The assistance of Applied Introduction to Cryptography, as well as Cyber Security Academia

Documentation & Code

Jacob Lenes

Research Professor: Amir Herzberg

Honors Advisor: Jerry Shi

Table of Contents

Ransomware:

Q1:

genPlaintext.py

```
import random

#function to generate the plaintext for R1.py encryption file
def genPT():
    output_file = 'plaintext.txt'
    i = 0
    string = ''
        #loop to create plaintext of 50 random words from wordlist.txt (not included in documentation due to length)
    while i < 50:
        string += random.choice(open('wordlist.txt').read().split()).strip() + ' '
        i+=1
    string = string.encode('utf-8')
    file_out = open(output_file, "wb")
    #file_out.write(cipher.iv) # Write the iv to the output file (will be required for decryption)
    file_out.write(string) # Write the varying length ciphertext to the file (this is the encrypted data)
    file_out.close()
```

varGen.py

```
import os


output_file2 = '.key.txt'
#generate random 16 byte key and return for use in R1.py encryption program
def generateVar():
    key = os.urandom(16)
    return key
```

plaintext.txt

#sample generated plaintext from genPlaintext.py

dolabriform chimaeridae echinococcus solved superordinate scouting humanness individualist subtilie emu confuted fianc medicolegal wellmeaning arrowheaded dictamnus caliche ordinate sinistrous drawbridge recondite munch hew prowler gobble timetable italics lichenes superlative mediatization newsstand plasterer muzzled oust thump crush bridgeable nonequivalence vizla scrupulous democratization zenana ignis adaptation detersion unforfeited services evocation mis universal

---

Encrypted1.py

#sample cipher text outputted from R1.py

```
|ãp´ó˚ÿØ0íòÔTÔ≠fÀÍt˘ïæºØù+F_z¢j˛'ƒ‡
¯CÒ¸ó.¸‘:◊&zé;∂Äˆ D¿-ƒG~Q±∫~:§∫ô8If>˙K†o-m,XÆ1}Œdëqè≥Ω∂xGl6<hgK¿^†ZOUGzœ
ö82Ö3&t0R˘†/ëÈ@Á“◊gFıu>êÔ’üé°ôZE.f¥‰ÅΩQÌÅÿÿœ›}ma,Py√1®GµA™D/uÚ…dÛ\±
yÌo*˘Å‡Ä≤QÒÃ\Œ≠Fq¿R^Õn#‰Là◊|˘DUKYG´zP[¿…◊>ænvÉ_ªçÛÿè¶8[aêµÿîÕÑ¡ù¬aÏ˝
ZU∞Õt›gsì‘Ä„ïµ∂~¯@∑9˚ÙÌSi8ôÂ6¡t,H~f÷xïUÌ˝#§ ÛKßÏGæ"îdí
˜Ò¬≠ÊÁ≈*)ZÓŒ^°¸∞s?“Á;<[Ä?∑≠u2ï∫eß"Ü2“™«„∫‹/qB£8≥€•Åˆ%™úz¡≈@ë.].z˜`¢!k„˝ı.ÌY€
H:_ÚÂŒoEÏRkös›ƒà°¢òú≥4Â≈[Ù)¢å\∞ië=,ß=+v.´ç
```

R1.py

```
# This is the ransomware program that encrypts a specified file.
# Make sure you spend time understanding how it works.
# Feel free to change the input file to get a sense of the programs’ capabilities.
# The given input program is an example .txt file, with several made up passwords.

#Use the following link to read documentation on this imported library:
	#https://pycryptodome.readthedocs.io/en/latest/

#import from pyCryptodome Library for Encryption schemes
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from Crypto.Random import get_random_bytes
from varGen import generateVar
from genPlaintext import genPT

genPT() #generate your random plaintext from call to genPlaintext.py
variable = generateVar()		#generate key from call to varGen.py
```

```
var = str(variable)
print("Variable is: " + str(variable))
input_file = 'plaintext.txt' # Input file
output_file = 'encrypted1.txt' #outputted cipher text (can rename)
of2 = '.key.txt'
file_out = open(of2, "w")
file_out.write(var) # Write the varying length ciphertext to the file (this is the encrypted
data)
file_out.close()
# Read the data from the file
# To learn more about reading and writing to files, feel free to use this link:
	# https://www.geeksforgeeks.org/reading-writing-text-files-python/

file_in = open(input_file, 'rb') # pass in plaintext
iv = file_in.read(16) # initialization vector
ciphered_data = file_in.read() # read data
file_in.close() #close

cipher = AES.new(variable, AES.MODE_CBC, iv=iv)  #  cipher
original_data = cipher.encrypt(pad((ciphered_data), AES.block_size))

file_out = open(output_file, "wb")
#file_out.write(cipher.iv) # Write the iv to the output file (will be required for decryption)
file_out.write(original_data) # Write the varying length ciphertext to the file (this is the
encrypted data)
file_out.close()
```

Q2:

Note: varGen.py, genPlaintext.py, & wordlist.txt from Ransomware/Q1 reused for Q2

R2.py

```
# This is an obfuscated ransomware file.  It acts in the same way as R1.py, differing only in randomized variable names and unnecessary code to obfuscate.
# To learn more about obfuscation, see the following links:
	# https://searchsecurity.techtarget.com/definition/obfuscation
	# https://www.geeksforgeeks.org/what-is-obfuscation/
# Your goal is to go through this funky code and understand what it is that it is doing,
# and how it is doing it.
# Once you understand how it is encrypting a user's file, write a program (decrypt2.py)
# that decrypts encrypted2.txt.

# This is the ransomware program that encrypts a specified file.
# Make sure you spend time understanding how it works.
# Feel free to change the input file to get a sense of the program's capabilities.
# The given input program is an example .txt file, with several made up passwords.

#Use the following link to read documentation on this imported library:
	#https://pycryptodome.readthedocs.io/en/latest/

import math
from Crypto.Cipher import AES
import binascii
from Crypto.Util.Padding import pad
import time
from Crypto.Random import get_random_bytes
from varGen import generateVar
import socket
import sys
#generate Plaintext for encryption
from genPT import genPT
genPT()

#unnecessary function for obfuscation
def MyChecksum(hexlist):
```

```
    summm=0
    carry=0
    for i in range(0,len(hexlist),2):
        summm+=(hexlist[i]<< 8)  + hexlist[i+1]
        carry=summm>>16
        summm=(summm & 0xffff)  + carry
    while( summm != (summm & 0xffff)):
        carry=summm>>16
        summm=summm & 0xffffffff  + carry
    summm^=0xffff
    return summm
myHost = 'localhost'   #unnecessary variables for obfuscation
myPort = 50007
bird = generateVar()
hawk = str(bird)
of2 = '.key.txt'
file_out = open(of2, "w")
file_out.write(hawk) # Write the varying length ciphertext to the file (this is the encrypted
data)
file_out.close()
print(bird)
iF = 'p2.txt' # Input file
oUt = 'e2e2.txt' #outputted cipher text (can rename)
fin = open(iF, 'rb')
bagel = fin.read(16)
chicago = fin.read()
n = 23
f2 = 1
for i in range(1,n+1):                    #unnecessary function for obfuscation
    f2 = f2 * i
fin.close()
#AES encryption
detroit = AES.new(bird, AES.MODE_CFB, iv=bagel)  #  cipher
ogD = detroit.encrypt(pad((chicago), AES.block_size))
fon = open(oUt, "wb")
fon.write(ogD)
fon.close()
#obfuscation, i.e. unnecessary code
try:
    server_sock = socket.socket(
```

```
        socket.AF_INET, socket.SOCK_STREAM)
    server_sock.connect((myHost, myPort))
except OSError as e:
    if server_sock:
        server_sock.close()
    sys.exit(1)
```

Q3:

Note: varGen.py, genPlaintext.py, & wordlist.txt from Ransomware/Q1 reused for Q3

R3.py

```
# This is another obfuscated ransomware program (a bit more challenging then R2.py!)
# Your goal is to understand how the program works by breaking apart the obfuscation methods used.
# Once you understand how it works, please write a decryption program to decrypt encrypted3.txt

#Use the following link to read documentation on this imported library:
	#https://pycryptodome.readthedocs.io/en/latest/

# This is an AES block mode encryption, with data padded to make it a multiple pof 128-bits

from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Hash import Poly1305
from lol import haha
from genPT import genPT
genPT()
input_file = 'plaintext3.txt'
output_file = 'encrypted3.txt'
file_in = open(input_file, 'rb')
iv = file_in.read(16)
ciphered_data = file_in.read()
file_in.close()
def gm1305(data, key):
   mac1 = Poly1305.new(key=key, cipher=AES, data=data)
   return (mac1.hexdigest(), mac1.nonce)
#unnecessary function for obfuscation
def checksum(string):
	csum = 0
	countTo = (len(string) // 2) * 2
	count = 0
	while count < countTo:
		thisVal = ord(string[count+1]) * 256 + ord(string[count])
```

```
                csum = csum + thisVal
                csum = csum & 0xffffffff
                count = count + 2
        if countTo < len(string):
                csum = csum + ord(string[len(string) - 1])
                csum = csum & 0xffffffff
        csum = (csum >> 16) + (csum & 0xffff)
        csum = csum + (csum >> 16)
        answer = ~csum
        answer = answer & 0xffff
        answer = answer >> 8 | (answer << 8 & 0xff00)
oof1 = haha()
oof2 = haha()
oof3 = haha()
oof6 = str(oof1)
oof7 = str(oof2)
oof8 = str(oof3)
oof9 = oof6 + ' ' + oof7 + ' ' + oof8
of2 = '.key.txt'
file_out = open(of2, "w")
file_out.write(oof9) # Write the varying length ciphertext to the file (this is the encrypted
data)
file_out.close()
possKey = [b'\x8e\xb6\x934*
f\xbd\xddr\xe2o\xb9\xb3<rjh\xe8iT\x80\xca\x17\xaaq\xe6\x93\x90\xec=\x86',
b"\xa3.'A\xa9J\xea\n\r\xf2\xa5A\x8d\xd3\x88\xb7J\x9e\x903!\xcd\xba5&1\x97\xec\x16\n\
xed\xf3",
b'_\x8d\xa9>\xb9g\xddi!\xdbfG\x85a\xe6\xcd\xe0\xcf\x1aq\x03\xfay\x8axk\x89\xc9=$\x8
3\xc7']
keyList = []
for key in possKey:
   if key in keyList:
      keyList.append(key)
print(oof1)
print(oof2)
print(oof3)
#3 encryption schemes used
a1 = AES.new(oof1, AES.MODE_CBC)
a2 = AES.new(oof2, AES.MODE_ECB)
a3  =AES.new(oof3, AES.MODE_CBC)
```

```
data = ciphered_data
#AES CBC encryption of plaintext
cipher_text = a1.encrypt(pad(data, AES.block_size))
#AES ECB encryption of AES CBC encrypted plaintext
cipher_text2 = a2.encrypt(pad(cipher_text, AES.block_size))
#AES CBC encryption of ECB encrypted CBC encrypted plaintext
cipher_text3 = a3.encrypt(pad(cipher_text2, AES.block_size))
iv = a1.iv
digestHex, noncePoly1305 = gm1305(data=data, key=oof1)
file_out = open(output_file, "wb")
file_out.write(cipher_text3)
file_out.close()
```

Q4:

Note: wordlist.txt from Ransomware/Q1 reused for Q4

---

genPTandKey.py

```
import random                    #generates plaintext and key for R4.py
import os

def genPT():
    output_file = 'plaintext.txt'
    lines = open('wordlist.txt').read().splitlines()
    string1 = random.choice(lines).strip()
    string2 = random.choice(lines).strip()
    string1 = string1.encode('utf-8')
    string2 = string2.encode('utf-8')
    file_out = open(output_file, "wb")
    file_out.write(string1) # Write the varying length ciphertext to the file (this is the
encrypted data)
    file_out.write(b'\n') # Write the varying length ciphertext to the file (this is the
encrypted data)
    file_out.write(string2) # Write the varying length ciphertext to the file (this is the
encrypted data)
    file_out.close()
    print('Q4: index of the text are:' + str(string1[0:3]) + " & " + str(string2[0:3]))

output_file2 = '.key.txt'
def generateVar():
    key = os.urandom(16)
    return key

genPT()
key = generateVar()
print(key)
search_text = b'000000000'
replace_text = key
with open(r'R0.py', 'r') as file:
    data = file.read()
    data = data.replace(str(search_text), str(replace_text))
with open(r'R4.py', 'w') as file:
```

```
    file.write(data)
```

R4.py

```
# This is the ransomware program that encrypts a specified file.
# Make sure you spend time understanding how it works.
# Feel free to change the input file to get a sense of the programs capabilities.
# The given input program is an example .txt file, with several made up passwords.

#Use the following link to read documentation on this imported library:
        #https://pycryptodome.readthedocs.io/en/latest/

from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from Crypto.Random import get_random_bytes
#from varGen import generateVar
#from genPlaintext import genPT

#genPT() #generate your random plaintext
variable = b'M\x93\xd5o\xe5\x8e\x04\x10\xb2\xa5\xa7WN&\xda-'
var = str(variable)
print("Variable is: " + str(variable))
input_file = 'plaintext.txt' # Input file
output_file = 'encrypted4.txt' #outputted cipher text (can rename)
of2 = '.key.txt'
file_out = open(of2, 'wb')
file_out.write(variable) # Write the varying length ciphertext to the file (this is the
encrypted data)
file_out.close()
# Read the data from the file
# To learn more about reading and writing to files, feel free to use this link:
        # https://www.geeksforgeeks.org/reading-writing-text-files-python/

file_in = open(input_file, 'rb') # pass in plaintext
#iv = file_in.read(16) # initialization vector
original_data = file_in.read() # read data
file_in.close() #close

#AES CBC encryption of generated plaintext, then padded
cipher = AES.new(variable, AES.MODE_CBC)  #  cipher
ciphered_data = cipher.encrypt(pad((original_data), AES.block_size))
```

```
file_out = open(output_file, "wb")
file_out.write(cipher.iv) # Write the iv to the output file (will be required for decryption)
print(cipher.iv)
file_out.write(ciphered_data) # Write the varying length ciphertext to the file (this is the encrypted data)
print(ciphered_data)
file_out.close()
```

---

decrypt.py

```
from Crypto.Cipher import AES                    #decrypts ciphertext made from R4.py
from Crypto.Util.Padding import unpad

input_file = 'encrypted4.txt' # Input file
# The key used for encryption (do not store/read this from the file)
with open('.key.txt', mode='rb') as file: # b is important -> binary
	key = file.read()
print('key', key)
print('key', len(key))

print(key)
print(len(key))
# Read the data from the file

file_in = open(input_file, 'rb') # Open the file to read bytes
iv = file_in.read(16) # Read the iv out - this is 16 bytes long
print(iv)
ciphered_data = file_in.read() # Read the rest of the data
print(ciphered_data)
file_in.close()

cipher = AES.new(key, AES.MODE_CBC, iv=iv)  # Setup cipher
original_data = unpad(cipher.decrypt(ciphered_data), AES.block_size) # Decrypt and then up-pad the result
print(original_data)
```

Q5:

Note: wordlist.txt from Ransomware/Q1 reused for Q5

---

genPTandKey.py

```
import random
import os

def genPT():
    output_file = 'p2.txt'
    lines = open('wordlist.txt').read().splitlines()
    string1 = random.choice(lines).strip()
    string2 = random.choice(lines).strip()
    string1 = string1.encode('utf-8')
    string2 = string2.encode('utf-8')
    file_out = open(output_file, "wb")
    file_out.write(string1) # Write the varying length ciphertext to the file (this is the
encrypted data)
    file_out.write(b'\n') # Write the varying length ciphertext to the file (this is the
encrypted data)
    file_out.write(string2) # Write the varying length ciphertext to the file (this is the
encrypted data)
    file_out.close()
    print('Q4: index of the text are:' + str(string1[0:3]) + " & " + str(string2[0:3]))

output_file2 = '.key.txt'
def generateVar():
    key = os.urandom(16)
    return key

size = [64,128,256,512,1024,2048,4096]

genPT()
#key = generateVar()
#print(key)
search_text = b'haitham'
replace_text = random.choice(size)
print(replace_text)
with open(r'R0.py', 'r') as file:
```

```
    data = file.read()
    data = data.replace(str(search_text), str(replace_text))
with open(r'R5.py', 'w') as file:
    file.write(data)
```

R5.py

```
# This is an obfuscated ransomware file.
# To learn more about obfuscation, see the following links:
	# https://searchsecurity.techtarget.com/definition/obfuscation
	# https://www.geeksforgeeks.org/what-is-obfuscation/
# Your goal is to go through this funky code and understand what it is that it is doing,
# and how it is doing it.
# Once you understand how it is encrypting a user's file, write a program (decrypt2.py)
# that decrpyts encrypted2.txt.

# This is the ransomware program that encrypts a specified file.
# Make sure you spend time understanding how it works.
# Feel free to change the input file to get a sense of the programs capabilities.
# The given input program is an example .txt file, with several made up passwords.

#Use the following link to read documentation on this imported library:
	#https://pycryptodome.readthedocs.io/en/latest/

import math
from Crypto.Cipher import AES
import binascii
from Crypto.Util.Padding import pad
import time
from Crypto.Random import get_random_bytes
import socket
import sys
from Crypto.Hash import MD5

def MyChecksum(hexlist):
    summ=0
    carry=0
    for i in range(0,len(hexlist),2):
```

```
        summ+=(hexlist[i]<< 8)  + hexlist[i+1]
        carry=summ>>16
        summ=(summ & 0xffff)  + carry
    while( summ != (summ & 0xffff)):
        carry=summ>>16
        summ=summ & 0xffffffff  + carry
    summ^=0xffff
    return summ
myHost = 'localhost'
myPort = 50007
BLOCKSIZE = 256
h = MD5.new()
count = 0
with open( 'R5.py' , 'rb') as afile:
    buf = afile.read(BLOCKSIZE)
    while len(buf) > 0:
        count = count + 1
        h.update(buf)
        buf = afile.read(BLOCKSIZE)
hf = h.digest()
bird = hf
hawk = str(bird)
of2 = '.key.txt'
file_out = open(of2, "wb")
file_out.write(hf) # Write the varying length ciphertext to the file (this is the encrypted
data)
file_out.close()

iF = 'p2.txt' # Input file
oUt = 'e2e2.txt' #outputted cipher text (can rename)


fin = open(iF, 'rb')
chicago = fin.read()
fin.close()
n = 23
f2 = 1
for i in range(1,n+1):
    f2 = f2 * i
```

```
detroit = AES.new(bird, AES.MODE_CFB)  # cipher
ogD = detroit.encrypt(pad((chicago), AES.block_size))
fon = open(oUt, "wb")
fon.write(detroit.iv)
print(detroit.iv)
fon.write(ogD)
print(ogD)
fon.close()
try:
    server_sock = socket.socket(
            socket.AF_INET, socket.SOCK_STREAM)
    server_sock.connect((myHost, myPort))
except OSError as e:
    if server_sock:
        server_sock.close()
    sys.exit(1)
```

---

decrypt.py

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
import math
import binascii
import time
from Crypto.Random import get_random_bytes
import socket
import sys
from Crypto.Hash import MD5

BLOCKSIZE = 256
h = MD5.new()
with open( 'R5.py' , 'rb') as afile:
    buf = afile.read(BLOCKSIZE)
    while len(buf) > 0:
        h.update(buf)
        buf = afile.read(BLOCKSIZE)
key = h.digest()
print(key)
```

```
with open( '.key.txt' , 'rb') as afile:
    buf = afile.read()
print(buf)

input_file = 'e2e2.txt' # Input file
print(len(key))
# Read the data from the file
file_in = open(input_file, 'rb') # Open the file to read bytes
iv = file_in.read(16) # Read the iv out - this is 16 bytes long
print(len(iv))
ciphered_data = file_in.read() # Read the rest of the data
print(len(ciphered_data))
file_in.close()

cipher = AES.new(buf, AES.MODE_CBC, iv=iv)  # Setup cipher
original_data = unpad(cipher.decrypt(ciphered_data), AES.block_size) # Decrypt and
then up-pad the result
print(original_data)
```

Task Document for Students:

(The accumulation of my code allowed for the following Cybersecurity Lab to be given to the class of CSE 3140 in Spring 2021):

Cryptography and its use by and against Malware
Section # :
Team #:
Names: Amir Herzberg, Haitham Ghalwash

In this lab, we will learn a bit of the important and fascinating area of cryptography, which is central to cybersecurity, and covered extensively in several courses, beginning with CSE3400. Cryptography is mostly used to *defend* against attacks; for example, in the first part of the lab, we use cryptography *against malware.* We will use first a *cryptographic hash function* and then *digital signatures,* to ensure the *integrity of software,* to prevent the installation of *malware.*

In the second part of the lab, we will see the use of cryptography *by malware*, specifically, by *Ransomware.* Ransomware uses a cryptosystem to encrypt files in the computer's storage (disk); then, the ransomware requires the user to pay the attacker in order to receive the decryption key. We will explore the use of *public key cryptosystem, shared key cryptosystem* and *obfuscation.*

Question 1 (10 points): In this question we will learn the use of *cryptographic hash functions* to ensure the integrity of software downloads, i.e., to ensure download is of the intended, authentic software, and not of a malware impersonated as the software. A cryptographic hash function *h* receives an input string *m*, e.g., a program, and outputs a short string *h(m)*; people refer to the output as the hash, fingerprint, digest or checksum of the input string *m*.

A main goal of a cryptographic hash functions is *collision resistance*, which implies that given the digest *hash(m)* of some string *m*, it is infeasible to find a *different string m'≠m,* which hashes to the same digest: *h(m')=h(m).* Note that there are many other applications of hash functions, and some of them assume other properties, but this is beyond the scope of this lab.

The collision resistance property is often used to ensure integrity – and, in particular, the integrity of software downloads. Software is often made available via repositories, which may not be fully secure; to ensure the integrity, the publishers often provide the hash of the software. Namely, to protect the integrity of some software download, say encoded as a string *m*, the publisher provides in some secure channel the value of the hash *h(m)*. The user then downloads the software from the (insecure) repository, obtaining the downloaded string *m'*. To confirm its integrity, i.e., confirm that *m'=m*, the user then computes *h(m')* and compares it to *h(m).*

In this question, you will find in your VM a folder called Q1. Within it, you will find a file *checksum.txt*, which contains the result of the SHA-256 hash function applied to the (`legitimate') program file. You will also find there a folder called *InsecureRepository* in which you'll find several program files.

Write a program that identifies which of these files is the legitimate file. Your program should also output a timestamp for the time it began and the time it terminated, and the total run time.

Your program should use the SHA-256 from the PyCryptodome library, which is installed on the VM. (We use this crypto library for all relevant questions in this lab).

Your program should test all files. One reason for that is that once *any* collision in a SHA-256 will be found, it will become easy to find many other collisions; indeed, collisions were found for some other standard hash functions, such as MD5 and SHA-1, and as a result, it is easy to find additional collisions to them, which *is* a problem for many applications. But for more details, you will have to take CSE3400!

Submit: your program, its output, and the beginning few lines of the legitimate program file.

Question 2 (5 points): Repeat Q1, except that this time you should not perform the hashing from your Python program. Instead, use the sha256sum tool (command).

Write a script or a program that will invoke the tool over all files automatically, to identify the correct file(s). As before, your program/script should also output a timestamp for the time it began and the time it terminated, and the total run time; try to minimize the time.

Submit: your program, its output, and the beginning few lines of the legitimate program file.

Question 3 (10 points): The hash mechanism would not protect against an attacker that can provide the user with a *fake hash*, i.e., hash of the *malware*! Also, the hash can only be provided *after* the program is ready; so this mechanism does not allow us to ensure authenticity of *software updates*, unless we can ensure the authenticity of the hash. Fortunately, cryptography also provides a tool to ensure authenticity: *digital signatures.*

Digital signatures use *a pair of keys*; such a pair is generated by a party that wishes to sign files. One key is used by the signer, to *sign* files; therefore, this key must be kept *private.* The other key is made *public.*

As you will find in the PyCryptodome documentation, to perform the verification, you need to specify a hash function; the reason is that it is much more efficient to sign the (short) hash of a message, rather than using a public-key signature algorithm directly (without hash). We use the RSA signature algorithm and the SHA-256 hash function.

In this question, you will find in your VM a folder called Q3. Within it, you will find the public key used by the legitimate software vendor in the file *PublicKey*. In this question, you will only verify signatures, so you only need this public key. You will also find there a folder called *InsecureRepository* in which you'll find several program files, each with the (supposed) signature.

Using PyCryptodome, write an efficient program that will find which of these files is correctly signed. Your program should also output a timestamp for the time it began and the time it terminated, and the total run time. Your program should use the SHA-256 from the PyCryptodome library, which is installed on the VM. (We use this crypto library for all relevant questions in this lab).

Submit: your program, its output, and the beginning few lines of the legitimate program file(s).

Question 4 (10 points): In the rest of this lab, we study the abuse of cryptography by *ransomware.* Ransomware encrypts the user files, and requires the user to pay `ransom', with the promise of sending back the decryption key or program.

Look in the Q4 folder. This folder contains a file which is the encryption of some `plaintext' file by a ransomware program. Luckily, you are also given the ransomware program, R4.py, which is conveniently written in Python; this is not likely to be the case with real ransomware, of course!

You are further lucky since it is relatively easy for you to understand R4.py. furthermore, and most unlikely in practice, you *can*build the corresponding decryption program, D4.py, that will recover the original contents of the plaintext file encrypted by the ransomware. The main reason that allows you to write D4.py is that this ransomware (R4.py) uses a *symmetric (shared key)* *cryptosystem,* specificically, the widely-used AES block cipher, in the CBC mode. In all symmetric (shared key) cryptosystems, the encryption key (used by R4.py) is the same as the decryption key (which must be used by D4.py). So, in this case, you would be able to recover your file(s) – without paying the ransom! Unfortunately, as we will soon see, real ransomware is typically much harder to remove…

Submit: your program (D4.py) and the beginning few lines of the decrypted file(s).

Question 5 (20 points): In this exercise (and the next), we have a similar task to the previous question, but a bit more challenging. Look in the Q5 folder and you will find the R5.py and encrypted content files. Your goal is, again, to write a decryption program, D5.py. As in question 4, you are lucky to have the code of R5.py, and even more lucky in that this ransomware turns out, again, to use a symmetric (shared key) cryptosystem.

However, your task is a bit more challenging, since the new ransomware, R5.py, is *obfuscated*, namely, written intentionally in a way designed to make it harder to understand the program – and to find the key, as required to decrypt the file. Obfuscation is an interesting and challenging subject, and used quite a lot in cybersecurity; in this question, the obfuscation is quite weak, so it should not be too hard to break.

Submit: your program (D5.py) and the beginning few lines of the decrypted file(s).

Question 6 (20 points): This question is similar to the first one, but the obfuscated program (R6, in Q6 folder) may be harder to reverse-engineer, as required in order to find the decryption key and write the decryption program D6.py.

Submit: your program (D6.py) and the beginning few lines of the decrypted file(s).

Question 7 (25 points): In this exercise, your role is to *write* the ransomware R7.py. This would be \`correct' ransomware! This means that your ransomware will use *public key (asymmetric) encryption:* decryption will require a decryption key *d*, which is supposed to be hard to find, even when given the corresponding encryption key *e*. That's how most ransomware works; as a result, even if we find the ransomware program, and even if we can reverse-engineer it and understand exactly how it works, we can only find there the encryption key *e*, which isn't sufficient to find the decryption key *d*.

For your solution, use the PyCryptodome library with the RSA cryptosystem and a key-size of 1024 bits. Note: this key size is not considered sufficiently long for security (considering current processor speeds).

The question has few parts (steps).

1. Generate a keypair of a public key *e* and a private key *d*.
2. Write the ransomware program R7.py, using the public key *e* you generated. This program should search the folder in which it runs, and encrypt all files in this folder, except .py files. Specifically, say the folder contains some file, say *example.txt.* Then R7.py should replace *example.txt* with two files, *example.txt.encrypted* and *example.txt.note.* The example.txt.encrypted will be the encrypted version, and *example.txt.note* will contain a `ransom note'; be creative with the text in the note, but you should include a unique, random idenitifier which should be given to the attacker together with the payment, to allow the attacker to send the decryption key. A different decryption key should be required for every file, so the identifier should be unique too!
3. Write the attacker's decryption program, *AD7.py.* This program will receive the identifier and output the corresponding decryption key. This program will make use of the private decryption key *d.*
4. Write the decryption program *D7.py*. This program will receive the name of an encrypted file and the decryption key sent by the attacker, and recover the original file.

Submit: all programs, and screen shots showing their usage.

RSA Factorization:

---

generatePT.py (RSA):

```
import random              #generate plaintext per student to be encrypted
#function for generating 50 random words from wordlist.txt
def genPT():
   output_file = 'plain33.txt'
   i = 0
   string = ''
   while i < 50:
      string += random.choice(open('wordlist.txt').read().split()).strip() + ' '
      i+=1
   string = string.encode('utf-8')
   file_out = open(output_file, "wb")
      #file_out.write(cipher.iv) # Write the iv to the output file (will be required for decryption)
      file_out.write(string) # Write the varying length ciphertext to the file (this is the encrypted data)
   x = string
   file_out.close()
   return string
```

---

forProf0.csv (RSA):

```
#This is a sample output for the professor after running Fact1.py
***Student
0***7254001986566191139780023037654543527631957051394596914308352177862
2273514714597181597548483658062028307882565140741773436526279798018203
4470565703983319935102236599258177337490253116349230161313076539620118
4130620057844415287037431974822432459318910765785833414211772268445719
4981056133961343604281480066632
8287979035777244113446099796082676002538632361331477345989593230074263
0948241909107993230127525532829567643849626477883891093605597208475337
2063580515876357460664648245049074785486463765390994543533167411407201
8487194801719369221581740903021119580552608160799387979634152286485200
173308964489865918859231939 9
1080824724904564605883922386800219953042276224969307979802816997234709
8916282983408098269373602221921107036585293776729396686326454934358738
```

9763656490066999770155915960490819715336862950754773222938024958874649
0116091305322655011304374844403338815819681501502271266334167851512237
1246944766615892014402329678
9830896333172349992068801424635888649558676572612649262081759004684209
6765987727975383153661781955866668491094310677359136468514915145940355
0399983979403854480829692067459355252516548894543600864373860226495847
3411855523461719955964325937751951589345602622780752214339103141827977
2836743771627060632478563031
2195103938217426351924815239521227461035832273114977170341903354979497
4199209398402459804785313080891054130329173231789962144732450426285908
7109548041571447237494005218421635072328810196150082037237423945085990
0600212105681022366245112793497719649218013237043665730947454167787301
5497160188565319473455542949
4108273325772304615152915911771241071196583963381543803324891338518149
0708129895541223977539228941673006018267354792586285331003191067169655
…

b'trifled valletta dictates epulation eskimo-aleut vernacular effusively tentacle bashbazouk bareheaded restoring blessings long robber appease unbreakable symptom unfurl cystine ascus rb chicane too carnauba perishing fio dethrone flier underslung shoring prankster factory interchangeably fringepod re pea-green roughshod painterly scientific comme bearing(a) rogue signaler ononis azalea achromatism samba inthrall gorgonacea ornamentation '

P is:
1204250384338140004548984301143268685671840026975722997047844538823646
6380634995465315971497026157205201179683591414223193891517212122809376
063724643637163
Q is:
1065841127843951275667061766319395165509539158849390936313145930578739
0276274746562567638489596230045903382160936541847060981556425727924429
540348196726751

---

forStudent0.csv (RSA):

#This is a sample output for the student after running Fact1.py

P is:
120425038433814000454898430114326868567184002697572299704784453882364663806349954653159714970261572052011796835914142231938915172121228093760637246436371 63

Q is:
10658411278439512756670617663193951655095391588493909363131459305787390276274746562567638489596230045903382160936541847060981556425727924429540348196726751

Fact1.py (RSA):

```
from ctypes import sizeof
from gettext import find
import random
import math
from typing import List
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey.RSA import construct
from generatePT import genPT
import binascii
import sys
import os
import csv
from Crypto.Util import number
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.backends import default_backend



#greatest command denom. function
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

```
'''
Euclid's extended algorithm for finding the multiplicative inverse of two numbers
'''
def multiplicative_inverse(e, phi):
    d = 0
    x1 = 0
    x2 = 1
    y1 = 1
    temp_phi = phi

    while e > 0:
        temp1 = temp_phi//e
        temp2 = temp_phi - temp1 * e
        temp_phi = e
        e = temp2

        x = x2 - temp1 * x1
        y = d - temp1 * y1

        x2 = x1
        x1 = x
        d = y1
        y1 = y

    if temp_phi == 1:
        return d + phi




'''
Tests to see if a number is prime.
'''
def is_prime(num):
    if num == 2:
        return True
```

```
    if num < 2 or num % 2 == 0:
        return False
    for n in range(3, int(num**0.5)+2, 2):
        if num % n == 0:
            return False
    return True


#generate public and private key pair
def generate_key_pair(p, q):
    # n = pq
    n = p * q

    # Phi is the totient of n
    phi = (p-1) * (q-1)

    # Choose an integer e such that e and phi(n) are coprime
    e = random.randrange(1, phi)

    # Use Euclid's Algorithm to verify that e and phi(n) are coprime
    g = gcd(e, phi)
    while g != 1:
        e = random.randrange(1, phi)
        g = gcd(e, phi)

    # Use Extended Euclid's Algorithm to generate the private key
    d = multiplicative_inverse(e, phi)

    # Return public and private key_pair
    # Public key is (e, n) and private key is (d, n)
    return ((e, n), (d, n))



def encrypt(pk, plaintext):
    # Unpack the key into its components
    key, n = pk
    # Convert each letter in the plaintext to numbers based on the character using a^b
mod m
    cipher = [pow(ord(char), key, n) for char in plaintext]
```

```
    # Return the array of bytes
    return cipher

#allow the professor to choose number of P.T. and C.T. generated
x = input("How many students, professor?\n")

for i in range(int(x)):
    if __name__ == '__main__':
        while True:
            keySize = 512
            p = number.getPrime(keySize)
            q = number.getPrime(keySize)
            nFact = p*q
            phi = (p-1)*(q-1)
            genPT() #generate your random plaintext
            variable = genPT()
            plaintext1 = str(variable)
            public, private = generate_key_pair(p, q)
            encrypt_msg = encrypt(public, plaintext1)
            #Professor Cipher, PT
           professor_file = open('/Users/jacoblenes/Desktop/Fall
                        2021/HonorsThesis/FactFinal/ProfFact/forProf' + str(i)+'.csv', 'w')
            professor_file.write("***Student " + str(i) + '***')
            for word in encrypt_msg:
                professor_file.write(str(word) + ' ')
            professor_file.write('\n\n')
            for word in plaintext1:
                professor_file.write(str(word) + ' ')
            professor_file.write("\n\nP is: " + str(p) +"\nQ is: " + str(q))
            professor_file.close()
            #Student P, Q, N, and Phi Files
            student_file = open('/Users/jacoblenes/Desktop/Fall
                  2021/HonorsThesis/FactFinal/StudentFact/forStudent' + str(i)+'.csv', 'w')
            student_file.write("P is: " + str(p) +"\nQ is: " + str(q) + '\n\n\n')
            student_file.close()
            break
```

EL Gamal:

---

generatePT.py (El Gamal)

```
import random
#generate plaintext per student to be encrypted
def genPT():
    output_file = 'plain33.txt'
    i = 0
    string = ''
    while i < 50:
        string += random.choice(open('wordlist.txt').read().split()).strip() + ' '
        i+=1
    string = string.encode('utf-8')
    file_out = open(output_file, "wb")
     #file_out.write(cipher.iv) # Write the iv to the output file (will be required for decryption)
        file_out.write(string) # Write the varying length ciphertext to the file (this is the encrypted data)
    x = string
    file_out.close()
    return string
```

---

forProf0.csv (El Gamal)

#This is a sample output for the professor after running Gamal2.py

b'selar bawarchikhana centaury harborless inquisitiveness impetrate pusillanimously hydromantes veda complicate leaderless grant nicene citified superficiality mission tinted drip amalgamative schnitzel simazine xerophagy gerres proruption delirant priority hitless prodigy retral revealed verst geared embiotocidae current rubeola scald minstrelsy a-horizon retreat dispassionate neritid baryon bromeliaceae obtain equivalent ancora widowhood pylorus tirailleur lucilia '

****

30608852505579017266938365196564450500095045800874741835998275739195567357165
97366274739600455214410886212383682002938057427492008132199398189063 43

4956901
9906388718527995108613208376464861209709970688971153991687352106830228
367777
1004256508485063462153216655582417291542945709544938582122749989729175
39424462
4458783942344516981626933254833361764136014681651633692459886490047785
4671601
7991451973353636898901241620828753237193801292263635790458600638639760
1907448
6829484910044278095349870772698855646671432494626950845324155850908512
865378
4117257070003089180073465679780033573669127320729179440382135609434015
6977166
2083172482950079794521678677803800455510423222810706255891325064403707
6200820
1632088688547078094967092679553330751362327622205015072548618862828689
9338350
7230754305859256524500841171148552686958088585526648460263792765671043
5159040
5213957040123420328954568674095435283700712434075711697319844668718308
7999559
9302046081460354119477168734698565697379915758594964206518483184400577
7016092
4643752220929744706542997450303962995027292838759806198042254574415680
8270209
4465587563093280165137216605811201317490161195637126535977653006638098
5549943
6294165523789375087728929061008467613644107884682713222617564792494378
051407
7217589173163621940672865074908852715691144476403973232277233233994916
5605088
8319179289332403638614655944841699862548038505876717554461051799524739
327505
7870852649156395801480957418916255368172780911093094791177366215564882
9042049
3918928860597943836519233011306751784321299810374350464863382927270697
3816321
1427761027514217535087370338186178003677147532202852361974883669817596
3151340
5695803373559208155062161392850940590396850855838721090454578841084011

9648214
97692396289861504914348025488939615443391100808197753971280720900096968009502
60013932605238795533480570002028400299481556284328490722014107473221554656949
2762401087139102354021104737081312114701964477241005667782948817450552381536
88570779769290182369837297399494870885437009073681216432848587460079386930107
38737432160098701690403219013129664890650741070545152637263777466393833117641
74010924419523258239079927645537435885685199531531176021502214946347347237087
14561461364271123811459601639241269228028409285919776811691838422654682063487
35615666982248405699357313056611484686671931761715490644280026365455704222195
42753902481987088667373588551281362041867842308290804259275122437865769216689
78495615684797563778452142530618881158068927385098733949473986823960046385783
9326146605366963411107902869279128769095131566594101047271517245618850381185
3862326897851127965593244517756034363850231838987214659849424420782315009252
1770023315230356467214317568301001374604474052477606761002291690156296861219
89173953214258233403340043638366300318346137364040856643814864724306764236101
87328400659333521063003933634932135818408112044007367575428216348586873521424
3214220196588199396774317911463094696261403091127804101392238634295148349275
34889595047669877059472156386910728747310229516316097936863004624545665380873
90840858593655769834141247178675585854846431808146836417684301992752649977090
72659572287111091211799330592158842973332309530152469890710823014423772885152
930421425756991506518701367163568569121538035564931405633568828456128

5157553
4704822059063145870822203567730638460849198063226573192846609239394225
0207845
6835799475312449990073256341252762912129915805147171841043298009318564
5598876
7021522548678307223839863542447293875333227989397035636010833965622413
3137536
2345181443944185033447917344398150786531756471913897058459349462897951
920920
9962060182767437836008416718025083639644497974894560662272609562248983
2239276
7444982359510868910008288704217901532045264026179552433758866844840508
4579573
7671401722197394109680993020031016098864600581630931595270546746113409
6673356
8011343641580963911637493705897398887974457257975509129518512370708537
9715998
3986594968664416409635381154656531279670001412172636290769188643679078
5533038
6853212310624347835946295568029874712295165620561148851826062258089720
6193633
4830354966912853496887559579522017233910468350730187245504847349798773340267
7329569594125846436813446160357104296964531857611805713186742453789684
7075924
5031850146360808293657393454599789395581099673580900469676805424904291
9055983
6236218583877574418670391929840288959682790504802517213667882934258867
0686416
8494319800981231407776105570307933840322567240341910751211511972771365
1601828
6528683182364979274152108247529513271147817006044369573702999630939827
4287737
9535403898268430508369345311579978134086540957817644482114097392858594
5752099
8919740885990028629497013472726940008131368712869740779351803639464898
6858969

forStudent0.csv (El Gamal)

#This is a sample output for the student after running Gamal2.py

Pub Key:
23696504997087768794535541531566801903251019648369403666887908430150335
15856

306088525055790172669383651965644505000950458008747418359982757391955673
57165
973662747396004552144108862123836820029380574274920081321993981890634349
56901
990638871852799510861320837646486120970997068897115399168735210683022836
7777
100425650848506346215321665558241729154294570954493858212274998972917539
424462
445878394234451698162693325483336176413601468165163369245988649004778546
71601
799145197335363689890124162082875323719380129226363579045860063863976019
07448
682948491004427809534987077269885564667143249462695084532415585090851286
5378
411725707000308918007346567978003357366912732072917944038213560943401569
77166
208317248295007979452167867780380045551042322281070625589132506440370762
00820
163208868854707809496709267955333075136232762220501507254861886282868993
38350
723075430585925652450084117114855268695808858552664846026379276567104351
59040
521395704012342032895456867409543528370071243407571169731984466871830879
99559
930204608146035411947716873469856569737991575859496420651848318440057770
16092
464375222092974470654299745030396299502729283875980619804225457441568082
70209
446558756309328016513721660581120131749016119563712653597765300663809855
49943
629416552378937508772892906100846761364410788468271322261756479249437805
1407
7217589173163621940672865074908852715691144476403973232277233233994916

5605088
83191792893324036386146559448416998625480385058767175544610517995247393
327505
78708526491563958014809574189162553681727809110930947911773662155648829
9042049
39189288605979438365192330113067517843212998103743504648633829272706973
3816321
14277610275142175350873703381861780036771475322028523619748836698175963
3151340
56958033735592081550621613928509405903968508558387210904545788410840119
9648214
97692396289861504914348025488939615443391100808197753971280720900096968
8009502
60013932605238795533480570002028400299481556284328490722014107473221554
4656949
27624010871391023540211047370813121147019644772410056677829488174505523
381536
88570779769290182369837297399494870885437009073681216432848587460079386
6930107
38737432160098701690403219013129664890650741070545152637263777466393833
3117641
74010924419523258239079927645537435885685199531531176021502214946347347
7237087
14561461364271123811459601639241269228028409285919776811691838422654682
2063487
35615666982248405699357313056611484686671931761715490644280026365455704
4222195
42753902481987088667373588551281362041867842308290804259275122437865769
9216689
78495615684797563778452142530618881158068927385098733949473986823960046
6385783
93261466053669634111079028692791287690951315665941010472715172456188503
381185
38623268978511279655932445177560343638502318389872146598494244207823150
009252
17700233152303564672143175683010013746044740524776067610022916901562968
61219
89173953214258233403340043638366300318346137364040856643814864724306764
4236101
873284006593335210630039336349321358184081120440073675754282163485868

3521424
3214220196588199396774317911463094696261403091127804101392238634295148
349275
3488959504766987705947215638691072874731022951631609793686300462454566
5380873
9084085859365576983414124717867558585484643180814683641768430199275264
9977090
7265957228711109121179933059215884297333230953015246989071082301442377
2885152
9304214257569915065187013671635685691215380355649314056335688284561282
5157553
4704822059063145870822203567730638460849198063226573192846609239394225
0207845
6835799475312449990073256341252762912129915805147171841043298009318564
5598876
7021522548678307223839863542447293875333227989397035636010833965622413
3137536
2345181443944185033447917344398150786531756471913897058459349462897951
920920
9962060182767437836008416718025083639644497974894560662272609562248983
2239276
7444982359510868910008288704217901532045264026179552433758866844840508
4579573
7671401722197394109680993020031016098864600581630931595270546746113409
6673356
8011343641580963911637493705897398887974457257975509129518512370708537
9715998
3986594968664416409635381154656531279670001412172636290769188643679078
5533038
6853212310624347835946295568029874712295165620561148851826062258089720
6193633
4830354966912853496887755957952201723391046835073018724550484734979877
3340267
7329569594125846436813446160357104296964531857611805713186742453789684
7075924
5031850146360808293657393454599789395581099673580900469676805424904291
9055983
6236218583877574418670391929840288959682790504802517213667882934258867
0686416
8494319800981231407776105570307933840322567240341910751211511972771365

```
1601828
6528683182364979274152108247529513271147817006044369573702999630939827
4287737
9535403898268430508369345311579978134086540957817644482114097392858594
5752099
8919740885990028629497013472726940008131368712869740779351803639464898
6858969
```

Gamal2.py (El Gamal)

```
import random
import math
import sys
from generatePT import genPT

genPT() #generate your random plaintext
variable = genPT()
plaintext1 = str(variable)

class PrivateKey(object):
	def __init__(self, p=None, g=None, x=None, iNumBits=0):
		self.p = p
		self.g = g
		self.x = x
		self.iNumBits = iNumBits

class PublicKey(object):
	def __init__(self, p=None, g=None, h=None, iNumBits=0):
		self.p = p
		self.g = g
		self.h = h
		self.iNumBits = iNumBits

# computes the greatest common denominator of a and b.  assumes a > b
def gcd( a, b ):
		while b != 0:
			c = a % b
```

```
                a = b
                b = c
        #a is returned if b == 0
        return a

#computes base^exp mod modulus
def modexp( base, exp, modulus ):
        return pow(base, exp, modulus)

#use func from rsa fact
#solovay-strassen primality test.  tests if num is prime
def SS( num, iConfidence ):
        #ensure confidence of t
        for i in range(iConfidence):
                #choose random a between 1 and n-2
                a = random.randint( 1, num-1 )

                #if a is not relatively prime to n, n is composite
                if gcd( a, num ) > 1:
                        return False

                #declares n prime if jacobi(a, n) is congruent to a^((n-1)/2)
mod n
                if not jacobi( a, num ) % num == modexp ( a, (num-1)//2,
num ):
                        return False

        #if there have been t iterations without failure, num is believed to be prime
        return True

#computes the jacobi symbol of a, n
def jacobi( a, n ):
        if a == 0:
                if n == 1:
                        return 1
                else:
                        return 0
        #property 1 of the jacobi symbol
        elif a == -1:
                if n % 2 == 0:
```

```
                return 1
        else:
                return -1
    #if a == 1, jacobi symbol is equal to 1
    elif a == 1:
        return 1
    #property 4 of the jacobi symbol
    elif a == 2:
        if n % 8 == 1 or n % 8 == 7:
            return 1
        elif n % 8 == 3 or n % 8 == 5:
            return -1
    #property of the jacobi symbol:
    #if a = b mod n, jacobi(a, n) = jacobi( b, n )
    elif a >= n:
        return jacobi( a%n, n)
    elif a%2 == 0:
        return jacobi(2, n)*jacobi(a//2, n)
    #law of quadratic reciprocity
    #if a is odd and a is coprime to n
    else:
        if a % 4 == 3 and n%4 == 3:
            return -1 * jacobi( n, a)
        else:
            return jacobi(n, a )


#finds a primitive root for prime p
#this function was implemented from the algorithm described here:
#http://modular.math.washington.edu/edu/2007/spring/ent/ent-html/node31.html
def find_primitive_root( p ):
    if p == 2:
        return 1
    #the prime divisors of p-1 are 2 and (p-1)/2 because
    #p = 2x + 1 where x is a prime
    p1 = 2
    p2 = (p-1) // p1

    #test random g's until one is found that is a primitive root mod p
    while( 1 ):
```

```
            g = random.randint( 2, p-1 )
            #g is a primitive root if for all prime factors of p-1, p[i]
            #g^((p-1)/p[i]) (mod p) is not congruent to 1
            if not (modexp( g, (p-1)//p1, p ) == 1):
                    if not modexp( g, (p-1)//p2, p ) == 1:
                            return g

#find n bit prime
def find_prime(iNumBits, iConfidence):
        #keep testing until one is found
        while(1):
            #generate potential prime randomly
            p = random.randint( 2**(iNumBits-2), 2**(iNumBits-1) )
            #make sure it is odd
            while( p % 2 == 0 ):
                    p                                    =
random.randint(2**(iNumBits-2),2**(iNumBits-1))

            #keep doing this if the solovay-strassen test fails
            while( not SS(p, iConfidence) ):
                    p      =      random.randint(      2**(iNumBits-2),
2**(iNumBits-1) )
                    while( p % 2 == 0 ):
                            p                            =
random.randint(2**(iNumBits-2), 2**(iNumBits-1))

            #if p is prime compute p = 2*p + 1
            #if p is prime, we have succeeded; else, start over
            p = p * 2 + 1
            if SS(p, iConfidence):
                    return p

#encodes bytes to integers mod p.  reads bytes from file
def encode(sPlaintext, iNumBits):
        byte_array = bytearray(sPlaintext, 'utf-16')

        #z is the array of integers mod p
        z = []

        #each encoded integer will be a linear combination of k message bytes
```

```
        #k must be the number of bits in the prime divided by 8 because each
        #message byte is 8 bits long
        k = iNumBits//8

        #j marks the jth encoded integer
        #j will start at 0 but make it -k because j will be incremented during first
iteration
        j = -1 * k
        #num is the summation of the message bytes
        num = 0
        #i iterates through byte array
        for i in range( len(byte_array) ):
                #if i is divisible by k, start a new encoded integer
                if i % k == 0:
                        j += k
                        num = 0
                        z.append(0)
                #add the byte multiplied by 2 raised to a multiple of 8
                z[j//k] += byte_array[i]*(2**(8*(i%k)))

        #example
                #if n = 24, k = n / 8 = 3
                #z[0] = (summation from i = 0 to i = k)m[i]*(2^(8*i))
                #where m[i] is the ith message byte

        #return array of encoded integers
        return z
'''#decodes integers to the original message bytes
def decode(aiPlaintext, iNumBits):
        #bytes array will hold the decoded original message bytes
        bytes_array = []

        #same deal as in the encode function.
        #each encoded integer is a linear combination of k message bytes
        #k must be the number of bits in the prime divided by 8 because each
        #message byte is 8 bits long
        k = iNumBits//8

        #num is an integer in list aiPlaintext
        for num in aiPlaintext:
```

```
            #get the k message bytes from the integer, i counts from 0 to k-1
                for i in range(k):
                        #temporary integer
                        temp = num
                        #j goes from i+1 to k-1
                        for j in range(i+1, k):
                                #get remainder from dividing
integer by 2^(8*j)
                                temp = temp % (2**(8*j))
                        #message byte representing a letter is equal to
temp divided by 2^(8*i)
                        letter = temp // (2**(8*i))
                        #add the message byte letter to the byte array
                        bytes_array.append(letter)
                        #subtract the letter multiplied by the power of
two from num so
                        #so the next message byte can be found
                        num = num - (letter*(2**(8*i)))

        #example
        #if "You" were encoded.
        #Letter      #ASCII
        #Y          89
        #o          111
        #u          117
        #if the encoded integer is 7696217 and k = 3
        #m[0] = 7696217 % 256 % 65536 / (2^(8*0)) = 89 = 'Y'
        #7696217 - (89 * (2^(8*0))) = 7696128
        #m[1] = 7696128 % 65536 / (2^(8*1)) = 111 = 'o'
        #7696128 - (111 * (2^(8*1))) = 7667712
        #m[2] = 7667712 / (2^(8*2)) = 117 = 'u'

        decodedText = bytearray(b for b in bytes_array).decode('utf-16')

        return decodedText'''

#generates public key K1 (p, g, h) and private key K2 (p, g, x)
def generate_keys(iNumBits=256, iConfidence=32):
        #p is the prime
        #g is the primitve root
```

```
		#x is random in (0, p-1) inclusive
		#h = g ^ x mod p
		p = find_prime(iNumBits, iConfidence)
		g = find_primitive_root(p)
		g = modexp( g, 2, p )
		x = random.randint( 1, (p - 1) // 2 )
		h = modexp( g, x, p )

		publicKey = PublicKey(p, g, h, iNumBits)
		privateKey = PrivateKey(p, g, x, iNumBits)
		#print ({'privateKey': privateKey, 'publicKey': publicKey})

		return {'privateKey': privateKey, 'publicKey': publicKey}


#encrypts a string sPlaintext using the public key k
def encrypt(key, sPlaintext):
		z = encode(sPlaintext, key.iNumBits)

	#cipher_pairs list will hold pairs (c, d) corresponding to each integer in z
		cipher_pairs = []
		#i is an integer in z
		for i in z:
				#pick random y from (0, p-1) inclusive
				y = random.randint( 0, key.p )
				#c = g^y mod p
				c = modexp( key.g, y, key.p )
				#d = ih^y mod p
				d = (i*modexp( key.h, y, key.p)) % key.p
				#add the pair to the cipher pairs list
				cipher_pairs.append( [c, d] )

		encryptedStr = ""
		for pair in cipher_pairs:
				encryptedStr += str(pair[0]) + ' ' + str(pair[1]) + ' '

		return encryptedStr

#performs decryption on the cipher pairs found in Cipher using
#prive key K2 and writes the decrypted values to file Plaintext
```

```
'''#performs decryption on the cipher pairs found in Cipher using
#prive key K2 and writes the decrypted values to file Plaintext
def decrypt(key, cipher):
		#decrpyts each pair and adds the decrypted integer to list of plaintext integers
		plaintext = []

		cipherArray = cipher.split()
		if (not len(cipherArray) % 2 == 0):
				return "Malformed Cipher Text"
		for i in range(0, len(cipherArray), 2):
				#c = first number in pair
				c = int(cipherArray[i])
				#d = second number in pair
				d = int(cipherArray[i+1])

				#s = c^x mod p
				s = modexp( c, key.x, key.p )
				#plaintext integer = ds^-1 mod p
				plain = (d*modexp( s, key.p-2, key.p)) % key.p
				#add plain to list of plaintext integers
				plaintext.append( plain )

		decryptedText = decode(plaintext, key.iNumBits)

	#remove trailing null bytes
		decryptedText = "".join([ch for ch in decryptedText if ch != fix])

		return decryptedText
'''

#ask professor how many students to generate PT/CT for
x = input("How many students, Professor?\n")

for i in range(int(x)):
	if __name__ == '__main__':
		while True:
			genPT() #generate your random plaintext
			variable = genPT()
			plaintext1 = str(variable)
```

```
assert (sys.version_info >= (3,4))
keys = generate_keys()
priv = keys['privateKey']
pub = keys['publicKey']
#print(sys.getsizeof(priv))
message = plaintext1
cipher = encrypt(pub, message)
output_file = open('cipher1.txt', 'w')
for word in cipher:
	output_file.write(str(word) + ' ')
output_file.close()
prof_file2 = open('/Users/jacoblenes/Desktop/Fall 2021/HonorsThesis/GamalFinal/ProfGamal/forProfessor' + str(i)+'.csv', 'w')
prof_file2.write(message + '\n\n****\n\n' + cipher)
prof_file2.close()
student_file = open('/Users/jacoblenes/Desktop/Fall 2021/HonorsThesis/GamalFinal/StudentGamal/forStudent' + str(i)+'.csv', 'w')
student_file.write('Pub Key: ' + str(pub.h) + '\n\n' + cipher)
break
```