山东大学 软件 学院

Python密码学编程课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202100150209 | 姓名： 杨佳庆 | | 班级： 21级网安班 |
| 实验题目：实验4：实现非对称加密编程 | | | |
| 实验学时：2 | | 实验日期： 2023.10.10 | |
| 实验目的：   * 实现非对称加密密钥生成 * 实现非对称密钥对象和字节流的相互转换 * 实现RSA非对称加解密（1.RAW-RSA，2.随机padding后的安全RSA） | | | |
| 硬件环境：  PC机 | | | |
| 软件环境：  PyCharm | | | |
| 实验步骤与内容：   1. 安装gmpy2（若之前没有安装）      1. 利用cryptography中的 asymmetric的rsa模块实现rsa密钥对的生成   生成RSA密钥对：使用cryptography库的as.rsa模块实现RSA密钥对的生成可以使用rsa.generate\_private\_key函数生成私钥使用私钥的public\_key属性获取公钥。  from cryptography.hazmat.backends import default\_backend from cryptography.hazmat.primitives.asymmetric import rsa  private\_key = rsa.generate\_private\_key( public\_exponent=65537, key\_size=2048, backend=default\_backend() )   public\_key = private\_key.public\_key()   1. 利用cryptography中的serialization模块实现rsa密钥字节数组到密钥对象的相互转换   密钥对象和字节流的相互转换：使用cryptography库serialization模块实现RSA密字节数组到密钥对象的相互转换。可以使用load\_pem\_private\_key和load\_pem\_public\_key函数将密钥节数组转换为私钥和公钥对象，使用private\_key.private\_bytes和public\_key.public\_bytes函数将密钥对象转换字节数组。  # Convert the private key into bytes. We won't encrypt it this time. private\_key\_bytes = private\_key.private\_bytes(  encoding=serialization.Encoding.PEM,  format=serialization.PrivateFormat.TraditionalOpenSSL,  encryption\_algorithm=serialization.NoEncryption() )  # Convert the public key into bytes. public\_key\_bytes = public\_key.public\_bytes(  encoding=serialization.Encoding.PEM,  format=serialization.PublicFormat.SubjectPublicKeyInfo )  # Convert the private key bytes back to a key. # Because there is no encryption of the key, there is no password. private\_key = serialization.load\_pem\_private\_key(  private\_key\_bytes,  backend=default\_backend(),  password=None)  public\_key = serialization.load\_pem\_public\_key(  public\_key\_bytes,  backend=default\_backend())   1. 利用gmpy2库以及cryptography中的rsa实现简单Raw Rsa加密和解密   简单的RAW-RSA加密和解密：使用gmpy2库和cryptography库的rsa模块实现简单的RAW-RSA（无padding）加密和解密。可以使用gmpy2库的powmod函数进行密和解密操作，注意要将明文和密文表示为整数。  class RSACrypto(object):  *'''  RSACrypto  '''* def \_\_init\_\_(self, public\_key, private\_key):  self.public\_key = public\_key  self.private\_key = private\_key   @staticmethod  def int\_to\_bytes(i):  # i might be a gmpy2 big integer; convert back to a Python int  i = int(i)  return i.to\_bytes((i.bit\_length()+7)//8, byteorder='big')   @staticmethod  def bytes\_to\_int(b):  return int.from\_bytes(b, byteorder='big')   @staticmethod  def simple\_rsa\_encrypt(m, publickey):  numbers = publickey.public\_numbers()  return gmpy2.powmod(m, numbers.e, numbers.n)   @staticmethod  def simple\_rsa\_decrypt(c, private\_key):  numbers = private\_key.private\_numbers()  return gmpy2.powmod(c, numbers.d, numbers.public\_numbers.n)   def encrypt(self, message):  public\_key = self.public\_key  if not isinstance(message, bytes):  message = message.encode()  if not public\_key:  print("\nNo public key loaded\n")  else:  message\_as\_int = self.bytes\_to\_int(message)  cipher\_as\_int = self.simple\_rsa\_encrypt(message\_as\_int,public\_key)  cipher = self.int\_to\_bytes(cipher\_as\_int)  return cipher   def decrypt(self, cipher\_hex):  private\_key = self.private\_key  if not isinstance(cipher\_hex, bytes):  cipher\_hex = cipher\_hex.encode()  if not private\_key:  print("\nNo private key loaded\n")  else:  cipher = binascii.unhexlify(cipher\_hex)  cipher\_as\_int = self.bytes\_to\_int(cipher)  message\_as\_int = self.simple\_rsa\_decrypt(cipher\_as\_int, private\_key)  message = self.int\_to\_bytes(message\_as\_int)  return message  (message\_as\_int)  return message  加密：  逻辑：string->byte->int->encrypt->byte  message\_as\_int = self.bytes\_to\_int(message) cipher\_as\_int = self.simple\_rsa\_encrypt(message\_as\_int,public\_key) cipher = self.int\_to\_bytes(cipher\_as\_int) return cipher  输出时转换为16进制输出：  binascii.hexlify(cipher)    解密：逻辑：hex->byte->int->decrypt->byte  cipher = binascii.unhexlify(cipher\_hex) cipher\_as\_int = self.bytes\_to\_int(cipher) message\_as\_int = self.simple\_rsa\_decrypt(cipher\_as\_int, private\_key) message = self.int\_to\_bytes(message\_as\_int) return message     1. 利用crypto的PKCS1\_OAEP实现安全rsa加解密   安全RSA加解密：使用c库的rsa模块实现安全RSA加解密，使用PKCS1AEP padding进行加解密操作。可以使用rsa模块的PKCS1\_OAEP.encrypt和PKCS1\_OAEP.decrypt函数进行加密和解密操作。  def rsa\_padding\_OAEP\_encrypt(self, message):  if not isinstance(message, bytes):  message = message.encode()  cipher = public\_key.encrypt(  message,  padding.OAEP(  mgf=padding.MGF1(algorithm=hashes.SHA256()),  algorithm=hashes.SHA256(),  label=None  )  )  return cipher  def rsa\_padding\_OAEP\_decrypt(self, cipher\_hex):  if not isinstance(cipher\_hex, bytes):  cipher\_hex = cipher\_hex.encode()  message = private\_key.decrypt(  cipher\_hex,  padding.OAEP(  mgf=padding.MGF1(algorithm=hashes.SHA256()),  algorithm=hashes.SHA256(),  label=None # rarely used. Just leave it 'None'  ))  return message  在加密过程中，rsa\_padding\_OAEP\_encrypt()函数首先将输入的消息（message）转换为字节流然后，使用公钥（public\_key）对消息进行加密，采用了PKCS1\_OAEP填充方案（padding.OAEP）来保护加密数据。  PKCS1\_OAEP填充方案使用了两个独立的散列算法，其中mgf参数指定用于生成掩码的散列算法，而algorithm参数指定了用于加密的散列算法。mgf参数和algorithm参数都选择了SHA256散列算法。  在解密过程中，rsa\_padding\_OAEP\_decrypt()函数首先将输入的密文（cipher\_hex）转换为字节流。然后，使用私钥（private\_key）对密文进行解密，同样采用了PKCS1\_OAEP填充方案（padding.OAEP）。  在解密过程中，与加密相对应，指定mgf参数和algorithm参数采用相同的SHA256散列算法。     1. 实现共模攻击和同态攻击（选做）   共模攻击：  def common\_modulus\_attack(cipher1, cipher2, public\_key1, public\_key2):  # 获取两个公钥的模数和指数  n1 = public\_key1.public\_numbers().n  e1 = public\_key1.public\_numbers().e  n2 = public\_key2.public\_numbers().n  e2 = public\_key2.public\_numbers().e   # 使用中国剩余定理（CRT）求解明文  gcd, s1, s2 = extended\_gcd(e1, e2)  m1 = pow(cipher1, s1, n1)  m2 = pow(cipher2, s2, n2)   # 计算明文乘积  message =(m1 \* m2) % (n1 \* n2)  return message  def extended\_gcd(a, b):  if a == 0:  return b, 0, 1  else:  gcd, x, y = extended\_gcd(b % a, a)  return gcd, y - (b // a) \* x, x  # 假设Alice和Bob分别使用不同的公钥加密了相同的明文 ciphertext\_alice = 123 ciphertext\_bob = 123 private\_key\_alice = rsa.generate\_private\_key(  public\_exponent=65537,  key\_size=2048,  backend=default\_backend()  ) public\_key\_alice = private\_key\_alice.public\_key() private\_key\_bob = rsa.generate\_private\_key(  public\_exponent=65537,  key\_size=2048,  backend=default\_backend()  ) public\_key\_bob = private\_key\_bob.public\_key()  # 共模攻击 plaintext = common\_modulus\_attack(ciphertext\_alice, ciphertext\_bob, public\_key\_alice, public\_key\_bob) print("明文:", plaintext)    同态攻击：  def homomorphic\_attack(message1, message2, public\_key):  # 获取公钥的模数和指数  n = public\_key.public\_numbers().n  e = public\_key.public\_numbers().e   cipher1 = simple\_rsa\_encrypt(message1, public\_key)  cipher2 = simple\_rsa\_encrypt(message2, public\_key)   # 使用同态性质进行运算  ciphertext\_product = (cipher1 \* cipher2) % public\_key.public\_numbers().n   # 密文乘积的解是明文的乘积  # 密文乘积共模取余的解仍然是明文的乘积  val1 = simple\_rsa\_decrypt(ciphertext\_product, private\_key)  print(val1)  val2 = simple\_rsa\_decrypt(cipher1 \* cipher2, private\_key)  print(val2)  val3 = message1 \* message2  print(val3)  # 解密得到结果 plaintext\_product = pow(ciphertext\_product, e, n) return plaintext\_product  # 假设Alice和Bob分别使用相同的公钥加密了不同的明文 ciphertext\_alice = 123 ciphertext\_bob = 456 private\_key = rsa.generate\_private\_key(  public\_exponent=65537,  key\_size=2048,  backend=default\_backend()  ) public\_key = private\_key.public\_key()  homomorphic\_attack(ciphertext\_alice, ciphertext\_bob, public\_key)     1. 通过代码4-10了解PKCS1v15攻击（选做）   **填充攻击Oracle**  攻击的总体数学思路：  基本加解密原理：  加密：c = (m^e) % n  解密：m = (c^d) % n  基本定理：  (a \* b) % p = (a % p \* b % p) % p  (a ^ b) % p = ((a % p)^b) % p  假设此时有一个随机的明文s，通过构造这样一种密文c\_x，使得：  c\_x = (c \* s^e) % n  =( c % n \* s^e % n ) % n  = (c \* c\_s) % n  其中c\_s为明文s加密后对应的密文  进而反推对c\_x进行解密，假设其明文为s\_m，此时有：  s\_m = (c\_x^d) % n  s\_m = ((c \* s^e) % n)^d % n # 定理：(a ^ b) % p = ((a % p)^b) % p  = (c \* s^e) ^ d % n  = (((m^e)%n) \* s^e)^d%n  = ((m^e)%n)^d \* s^e^d) % n # 定理：(a \* b) % p = (a % p \* b % p) % p  = ((m^e)%n)^d%n \* s^e^d % n) %n # m = (m^e)%n)^d%n  = (m \* s^e^d % n) % n  = (m \* (s^e%n)^d%n) % n # s = (s^e%n)^d%n  = (m \* s) % n  总体上来说，攻击者可以通过不断的发送特定的s给RSA解密服务端，通过服务端返回的解密明文ms是否符合PKCSv1.5规范来缩小明文m的取值范围，直到最后得到精确的明文m。  攻击者会设置一个假oracle，检查解密后的密文是否有适当的填充（以字节[0, 2]开头）。  下面是 RSA Oracle 攻击的步骤分解：  攻击者通过初始化 B（区间大小 s（致盲值序列））、（区间集）、i（迭代计数器）和 n（公共模数）的值来设置致盲步骤(blinding step)。 攻击者执行搜索步骤有三种变化： 步骤 2a：使用 find\_s() 函数找到合适的 s 值，开始搜索。 步骤 2b： 通过递增前一个 s 值，搜索一个以上的剩余区间。 步骤 2c：通过计算新的 ri 值并在给定的限制条件下找到合适的 s 值，在还剩一个区间时进行搜索。 .攻击者根据当前区间和缩小解集步骤中选择的 s 值计算新区间，从而缩小解集。  攻击者通过仅剩的区间计算解，并返回该区间的边界。 最后，攻击者可以使用 RSAOracleAttacker 类对密文进行攻击，并恢复原始明文。 | | | |
| 结论分析与体会：  在这个实验中，我们学习了如何使用Python的cryptography库实现非对称加密的一些基本操作。主要包括密钥生成、密钥对象和字节流之间的转换、以及基于RSA的加解密算法。  在密钥生成部分，我们使用了cryptography库中的rsa.generate\_private\_key()函数生成了RSA密钥对。通过指定公钥指数（public\_exponent）和密钥长度（key\_size），我们可以生成不同的密钥对。  然后，我们学习了如何使用cryptography库中的serialization模块将RSA密钥对象转为字节流表示，并可以将字节流转换回密钥对象。这种转换非常重要，因为密钥在网络传输或存储时通常以字节流的形式进行。  接下来，我们学习了如何使用gmpy2库和cryptography库中的RSA模块来实现简单的RAW-RSA加和解密。注意，这是一个非常不安全且易受攻击的实现。在实际应用中，我们应该使用更安全的填充方案来保护RSA加密。  在下一部分，我们学习了如何使用cryptography库中的PKCS1\_OAEP填充案来实现更安全的RSA加解密。PKCS1\_OAEP填充方案提供了更高的安全性，并且防止一些常见的RSA加密攻击。在实践中，我们应该使用更安全的加密库和填充方案来保护加密通信。  在最后两个部分，我们学习了共模攻击和同态攻击的概念。这些是高级的攻击技术，可以利用特定条件下的RSA实现中的漏洞来实现攻击。  通过这个实验，我们深入了解了非对称加密的基本原理、密钥、加解密操作、填充方案的选择等方面。我们还学习了一些基本的攻击技术和如何使用cryptography库来实现安全的加密操作。对于进一步加强我们对非对称加密算法的理解和应用具有很大帮助。 | | | |