = ''
    clen = len(c)
    for i in range(clen):
        if c[i].isupper():
            m += chr((ord(c[i]) - ord('A') - k + 26) % 26 + ord('A'))
        elif c[i].islower():
            m += chr((ord(c[i]) - ord('a') - k + 26) % 26 + ord('a'))
        else:
            m += c[i]
    return m
```

1.1.3 正确性验证与性能分析

(1) 正确性验证

① 加密:

```
Please choose a mood:
[E]Encrypt [D]Decrypt [C]Crack [Q]Quit
E
Please input the message:
My name is YuanXiaojian.
Please input the key:
12
The cipher is:
Yk zmyq ue KgmzJumavumz.
```

② 解密:

```

Please choose a mood:
[E]Encrypt [D]Decrypt [C]Crack [Q]Quit
D
Please input the cipher:
Yk zmyq ue KgmzJumavumz.
Please input the key:
12
The message is:
My name is YuanXiaojian.

```

(2) 性能分析

加密解密均只需要遍历一遍明文/密文，因此加密解密算法的时间复杂度为 $O(n)$

1.1.4 安全性与可用性分析

凯撒密码通常被作为其他更复杂的加密方法如维吉尼亚密码中的一个步骤，以及还在现代的 ROT13 系统中被应用。但是和所有的利用字母表进行替换的加密技术一样，凯撒密码非常容易被破解，而且在实际应用中也无法保证通信安全。

1.1.5 破解或攻击方式分析

现在而言，凯撒密码可以轻易的被唯密文攻击，由于使用恺撒密码进行加密的语言一般都是字母文字系统，因此密码中可能是使用的偏移量也是有限的，例如使用 26 个字母的英语，它的偏移量最多就是 25（偏移量 26 等同于偏移量 0，即明文；偏移量超过 26，等同于偏移量 1-25）。因此可以通过穷举法，很轻易地进行破解。

穷举法破解凯撒密码的代码如下：

```

def Crack(c):
    for i in range(1, 26):
        print("k = " + str(i).zfill(2) + ": ", Decrypt(c, i))

```

破解结果如下：

```

Please choose a mood:
[E]Encrypt [D]Decrypt [C]Crack [Q]Quit
C
Please input the cipher:
Yk zmyq ue KgmzJumavumz.
Possible messages:

k = 01: Xj ylxp td JflyItlzutly.
k = 02: Wi xkwo sc IekxHskytskx.
k = 03: Vh wjvn rb HdjwGrjxsrjw.
k = 04: Ug vium qa GcivFqiwrqiv.
k = 05: Tf uhtl pz FbhuEphvqphu.
k = 06: Se tgs k oy EagtDogupogt.
k = 07: Rd sfrj nx DzfsCnftonfs.
k = 08: Qc reqi mw CyerBmesmer.
k = 09: Pb qdph lv BxdqAldrml dq.
k = 10: Oa pcog ku AwcpZkcqlkcp.
k = 11: Nz obnf jt ZvboYjbpkjbo.
k = 12: My name is YuanXiaojian.
k = 13: Lx mzld hr XtzmWhznihzm.

```

1.2 维吉尼亚密码

1.2.1 算法原理

维吉尼亚密码是使用一系列凯撒密码组成密码字母表的加密算法，属于多表密码的一种简单形式。

用数字 0-25 代替字母 A-Z，维吉尼亚密码的加密算法可以写成同余的形式：

$$C_i \equiv P_i + K_i \pmod{26}$$

解密的过程则与加密相反：

$$P_i \equiv C_i - K_i \pmod{26}$$

1.2.2 代码实现

(1) 加密函数

```
def Encrypt(m, k):
    """
    维吉尼亚密码加密
    :param m: 明文
    :param k: 密钥
    :return: 密文
    """
    c = ''
    j = 0
    for i in range(len(m)):
        if m[i].isupper():
            c += chr(((str2num(m[i]) + str2num(k[j % len(k)])) % 26) + ord('A'))
            j += 1
        elif m[i].islower():
            c += chr(((str2num(m[i]) + str2num(k[j % len(k)])) % 26) + ord('a'))
            j += 1
        else:
            c += m[i]
    return c
```

(2) 解密函数

```
def Decrypt(c, k):
    """
    维吉尼亚密码解密
    :param c: 密文
    :param k: 密钥
    :return: 明文
    """
    m = ''
    j = 0
    for i in range(len(c)):
        if c[i].isupper():
            m += chr(((str2num(c[i]) - str2num(k[j % len(k)]) + 26) % 26) + ord('A'))
            j += 1
        elif c[i].islower():
            m += chr(((str2num(c[i]) - str2num(k[j % len(k)]) + 26) % 26) + ord('a'))
            j += 1
        else:
            m += c[i]
    return m
```

1.2.3 正确性验证与性能分析

(1) 正确性验证

① 加密:

```
Please choose a mode:
[E]Encrypt [D]Decrypt [C]Crack [Q]Quit
E
Please input the message:
My name is YuanXiaojian,I'm from CUMT.
Please input the key:
crypto
The cipher is:
Op lpfs kj WjtbZzydcwce,G'b yfqd AJFH.
```

```
Please choose a mode:
[E]Encrypt [D]Decrypt [C]Crack [Q]Quit
```

② 解密:

```
Please choose a mode:
[E]Encrypt [D]Decrypt [C]Crack [Q]Quit
D
Please input the cipher:
Op lpfs kj WjtbZzydcwce,G'b yfqd AJFH.
Please input the key:
crypto
The message is:
My name is YuanXiaojian,I'm from CUMT.
```

(2) 性能分析

维吉尼亚密码也是一种加密解密简洁迅速的密码，加密解密均只需要一层 for 循环，时间复杂度为 $O(n)$ ，也可以通过提前建立二维表牺牲空间换取时间。

1.2.4 安全性与可用性分析

(1) 安全性

维吉尼亚是一种多表代换的密码，相对于单表替换多打破了原语言的字符出现规律，一定程度上掩盖了明文的统计特性，无法直接使用频率分析法，同时密钥空间大大增大，可以是安全性大大提高。

但是维吉尼亚密码仍然可以用重合指数法等方法进行唯密文攻击，但是前提是密文足够的长，较短的密文几乎是不可破译的。

(2) 可用性

维吉尼亚密码加解密算法简单，时间耗费少，适合于资源有限的情况下对短信息的加密。而对于较重要的信息以及大量的明文情况下，则不适合使用维吉尼亚密码进行，相对于对于现代的计算能力来说，其实际上已无太大实际用途。

1.2.5 破解或攻击方式分析

实际上，通过分析可以知道维吉尼亚密码有着比较明显的漏洞，即如果攻击者知道了密钥的长度，那密文就可以被看作是交织在一起的恺撒密码，而其中每一个都可以单独破解。所以要想破解维吉尼亚密码，最关键的步骤就使要得出密钥的长度。

(1) 密钥长度获取

因为维吉尼亚密码是多表循环加密的，所以若假设密钥长度为 d ，那么可以知道第 $(1 + d), \dots, (1 + k \times d)$ 都是使用密钥中的同一个字母进行加密的，如果第 i 和第 $(i + k \times d)$ 个字母相同的话，他们的密文一定是一样的。利用这个事实，我们可以假设密钥长度为 3~15 逐个进行测试，找出在每个猜测的密钥长度下，两个字母间隔密钥长度且相同组数最多的那一个密钥长度，则其最有可能为真实密钥长度。实际上此方法也是基于重合指数的原理，但是不用具体计算重合指数，代码实现简单，且在存在大量密文的情况下准确度也很高，所以我选择使用此方法。

具体代码实现如下：

```
# 获取密钥长度
# 遍历3-20位, 相同间隔的字母相同数最多的位数, 即最有可能为密钥长度
def get_keyLength(cipher):
    keyLen = 1
    maxCount = 0
    # 密钥长度3-20位
    for step in range(3, 21):
        count = 0
        for i in range(len(cipher) - step):
            if cipher[i] == cipher[i - step]:
                count += 1
        if count > maxCount:
            maxCount = count
            keyLen = step
    return keyLen
```

(2) 根据密钥长度确定密钥

```
def get_key(text, length):
    key = ''
    alphaFreq = [0.082, 0.015, 0.028, 0.043, 0.127, 0.022, 0.02, 0.061, 0.07, 0.002, 0.008, 0.04, 0.024,
                  0.067, 0.075, 0.019, 0.001, 0.06, 0.063, 0.091, 0.028, 0.01, 0.023, 0.001, 0.02, 0.001]
    cipher_matrix = cipherSplit(text, length)
    for i in range(length):
        col = [row[i] for row in cipher_matrix]
        freq = countFreq(col)
        IC = []
        for j in range(26):
            ic = 0.00000
            for k in range(26):
                ic += freq[k] * alphaFreq[k]
            IC.append(ic)
        freq = freq[1:] + freq[:1]
        temp = 1
        ch = ''
        for j in range(len(IC)):
            if abs(IC[j] - 0.065546) < temp:
                temp = abs(IC[j] - 0.065546)
                ch = chr(j + 97)
        key += ch
    return key
```

在求得密钥长度之后，通过穷举密钥字母的每一种可能取值（A 到 Z 总共 26 种），并且针对每一行求其在某一取值下重合指数，重合指数最高的情况下该行最有可能为明文原文，此时的穷举结果即为密钥字母。

（3）其他函数

统计a-z字母出现的频率

```
def countFreq(text):
    freq = []
    for i in range(97, 123):
        count = 0
        for ch in text:
            if ch == chr(i):
                count += 1
        freq.append(count / len(text))
    return freq
```

根据密钥长度对密文分组,用二维矩阵表示

每一行都是用按密钥加密,每一列都是用同一个字母移位加密的

```
def cipherSplit(text, length):
    cipher_matrix = [] # 二维矩阵
    row = [] # 每一行都是用按密钥加密的,长度为密钥长度
    i = 0
    for ch in text:
        row.append(ch)
        i += 1
        if i % length == 0:
            cipher_matrix.append(row)
            row = []
    return cipher_matrix
```

（4）正确性验证

选取一段 761 字的英文片段并使用密钥“IamFromCUMT”进行加密，对得到的密文进行破解如下：

```
Please choose a mode:
[E]Encrypt [D]Decrypt [C]Crack [Q]Quit
E
Please input the message:
More generally, cryptography is about constructing and analyzing protocols that prevent thi
Please input the key:
IamFromCUMT
The cipher is:
Uodj xszglmety, owpdfqadtxhk nj onqof vwneyiiovczz inp feoxatugo pdtkcoqfe mpaf uishghf mpi

Please choose a mode:
[E]Encrypt [D]Decrypt [C]Crack [Q]Quit
C
Please input the cipher:
Uodj xszglmety, owpdfqadtxhk nj onqof vwneyiiovczz inp feoxatugo pdtkcoqfe mpaf uishghf mpi
The key might be:
iamfromcumt
The message might be:
More generally, cryptography is about constructing and analyzing protocols that prevent thi
```


1.3 实验总结

1.3.1 存在的问题

（1）在凯撒密码的破解上，可以考虑使用 Python 的 NLP 库，实现直接从暴力破解的结果中输出符合自然语言规律的字符串即为密文。

（2）维吉尼亚密码的唯密文破解成功率仍有待提高，牺牲了具体计算重合指数的准确度而获取了代码层面的效率。

1.3.2 实验感想

此次实验通过对凯撒密码以及维吉尼亚密码的实现与分析，对单表、多表替换密码有了较深的学习，并第一次通过编程实现了加密、解密以及破解算法。同时，也认识到了古典密码对于现代计算机的运算能力而言，已经几乎无安全性可言，无论是穷举攻击还是利用统计分析方法，破译它们都没有太大的难度，我们需要认识到这一点。

第2章 流密码

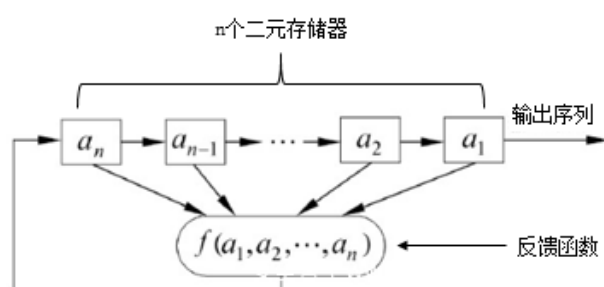
实验内容：

- 实现了线性反馈移位寄存器生成伪随机密钥流的代码
- 验证了本原多项式对 LFSR 最大周期的影响及其正确性
- 对 m 序列密码的破译进行了一定的分析
- 编写了 RC4 算法的加解密代码，并验证其正确性
- 分析了 RC4 算法的安全性、可用性
- 通过查阅资料，对 RC4 算法的现代破解方式进行一定的阐述

2.1 线性反馈移位寄存器（LFSR）

2.1.1 LFSR 的原理分析

LFSR 是属于 FSR（反馈移位寄存器）的一种，除了 LFSR 之外，还包括 NFSR（非线性反馈移位寄存器）。FSR 是流密码产生密钥流的一个重要组成部分，在 $GF(2)$ 上的一个 n 级 FSR 通常由 n 个二元存储器和一个反馈函数组成，如下图所示：



如果这里的反馈函数是线性的，我们则将其称为 LFSR，此时该反馈函数可以表示为：

$$f(a_1, a_2, \dots, a_n) = c_n a_1 \oplus c_{n-1} a_2 \oplus \dots \oplus c_1 a_n$$

其中 $c_i=0$ 或 1 ， \oplus 表示异或（模二加）。

2.1.2 LFSR 的代码实现

（1）反馈函数：

```
def feedback(reg, fb):
    """
    反馈函数
    :param reg: 移位寄存器的内容
    :param fb: 由抽头构成的列表
    :return: 最左端的输入
    """
    res = reg[fb[0] - 1]
    for i in range(1, len(fb)):
        res = int(res) ^ int(reg[fb[i] - 1])
    return res
```

(2) 移位寄存器:

```
def lfsr(p):
    """
    线性反馈移位寄存器
    :param p: 由本原多项式次数构成的列表
    :return:
    """
    reg_len = max(p)
    # 初始寄存器状态:00000....001
    shift_reg = '1'.zfill(reg_len)
    regs = [shift_reg] # 存放寄存器的各个状态
    output_ = [] # 存放输出序列
    for i in range(pow(2, reg_len) - 1):
        # 输出寄存器最右端的值
        output_.append(shift_reg[-1])
        # 计算抽头异或的值
        input_ = str(feedback(shift_reg, p))
        shift_reg = input_ + shift_reg[:-1]
        # 如果寄存器当前状态已经出现过了,说明一个周期结束
        if shift_reg in regs:
            break
        else:
            regs.append(shift_reg)
    return output_, regs
```

2.1.3 LFSR 的周期分析

(1) 本原多项式的选取

下列网站提供了本原多项式的在线查询,我选取 16 次本原多项式进行测试:

http://wims.unice.fr/wims/cn_tool~algebra~primpoly.cn.html

本原多项式

在 \mathbb{F}_2 上总共有 2048 个 16 次本原多项式.

搜索结果: [往后>>](#)

- 1 $x^{16}+x^5+x^3+x^2+1$
- 2 $x^{16}+x^5+x^4+x^3+1$
- 3 $x^{16}+x^5+x^4+x^3+x^2+x+1$
- 4 $x^{16}+x^6+x^4+x+1$
- 5 $x^{16}+x^7+x^5+x^4+x^3+x^2+1$
- 6 $x^{16}+x^7+x^6+x^4+x^2+x+1$
- 7 $x^{16}+x^8+x^5+x^3+x^2+x+1$
- 8 $x^{16}+x^8+x^5+x^4+x^3+x^2+1$
- 9 $x^{16}+x^8+x^6+x^3+x^2+x+1$
- 10 $x^{16}+x^8+x^6+x^4+x^3+x^2+1$

(2) 周期验证

选取 16 次本原多项式:

$$x^{16} + x^5 + x^3 + x^2 + 1$$

输入本原多项式的 x 系数，得到理论最大周期 65535，实际周期 65535，并保存每个输出时的寄存器状态。

请输入本原多项式 x 的系数(以空格分隔):

16 5 3 2

您输入的本原多项式为:

$x^{16} + x^5 + x^3 + x^2 + 1$

理论最大周期为: 65535

是否确认:

[Y]确定 [N]重新输入 [Q]退出

Y

周期为: 65535

是否查看输出序列:

[Y]是 [N]否

Y

100000000000000010111101000010110101101000011111011111110010101100000

是否输出周期内寄存器各状态:

[Y]是 [N]否

Y

成功! 周期内寄存器各状态保存在regs.txt

可以看到第 65535 个寄存器状态的下一个状态即为初始状态，为一个周期:

1	0000000000000001	65529	0000000010000000
2	1000000000000000	65530	0000000001000000
3	0100000000000000	65531	0000000000100000
4	1010000000000000	65532	0000000000010000
5	1101000000000000	65533	0000000000001000
6	1110100000000000	65534	0000000000000100
7	1111010000000000	65535	0000000000000010

2.1.4 m 序列密码的破译

由于 LFSR 是线性关系，由其输入输出之间的线性关系来决定。然而在密码体制中，线性关系会使密钥变得非常不安全，已知明文攻击通过求解可以对其进行破译。假设攻击者已知一段长度为 $2m$ 的明文 $x_0, x_1, x_2, \dots, x_{2m-1}$ ，它对应得密文 $y_0, y_1, y_2, \dots, y_{2m-1}$ ，此时攻击者可以重构开头得 $2m$ 个密钥序列:

$$s_i \equiv x_i + y_i \pmod{2}; i = 0, 1, 2, \dots, 2m - 1$$

由于每个 i 值都会生成不同的等式，因此我们可以得到 i 个等式:

$$\begin{aligned} i = 0, s_m &\equiv s_{m-1}p_{m-1} + \dots + s_1p_1 + s_0p_0 \pmod{2} \\ i = 1, s_{m+1} &\equiv s_m p_{m-1} + \dots + s_2p_1 + s_1p_0 \pmod{2} \\ i = 2, s_{m+2} &\equiv s_{m+1}p_{m-1} + \dots + s_3p_1 + s_2p_0 \pmod{2} \\ &\vdots \\ i = m - 1, s_{2m-2} &\equiv s_{m-2}p_{m-1} + \dots + s_m p_1 + s_{m-1}p_0 \pmod{2} \end{aligned}$$

对于这个等式，可以使用高斯消去法、矩阵求逆或其他方法来求解此线性等式系统。即使 m 的值非常大，现代计算机仍然能够在较短时间计算求解。当然这并不代表 LFSR 无法应用到密码体系中去，我们可以通过使用多个 LFSR 进行组合，构建一个健壮的密码体制。

2.2 RC4 算法

2.2.1 算法原理

RC4 是一种流加密算法，是有线等效加密（WEP）中采用的加密算法，也曾经是 TLS 可采用的算法之一。RC4 由伪随机数生成器和异或运算组成，其密钥长度可变，范围是[1,255]。

RC4 一个字节一个字节地加解密，给定一个密钥，伪随机数生成器接受密钥并产生一个 S 盒，盒用来加密数据，而且在加密过程中 S 盒会变化。

RC4 包含两个处理过程：一个是密钥调度算法（KSA），用来置乱 S 盒得初始排列；另一个是伪随机生成算法（PRGA），用来输出随机序列并修改 S 的当前排列顺序。

2.2.2 代码实现

（1）生成 S 盒

```
def get_sbox(k):
    # 将0~255装入S盒
    sBox = list(range(256))
    T = [ord(k[i % len(k)]) for i in range(256)]
    # 根据密钥打乱S盒
    j = 0
    for i in range(256):
        j = (j + sBox[i] + T[i]) % 256
        sBox[i], sBox[j] = sBox[j], sBox[i]
    return sBox
```

（2）生成密钥流并进行异或操作

```
def get_sbox(k):
    # 将0~255装入S盒
    sBox = list(range(256))
    T = [ord(k[i % len(k)]) for i in range(256)]
    # 根据密钥打乱S盒
    j = 0
    for i in range(256):
        j = (j + sBox[i] + T[i]) % 256
        sBox[i], sBox[j] = sBox[j], sBox[i]
    return sBox
```

(3) 加密/解密 (Base64 方式)

```
def encrypt(message, key):
    return base64.b64encode(prga(message, key).encode()).decode()

def decrypt(cipher, key):
    return prga(base64.b64decode(cipher).decode(), key)
```

2.2.3 正确性验证与性能分析

(1) 正确性验证

① 加密

使用“cumt_crypto”作为密钥加密“My name is YuanXiaojian”:

请选择模式:

[E]加密 [D]解密 [Q]退出

E

请输入明文:

My name is YuanXiaojian.

请输入密钥:

cumt_crypto

密文为:

wPCpCcDeXUAW6PCsjIjfGHDv80bIM00w5PDtcORw5rCoh/Cvg==

② 解密

使用相同的密钥对上述密文进行解密, 结果如下:

请选择模式:

[E]加密 [D]解密 [Q]退出

D

请输入密文:

wPCpCcDeXUAW6PCsjIjfGHDv80bIM00w5PDtcORw5rCoh/Cvg==

请输入密钥:

cumt_crypto

明文为:

My name is YuanXiaojian.

(2) 性能分析

RC4 的执行速度相当快, 它大约是分块密码算法 DES 的 5 倍, 是 3DES 的 15 倍, 且比高级加密算法 AES 也快很多。RC4 算法简单, 实现容易。

我们使用如下测试代码, 分别加密解密 10000 次, 来对 RC4 性能进行分析:

```
import time
message = 'YuanXiaoJian'
key = '06172151'
start = time.time()
for i in range(10000):
    cipher = encrypt(message, key)
end = time.time()
print("encrypt spend time: " + str(end - start) + " seconds.")
start = time.time()
for i in range(10000):
    message = decrypt(cipher, key)
end = time.time()
print("decrypt spend time: " + str(end - start) + " seconds.")
```

结果如下，因为加解密流程相同，因此所需时间也基本一致，同时可以看到使用 RC4 进行加解密具有效率高、速度快的特点：

```
D:\code\Anaconda3\python.exe F:/MyCode/Cryptography/StreamCipher/RC4.py
encrypt spend time: 0.8138253688812256 seconds.
decrypt spend time: 0.8297500610351562 seconds.
```

2.2.4 安全性与可用性分析

(1) 安全性

一个 256 位的 RC4 密码共有 $256!$ 种可能，即 2^{1600} 种可能，这样就使得穷举攻击变得不可能。但是 RC4 算法加密强度完全取决于密钥，即伪随机序列生成，而伪随机就不可避免的会存在密钥的重复问题，加密解密简单的异或操作就导致了一旦子密钥序列出现了重复，密文就极有可能被破解。

(2) 可用性

RC4 具有算法简单、容易实现、运行速度快等优点，且采用的是输出反馈工作方式，所以可以用一个短的密钥产生一个相对较长的密钥序列，除此之外，OFB 模式最大的优点是消息如果某一位发生了错误，其影响不会传递到产生的密钥序列上。RC4 的安全保证主要在于输入密钥的产生途径，只要在这方面不出现漏洞，使用 128bit 的密钥还是相对较为安全的。

2.2.5 破解或攻击方式分析

实际上有研究人员发现利用 RC4 中的统计偏差，可对加密信息中的一些伪随机字节能进行猜测。经查阅资料，破解过程大致如下：

若明文、密钥是任意长的字节，可以用“重合码计数法”找出密钥长度。把密文进行各种字节的位移，并与原密文进行异或运算，统计那些相同的字节。如果位移是密钥长度的倍数，那么超过 60% 的字节将是相同的；如果不是，则至多只有 0.4% 的字节是相同的，这叫做“重合指数”。找出密钥长度倍数的最小位移，按此长度移到密文，并且和自身异或。由

于明文每字节有 1.3 位的实际信息，因此有足够的冗余度去确定位移的解密。对所有的密钥，输出密钥流的前几个字节不是随机的，因此极有可能会泄露密钥的信息。

2.3 实验总结

2.3.1 存在的问题

(1) 在 LFSR 实验的过程种, 因为书本与课堂讲授的不一致, 导致产生了一定时间的迷惑, 但在整理思路、理解原理后, 成功安装正确的方式完成周期序列的验证。

(2) 在 RC 实验中, 由于 python3 对于编码的改变, 其 Base64 编码函数接收 byte 类型, 不接受 str 类型, 而输出的也是 byte 类型, 所以在编写程序的时候一定要注意类型的转换, 否则可能会遇到意料之外的问题。

2.3.2 实验感想

流密码（又称序列密码）是一类重要的对称密码体制，具有算法实现简单、速度快、传播错误少的特点，其简单的加解密算法需要对密钥流生成器有较高的要求。因此本次实验中，我首先实现了 LFSR 来深入了解伪随机密钥流生成器的原理，并对其周期性进行了一定的验证。而光有密钥流生成器是远远不够的，所以我又实现了使用非线性函数的 RC4 算法，并进行了加密、解密的实验。

流密码作为现代密码学的第一个学习内容，我深刻的知道课堂上及实验中涉及到的内容只是冰山一角，我仍需要在课后对其进行深一步的了解与学习。

第 3 章 分组密码

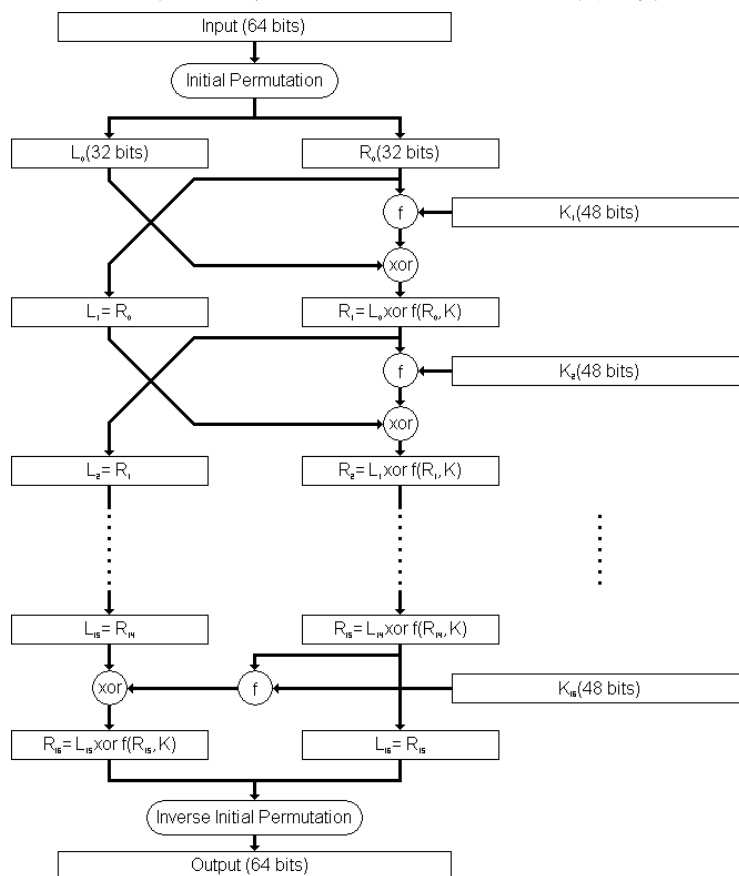
实验内容：

- 对数据加密标准（DES）进行了代码上的实现，并验证其正确性
- 将 DES 与 RC4 进行了效率上的对比，并简单分析
- 分析了在现代环境中，DES 的可用性与安全性
- 简单介绍了对 DES 的两种攻击方式
- 分别实现了 DES 在 ECB、CBC、CFB、OFB、CTR 五种工作模式下的加解密过程，并验证其正确性
- 对 DES 在五种工作模式下的可用性与安全性进行了分析
- 对比了五种工作模式下 DES 加解密的效率，并给出总结与分析
- 对比了 pyDes 官方库与自己实现的代码在效率上的差异，并分析原因

3.1 数据加密标准（DES）

3.1.1 原理分析

DES 使用的分组密码种常见的 Feistel 网络，分组长度 64 比特，密钥长度 64 比特（其中有效长度为 56 比特），第 8、16、24、40、48、56 和 64 位是奇偶校验位。DES 加密流程大致如下图所示，其中右半部分的每轮 48 位子密钥由密钥编排算法生成：



3.1.2 代码实现

我使用了函数式编程方式，将 DES 加密的每一个步骤封装成一个函数，下面将展示各步骤具体实现代码，而其它基本构件（盒子）定义见附录完整代码。

（1）预处理，保证密钥的长度为 64 位、明文的长度能够完整的分组

```
# 预处理保证输入的密钥长度位为64位
def key_process(key):
    # 小于64位补0
    if len(key) < 64:
        key += '0' * (64 - len(key))
    # 大于64位截断
    elif len(key) > 64:
        key = key[:64]
    return key

# 预处理保证输入的明文长度位数为num的倍数, 否则在后面补0
def message_process(message, num):
    # 不是num的倍数, 则补0
    if len(message) % num != 0:
        message += '0' * (num - (len(message) % num))
    return message
```

（2）密钥编排算法：PC-1 盒置换

```
def pc_1(key):
    res = ''
    for i in pc1_:
        res += key[i - 1]
    return res
```

（3）循环左移

```
# 循环左移num位
def rol(key, num):
    return key[num:] + key[:num]
```

（4）密钥编排算法：PC-2 盒置换

```
def pc_2(key):
    res = ''
    for i in pc2_:
        res += key[i - 1]
    return res
```

(5) 16 轮迭代得到 16 个子密钥

```
def get_key(key):
    key_list = []
    pc1_output = pc_1(key) # pc1置换
    c = pc1_output[:28]
    d = pc1_output[28:]
    # 循环左移16轮迭代
    for i in left_shift_:
        c = rol(c, i)
        d = rol(d, i)
        key = pc_2(c + d) # pc2置换得到一个密钥
        key_list.append(key)
    return key_list
```

(6) 初始 IP 置换

```
def ip(message):
    res = ''
    for i in ip_:
        res += message[i - 1]
    return res
```

(7) 末尾 IP 逆置换

```
def ip_inverse(message):
    res = ''
    for i in inverse_ip_:
        res += message[i - 1]
    return res
```

(8) 扩展置换 (E 盒)

```
def e_box(message):
    res = ''
    for i in e_box_:
        res += message[i - 1]
    return res
```

(9) 密钥加

```
def xor(message, key):
    res = ''
    for i in range(len(message)):
        res += str(int(message[i]) ^ int(key[i]))
    return res
```

(10) 代换盒 (S 盒)

```
def s_box(message):
    res = ''
    box_index = 0
    for i in range(0, len(message), 6):
        block = message[i:i + 6]
        row = int(block[0] + block[5], 2)
        col = int(block[1] + block[2] + block[3] + block[4], 2)
        out = bin(s_box_[box_index][row][col])[2:]
        if len(out) != 4:
            out = '0' * (4 - len(out)) + out
        res += out
        box_index += 1
    return res
```

(11) 置换运输 (P 盒)

```
def p_box(message):
    res = ''
    for i in p_box_:
        res += message[i - 1]
    return res
```

(12) F 函数

```
def f_func(message, key):
    e_out = e_box(message) # 扩展置换
    xor_out = xor(e_out, key) # 密钥加
    s_out = s_box(xor_out) # 代换盒
    p_out = p_box(s_out) # 置换运算
    return p_out
```

(13) DES 加密函数

```
def des_encrypt(message, key):
    message = message_process(message, 64)
    key = key_process(key)
    key_list = get_key(key)
    ip_out = ip(message)
    left_part = ip_out[:32]
    right_part = ip_out[32:]
    for i in range(15):
        left_temp = left_part
        right_temp = right_part
        left_part = right_part
        right_part = xor(left_temp, f_func(right_temp, key_list[i]))
    left_last = xor(left_part, f_func(right_part, key_list[15]))
    right_last = right_part
    cipher = ip_inverse(left_last + right_last)
    return cipher
```

(14) DES 解密函数

```
def des_decrypt(cipher, key):
    key = key_process(key)
    key_list_inverse = get_key(key)[::-1]
    ip_out = ip(cipher)
    left_part = ip_out[:32]
    right_part = ip_out[32:]
    for i in range(15):
        left_temp = left_part
        right_temp = right_part
        left_part = right_part
        right_part = xor(left_temp, f_func(right_temp, key_list_inverse[i]))
    left_last = xor(left_part, f_func(right_part, key_list_inverse[15]))
    right_last = right_part
    message = ip_inverse(left_last + right_last)
    return message.strip("\x00")
```

(15) 进制转换函数

```
# 字符串转2进制
def str2bin(text):
    res = ''
    for i in text:
        tmp = bin(ord(i))[2:].zfill(8)
        res += tmp
    return res

# 2进制转字符串
def bin2str(bin_text):
    res = ""
    tmp = re.findall(r'.{8}', bin_text)
    for i in tmp:
        res += chr(int(i, 2))
    return res

# 字符串转16进制
def str2hex(text):
    res = ''
    for ch in text:
        tmp = hex(ord(ch))[2:].zfill(2)
        res += tmp
    return res

# 16进制转字符串
def hex2str(hex_text):
    res = ''
    tmp = re.findall(r'.{2}', hex_text)
    for i in tmp:
        res += chr(int('0x' + i, 16))
    return res
```

3.1.3 正确型验证与性能分析

(1) 正确性验证

与其它块密码相似，DES 自身并不是加密的实用手段，而必须以某种工作模式进行实际操作，在下一节中我将使用 5 种工作模式分别对 DES 进行加解密并验证，所以这里仅对 64 位的明文进行简单的验证。

① 加密

```
Please input the message:
IamYuan.
Please input the key:
cumt1234
The cipher is:
0010111111111101100100011001101011100100100010010000111110101101
```

② 解密

```
Please input the cipher:
0010111111111101100100011001101011100100100010010000111110101101
Please input the key:
cumt1234
The message is:
IamYuan.
```

(2) 性能分析

本节种暂时不使用工作模式，单纯的用已实现的 DES 和 RC4 分别用密钥“cumt1234”对明文“IamYXJ.”加密解密 10000 次，结果如下：

① RC4

```
import time
message = 'IamYXJ.'
key = 'cumt1234'
start = time.time()
for i in range(10000):
    cipher = encrypt(message, key)
end = time.time()
print("encrypt spend time: " + str(end - start) + " seconds.")
start = time.time()
for i in range(10000):
    message = decrypt(cipher, key)
end = time.time()
print("decrypt spend time: " + str(end - start) + " seconds.")
```

时间耗费：

```
encrypt spend time: 0.943678617477417 seconds.
decrypt spend time: 0.963029146194458 seconds.
```

② DES

```

import time
message = str2bin("IamYXJ.")
key = str2bin("cunt1234")
start = time.time()
for i in range(10000):
    cipher = des_encrypt(message, key)
end = time.time()
print("Encrypt spend time: " + str(end - start) + " seconds.")
start = time.time()
for i in range(10000):
    message = des_decrypt(cipher, key)
end = time.time()
print("Decrypt spend time: " + str(end - start) + " seconds.")

```

时间耗费：

Encrypt spend time: 10.657340288162231 seconds.

Decrypt spend time: 10.531631231307983 seconds.

结果如下：

算法\时间	加密（秒）	解密（秒）
RC4	0.94368	0.96303
DES	10.65734	10.53163

可以看出：

由于 DES 的加解密过程也是相似的，因此其加解密所需的时间大致相等；除此之外，RC4 比 DES 快了 10 左右，与在网上查阅的速度对比情况一致，毕竟 DES 的加解密步骤要比 RC4 复杂许多。

3.1.4 安全性与可用性分析

（1）安全性

① 互补性：利用这一性质，如果用某个密钥加密一个明文分组得到一个密文分组，那么用该密钥的补密钥加密该明文分组的补便会得到该密文分组的补。互补性会使 DES 在选择明文攻击下所需的工作量减半，仅需要测试 2^{56} 个密钥的一半，及 2^{55} 个密钥。

② 弱密钥：DES 的解密过程需要用到 56 位初始密钥生成的 16 个子密钥，如果给定初始密钥 K，经过子密钥生成器得到的各个子密钥都相同，即有 $K_1 = K_2 = \dots = K_{16}$ ，则称给定的初始密钥 K 为弱密钥。如果 K 为弱密钥，则对任意的 64 位数据 M，有

$$E_K(E_K(M)) = M, (D_K(D_K(M)) = M$$

这说明以 K 对 M 加密两次或解密两次相当于恒等映射，结果仍是 M，这也意味着加密运算和解密运算没有区别。除此之外，还有半弱密钥、四分之一弱密钥和八分之一弱密钥。

③ 密钥空间大小：随着现代计算机运算能力的不断提升，DES 实际 56 位的密钥已经远远不够了，甚至可以用穷举法进行攻击。

(2) 可用性

尽管 DES 自诞生起就经历了许多很强的分析攻击，但是至今并未发现一种能够高效破解它的攻击方式。不过，由于现代计算能力利用穷举法就可能容易的破解单重 DES。因此，对大多数实际应用场景来说，单重 DES 都已经不在适用。

3.1.5 破解或攻击方式分析

除了最传统的暴力破解之外，还有如下两种已知方法可以对 DES 进行攻击：

- ① 差分分析，即系统地研究明文中的一个细小变化是如何影响密文的。
- ② 线性分析，即寻找明文、密文和密钥间的有效线性逼近，当该逼近的线性偏差足够大时，就可以由一定量的明密文对推测出部分密钥信息。

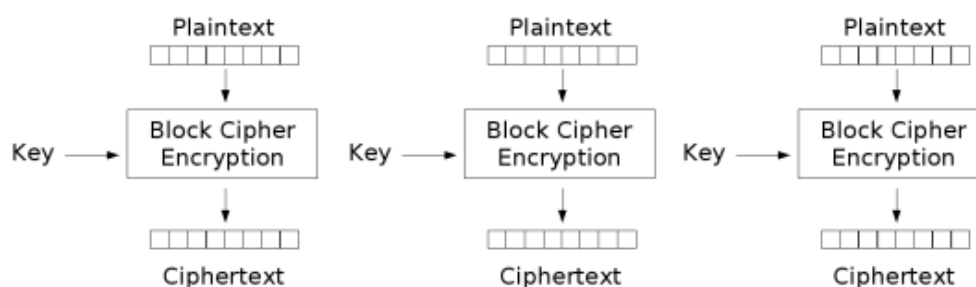
3.2 分组密码的工作模式

本节在上一节的基础上，分别实现了在 ECB、CBC、CFB、OFB、CTR 五种工作模式下的 DES 加解密，并验证正确性，然后再对这五种工作模式的可用性、安全性与性能效率进行分析。

3.2.1 电子密码本模式 (ECB)

3.2.1.1 原理分析

这是分组加密最简单的一种模式，即明文的每一个块加密成密文的每一个块。明文被分为若干块($M_1, M_2, M_3, M_4, \dots, M_n$)，通过加密方法 E_k ，得到密文($C_1, C_2, C_3, C_4, \dots, C_n$)，当最后一个明文分组小于分组长度时，需要用一些特定的数据进行填充。



Electronic Codebook (ECB) mode encryption

3.2.1.2 DES-ECB 代码实现

① 加密

```
def des_ecb_encrypt(message, key, output_mode='h'):
    cipher = ''
    message_bin = message_process(str2bin(message), 64)
    key_bin = str2bin(key)
    message_list = re.findall(r'.{64}', message_bin)
    cipher_bin = ''
    for i in message_list:
        cipher_bin += des_encrypt(i, key_bin)
    if output_mode == 'H' or output_mode == 'h':
        cipher = str2hex(bin2str(cipher_bin))
    elif output_mode == 'B' or output_mode == 'b':
        cipher = base64.b64encode(bin2str(cipher_bin).encode('latin')).decode('latin')
    return cipher
```

② 解密

```
def des_ecb_decrypt(cipher, key, input_mode='h'):
    if input_mode == 'h' or input_mode == 'H':
        cipher_bin = str2bin(hex2str(cipher))
    elif input_mode == 'b' or input_mode == 'B':
        cipher_bin = str2bin(base64.b64decode(cipher).decode('latin'))
    key_bin = str2bin(key)
    cipher_bin = ''
    cipher_list = re.findall(r'.{64}', cipher_bin)
    message_bin = ''
    for i in cipher_list:
        message_bin += des_decrypt(i, key_bin)
    message = bin2str(message_bin).strip("\x00")
    return message
```

3.2.1.3 正确性验证

(1) 加密

使用密钥“crypto”对明文使用 ECB 工作模式进行加密，并以 Base64 编码输出如下：

```
Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
E
Please input the message:
My name is YuanXiaojian,I am from CUMT.
Please input the key:
crypto
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
1
Please choose the output way:
[B]Base64 [H]Hex
B
The cipher is:
BAz5N6fGAIJE93/Vrug6a8Htiy8C6TnoB9f9J4EK2De3iqNcVQCbmQ==
```

可以看到输出结果与在线加密网站对应模式下所得结果一致：

DES加密模式: ECB ▼ 填充: zeropadding ▼ 密码: crypto 偏移量: iv偏移量, ecb模: 输出: base64 ▼

字符集: utf8编码 (unicode编码) ▼

待加密、解密的文本:  

My name is YuanXiaojian,I am from CUMT.

↑ 将你电脑文件直接拖入试试 ^-^

DES加密 DES解密

DES加密、解密转换结果(base64了):   

BAz5N6fGAIJE93/Vrug6a8Htiy8C6TnoB9f9J4EK2De3iqNcVQCbmQ==

(2) 解密

对上述所得密文进 ECB 模式的解密, 成功得到相应明文:

```
Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
D
Please input the cipher:
BAz5N6fGAIJE93/Vrug6a8Htiy8C6TnoB9f9J4EK2De3iqNcVQCbmQ==
Please input the key:
crypto
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
1
Please choose your input way:
[B]Base64 [H]Hex
B
The message is:
My name is YuanXiaojian,I am from CUMT.
```

3.2.1.4 可用性与安全性分析

(1) 可用性

ECB 模式拥有很多优点, 加密方和解密方之间的分组同步不是必须的, 每个明文分组可以独立地进行加密。ECB 还具有良好的差错控制, 位错误仅仅影响对应的分组, 其对后面的分组没有任何影响。所以, ECB 特别适合数据随机且较少的情况, 比如加密密钥。

(2) 安全性

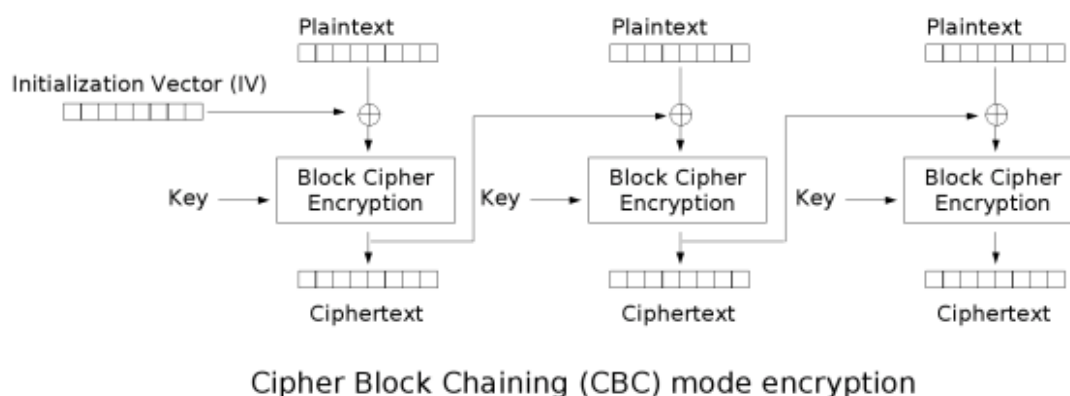
ECB 模式最大的问题在于它的加密是高度确定的, 因此只要密钥不变, 相同的明文分组总

是产生相同的密文分组，这也就使得攻击者容易实现统计分析攻击、分组重放攻击和代换攻击。在许多实际情形中，尤其是消息较长的时候，ECB 可能是不安全的。

3.2.2 密码分组链接模式（CBC）

3.2.2.1 原理分析

在 CBC 模式中，每个明文块先与前一个密文块进行异或后，再进行加密。在这种方法中，每个密文块都依赖于它前面的所有明文块。同时，为了保证每条消息的唯一性，在第一个块中需要使用初始化向量。



3.2.2.2 DES-CBC 代码实现

① 加密

```
def des_cbc_encrypt(message, key, iv, output_mode='h'):
    cipher = ''
    key_bin = str2bin(key)
    iv_bin = key_process(str2bin(iv))
    message_bin = message_process(str2bin(message), 64)
    message_list = re.findall(r'.{64}', message_bin)
    cipher_bin = ''
    # 第一组的反馈即为iv
    c = iv_bin
    for i in message_list:
        # 与反馈进行异或
        m = xor(c, i)
        c = des_encrypt(m, key_bin)
        # 链接密文分组
        cipher_bin += c
    if output_mode == 'H' or output_mode == 'h':
        cipher = str2hex(bin2str(cipher_bin))
    elif output_mode == 'B' or output_mode == 'b':
        cipher = base64.b64encode(bin2str(cipher_bin).encode('latin')).decode('latin')
    return cipher
```

② 解密

```

def des_cbc_decrypt(cipher, key, iv, input_mode='h'):
    cipher_bin = ''
    key_bin = str2bin(key)
    iv_bin = key_process(str2bin(iv))
    if input_mode == 'h' or input_mode == 'H':
        cipher_bin = str2bin(hex2str(cipher))
    elif input_mode == 'b' or input_mode == 'B':
        cipher_bin = str2bin(base64.b64decode(cipher).decode('latin'))
    # 对密文进行分组
    cipher_list = re.findall(r'.{64}', cipher_bin)
    message_bin = ''
    for i in cipher_list:
        temp = des_decrypt(i, key_bin)
        m = xor(iv_bin, temp)
        message_bin += m
        # 保存当前密文分组,用于下一个分组的解密异或
        iv_bin = i
    message = bin2str(message_bin).strip("\x00")
    return message

```

3.2.2.3 正确性验证

① 加密

使用 “crypto” 作为密钥，“06172151” 作为初始化向量 IV，对明文使用 CBC 工作模式进行加密，并以 Base64 编码输出如下：

```



Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
E
Please input the message:
My name is YuanXiaojian,I am from CUMT.
Please input the key:
crypto
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
2
Please input the IV:
06172151
Please choose the output way:
[B]Base64 [H]Hex
B
The cipher is:
D939SjRkRm1gGF4z9xwKoBSqUkXzv8TUtcfytxOMwrqO6BA+p2Rpyg==

```

可以看到输出结果与在线加密网站对应模式下所得结果一致：

DES加密模式: CBC 填充: zeropadding 密码: crypto 偏移量: 06172151 输出: base64



字符集: utf8编码 (unicode编码)

待加密、解密的文本:  

My name is YuanXiaojian,I am from CUMT.

↑ 将你电脑文件直接拖入试试^-^

DES加密 DES解密

DES加密、解密转换结果(base64了):   

D939SjRkRm1gGF4z9xwKoBSqUkXzv8TUtcfytxOMwrqO6BA+p2Rpyg==

② 解密

对上述所得密文进行 CBC 模式的解密,解密时同时需要密钥和 IV,成功得到相应明文:

```
Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
D
Please input the cipher:
D939SjRkRm1gGF4z9xwKoBSqUkXzv8TUtcfytxOMwrqO6BA+p2Rpyg==
Please input the key:
crypto
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
2
Please input the IV:
06172151
Please choose your input way:
[B]Base64 [H]Hex
B
The message is:
My name is YuanXiaojian,I am from CUMT.
```

3.2.2.4 可用性与安全性分析

(1) 可用性

CBC 模式的没有明文错误扩散,密文错误扩散很小,是最为常用的工作模式,确保互联网安全的通信协议之一 IPsec, 就是使用 CBC 模式来确保通信机密性的,此外, CBC 模式还被用于 Kerberos version 5 的认证系统中。

(2) 安全性

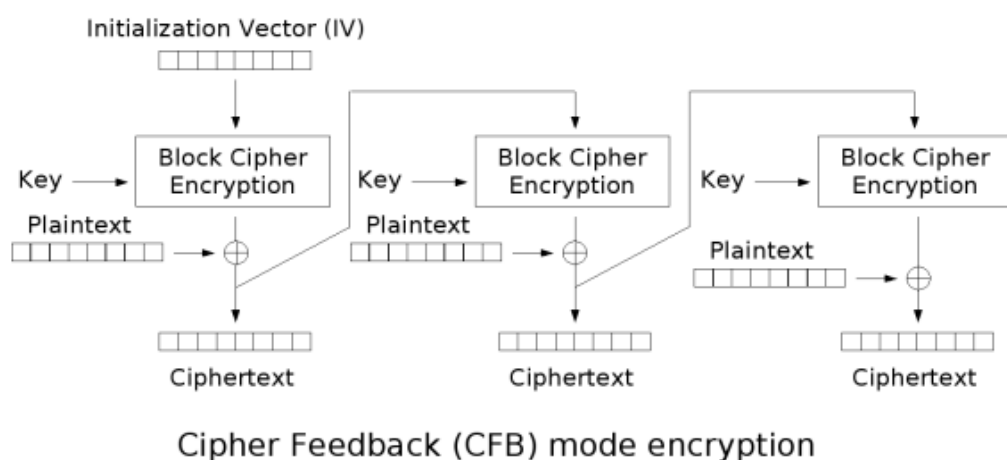
由于引入了反馈, CBC 可以将重复的明文分组加密成不同的密文,克服了 ECB 的弱点,并且使用了随机化向量 IV,有效防止了重放攻击,但是前提是 IV 应该同密钥一样被加以

保护，否则攻击者可以欺骗接收者，让他使用不同的 IV，从而使接收者解密出第一块明文分组的某些位取反。

3.2.3 密码反馈模式（CFB）

3.2.3.1 原理分析

CFB 模式可以看作是一种使用分组密码来实现流密码的方式，CFB 模式中由密码算法所生成的比特序列称为密钥流。在 CFB 模式中，密码算法就相当于用来生成密钥流的伪随机数生成器，而初始化向量就相当于伪随机数生成器的“种子”，它的明文数据可以被逐比特加密。



3.2.3.2 DES-CFB 代码实现

① 加密

```
def des_cfb_encrypt(message, key, iv, output_mode='h'):
    cipher = ''
    key_bin = str2bin(key)
    iv_bin = key_process(str2bin(iv))
    message_bin = message_process(str2bin(message), 8)
    message_list = re.findall(r'.{8}', message_bin)
    cipher_bin = ''
    # 初始化移位寄存器
    reg = iv_bin
    for i in message_list:
        # 对寄存器64位进行加密
        enc_out = des_encrypt(reg, key_bin)
        # 选择加密结果的前8位与明文分组异或
        c = xor(i, enc_out[:8])
        # 寄存器左移8位,并在最右边填充前一密文分组
        reg = reg[8:] + c
        cipher_bin += c
    if output_mode == 'H' or output_mode == 'h':
        cipher = str2hex(bin2str(cipher_bin))
    elif output_mode == 'B' or output_mode == 'b':
        cipher = base64.b64encode(bin2str(cipher_bin).encode('latin')).decode('latin')
    return cipher
```

② 解密

```
def des_cfb_decrypt(cipher, key, iv, input_mode='h'):
    cipher_bin = ''
    key_bin = str2bin(key)
    iv_bin = key_process(str2bin(iv))
    if input_mode == 'h' or input_mode == 'H':
        cipher_bin = str2bin(hex2str(cipher))
    elif input_mode == 'b' or input_mode == 'B':
        cipher_bin = str2bin(base64.b64decode(cipher).decode('latin'))
    cipher_list = re.findall(r'.{8}', cipher_bin)
    message_bin = ''
    reg = iv_bin
    for i in cipher_list:
        # 对移位寄存器进行加密
        enc_out = des_encrypt(reg, key_bin)
        # 选择加密结果的前8位与密文分组异或
        m = xor(i, enc_out[:8])
        # 寄存器左移8位,并在最右边填充前一密文分组
        reg = reg[8:] + i
        message_bin += m
    message = bin2str(message_bin).strip("\x00")
    return message
```

3.2.3.3 正确性验证

① 加密



使用 “crypto” 作为密钥，“06172151” 作为初始化向量 IV，对明文使用 CFB 工作模式进行加密，并以 Base64 编码输出如下：

```
Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
E
Please input the message:
My name is YuanXiaojian,I am from CUMT.
Please input the key:
crypto
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
3
Please input the IV:
06172151
Please choose the output way:
[B]Base64 [H]Hex
B
The cipher is:
luuWIIZNGqwO/EZ71F3Epw8jEPE8FycapmhEJOLfNVAQNtuH1Q81
```

可以看到输出结果与在线加密网站对应模式下所得结果一致（最后一个字符的区别为换行符的区别，对结果无影响）：

DES加密模式: CFB ▼ 填充: zeropadding ▼ 密码: crypto 偏移量: 06172151 输出: base64 ▼




utf8编码 (unicode编码) ▼

待加密、解密的文本:  

My name is YuanXiaojian,I am from CUMT.

↑ 将你电脑文件直接拖入试试^^^

DES加密 DES解密

DES加密、解密转换结果(base64了):   

luuWIIZNGqwO/EZ71F3Epw8jEPE8FycapmhEJOLfNVAQNtuH1Q81EA==

② 解密

对上述所得密文进行 CFB 模式的解密，解密时同时需要密钥和 IV，成功得到相应明文：

```
Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
D
Please input the cipher:
luuWIIZNGqwO/EZ71F3Epw8jEPE8FycapmhEJOLfNVAQNtuH1Q81
Please input the key:
crypto
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
3
Please input the IV:
06172151
Please choose your input way:
[B]Base64 [H]Hex
B
The message is:
My name is YuanXiaojian,I am from CUMT.
```

3.2.3.4 可用性与安全性分析

(1) 可用性

CFB 模式不会直接加密明文，只是使用密文与明文进行 XOR 来获得密文。所以在这种模式下，它不需要填充数据，以将块密码变为自同步的流密码，且具有有限的错误传播，这种可以进行实时流操作的模式，在某些情境下具有很大的优势，如面向数据流的通用传输及认证。

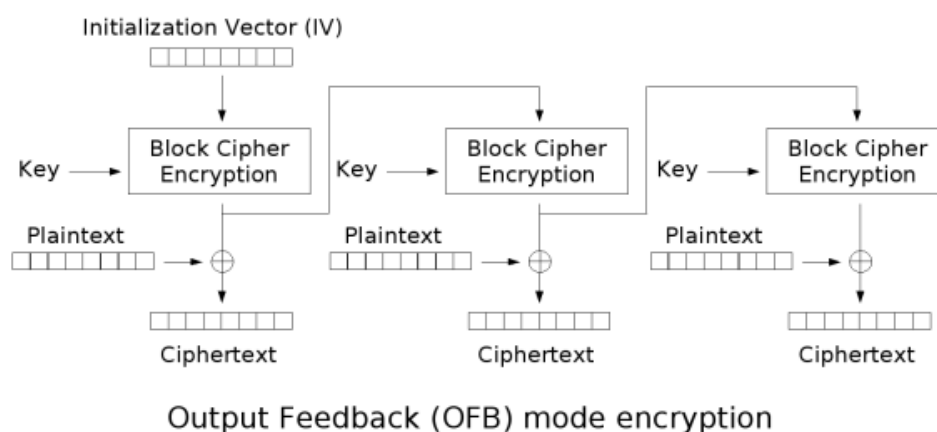
(2) 安全性

CFB 模式可以通过重放攻击进行攻击。例如，如果使用其他密文替换新的密文，则用户将获得错误的数据,但他不会知道数据是错误的。为确保安全性，需要为每 $2^{\frac{n+1}{2}}$ 个加密块更改此模式下的密钥。

3.2.4 输出反馈模式（OFB）

3.2.4.1 原理分析

在 OFB 模式中，密码算法的输出会反馈到密码算法的输入中。OFB 模式不是通过密码算法对明文直接进行加密的，而是通过将明文分组和密码算法的输出进行 XOR 来产生密文分组的。



3.2.4.2 DES-OFB 代码实现

① 加密

```
def des_ofb_encrypt(message, key, iv, output_mode='h'):
    cipher = ''
    key_bin = str2bin(key)
    iv_bin = key_process(str2bin(iv))
    message_bin = message_process(str2bin(message), 8)
    message_list = re.findall(r'.{8}', message_bin)
    cipher_bin = ''
    # 初始化移位寄存器
    reg = iv_bin
    for i in message_list:
        # 对寄存器64位进行加密
        enc_out = des_encrypt(reg, key_bin)
        # 选择加密结果的前8位与明文分组异或
        c = xor(i, enc_out[:8])
        # 寄存器左移8位,并在最右边填充加密结果的前8位
        reg = reg[8:] + enc_out[:8]
        cipher_bin += c
    if output_mode == 'H' or output_mode == 'h':
        cipher = str2hex(bin2str(cipher_bin))
    elif output_mode == 'B' or output_mode == 'b':
        cipher = base64.b64encode(bin2str(cipher_bin).encode('latin')).decode('latin')
    return cipher
```

② 解密

```
def des_ofb_decrypt(cipher, key, iv, input_mode='h'):
    cipher_bin = ''
    key_bin = str2bin(key)
    iv_bin = key_process(str2bin(iv))
    if input_mode == 'h' or input_mode == 'H':
        cipher_bin = str2bin(hex2str(cipher))
    elif input_mode == 'b' or input_mode == 'B':
        cipher_bin = str2bin(base64.b64decode(cipher).decode('latin'))
    cipher_list = re.findall(r'.{8}', cipher_bin)
    message_bin = ''
    # 初始化移位寄存器
    reg = iv_bin
    for i in cipher_list:
        # 对移位寄存器进行加密
        enc_out = des_encrypt(reg, key_bin)
        # 选择加密结果的前8位与密文分组异或
        m = xor(i, enc_out[:8])
        # 寄存器左移8位,并在最右边填充加密结果的前8位
        reg = reg[8:] + enc_out[:8]
        message_bin += m
    message = bin2str(message_bin).strip("\x00")
    return message
```

3.2.4.3 正确性验证

① 加密

使用 “crypto” 作为密钥，“06172151” 作为初始化向量 IV，对明文使用 OFB 工作模式进行加密，并以 Base64 编码输出如下：

```
Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
E
Please input the message:
My name is YuanXiaojian,I am from CUMT.
Please input the key:
crypto
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
4
Please input the IV:
06172151
Please choose the output way:
[B]Base64 [H]Hex
B
The cipher is:
1tZiY2CTSYFKeDvtgkZmNJ6n6B1jZ73oI4Jma654vPA2/RrdmAIv
```

可以看到输出结果与在线加密网站对应模式下所得结果一致（最后一个字符的区别为换行符的区别，对结果无影响）：

DES加密模式: OFB 填充: zeropadding 密码: crypto 偏移量: 06172151 输出: base64

utf8编码 (unicode编码)

待加密、解密的文本:  

My name is YuanXiaojian,I am from CUMT.

↑ 将你电脑文件直接拖入试试^-^

DES加密 DES解密

DES加密、解密转换结果(base64了):   

1tZlY2CTSYFKeDVtgkZmNJ6n6BljZ73oI4Jma654vPA2/RrdmAIvUA==

② 解密

对上述所得密文进行 OFB 模式的解密，解密时同时需要密钥和 IV，成功得到相应明文：

```

Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
D
Please input the cipher:
ltZlY2CTSYFKeDVtgkZmNJ6n6BljZ73oI4Jma654vPA2/RrdmAIvUA==
Please input the key:
crypto
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
4
Please input the IV:
06172151
Please choose your input way:
[B]Base64 [H]Hex
B
The message is:
My name is YuanXiaojian,I am from CUMT.

```

3.2.4.4 可用性与安全性分析

(1) 可用性

OFB 模式与 CFB 模式相似，但是将前一次加密产生的 s 比特分组中非密文分组送入移位寄存器的最右边，是一种“内部反馈”机制，其优点是传输过程中密文在某位上发生的错误不会影响解密后明文的其他未，但是在 OFB 中，失去同步是致命的。

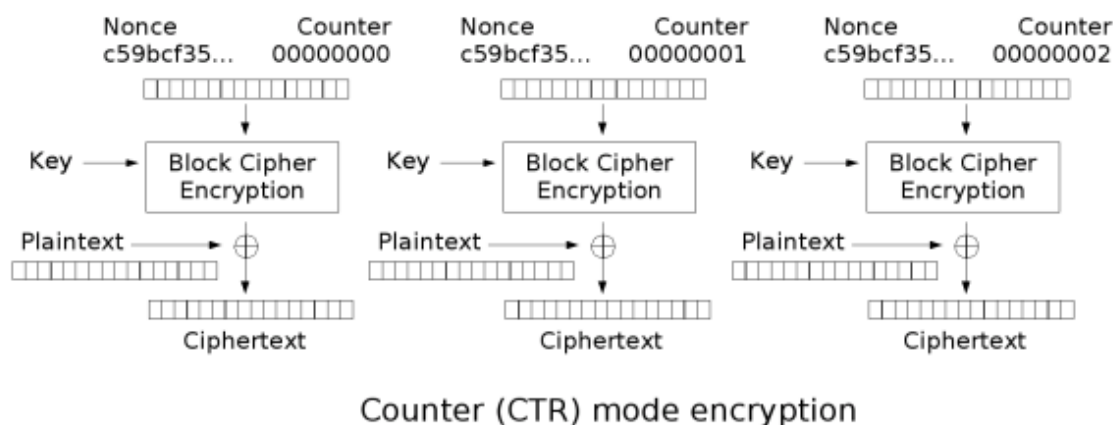
(2) 安全性

OFB 抗消息流篡改攻击的能力不如 CFB，因为密文中的某位取反，恢复出的明文的相应位也取反，所以攻击者有办法控制恢复明文的改变。这也，攻击者可以根据消息的改动而改动校验和，以使改动不被纠错码发现。

3.2.5 计数器模式 (CTR)

3.2.2.1 原理分析

CTR 模式是一种通过将逐次累加的计数器进行加密来生成密钥流的流密码。CTR 模式中，每个分组对应一个逐次累加的计数器，并通过对计数器进行加密来生成密钥流。也就是说最终的密文分组是通过将计数器加密得到的比特序列，与明文分组进行 XOR 而得到的。



3.2.2.2 DES-CTR 代码实现

① 加密

```
def des_ctr_encrypt(message, key, nonce, output_mode='h'):
    cipher = ''
    key_bin = str2bin(key)
    nonce_bin = key_process(str2bin(nonce))
    message_bin = message_process(str2bin(message), 64)
    message_list = re.findall(r'.{64}', message_bin)
    cipher_bin = ''
    # 初始化计数器
    counter = nonce_bin
    for i in message_list:
        enc_out = des_encrypt(counter, key_bin)
        cipher_bin += xor(i, enc_out)
        # 计数器加1(mod 2^64)
        counter = bin(int('0b' + counter, 2) + int('0b1', 2) % pow(2, 64))[2:].zfill(64)
    if output_mode == 'H' or output_mode == 'h':
        cipher = str2hex(bin2str(cipher_bin))
    elif output_mode == 'B' or output_mode == 'b':
        cipher = base64.b64encode(bin2str(cipher_bin).encode('latin')).decode('latin')
    return cipher
```

② 解密

```
def des_ctr_decrypt(cipher, key, nonce, input_mode='h'):
    cipher_bin = ''
    key_bin = str2bin(key)
    nonce_bin = key_process(str2bin(nonce))
    if input_mode == 'h' or input_mode == 'H':
        cipher_bin = str2bin(hex2str(cipher))
    elif input_mode == 'b' or input_mode == 'B':
        cipher_bin = str2bin(base64.b64decode(cipher).decode('latin'))
    cipher_list = re.findall(r'.{64}', cipher_bin)
    message_bin = ''
    # 初始化计数器
    counter = nonce_bin
    for i in cipher_list:
        enc_out = des_encrypt(counter, key_bin)
        message_bin += xor(i, enc_out)
        # 计数器加1(mod 2^64)
        counter = bin(int('0b' + counter, 2) + int('0b1', 2) % pow(2, 64))[2:].zfill(64)
    message = bin2str(message_bin).strip("\x00")
    return message
```

3.2.2.3 正确性验证

① 加密

使用 “crypto” 作为密钥，“06172151” 作为初始化随机数 nonce，对明文使用 CTR 工作模式进行加密，并以 Base64 编码输出如下：

```
Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
E
Please input the message:
My name is YuanXiaojian,I am from CUMT.
Please input the key:
crypto
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
5
Please input the nonce:
06172151
Please choose the output way:
[B]Base64 [H]Hex
B
The cipher is:
lpnfkiAvAQnksE0Ibv0/1Yyntj3twy0NG3Kzop78tkZep8QVvDBMGA==
```

可以看到输出结果与在线加密网站对应模式下所得结果一致（最后一个字符的区别为换行符的区别，对结果无影响）：

DES加密模式: CTR ▼ 填充: zeropadding ▼ 密码: crypto 偏移量: 06172151 输出: base64
utf8编码 (unicode编码) ▼
待加密、解密的文本:  
My name is YuanXiaojian,I am from CUMT.
↑ 将你电脑文件直接拖入试试^^
DES加密 DES解密
DES加密、解密转换结果(base64了):   
lpnfkiAvAQnksE0Ibv0/1Yyntj3twy0NG3Kzop78tkZep8QVvDBMGA==

② 解密

对上述所得密文进行 CTR 模式的解密，解密时同时需要密钥和初始化随机数 `nonce`，成功得到相应明文：

```
Please choose a mode:
[E]Encrypt [D]Decrypt [Q]Quit
D
Please input the cipher:
lpnfkiAvAQnksE0Ibv0/1Yyntj3twy0NG3Kzop78tkZep8QVvDBMGA==
Please input the key:
crypto
Please choose a working mode:
[1]ECB [2]CBC [3]CFB [4]OFB [5]CTR
5
Please input the nonce:
06172151
Please choose your input way:
[B]Base64 [H]Hex
B
The message is:
My name is YuanXiaojian,I am from CUMT.
```

3.2.2.4 可用性与安全性分析

(1) 可用性

CTR 模式的加密和解密使用了完全相同的结构，因此在软硬件实现上比较容易。

此外，CTR 模式中可以以任意顺序对分组进行加密和解密，能够以任意顺序处理分组，就

意味着能够实现并行计算。在支持并行计算的系统中，CTR 模式的速度是非常快的，具有很大的优势。

(2) 安全性

CTR 也具有一定的可证明安全性，能够证明 CTR 至少和上述 4 中工作模式一样安全。

3.2.6 各工作模式效率分析与对比

(1) 五种工作模式效率分析

下面对五种工作模式进行效率上的分析，使用 “My name is YuanXiaojian, I am from CUMT.” 并重复 2000 次来作为明文，使用 “crypto” 作为密钥，使用 “06172151” 作为 CBC、CFB、OFB 模式的初始向量 IV、CTR 模式的初始随机数 nonce。

则编写测试程序计算得到五种模式加解密所需时间结果如下：

ECB-Encrypt spend time: 11.349847078323364 seconds.

ECB-Decrypt spend time: 16.901703596115112 seconds.

CBC-Encrypt spend time: 11.364411354064941 seconds.

CBC-Decrypt spend time: 16.808470010757446 seconds.

CFB-Encrypt spend time: 85.59333181381226 seconds.

CFB-Decrypt spend time: 129.23327946662903 seconds.

OFB-Encrypt spend time: 86.61665058135986 seconds.

OFB-Decrypt spend time: 130.75028228759766 seconds.

CTR-Encrypt spend time: 11.776122808456421 seconds.

CTR-Decrypt spend time: 17.96898627281189 seconds.

如下：

工作模式\时间	加密（秒）	解密（秒）
ECB	11.34984	16.90170
CBC	11.36441	16.80847
CFB	85.59333	129.23328
OFB	86.61665	130.75028
CTR	11.77612	17.96899

由上述实验过程，得到如下总结：

- ① 五种模式下解密时间都要略长于加密时间。
- ② ECB、CBC、CTR 模式加解密所需要的时间大致相等，效率相对较高。
- ③ CFB、OFB 模式加解密所需要的时间大致相等，但是要大于其他三种模式，效率相对较低。

- ④ ECB 模式实现最为简单，但是安全性也最弱。
- ⑤ CBC 模式效率高，同时也具有一定的安全性保障，使用最为广泛。
- ⑥ CFB 与 OFB 模式虽然加解密的效率较低，但是它们以流的方式进行加密解密，在某些特殊的场景具有特别的优势。
- ⑦ 明文不易丢信号，对明文格式没有特殊要求的环境可选用 CBC 模式，需要完整性认证功能时可选用该模式；容易丢信号，或对明文格式有特殊要求的环境，可选用 CFB 模式；不易丢信号，但信号特别容易错、明文冗余特别多，可选用 OFB 模式。
- ⑧ CTR 效率较高，能并行处理，且只要求实现加密而不要求实现解密，在使用像 AES 这样加密和解密不同的算法时，更能体现 CTR 的简单性。

(3) 与官方库 pyDES 效率对比

使用 python 官方 DES 库 pyDES 编写 DES 加解密程序，使用 “My name is YuanXiaojian, I am from CUMT.” 重复 5000 次来作为明文、“crypto” 作为密钥、“06172151” 作为初始化向量，在 CBC 模式下进行加解密，代码如下：

```
from pyDes import *
import base64
import time

def DesEncrypt(str, key, iv):
    Des_Key = (key+"0000")[0:8]
    k = des(Des_Key, CBC, iv, pad=None, padmode=PAD_PKCS5)
    EncryptStr = k.encrypt(str)
    return base64.b64encode(EncryptStr)

def DesDecrypt(str, key, iv):
    Des_Key = (key+"0000")[0:8]
    EncryptStr = base64.b64decode(str)
    print(EncryptStr)
    k = des(Des_Key, CBC, iv, pad=None, padmode=PAD_PKCS5)
    DecryptStr = k.decrypt(EncryptStr)
    return DecryptStr

message = 'My name is YuanXiaojian, I am from CUMT.' * 5000
key = 'crypto'
iv = '06172151'
start = time.time()
c = DesEncrypt(message, key, iv)
end = time.time()
print("Encrypt spend time: " + str(end - start) + " seconds.")
start = time.time()
M = DesDecrypt(c, key, iv)
end = time.time()
print("Decrypt spend time: " + str(end - start) + " seconds.")
```


结果:

```
Encrypt spend time: 12.464011192321777 seconds.  
Decrypt spend time: 16.10939884185791 seconds.
```

再使用自己的之前的 CBC 加解密代码对同一明文使用相同的密钥和初始向量 IV 进行加解密, 最终结果如下:

代码\时间	加密 (秒)	解密 (秒)
pyDes	12.46401	16.10939
myDes	28.63284	41.85360

由上表可以看到, 官方的库在加解密的效率上比自己实现的代码要高不少, 分析原因可能如下:

- ① 我在实验过程中, 完全根据各个步骤的原理进行编程实现, 没有考虑算法的优化与改进; 而官方库则采取了很多优化手段。
- ② 官方库的底层代码使用 C 语言实现的, 而本实验中的代码为纯 python 语言实现的, 这两种语言分别为编译型语言和解释性语言, 在效率与性能上本身就存在着非常大的差距。

3.4 实验总结

3.4.1 存在的问题

(1) 刚开始编写 DES 代码的时候, 发现很多时候明文并不能够正好分组, 而书中并没有提及对应的解决方式。经过查阅资料, 实际上存在着许多种 Padding, 我在实验中选择了 ZeroPadding, 即用 0 进行填充。

(2) 由于 DES 对明文进行了填充, 因此在解密转成明文的时候, 实际上字符串最后会有填充的 '\00', 在输出显示的时候可能看不出区别, 但是将 DES 用于后续综合实验的一部分是, 可能会出现一些问题。

(3) 实验中的 DES 加解密对于大量明文来说还是非常慢的, 在不改变编程语言的情况下, 后续可以逐个步骤进行优化。

3.4.2 实验感想

分组密码是现代密码学的重要体制之一, 可以说时现代密码中最常用、应用最多的密码技术之一, 而数据加密标准 (DES) 则是密码学历史上第一个广泛应用于商用数据加密的密码算法, 所以在本次实验中我对 DES 进行了实现与分析, 并以此对分组密码中很常见的 Feistel 网络结构有了更深入的理解。

除此之外, 由于如 DES 的分组密码自身并不是加密的实用手段, 而必须以某种工作模式进行实际操作, 所以我对 ECB、CBC、CFB、OFB、CTR 五种常见的工作模式进行了实现,

通过对工作模式的深入学习，更加了解了 DES 及工作模式在实际场景下的应用与选择。本次实验由于时间有限，仅对 DES 进行了实现，虽然目前其安全性已不再足够了，但它的基本理论和设计思想仍值得学习与分析。而在之后的时间里，我会继续对分组密码进行学习，并实现 AES 在各工作模式下的加密与解密。

第 4 章 Hash 函数

4.1 MD5 算法

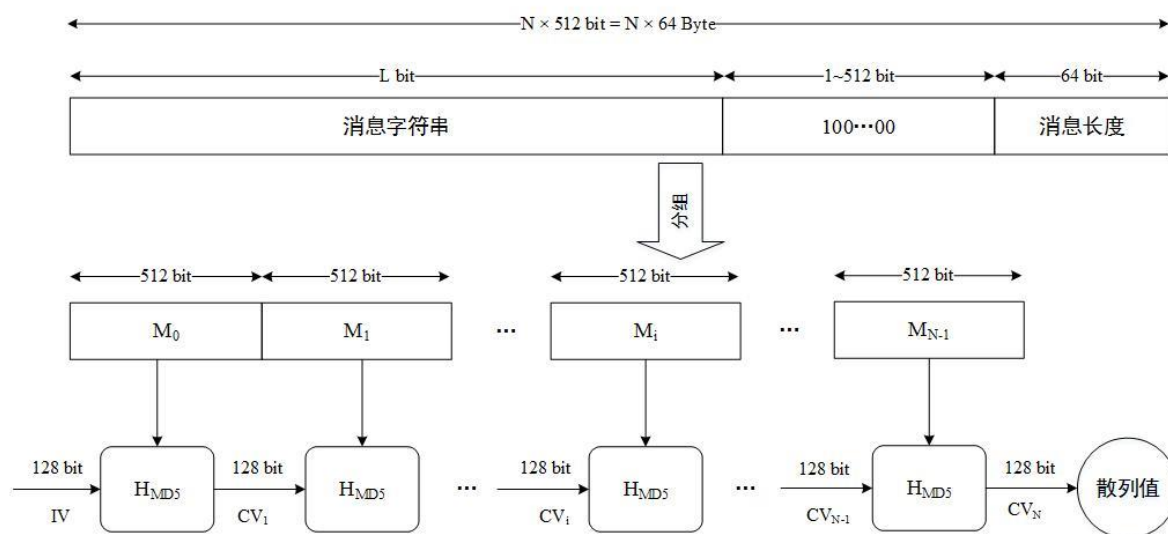
实验内容：

- 编写了 MD5 算法的过程，并验证其正确性
- 对 MD5 算法的可用性与安全性进行了分析
- 与官方库的 MD5 算法进行了效率上的对比，并分析了原因
- 查阅资料，对 MD5 的攻击方式进行了总结与阐述

4.1.1 原理分析

MD5 消息摘要算法 (MD5)，一种被广泛使用的密码散列函数，可以产生出一个 128 位（16 字节）的散列值，用于确保信息传输完整一致。

MD5 算法输入不定长度信息，输出固定长度 128-bits 的算法。经过程序流程，生成四个 32 位数据，最后联合起来成为一个 128-bits 散列。基本方式有求余、取余、调整长度、与链接变量进行循环运算，得出结果。基本流程如下图所示：



4.1.2 代码实现

(1) 基本构件

```

# 初始链接变量(小端序表示)
IV_A, IV_B, IV_C, IV_D = (0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476)

# 每轮步函数中循环左移的位数
shift_ = ((7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22),
          (5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20),
          (4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23),
          (6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21))

# 每步选择m得索引
M_index_ = ((0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15),
            (1, 6, 11, 0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12),
            (5, 8, 11, 14, 1, 4, 7, 10, 13, 0, 3, 6, 9, 12, 15, 2),
            (0, 7, 14, 5, 12, 3, 10, 1, 8, 15, 6, 13, 4, 11, 2, 9))

```

(2) 四个非线性函数的定义

$$F(X, Y, Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$

$$G(X, Y, Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$$

$$H(X, Y, Z) = X \oplus Y \oplus Z$$

$$I(X, Y, Z) = Y \oplus (X \vee \neg Z)$$

```
def F(x, y, z): return (x & y) | (~x & z)
```

```
def G(x, y, z): return (x & z) | (y & ~z)
```

```
def H(x, y, z): return x ^ y ^ z
```

```
def I(x, y, z): return y ^ (x | (~z))
```

(3) 伪随机常数的生成

$$T[i] = \lfloor 2^{32} \times \text{abs}(\sin(i)) \rfloor = \lfloor 4294967296 \times \text{abs}(\sin(i)) \rfloor$$

```
def T(i): return math.floor(abs(math.sin(i)) * pow(2, 32))
```

(4) 二进制的循环左移

```
def rol(x, n): return (x << n) | (x >> 32 - n)
```

(5) 为了方便使用，将模 32 加封装为一个函数

```
def mod_add(x, y): return (x + y) % pow(2, 32)
```

(6) 字符串转二进制

```
def str2bin(text):
    res = ''
    for i in text:
        tmp = bin(ord(i))[2:].zfill(8)
        res += tmp
    return res

```

(7) 大端序转小端序 (2 进制/16 进制)

```
def bin2little(x):
    res = re.findall(r'.{8}', x)[::-1]
    res = ''.join(res)
    return res

def hex2little(x):
    res = re.findall(r'.{2}', x)[::-1]
    res = ''.join(res)
    return res
```

(8) 对消息进行填充

```
def message_padding(m):
    # 计算附加的64为长度(小端序表示)
    len_padding = bin2little(bin(len(m))[2:].zfill(64))
    m += '1'
    while len(m) % 512 != 448:
        m += '0'
    return m + len_padding
```

(9) 压缩函数

```
def compress_func(a, b, c, d, m):
    A, B, C, D = a, b, c, d
    m_list_32 = re.findall(r'.{32}', m)
    for round_index in range(4):
        for step_index in range(16):
            AA, BB, CC, DD = A, B, C, D
            if round_index == 0:
                func_out = F(B, C, D)
            elif round_index == 1:
                func_out = G(B, C, D)
            elif round_index == 2:
                func_out = H(B, C, D)
            else:
                func_out = I(B, C, D)
            A, C, D = D, B, C
            B = mod_add(AA, func_out)
            B = mod_add(B, int(bin2little(m_list_32[M_index[round_index][step_index]]), 2))
            B = mod_add(B, T(16 * round_index + step_index + 1))
            B = rol(B, shift[round_index][step_index])
            B = mod_add(B, BB)
            print(str(16 * round_index + step_index + 1).zfill(2), end=" ")
            print(hex(A).replace("0x", "").replace("L", "").zfill(8), end=" ")
            print(hex(B).replace("0x", "").replace("L", "").zfill(8), end=" ")
            print(hex(C).replace("0x", "").replace("L", "").zfill(8), end=" ")
            print(hex(D).replace("0x", "").replace("L", "").zfill(8))
        print(" *" * 38)
    A = mod_add(A, a)
    B = mod_add(B, b)
    C = mod_add(C, c)
    D = mod_add(D, d)
    return A, B, C, D
```

(10) MD5 运算流程

```
def md5(m):
    m = str2bin(m)
    m = message_padding(m)
    m_list_512 = re.findall(r'.{512}', m)
    A, B, C, D = IV_A, IV_B, IV_C, IV_D
    # 对每512bit进行分组处理,前一组的4个输出连接变量作为下一组的4个输入链接变量
    for i in m_list_512:
        A, B, C, D = compress_func(A, B, C, D, i)
    # 把最后一次的分组4个输出连接变量再做一次大端小端转换
    A = hex2little(hex(A)[2:]).zfill(8)
    B = hex2little(hex(B)[2:]).zfill(8)
    C = hex2little(hex(C)[2:]).zfill(8)
    D = hex2little(hex(D)[2:]).zfill(8)
    return A + B + C + D
```

4.1.3 正确性验证与性能分析

(1) 正确性验证

我们对字符串“My name is YuanXiaojian, I am from CUMT.”计算其 md5 散列值,并打印出 64 轮循环操作过程中 A、B、C、D 四个 32bit 寄存器的变化情况如下,因为长度小于一个分组,因此只需要经过一次压缩函数,即 4 轮共 64 步处理。

① 第一轮(1~16步)

请输入要进行md5的内容:

My name is YuanXiaojian, I am from CUMT.

01	10325476	b55c8dab	efcdab89	98badcfe
02	98badcfe	9a8d3a1f	b55c8dab	efcdab89
03	efcdab89	302750af	9a8d3a1f	b55c8dab
04	b55c8dab	6f0f268d	302750af	9a8d3a1f
05	9a8d3a1f	569ad7ef	6f0f268d	302750af
06	302750af	7d1ecf3a	569ad7ef	6f0f268d
07	6f0f268d	0c41ee61	7d1ecf3a	569ad7ef
08	569ad7ef	f691440f	0c41ee61	7d1ecf3a
09	7d1ecf3a	1c67f797	f691440f	0c41ee61
10	0c41ee61	b2724c42	1c67f797	f691440f
11	f691440f	cfb50e8b	b2724c42	1c67f797
12	1c67f797	0879a6d1	cfb50e8b	b2724c42
13	b2724c42	1e2544f2	0879a6d1	cfb50e8b
14	cfb50e8b	daf02c8d	1e2544f2	0879a6d1
15	0879a6d1	8f434d3c	daf02c8d	1e2544f2
16	1e2544f2	7f7e71aa	8f434d3c	daf02c8d

② 第二轮（17~32 步）

17	daf02c8d	03059f5c	7f7e71aa	8f434d3c
18	8f434d3c	1fd2b63b	03059f5c	7f7e71aa
19	7f7e71aa	7955ab78	1fd2b63b	03059f5c
20	03059f5c	5b24fe4d	7955ab78	1fd2b63b
21	1fd2b63b	703f0bbd	5b24fe4d	7955ab78
22	7955ab78	0893a2e5	703f0bbd	5b24fe4d
23	5b24fe4d	d60f4169	0893a2e5	703f0bbd
24	703f0bbd	a47fa0e8	d60f4169	0893a2e5
25	0893a2e5	7720bcfa	a47fa0e8	d60f4169
26	d60f4169	edba837e	7720bcfa	a47fa0e8
27	a47fa0e8	40f24a21	edba837e	7720bcfa
28	7720bcfa	2781a563	40f24a21	edba837e
29	edba837e	002f91ab	2781a563	40f24a21
30	40f24a21	dc01680	002f91ab	2781a563
31	2781a563	3d2bde5d	dc01680	002f91ab
32	002f91ab	bc94f4d5	3d2bde5d	dc01680

③ 第三轮 16 步（33~48 步）

33	dc01680	570b7abd	bc94f4d5	3d2bde5d
34	3d2bde5d	0d60a6ac	570b7abd	bc94f4d5
35	bc94f4d5	75a43874	0d60a6ac	570b7abd
36	570b7abd	b9195d7d	75a43874	0d60a6ac
37	0d60a6ac	99f2bdfa	b9195d7d	75a43874
38	75a43874	8f7846c1	99f2bdfa	b9195d7d
39	b9195d7d	d9ffd51a	8f7846c1	99f2bdfa
40	99f2bdfa	21237c3e	d9ffd51a	8f7846c1
41	8f7846c1	c4564691	21237c3e	d9ffd51a
42	d9ffd51a	eb142fb7	c4564691	21237c3e
43	21237c3e	67404575	eb142fb7	c4564691
44	c4564691	c2a7ac7c	67404575	eb142fb7
45	eb142fb7	79da5a4d	c2a7ac7c	67404575
46	67404575	e5c15fbe	79da5a4d	c2a7ac7c
47	c2a7ac7c	51be455d	e5c15fbe	79da5a4d
48	79da5a4d	ce155238	51be455d	e5c15fbe

④ 第四轮 16 步（49~64 步）

```

49 e5c15fbe b78092d2 ce155238 51be455d
50 51be455d 5522c306 b78092d2 ce155238
51 ce155238 c8cfa2e4 5522c306 b78092d2
52 b78092d2 e1824174 c8cfa2e4 5522c306
53 5522c306 dcb52d03 e1824174 c8cfa2e4
54 c8cfa2e4 316b1e6e dcb52d03 e1824174
55 e1824174 4751f4b3 316b1e6e dcb52d03
56 dcb52d03 ee4da851 4751f4b3 316b1e6e
57 316b1e6e 30db311f ee4da851 4751f4b3
58 4751f4b3 5b3ca338 30db311f ee4da851
59 ee4da851 ff00fea4 5b3ca338 30db311f
60 30db311f f8dd0c66 ff00fea4 5b3ca338
61 5b3ca338 9849dfcb f8dd0c66 ff00fea4
62 ff00fea4 01ee89ad 9849dfcb f8dd0c66
63 f8dd0c66 368d9afd 01ee89ad 9849dfcb
64 9849dfcb 4a1ebc29 368d9afd 01ee89ad

```

(5) 最终结果:

md5后的散列值为:

cc028fffb267ec39fb7748cf23de2012


(6) 与在线 md5 网站验证



MD5在线加密

要加密的字符串:

[加密](#)

字符串	My name is YuanXiaojian, I am from CUMT.
16位 小写	b267ec39fb7748cf
16位 大写	B267EC39FB7748CF
32位 小写	cc028fffb267ec39fb7748cf23de2012 
32位 大写	CC028FFFB267EC39FB7748CF23DE2012

可以看到查询的结果，与我们得到的结果一致，正确性得以验证。

（另：该网站使用“加密”一词描述 md5 并不严谨，因为 md5 为单向处理，无法进行解

密，不属于加密范畴。)

(2) 性能分析

因为 md5 的速度非常快且具有单向性，所以其性能不适合与上述其他加密算法进行对比。因此，我选择与 python 的 md5 官方库进行效率上的对比，使用官方库编写脚本如下，对上述测试字符串重复 3000 次作为消息进行 md5:

```
import hashlib
import time

message = "My name is YuanXiaojian, I am from CUMT." * 3000
digest = hashlib.md5()
digest.update(message.encode())

start = time.perf_counter()
res = digest.hexdigest()
end = time.perf_counter()
print("md5: " + res)
print("Spend time: " + str(end - start) + " seconds.")
```

结果如下:

```
md5: 8fccc0c08ad041806f95025b1140b5ae
Spend time: 3.4000000000145025e-06 seconds.
```

而我写的 md5 程序所需时间如下:

```
md5: 8fccc0c08ad041806f95025b1140b5ae
Spend time: 0.7607792 seconds.
```

对比如下:

代码\时间	时间 (秒)
myMD5	0.76078
pyMD5	3.4×10^{-6}

可以看到，虽然 md5 后得到的散列值是一致的，但是在效率上与 python 的官方库的差距还是非常巨大的。经过查阅资料，分析原因可能有以下几点:

- ① 语言差异。与 DES 相似，md5 的官方库底层也是采用 C 语言进行编写的，属于编译型语言；而本实验由纯 python 代码实现，属于解释型语言，两种语言在性能与效率上本身就存在巨大的差距。
- ② 多线程计算。查阅资料可知，官方库在计算 md5 时使用了多线程进行计算，将计算过程拆解为可并行执行的部分，由多个线程同时进行计算，因此大大加快了其效率。
- ③ 优化策略。除了上述两点之外，官方库还在算法与代码的实现上通过编程技巧与策略进

行了优化，如零复制优化，可以节省数拷贝数组等对象的时间，从而加快了运行效率。

4.1.4 可用性与安全性分析

(1) 可用性

MD5 不依赖任何密码系统和假设条件，算法简洁、计算速度快，特别适合 32 位计算机软件实现。当前，MD5 算法因其普遍、稳定、快速的特点，广泛应用于普通数据的错误检查、数字签名、密码存储等领域，但是，因为其已经被发现可以在有限时间内找到其碰撞，安全性保障大大下降，在需要对重要消息进行摘要的情况下，应选择安全系数更高的其他哈希函数。

(2) 安全性

本质上，对于任何一个哈希函数来说，碰撞是无可避免的，从一个规模较大的集合映射到一个规模较小的集合，必然会存在相同映射的情况。对于 MD5 而言，由于其抗密码分析能力较弱以及计算机运算能力的不断提升，在有限时间实现 MD5 的碰撞已经被证明为可能，这使得 MD5 算法在目前的安全环境下有一点落伍。但是从实践角度，不同信息具有相同 MD5 的可能性还是非常低的，通过碰撞的方法也很难碰撞出复杂信息的 MD5 数值。

4.1.5 破解或攻击方式分析

4.1.5.1 学术界成果

经过查阅资料，目前对 MD5 的攻击已经取得以下结果：

(1) T. Berson (1992) 已经证明，对单轮的 MD5 算法，利用差分密码分析，可以在合理的时间内找出散列值相同的两条消息。这一结果对 MD5 四轮运算的每一轮都成立。但是，目前尚不能将这种攻击推广到具有四轮运算的 MD5 上。

(2) B. Boer 和 A. Bosselaers (1993) 说明了如何找到消息分组和 MD5 两个不同的初始值，使它们产生相同的输出。也就是说，对一个 512 位的分组，MD5 压缩函数对缓冲区 ABCD 的不同值产生相同的输出，这种情况称为伪碰撞 (pseudo-collision)，但是目前尚不能用该方法成功攻击 MD5 算法。

(3) H. Dobbertin (1996) 找到了 MD5 无初始值的碰撞 (pseudo-collision)。给定一个 512 位的分组，可以找到另一个 512 位的分组，对于选择的初始值 IV0，它们的 MD5 运算结果相同。到目前为止，尚不能用这种方法对使用 MD5 初始值 IV 的整个消息进行攻击。

(4) 我国山东大学王小云教授 (2004) 提出的攻击对 MD5 最具威胁。对于 MD5 的初始值 IV，王小云找到了许多 512 位的分组对，它们的 MD5 值相同，即利用差分分析，只需 1 小时就可找出 MD5 的碰撞。

(5) 国际密码学家 Lenstra 利用王小云等提供的 MD5 碰撞，伪造了符合 X.509 标准的数字证书。

4.1.5.2 生日攻击

生日攻击是一种密码学攻击手段，所利用的是概率论中生日问题的数学原理。此攻击依赖于在随机攻击中的高碰撞概率和固定置换次数（鸽巢原理）。使用生日攻击，攻击者可在 $\sqrt{2^n} = 2^{n/2}$ 中找到散列函数碰撞， 2^n 为原像抗性安全性。MD5 算法抗密码分析能力较弱，对 MD5 的生日攻击所需代价为 2^{64} 数量级。

4.2 实验总结

4.2.1 存在的问题

在编写 md5 算法的过程种，遇到的最大的麻烦就是关于大小端序的转换问题，初始链接遍历应该是小端序输入的，消息的填充也应该以小端序方式，并且在最后运算结束后还要再进行一次大小端序的转换。这方面一定要注意，否则就可能的不到正确的结果。

4.2.2 实验感想

本次实验，我对 MD5 的整个流程进行了代码层面的实现，对 MD5 算法有了进一步的认识，同时也让我对字节序方面的有了更多的理解。整个过程还是比较繁琐的，尤其得注意细节方面的处理，在正确实现的基础上，下一步应考虑对代码进行进一步的优化与改进。

第 5 章 公钥密码

实验内容：

- 编写了 RSA 算法的加密解密过程，并验证其正确性
- 对 RSA 公钥加密体制的可用性与安全性进行了分析
- 比较了快速幂算法与内置函数的效率
- 对 Miller-Rabin 素性检验的迭代轮数选择进行了分析
- 对 RSA 的低加密指数攻击、低加密指数广播攻击、共模攻击进行了分析，并给出攻击脚本代码

本代码

- 编写了基于 RSA 与 md5 的数字签名体制流程，并对验证其正确性
- 对 RSA 数字签名体制的可用性与安全性进行了分析

5.1 RSA 公钥加密体制

5.1.1 原理分析

5.1.1.1 RSA 算法

RSA 是 1977 年由 Ron Rivest、Adi Shamir 和 Leonard Adleman 共同提出，其安全性基于大素数因子分解的困难问题，整体算法流程简单清晰，分为如下三个部分：

（1）密钥生成算法：

- ① 选取两个保密的大素数 p 和 q ，满足 $p \neq q$ ，计算 $n = p \times q$ ， $\varphi(n) = (p - 1)(q - 1)$ ， $\varphi(n)$ 为 n 的欧拉函数。
- ② 随机选取整数 e ，满足 $1 < e < \varphi(n)$ 且 $\gcd(e, \varphi(n)) = 1$ ，即 e 与 $\varphi(n)$ 互素。
- ③ 计算 d ，满足 $ed \equiv 1 \pmod{\varphi(n)}$ ，则公钥为 (e, n) ，私钥为 d 。

（2）加密

对明文进行比特串分组，使每个分组十进制小于 n ，然后对每个分组 $m(0 \leq m < n)$ ，计算

$c = m^e \pmod{n}$ ，则得到密文 c 。

（3）解密

对于密文 $c(0 \leq c < n)$ ，计算 $m = c^d \pmod{n}$ ，得到对应明文 m 。

5.1.1.2 Miller-Rabin 素性检验

除了密钥生成以及加密解密算法之外，RSA 加密体制还有一个重要的问题，那就使大素数的生成，这就要求了还需要掌握素性检验的算法。在本实验中，我使用了 Miller-Rabin 素性检验结合随机数的生成来得到所需要的大素数，它是一个基于概率的算法，是费马小定理的一个改进。

简单来说,要测试 n 是否为素数,首先将 $n-1$ 分解为 $2^s d$ 。在每次测试开始时,先随机选一个介于 $[1, n-1]$ 的整数 a ,之后如果对所有的 $r \in [0, s-1]$,若 $a^d \not\equiv 1 \pmod{n}$ 且 $a^{2^r d} \not\equiv -1 \pmod{n}$,则 n 是合数。否则, n 有 $\frac{3}{4}$ 的概率为素数,随着增加测试的次数,是素数的概率会越来越高,当达到某一

5.1.2 代码实现

(1) 扩展的欧几里得算法

```
def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0
    else:
        g, x, y = extended_gcd(b, a % b)
        return g, y, x - (a // b) * y
```

(2) 模逆计算

```
def mod_inverse(a, m):
    g, x, y = extended_gcd(a, m) # ax + my = 1
    # 若a,m不互素,则不可逆
    if g != 1:
        raise Exception(str(a) + ' is not invertible!')
    else:
        return x % m
```

(3) 快速幂

```
def fast_mod(x, n, p):
    x = x % p
    res = 1
    while n != 0:
        if n & 1:
            res = (res * x) % p
        n >>= 1 # 相当于 n //= 2
        x = (x * x) % p
    return res
```

(4) Miller Rabin 素性检验

```
def miller_rabin(n, k=10):
    if n % 2 == 0:
        return False
    s, d = 0, n - 1
    while d % 2 == 0:
        s += 1
        d //= 2
    for i in range(k):
        a = randint(1, n - 1)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        else:
            flag = 0
            for r in range(s):
                x = pow(x, 2, n)
                if x == n - 1:
                    flag = 1
                    break
            if flag == 0:
                return False
    return True
```

(5) 素数生成

```
def get_prime(n):
    while True:
        # 最高位为1, 保证是n位
        num = '1'
        for i in range(n - 2):
            x = randint(0, 1)
            num += str(x)
        # 最低位为1, 保证是奇数
        num += '1'
        num = int(num, 2)
        if miller_rabin(num):
            return num
```

(6) 密钥生成

```
def get_keys(nbits):
    nbits = int(nbits)
    while True:
        p = get_prime(nbits)
        q = get_prime(nbits)
        if p == q:
            continue
        N = p * q
        phiN = (p - 1) * (q - 1)
        e = randint(500, 10000)
        if extended_gcd(e, phiN)[0] == 1:
            d = mod_inverse(e, phiN)
            return e, N, d
```

(7) 加密函数

```
def Encrypt(m, e, n):
    e = int(e)
    n = int(n)
    c = pow(m, e, n)
    return c
```

(8) 解密函数

```
def Decrypto(c, d, n):
    d = int(d)
    n = int(n)
    m = pow(c, d, n)
    return m
```

5.1.3 正确性验证

(1) 正确性验证

① 加密

选择生成 2048 位的密钥，在符合 RSA 算法要求的下随机产生 $\{e, n\}$ 和私钥 $\{d\}$ ，并用公钥对明文 “My name is YuanXiaojian, I am from CUMT.” 进行加密，得到对用的密文。

请选择模式：

[E]加密 [D]解密

E

请输入明文：

My name is YuanXiaojian, I am from CUMT.

请输入位数：

2048

公钥e为：7357

公钥n为：82754313508045979601063887428670549514598379588884413428635756707

私钥d为：69334999111539407130754084832176874705448472446089917680319533008

明文为：My name is YuanXiaojian, I am from CUMT.

密文为(10进制)：7458237419705883464854753680314018010279185218081632913125

② 解密

使用上述生成的私钥 d 和 n 对得到的密文进行解密，并成功得到明文如下。

请选择模式：

[E]加密 [D]解密

D

请输入密文(10进制)：

7458237419705883464854753680314018010279185218081632913125474841717882537495

请输入私钥d：

6933499911153940713075408483217687470544847244608991768031953300878560048995

请输入公钥n：

8275431350804597960106388742867054951459837958888441342863575670759825807991

密文为：My name is YuanXiaojian, I am from CUMT.

5.1.4 性能分析

5.1.4.1 快速幂算法与内置函数效率对比

在实验中我虽然根据书中原理实现了计算快速幂的算法，但是在实际应用中，发现我编写的快速幂算法在计算效率上并没有超过、甚至不如 python 内置的 pow 函数，所以我最终使用了内置函数来进行大数的模幂计算，下面进行对此进行分析，使用是用上述验证过程中得到 c 、 d 、 n 来进行测试 $c^d \pmod n$ ，代码如下：

```
import time
```

```
def fast_mod(x, n, p):
    x = x % p
    res = 1
    while n != 0:
        if n & 1:
            res = (res * x) % p
        n >>= 1      # 相当于 n //= 2
        x = (x * x) % p
    return res
```

```
n = 8275431350804597960106388742867054951459837958888441342863575
c = 7458237419705883464854753680314018010279185218081632913125474
d = 6933499911153940713075408483217687470544847244608991768031953
```



```
start = time.perf_counter()
m = pow(c, d, n)
end = time.perf_counter()
print("pow spend " + str(end - start) + " seconds.")
start = time.perf_counter()
m = fast_mod(c, d, n)
end = time.perf_counter()
print("fast_mod spend " + str(end - start) + " seconds.\n")

start = time.perf_counter()
for i in range(10):
    m = pow(c, d, n)
end = time.perf_counter()
print("10 times pow spend " + str(end - start) + " seconds.")
start = time.perf_counter()
for i in range(10):
    m = fast_mod(c, d, n)
end = time.perf_counter()
print("10 times fast_mod spend " + str(end - start) + " seconds.\n")

start = time.perf_counter()
for i in range(100):
    m = pow(c, d, n)
end = time.perf_counter()
print("100 times pow spend " + str(end - start) + " seconds.")
start = time.perf_counter()
for i in range(100):
    m = fast_mod(c, d, n)
end = time.perf_counter()
print("100 times fast_mod spend " + str(end - start) + " seconds.")
```

结果如下：

pow spend 0.1701686 seconds.

fast_mod spend 0.23112480000000002 seconds.

10 times pow spend 1.7253087 seconds.

10 times fast_mod spend 2.2715989000000003 seconds.

100 times pow spend 17.452038299999998 seconds.

100 times fast_mod spend 22.5601474 seconds.

函数\时间	1 次（秒）	10 次（秒）	100 次（秒）
pow	0.17017	1.7253	17.452
fast_mod	0.23112	2.2716	22.560

可以看到，我编写的快速幂算法确实比 pow 函数在单次计算的效率上要低一些，虽然差距并不是特别大。但是若需要多次进行模幂运算的时候，如 Miller Rabin 素性检验过程中以及对多个分组进行加密解密时，使用内置函数 pow 还是会有一定效率上的提升的。

5.1.4.2 Miller-Rabin 素性检验迭代轮数分析

在进行 Miller Rabin 素性检验算法的过程，存在的一个比较大的疑问就是检测的轮数到底改选为多少。若检验的轮数选择太多，则会影响算法的效率；而检验的轮数选择太少，则不能保证素数生成的正确性。

经过查阅资料，在论文《Average case error estimates for the strong probable prime test》中给出了相应的概率界如下：

$$\text{当 } 3 \leq t \leq k/9, k \geq 2 \text{ 时, } p_{k,t} < k^{3/2} 2^t t^{-1/2} 4^{2-\sqrt{tk}}$$

其中 t 为迭代轮数， k 为被检验数的位数， $p_{k,t}$ 为结果是素数的概率上界

我们分别计算 1024 位和 2048 位大数的概率上界，测试代码如下：

```
from math import *

def p(k, t):
    return pow(k, 3 / 2) * pow(2, t) * pow(t, -1 / 2) * pow(4, 2 - sqrt(t * k))

for t in range(1, 11):
    print(str(t).zfill(2) + ": " + str(p(1024, t)) + "\t" + str(p(2048, t)))
```

结果如下：

$t \backslash p_{k,t}$	$p_{1024,t}$	$p_{2048,t}$
1	5.6843×10^{-14}	1.6827×10^{-21}
2	8.4137×10^{-22}	1.2325×10^{-32}
3	1.0340×10^{-27}	4.4052×10^{-41}
4	1.2325×10^{-32}	3.8190×10^{-48}
5	6.2420×10^{-37}	2.5269×10^{-54}
6	8.8105×10^{-41}	7.0650×10^{-60}
7	2.7002×10^{-44}	5.8790×10^{-65}
8	1.5276×10^{-47}	1.1591×10^{-69}
9	1.4254×10^{-50}	4.6226×10^{-74}
10	2.0215×10^{-53}	3.3235×10^{-78}

由上表可以看出，1024 位的大数，在 $t = 6$ 的情况下，已经能够保证错误概率小于 10^{-40} ；而 2048 位的大数，在 $t = 3$ 的时候，错误概率已经小于 10^{-40} 了。事实上，计算机发生随机错误的概率大约是 1.8×10^{-24} ，再增加迭代次数，将 Miller-Rabin 算法的准确读提的更高并无意义。

5.1.5 可用性与安全性分析

(1) 可用性

RSA 是第一个安全、实用的公钥加密算法，已经成为国际标准，是目前应用最广泛的公钥加密体制，其算法数学原理可靠，实现简单。虽然 RSA 与 DES 等分组密码相比加密速度要慢很多，但是仍可应用在密钥加密、数字签名等领域，而不适合直接对大量明文进行加密。

(2) 安全性

RSA 算法的安全性取决于对大整数的因子分解难题，对一极大整数做因数分解越困难，RSA 算法的安全性愈可靠，假如有人找到一种快速因数分解的算法的话，那么 RSA 的安全性将会不复存在。然而目前来看，只要其密钥的长度足够长，用 RSA 加密的信息实际上是不能被破解的。

但是，RSA 的安全性很多时候也受到各参数选择的影响，具体来说：

- ① p 和 q 的长度相差不能太大，以避免椭圆曲线因子分解法。
- ② p 和 q 的差值不能太小，以防止 n 被所有接近 \sqrt{n} 的奇整数试除而被有效分解。
- ③ p-1 和 q-1 都应有大的素因子，以避免循环攻击。
- ④ e 和 d 不能选取太小，以避免低指数攻击。
- ⑤ 同一通信网络的多个用户不应使用同一个 n，以避免共模攻击。

5.1.6 破解或攻击方式分析

5.1.6.1 低加密指数攻击

(1) 攻击原理

有时候为了增加加密的高效性，希望选择较小的加密密钥 e，而当选取的 e 过小,如 $e = 3$ ，导致明文 m^3 依然小于n，那么直接对密文 c 开三次方 $\sqrt[3]{c}$ 即得到 m

(2) 攻击脚本

```
import gmpy2
import libnum
e = 3
n= 22885480907469109159947272333565375109310485067211461543881386718201442106967914852474
c= 15685364647213619014219110070569189770745535885901269792039052046431067708991036961644
print('n=', n)
print('c=', c)
result = gmpy2.iroot(c, 3)
if result[1]:
    print('m= ', libnum.n2s(result[0]))
```

得到结果：

```
n= 228854809074691091599472723335653751093104850672114615438813867182014421069679148524
c= 156853646472136190142191100705691897707455358859012697920390520464310677089910369616
m= Tr0y{e=3_And_Sma11_M_1s_danger0us}
```

5.1.6.2 低加密指数广播攻击

(1) 攻击原理

如果选取的加密指数较低，并且使用了相同的加密指数给一个接受者的群发送相同的信息，那么可以进行广播攻击得到明文。选取了相同的加密指数 e (这里取 $e=3$)，对相同的明文 m 进行了加密并进行了消息的传递，那么有：

$$\begin{aligned}c_1 &\equiv m^e \pmod{n_1} \\c_2 &\equiv m^e \pmod{n_2} \\c_3 &\equiv m^e \pmod{n_3}\end{aligned}$$

对上述等式运用中国剩余定理，在 $e=3$ 时，可以得到：

$$c_x \equiv m^3 \pmod{n_1 n_2 n_3}$$

通过对 c_x 进行三次开方就可以求得明文

(2) 攻击脚本

```
import gmpy2
import gmpy
import libnum

question = [c1,c2,c3,...,n1,n2,n3,...]
N = 1
e=10
for i in range(len(question)):
    N*=question[i]['n']
N_list = []
for i in range(len(question)):
    N_list.append(N/question[i]['n'])
t_list = []
for i in range(len(question)):
    t_list.append(int(gmpy2.invert(N_list[i],question[i]['n'])))
sum = 0
for i in range(len(question)):
    sum = (sum+question[i]['c']*t_list[i]*N_list[i])%N
sum = gmpy.root(sum,e)[0]
print libnum.n2s(sum)
```

5.1.6.3 共模攻击

(1) 攻击原理

在实现 RSA 时，有时候为了方便起见，可能会让多个用户使用相同的模数 n ，但他们的公、

私钥对不同，然而这就会造成共模攻击。

设两个用户公钥为 e_1 和 e_2 且互素，明文消息为 m ，那么可以得到：

$$c_1 \equiv m^{e_1} (\text{mod } n)$$

$$c_2 \equiv m^{e_2} (\text{mod } n)$$

当攻击者截取到 c_1 和 c_2 时，用扩展欧几里得算法求出满足 $re_1 + se_2 = 1$ 的两个整数 r 和 s ，由此可得：

$$c_1^r c_2^s \equiv m^{re_1} m^{se_2} \equiv m^{(re_1 + se_2)} \equiv m (\text{mod } n)$$

(2) 攻击脚本

```
import libnum
import gmpy2
c1 = 845418396732511470287160602037189304464373084819442108780055532995174
c2 = 347489654529391334766720222345156515385286212979434403020079868816620
n = 1540917915770938724155782268542684144357931315167269654437803582062746
e1 = 35807
e2 = 64109
_, s1, s2 = gmpy2.gcdext(e1, e2)
if s1 < 0:
    s1 = - s1
    c1 = gmpy2.invert(c1, n)
elif s2 < 0:
    s2 = - s2
    c2 = gmpy2.invert(c2, n)
m = pow(c1, s1, n) * pow(c2, s2, n) % n
print(libnum.n2s(m))
```

结果：

```
D:\code\Anaconda3\python.exe F:/MyCode/Cryptography/PublickeyCipher/common_n.py
CUMTCTF{037b1a9c1753987d51845372f6cf18b1}
```

5.2 基于 RSA 的数字签名

5.2.1 原理分析

基于 RSA 的数字签名方案，密钥的生成算法与 RSA 公钥密码机制相同，签名者的公钥为 $\{n, e\}$ ，私钥为 $\{d\}$ 。大致流程如下：

(1) 当被签名的消息为 m 时，则 m 的 RSA 签名为 $s \equiv m^d (\text{mod } n)$ ；

(2) 而当接收方接收到签名 s 和消息 m 后，计算 $m' \equiv s^e (\text{mod } n)$ ，若 $m' = m$ ，则验证通过，否则验证失败

然而实际应用中，是对消息的哈希值进行签名来保证安全性。

5.2.2 代码实现

由于在之前的实验中，已经实现了 RSA 加解密算法以及哈希函数 md5，因此这里直接导入

调用即可。

(1) 签名函数

```
from Hash.md5 import md5
import PublickeyCipher.RSA as RSA
import libnum
import sys

def sign(m, d, n):
    d = int(d)
    n = int(n)
    digest = md5(m)
    digest = libnum.s2n(digest)
    s = pow(digest, d, n)
    return hex(s)[2:]
```

(2) 验证函数

```
def check(m, s, e, n):
    s = int(s, 16)
    e = int(e)
    n = int(n)
    digest = md5(m)
    digest = libnum.s2n(digest)
    temp = pow(s, e, n)
    if digest == temp:
        return True
    else:
        return False
```

5.2.3 正确性验证

(1) 生成签名

开始签名

请输入您要签名的消息：

My name is YuanXiaojian, I am from CUMT.

你的消息为：

My name is YuanXiaojian, I am from CUMT.

对应的签名为：

54e62d72dd0bf8b14ac552d06fec1b42fa01c1873534aae2c0edd329a9baa6007

(2) 验证签名

对签名进行验证

请输入您所得到的签名：

54e62d72dd0bf8b14ac552d06fec1b42fa01c1873534aae2c0edd329a9bae

请输入与之对应的消息：

My name is YuanXiaojian, I am from CUMT.

验证成功！

5.2.4 可用性与安全性分析

(1) 可用性

RSA 签名方案是目前使用较多的一个签名方案，它的安全性是基于大整数因子分解的困难性，其签名方案的密钥生成算法与 RSA 加密方案完全相同，签名与验证算法清晰可靠。随着计算机网络技术的发展，电子商务、电子政务和电子金融等系统得到广泛应用，人们需要通过网络信息传输对电子的文件、契约、合同、信件和张单等进行数字签名以替代手写签名，而 RSA 签名方案可以很好的承担这一责任。

(2) 安全性

可以看到我在代码实现的时候，是对消息的哈希值进行签名，因为如果不适用哈希函数，由于 RSA 的同态特性，那么攻击者在一定条件下很容易进行签名伪造，而使用安全性较好的哈希函数，则可以避免类似的攻击，从而提高签名体制的安全性。此外，RSA 的签名方案还存在签名可重用的问题，即对同一消息在不同时刻的签名是相同的，所以可以在每次签名中引入随机数来解决整个问题。总而言之，基于 RSA 的签名方案在目前看来，还是比较安全可靠的。

5.3 实验总结

5.3.1 存在的问题

在本次实验中存在的问题在前面都有题过，一个是关于快速幂算法与 python 内置函数的选择；还有一个就是关于使用 Miller-Rabin 素性检验时，关于检验轮数的确定，但是通过查阅资料与验证，对于这两个问题，已经有了一定的解决。初次之外，我实现的代码在生成 2048 位素数时，所用的时间还是比较长的，之后需要思考优化方法。

5.3.2 实验感想

公钥密码体制为密码学的发展提供了新的理论和技术基础，它的出现时迄今为止整个密码学发展史上一次最伟大的革命。而本次实验，选择了现代密码学中最具代表性的公钥加密体制 RSA 来进行实现与分析。实验过程中，用到了很多数论中的知识，例如快速幂

的计算、费马小定理以及 Miller-Rabin 素性检验，这种学有所用的方式才是我们应该追求的。之后，还结合使用了已实现的 RSA 与 md5 算法，进行了数字签名的实现。整个实验下来，我对 RSA 公钥密码体制有了一个更全面、更深入的认识。

第 6 章 综合实验——基于 C/S 架构的文件加密传输系统

6.1 开发工具选择

6.1.1 开发语言选择

本次实验中，我选用 Python 作为开发语言，原因如下：

（1）对“大数”的支持

Python 默认支持大数的运算，这使得在公钥密码学相关方面开发中，非常的简单快捷，不需要经过任何其他处理，因此可以将重点集中于算法本身的实现上。

（2）丰富的第三方库

由于 Python 语言开源的特性，其拥有大量的第三方库，且安装简单、调用方便，能够大大的提高开发效率与可扩展性。

（3）语法简单

Python 的优势是语法简单、易学，实现相同的功能，使用 Python 编写的代码量远远小于使用 C/C++ 程序的代码量，因此在时间有限的实验过程中，可以减少代码编写时间、提高效率，从而集中于功能及算法的实现。

6.1.2 开发 IDE 选择

在选择 Python 作为开发语言后，我果断选择了 PyCharm 作为主要 IDE 进行程序的编写：

（1）学生认证免费使用

PyCharm 的非社区版并不是免费软件，不菲的价格反而说明了其功能的强大，而作为大学生，我可以通过进行学生身份的认证而免费使用它，并享用其所有强大的功能，节约开发时间、提高开发效率。

（2）良好的 Python 支持性

PyCharm 由于是专门为 Python 开发的 IDE，其对 Python 有着非常良好的支持性，除了完美的代码高亮与智能的代码补全外，内置集成的 Debug、Terminal 与 Python Console 让我在实验的过程中，能够非常方便的验证代码、进行简单的测试等，可以大大提高效率。

6.2 设计目的与要求

本系统以文件的加密安全传输为核心，以网络为支撑，满足用户在互联网中对于文件安全传输的要求。预期基于 C/S 架构，使用 Socket TCP 编程实现，并结合前期实验的相关密码学知识，设计基本目标如下：

- 服务端与客户端能够成功的建立 TCP 连接
- 客户端能够根据服务端提供的功能做出指令

- 服务端能够根据要求响应客户端的命令
- 客户端能够要求服务端列出工作目录下的文件列表，以及指定文件的发送
- 服务端按照客户端指令，列出文件列表并以加密传输的方式发送指定文件
- 客户端在接收到加密文件，能够成功的进行解密，并恢复成原文件
- 服务端与客户端能完成对文件的数字签名与认证

6.3 概要设计

6.3.1 系统功能描述

因为本次实验的主要目的为文件的加密传输部分，因此仅对此进行介绍。

(1) 服务端计算对文件进行签名并发送；客户端能根据收到的签名对其进行验证，从而保证文件传输过程的完整性。

(2) 服务端在每次发送文件时，能够使用随机生成的 64 位密钥 key 和初始化向量 IV 对文件在 CBC 模式下进行 DES 的加密。

(3) 在对文件加密完成后，服务端再利用 RSA 公钥加密体制，使用客户端的公钥 $\{e, n\}$ ，对分组密码的对称密钥 key 及初始化相邻 IV 进行加密。

(4) 服务端将文件的加密结果、对称密钥及初始化向量的加密结果以及原文件的 md5 哈希值，一起发送给客户端。

(6) 客户端再收到密文后，首先能够使用自己的私钥 $\{d\}$ 对对称密钥进行解密，得到密钥 key 和初始化向量 IV ，再用对称密钥 key 和 IV 对文件进行解密。

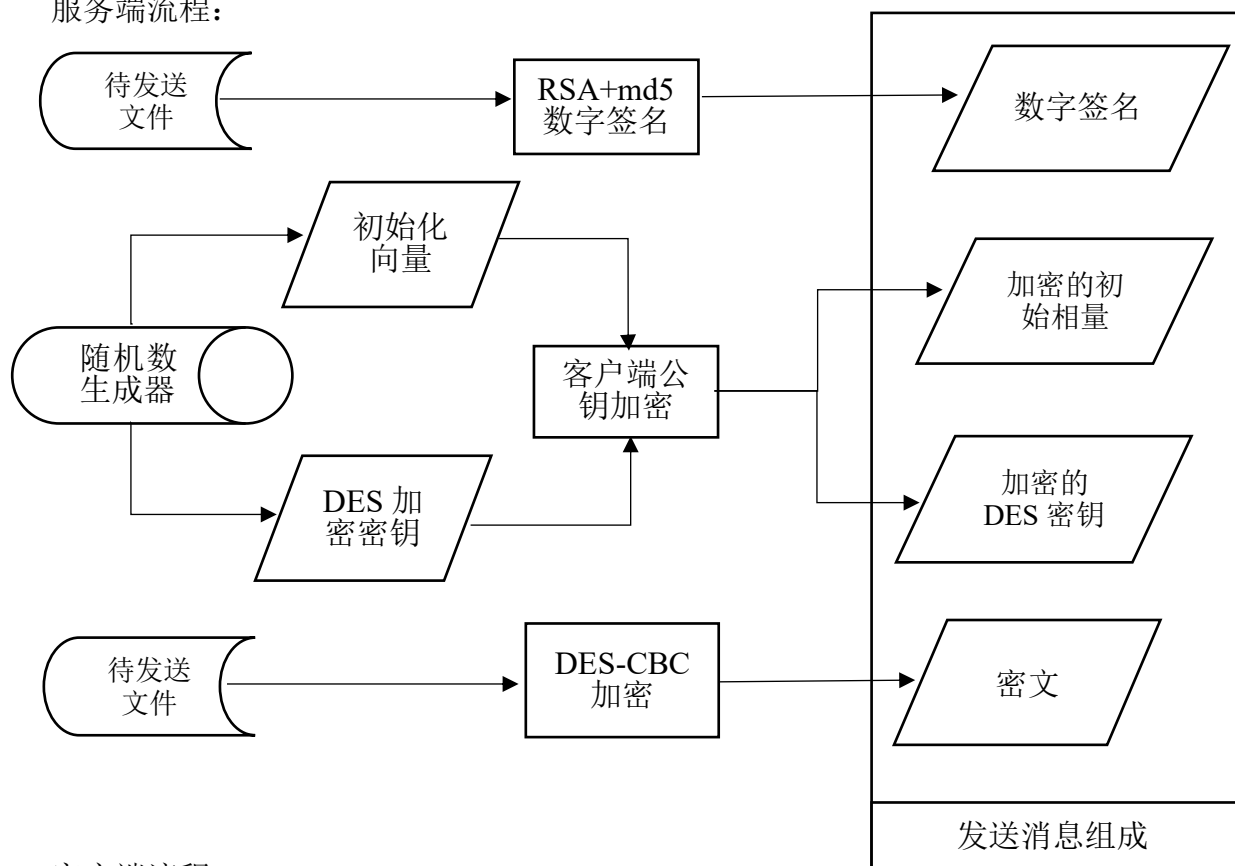
(7) 另外，在文件加密传输部分的实现中有如下两种思路：

① 服务端将整个文件先进行加密，然后分组传输密文文件和加密后的密钥；客户端再收到密文文件和加密后的密钥后，先解密得到密钥，再对密文文件解密得到原文件，最后再验证文件完整性。

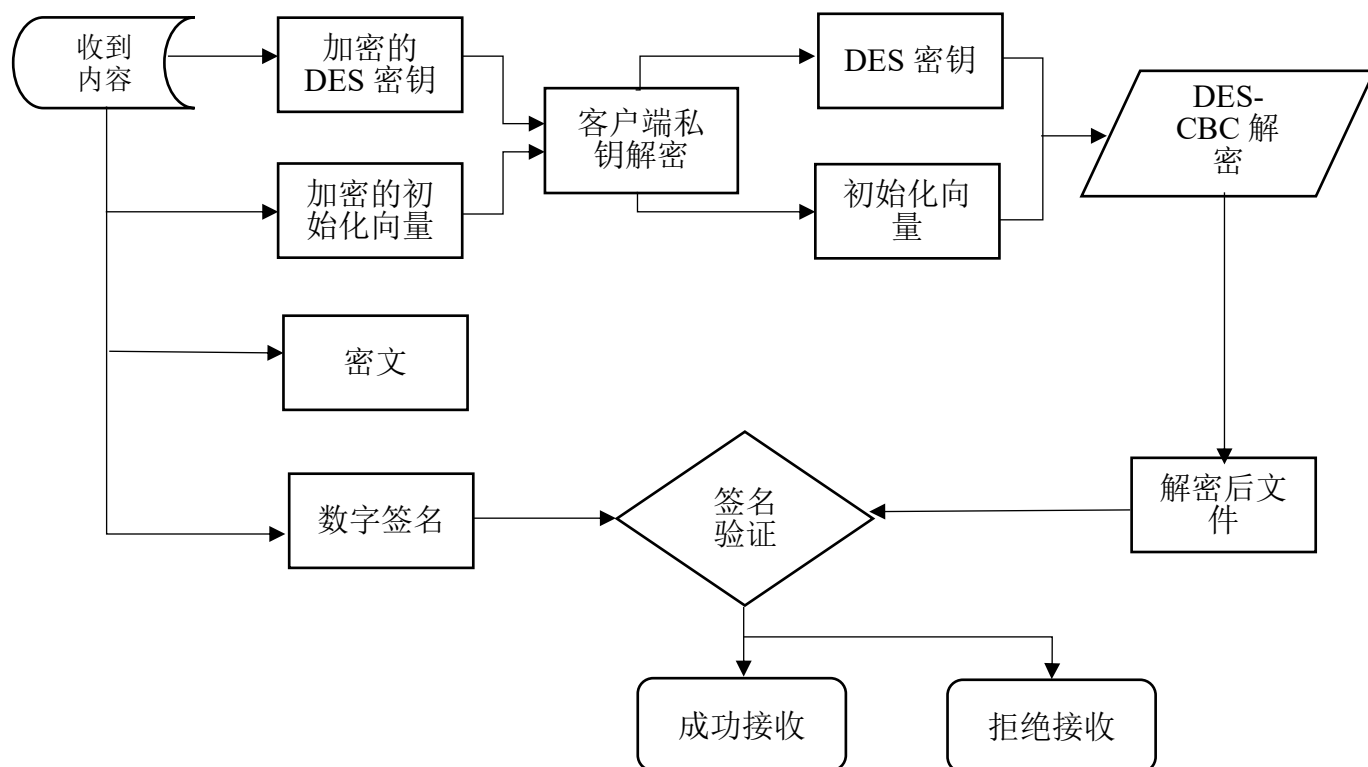
② 服务端将明文文件分组传输，对每个分组使用随机 key 和 IV 进行加密，然后与加密后的密钥一起发送；客户端接收到每个加密分组后就解密并保存，等所有分组传输完毕，即得到原文件，最后再验证文件完整性。

6.3.2 系统功能流程图

服务端流程:



客户端流程:



6.4 系统代码实现

这里只展示关键的部分代码，完整代码见附录地址。

(1) 服务端相关配置 (Server)

```
#####
SERVER_WORKDIR = "F:\\MyCode\\Cryptography\\FileTransferSystem\\ServerData\\"
SERVER_PUB_PATH = "F:\\MyCode\\Cryptography\\FileTransferSystem\\ServerKeys\\server.pub"
SERVER_KEY_PATH = "F:\\MyCode\\Cryptography\\FileTransferSystem\\ServerKeys\\server.key"
CLIENT_PUB_PATH = "F:\\MyCode\\Cryptography\\FileTransferSystem\\ServerKeys\\client.pub"
MAXBUFSIZE = 51200
BUFSIZE = 20480
SERVER_ADDRESS = ("127.0.0.1", 12345)
#####

serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serverSocket.bind(SERVER_ADDRESS)
serverSocket.listen(3)
```

(2) 客户端相关配置 (Client)

```
#####
CLIENT_WORKDIR = "F:\\MyCode\\Cryptography\\FileTransferSystem\\ClientData\\"
CLIENT_PUB_PATH = "F:\\MyCode\\Cryptography\\FileTransferSystem\\ClientKeys\\client.pub"
CLIENT_KEY_PATH = "F:\\MyCode\\Cryptography\\FileTransferSystem\\ClientKeys\\client.key"
SERVER_PUB_PATH = "F:\\MyCode\\Cryptography\\FileTransferSystem\\ClientKeys\\server.pub"
MAXBUFSIZE = 51200
BUFSIZE = 40960
BUFSIZE_SENT = 20480
SERVER_ADDRESS = ("127.0.0.1", 12345)
#####

clientServer = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

(3) 随机数生成函数 (Server、Client)

```
def get_des_param():
    while True:
        key = ''.join(random.sample(string.ascii_letters + string.digits, 8))
        iv = ''.join(random.sample(string.ascii_letters + string.digits, 8))
        if key != iv:
            break
    return key, iv
```

(4) 读取密钥 (Server、Client)

```
def get_private_key(keypath):
    with open(keypath, 'r') as f:
        d = int(f.read())
    return d

def get_public_key(keypath):
    with open(keypath, 'r') as f:
        keys = f.readlines()
        e = int(keys[0])
        n = int(keys[1])
    return e, n
```

(5) 计算文件 md5 值 (Server、Client)

```
def get_file_md5(filepath):
    with open(filepath, 'rb') as fs:
        digest = md5(fs.read().decode('latin'))
    return digest
```

(6) 传输进度条打印 (Server、Client)

```
def progress_bar(loaded, total):
    ratio = float(loaded) / total
    percentage = int(ratio * 100)
    r = '\r[%s%s]%d%%' % (">" * percentage, " " * (100 - percentage), percentage)
    sys.stdout.write(r)
    sys.stdout.flush()
```

(7) 列出工作目录下文件相对路径 (Server)

```
def get_filelist(path):
    allfiles = []
    if not os.path.exists(path):
        return -1
    for root, dirs, names in os.walk(path):
        for filename in names:
            allfiles.append(os.path.join(root, filename).replace(path, ''))
    return allfiles
```

(8) 对每个传输分组进行加密的函数 (Server)

```
def transfer_encrypt(message):
    key, iv = get_des_param()
    cipher = DES.des_cbc_encrypt(message, key, iv, 'b')
    key_encrypted = RSA.Encrypt(libnum.s2n(key), client_e, client_n)
    iv_encrypted = RSA.Encrypt(libnum.s2n(iv), client_e, client_n)
    return key_encrypted, iv_encrypted, cipher
```

(9) 对每个传输分组进行解密的函数 (Client)

```
def transfer_decrypt(cipher, key_encrypted, iv_encrypted):
    key = RSA.Decrypto(key_encrypted, server_d, server_d)
    iv = RSA.Decrypto(iv_encrypted, server_d, server_d)
    message = DES.des_cbc_decrypt(cipher, libnum.n2s(key), libnum.n2s(iv), 'b')
    return message
```

(10) 处理客户端的下载请求 (Server)

```
def get_file_stream(filename):
    print("\n*****")
    filepath = SERVER_WORKDIR + filename
    print("Send: " + filepath)
    if os.path.isfile(filepath):
        file_size = str(os.path.getsize(filepath))
        mainSocket.send(file_size.encode())
        while True:
            confirm = mainSocket.recv(10).decode()
            if confirm == "yes":
                break
```

```

count = (int(file_size) // BUFFSIZE) + 1
file_digest = get_file_md5(filepath)
send_size = 0
f = open(filepath, 'rb')
print("Start transferring...")
while count:
    filedata = b64encode(f.read(BUFFSIZE)).decode()
    key_encrypted, iv_encrypted, allfiles_encrypted = transfer_encrypt(filedata)
    mainSocket.send(allfiles_encrypted.encode())
    while True:
        confirm = mainSocket.recv(10).decode()
        if confirm == "yes":
            break
    mainSocket.send(str(key_encrypted).encode())
    while True:
        confirm = mainSocket.recv(10).decode()
        if confirm == "yes":
            break
    mainSocket.send(str(iv_encrypted).encode())
    while True:
        confirm = mainSocket.recv(10).decode()
        if confirm == "yes":
            break
    send_size += BUFFSIZE
    if send_size > int(file_size):
        send_size = file_size
    progress_bar(send_size, int(file_size), 10))
    count -= 1
print("\nGet the file Successfully.")
f.close()
mainSocket.send(file_digest.encode())
while True:
    confirm = mainSocket.recv(10).decode()
    if confirm == "yes":
        break
else:
    print("[Error]: Can't find the file")
    mainSocket.send("0".encode())
print("*****")
return 0

```

(11) 请求服务端传输文件 (Client)

```

def get_file_stream(filename):
    print("\n*****")
    print("Ready to transfer the file")
    start = time.time()
    clientServer.settimeout(10000)
    filepath = CLIENT_WORKDIR + filename
    try:
        file_size = clientServer.recv(BUFFSIZE).decode()
        clientServer.send("yes".encode())
        file_size = int(file_size, 10)
        count = (int(file_size) // BUFFSIZE_SENT) + 1
        if file_size > 0:
            f = open(filepath, 'wb')
            recv_size = 0
            print("Start transferring...")
            print("The file needs to be encrypted/decrypted, please wait...")

```

```

while count:
    filedata_encrypted = clientServer.recv(BUFSIZE).decode()
    clientServer.send("yes".encode())
    key_encrypted = int(clientServer.recv(BUFSIZE).decode())
    clientServer.send("yes".encode())
    iv_encrypted = int(clientServer.recv(BUFSIZE).decode())
    clientServer.send("yes".encode())
    filedata_decrypted = transfer_decrypt(filedata_encrypted, key_encrypted, iv_encrypted)
    f.write(b64decode(filedata_decrypted.encode()))
    recv_size += BUFSIZE_SENT
    if recv_size > file_size:
        recv_size = file_size
    progress_bar(recv_size, file_size)
    count -= 1
f.close()
recv_digest = clientServer.recv(32).decode()
clientServer.send("yes".encode())
if recv_digest != get_file_md5(filepath):
    print("\n[Warning]: The file you received may be broken! ")
else:
    end = time.time()
    print("\nGet the file successfully!")
    print("Spend time: " + str(int(end - start)) + "s")
else:
    print("[ERROR]: Can't find file")
except:
    print("\n[Error]: Transfer failed")
    clientServer.close()
print("*****")
return 0

```

6.5 运行结果及分析

6.5.1 基本功能运行

(1) 服务端运行

Waiting for connection...

(2) 客户端运行并请求列出文件列表

Input one of the following command:

```

[+] list -----list the file
[+] get1 [filename] -----get the file by stream
[+] get2 [filename] -----get the file after encrypt
[+] exit -----close socket and exit
[+] list

```

```

*****:
The list of files:
1.png
A_DeepLearning_Approachfor_Network_Intrusion_Detection_System.pdf
cumt.txt
ISC_Ch9.pptx
Learning_A_Static_Analyzer-A_Case_Study_on_a_Toy_Language.pdf
paper.pdf
paper.pdf.encrypted
*****:

```


对于这个问题，一开始的解决办法是使用 `sleep` 函数在发送后停顿一段时间，但是有时仍会出现错误。后来改为客户端每收到一个消息，就回应一个“yes”，服务端确认收到后再发送下一个消息，这也虽然不会发生错误，但是明显是会减低通信的效率。

(2) 关于签名提到的加密传输过程的两种思路，我在代码中都进行了实现看，即一种是服务端整个文件加密完成后再传输，客户端整个文件接收后再解密；一种是服务端对每个原文件分组加密传输，客户端对收到的每个原文件分组解密并写入。

实验中，我一开始使用的是第一种方式，但是当文件较大时，由于是使用的自己编写的分组加密算法，加密效率较低、时间较长，会造成客户端长时间未收到响应而超时。于是实现了第二种方式，此方式每个传输分组都使用了不同的 `key` 和 `IV`，安全性有所提高，且是以文件流的形式传输，即使文件很大也可以传输而不会超时，只不过时间会比较久。

6.6.2 改进方案

综合上面的实验，我们可以对文件加密传输系统进一步的改进：

(1) 使用安全的流密码代替分组密码

本系统使用自己编写的分组密码进行加密，效率还是比较低的，在传输大文件时比较耗时。因此，考虑可以使用加密效率更高的流密码来进行加密，而流密码的安全性也是比较高，这也可以提高文件传输的速度。

(2) 对消息的验证方面可以使用安全性更高的 SHA512

6.6.3 实验感想

本次综合实验中，可以说是对前面实验的一次总结与应用，并且对密码协议也有了一定的了解。通过分组密码加密大文件，公钥密码加密短密钥，md5 和签名实现文件的完整性验证，可以说在实际应用的大多数情况下，都是各种密码体制相互配合，共同形成一个安全的系统。通过这次实验，让我对密码学有了更加整体的学习与认识，比较有利于形成一个完整的知识体系。

附录

[1]DES 在线加密网站: <http://tool.chacuo.net/cryptdes>

[2]md5 在线加密网站: <https://md5jiami.51240.com>

[3]Average case error estimates for the strong probable prime test (<https://www.math.dartmouth.edu/~carlp/PDF/paper88.pdf>)

[4]课程设计完整代码: <https://github.com/LetheSec/Cryptography-Course-Design>