**江苏科技大学**

**课程实验报告**

课 程： 计算机网络与安全

课 题： 对称加密算法的网络应用

学 院： 计算机学院

姓 名： 陈四贵

班 级： 1822107101

学 号： 182210710119

指导老师： 张笑非

目 录

[一、 实验目的 1](#_Toc55916465)

[二、 实验内容 1](#_Toc55916466)

[三、 实验原理 1](#_Toc55916467)

[四、 实验步骤 1](#_Toc55916468)

[五、 实验结果分析 2](#_Toc55916469)

[六、 思考题 2](#_Toc55916470)

# 实验目的

1.掌握基于Python DES的Key定义方法；

2.掌握基于Python DES的加密/解密方法；

3.掌握基于Python RSA的加密/解密方法；

4.掌握基于Python RSA的签名签署/签名验证方法。

# 实验条件

1.Python运行环境；

2.DES包(<https://github.com/0x10001/des>)；

3.Python RSA.

# 实验步骤

**对称加密算法的网络算法部分：**

步骤1·安装DES：

$ pip install des

步骤2·定义Key：

首先，通过传递您的加密/解密密钥来定义DesKey对象。 密钥的长度应为8、16或24。将自动为您选择算法。 请注意，密钥应在Python 3中以字节形式写入。

1. **from** des **import** DesKey
2. key0 = DesKey(b"some key")                  # for DES
3. key1 = DesKey(b"a key for TRIPLE")          # for 3DES, same as "a key for TRIPLEa key fo"
4. key2 = DesKey(b"a 24-byte key for TRIPLE")  # for 3DES
5. key3 = DesKey(b"1234567812345678REAL\_KEY")  # for DES, same as "REAL\_KEY"

您可以通过调用is\_single()或is\_triple()方法来知道密钥是用于DES还是3DES算法。

1. key0.is\_single()  # -> True
2. key1.is\_triple()  # -> True
3. key2.is\_single()  # -> False
4. key3.is\_triple()  # -> False

步骤3·加密/解密：

通过从DesKey对象调用encrypt()方法来加密消息，或者通过调用decrypt()来解密消息。请注意，消息应在Python 3中写为字节

key0.encrypt(b"any long message") # -> b"\x14\xfa\xc2 '\x00{\xa9\xdc;\x9dq\xcbr\x87Q"

默认情况下，使用ECB模式。 您可以通过传递参数initial初始值来启用CBC模式。参数可以是长度为8的字节对象，也可以是使用big-endian的整数。

1. key0.encrypt(b"any long message", initial=0)        # -> b"\x14\xfa\xc2 '\x00{\xa9\xb2\xa5\xa7\xfb#\x86\xc5\x9b"
2. key0.encrypt(b"any long message", initial=b"\0"\*8)  # same as above

DES算法要求消息的长度为8的倍数。默认情况下，要加密/解密的消息的长度由用户确定。 您可以选择打开PKCS#5填充模式（通过将参数padding传递给TRUTHY值），告诉Python为您加密之前进行填充。

key0.encrypt(b"abc", padding=True) # -> b"%\xd1KU\x8b\_A\xa6"

在使用PKCS5填充进行解密时，仍然需要消息的长度为8的倍数。但是在解密之后，Python会丢弃填充字符。

key0.decrypt(b"%\xd1KU\x8b\_A\xa6", padding=True) # -> b"abc"

**非对称加密算法的网络算法部分：**

步骤1·安装RSA：

pip install rsa

步骤2·生成Key：

您可以使用rsa.newkeys()函数创建一个密钥对：

1. >>> **import** rsa
2. >>> (pubkey, privkey) = rsa.newkeys(512)

或者，您可以使用rsa.PrivateKey.load\_pkcs1()和rsa.PublicKey.load\_pkcs1()从文件中加载密钥：

1. >>> **import** rsa
2. >>> with open('private.pem', mode='rb') as privatefile:
3. ...     keydata = privatefile.read()
4. >>> privkey = rsa.PrivateKey.load\_pkcs1(keydata)

步骤3·加密/解密：

要加密或解密消息，请使用rsa.encrypt() 和rsa.decrypt()。假设Alice想发送一条消息，只有Bob可以阅读。

3.1 Bob生成一个密钥对，并将公钥提供给Alice。这样做是为了让Alice肯定知道密钥确实是Bob的密钥(例如，通过移交包含密钥的U盘)。

1. >>> **import** rsa
2. >>> (bob\_pub, bob\_priv) = rsa.newkeys(512)

3.2 Alice编写一条消息，并以UTF-8对其进行编码。RSA模块仅对字节进行操作，而不对字符串进行操作，因此此步骤是必需的。

>>> message = 'hello Bob!'.encode('utf8')

3.3 Alice使用Bob的公钥加密邮件，然后发送加密后的邮件。

1. >>> **import** rsa
2. >>> crypto = rsa.encrypt(message, bob\_pub)

3.4 Bob收到消息，并用他的私钥解密。

由于Bob拥有该私人密，因此Alice可以确保他是唯一可以阅读该消息的人。 但Bob不确定是不是Alice发送了该消息，因为她没有签名。

RSA只能加密小于密钥的消息。随机填充会丢失几个字节，其余字节可用于消息本身。例如，一个512位密钥可以编码一个53字节的消息（512位=64字节，其中11个字节用于随机填充和其他填充）。请参阅使用大文件以获取有关如何使用大文件的信息。

更改加密信息可能会导致rsa.pkcs1.DecryptionError。如果要确定，请使用rsa.sign()。

1. >>> crypto = rsa.encrypt(b'hello', bob\_pub)
2. >>> crypto = crypto[:-1] + b'X' # change the last byte
3. >>> rsa.decrypt(crypto, bob\_priv)
4. Traceback (most recent call last):
5. ...
6. rsa.pkcs1.DecryptionError: Decryption failed

注意：切勿显示 rsa.pkcs1.DecryptionError异常的堆栈跟踪。 它显示了异常在代码中的何处发生，从而泄漏了有关密钥的信息。 这只是一小部分信息，但每一点都使破解密钥变得更加容易。

步骤4·签名/验证：

您可以使用 rsa.sign()函数为邮件创建分离的签名：

1. >>> (pubkey, privkey) = rsa.newkeys(512)
2. >>> message = b'Go left at the blue tree'
3. >>> signature = rsa.sign(message, privkey, 'SHA-1')

这将使用SHA-1对消息进行哈希处理。 也可以使用其他哈希方法，有关详细信息，请查看 rsa.sign() 函数文档，然后用私钥对哈希签名。

可以在单独的操作中计算哈希和签名（即，在客户端计算机上生成哈希，然后在远程服务器上使用私钥签名）。 要对消息进行哈希处理，请使用rsa.compute\_hash() 函数，然后使用 rsa.sign\_hash() 函数对哈希进行签名：

1. >>> message = b'Go left at the blue tree'
2. >>> hash = rsa.compute\_hash(message, 'SHA-1')
3. >>> signature = rsa.sign\_hash(hash, privkey, 'SHA-1')

为了验证签名，请使用 rsa.verify() 函数。 如果验证成功，此函数将返回True：

1. >>> message = b'Go left at the blue tree'
2. >>> rsa.verify(message, signature, pubkey)
3. True

修改消息，签名不再有效，并抛出rsa.pkcs1.VerificationError：

1. >>> message = b'Go right at the blue tree'
2. >>> rsa.verify(message, signature, pubkey)
3. Traceback (most recent call last):
4. File "<stdin>", line 1, **in** <module>
5. File "/home/sybren/workspace/python-rsa/rsa/pkcs1.py", line 289, **in** verify
6. **raise** VerificationError('Verification failed')
7. rsa.pkcs1.VerificationError: Verification failed

注意: 切勿显示rsa.pkcs1.VerificationError异常的堆栈跟踪。它显示了异常在代码中的何处发生，从而泄漏了有关密钥的信息。这只是一小部分信息，但每一点都使破解密钥变得更加容易。

除了消息，您还可以使用类似文件的对象调用rsa.sign()和rsa.verify()。如果消息对象具有read(int)方法，则假定它是一个文件。 在这种情况下，文件当时将以1024字节的块进行哈希处理。

1. >>> with open('somefile', 'rb') as msgfile:
2. ...     signature = rsa.sign(msgfile, privkey, 'SHA-1')
3. >>> with open('somefile', 'rb') as msgfile:
4. ...     rsa.verify(msgfile, signature, pubkey)

步骤5·处理大文件：

RSA只能加密小于密钥的消息。 随机填充会丢失几个字节，其余字节可用于消息本身。 例如，一个512位密钥可以编码一个53字节的消息（512位= 64字节，其中11个字节用于随机填充和其他填充）。

将RSA用于较大文件的最常见方法是使用诸如AES或DES3之类的密码，用随机密钥加密文件，然后使用RSA加密随机密钥，您将把加密文件和加密密钥一起发送给收件人。完整的流程是：

5.1 生成随机密钥

1. >>> **import** rsa.randnum
2. >>> aes\_key = rsa.randnum.read\_random\_bits(128)

5.2 使用该密钥通过AES加密文件。

5.3 使用RSA加密AES密钥

>>> encrypted\_aes\_key = rsa.encrypt(aes\_key, public\_rsa\_key)

5.4 将加密文件与cryptod\_aes\_key一起发送

5.5 接收者现在可以逆转此过程以获得加密文件。

# 实验记录

# 思考题

1.ECB模式和CBC模式分别在什么场景上更加适合采用？

2.PKCS#5填充模式在实际应用中有哪些优势？

3.如何利用rsa.pkcs1.VerificationError异常的堆栈跟踪来破解秘钥？

4.试设计并完成一个完整的大文件加密流程代码设计。