ECE 676 - D

CRYPTOGRAPHY LAB REPORT

November 1, 2019

Craig Contreras
University of Miami
Department of Electrical and Computer Engineering

Contents

*Disclaimer: The codes provided in the figures may not be readable. This is due to the formatting when adding the picture to the lab report. Thus, some of the longer codes are provided as hyperlinks. All you have to do is click the blue 'click here'. In addition, a Github link to the repository containing all the files is included in the Appendix to view the codes and outputs that were written.

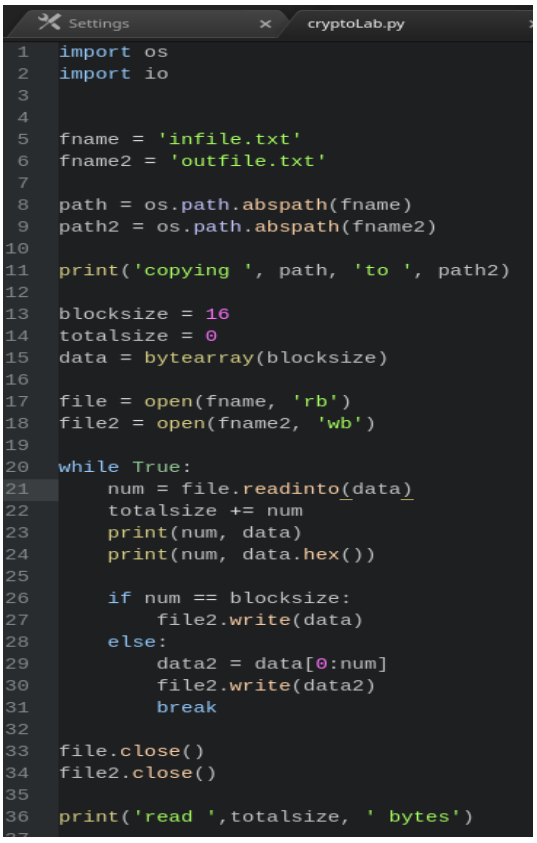
CRYPTOLAB1

0.0.1 Task 1: Read/Write Files in Python

0.0.1.1 Task 1.1: Using buffered Byte Streams

Program: CryptoLab.py

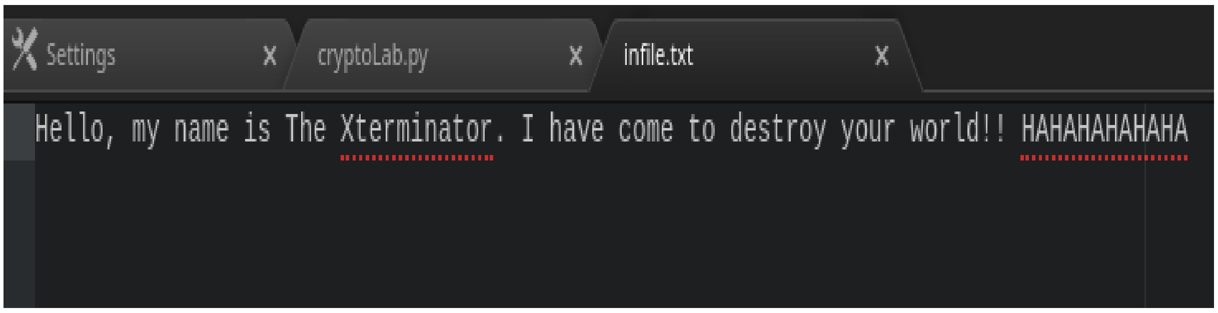
For this task, I had to write a python code that reads data from one file and writes to another file. So, I had to make a file called "infile.txt" and copy its contents into another file called "outfile.txt". A screenshot of the python code is shown below:



```python
import os
import io


fname = 'infile.txt'
fname2 = 'outfile.txt'

path = os.path.abspath(fname)
path2 = os.path.abspath(fname2)

print('copying ', path, 'to ', path2)

blocksize = 16
totalsize = 0
data = bytearray(blocksize)

file = open(fname, 'rb')
file2 = open(fname2, 'wb')

while True:
    num = file.readinto(data)
    totalsize += num
    print(num, data)
    print(num, data.hex())

    if num == blocksize:
        file2.write(data)
    else:
        data2 = data[0:num]
        file2.write(data2)
        break

file.close()
file2.close()

print('read ',totalsize, ' bytes')
```

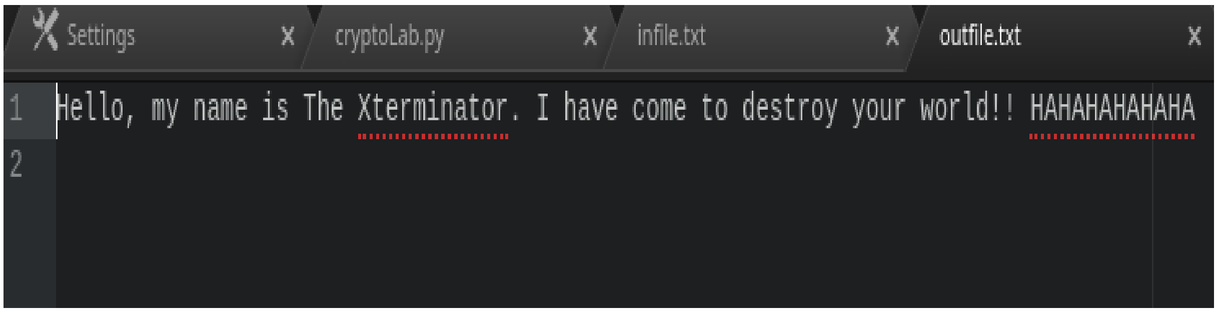
Figure 1: Code for CryptoLab.py

After running the file, the contents of the infile document were copied to the outfile document. A screenshot of both documents and their contents are shown below:



```
Hello, my name is The Xterminator. I have come to destroy your world!! HAHAHAHAHAHA
```

Figure 2: Output of infile.txt



```
Hello, my name is The Xterminator. I have come to destroy your world!! HAHAHAHAHAHA

```

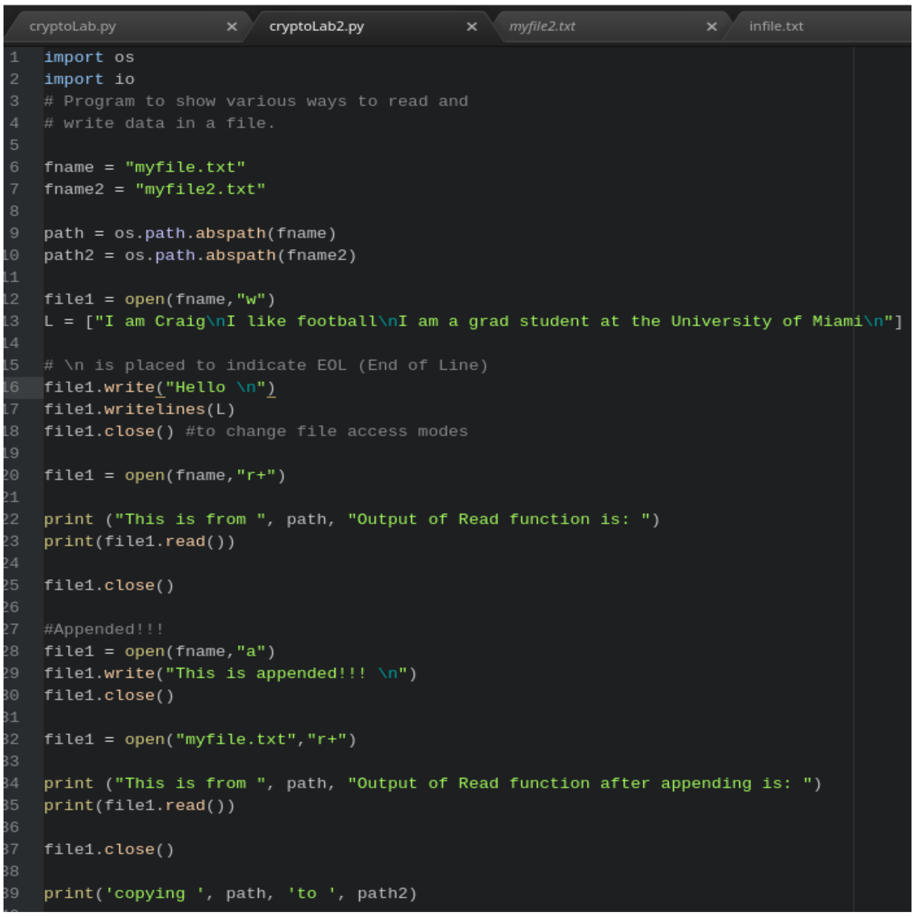
Figure 3: Output of outfile.txt

Thus, the code works perfectly because it does what it is supposed to.

0.0.1.2 Task 1.2: Using Character Streams

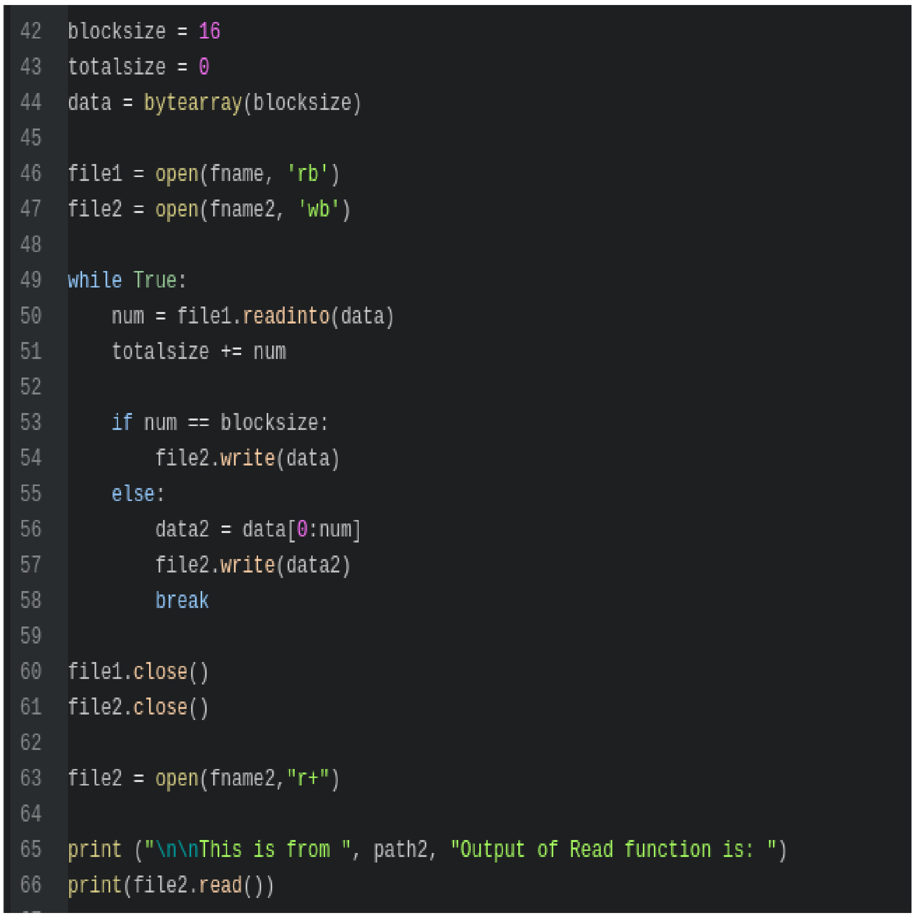
Program: CryptoLab2.py

For this task, the task was to create a new version of the program that reads text data and copies it to another file. A screenshot of the new program is shown below:



```python
import os
import io
# Program to show various ways to read and
# write data in a file.

fname = "myfile.txt"
fname2 = "myfile2.txt"

path = os.path.abspath(fname)
path2 = os.path.abspath(fname2)

file1 = open(fname,"w")
L = ["I am Craig\nI like football\nI am a grad student at the University of Miami\n"]

# \n is placed to indicate EOL (End of Line)
file1.write("Hello \n")
file1.writelines(L)
file1.close() #to change file access modes

file1 = open(fname,"r+")

print ("This is from ", path, "Output of Read function is: ")
print(file1.read())

file1.close()

#Appended!!!
file1 = open(fname,"a")
file1.write("This is appended!!! \n")
file1.close()

file1 = open("myfile.txt","r+")

print ("This is from ", path, "Output of Read function after appending is: ")
print(file1.read())

file1.close()

print('copying ', path, 'to ', path2)
```

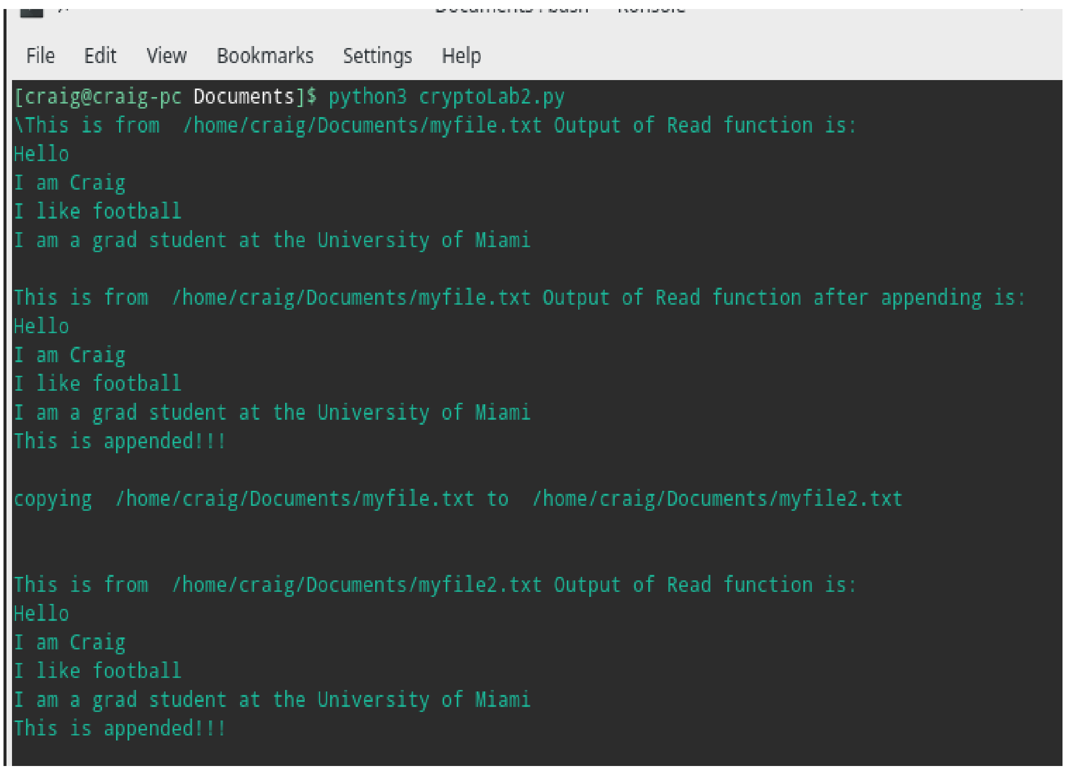
Figure 4: Code for cryptoLab2.py (part 1)



```python
blocksize = 16
totalsize = 0
data = bytearray(blocksize)

file1 = open(fname, 'rb')
file2 = open(fname2, 'wb')

while True:
    num = file1.readinto(data)
    totalsize += num

    if num == blocksize:
        file2.write(data)
    else:
        data2 = data[0:num]
        file2.write(data2)
        break

file1.close()
file2.close()

file2 = open(fname2,"r+")

print ("\n\nThis is from ", path2, "Output of Read function is: ")
print(file2.read())
```

Figure 5: Code for cryptoLab2.py (part 1)

Note that the new program writes the content of variable "L" to "myfile.txt". After it does that, it appends another line of text to that same file. Lastly, the contents of the "myfile.txt" file is copied onto "myfile2.txt". Thus, the program can write data to a new file, read that same data and append even more data, and copy that data onto a new file. A example of the running program is shown below:



```
[craig@craig-pc Documents]$ python3 cryptoLab2.py
\This is from  /home/craig/Documents/myfile.txt Output of Read function is:
Hello
I am Craig
I like football
I am a grad student at the University of Miami

This is from  /home/craig/Documents/myfile.txt Output of Read function after appending is:
Hello
I am Craig
I like football
I am a grad student at the University of Miami
This is appended!!!

copying  /home/craig/Documents/myfile.txt to  /home/craig/Documents/myfile2.txt


This is from  /home/craig/Documents/myfile2.txt Output of Read function is:
Hello
I am Craig
I like football
I am a grad student at the University of Miami
This is appended!!!
```

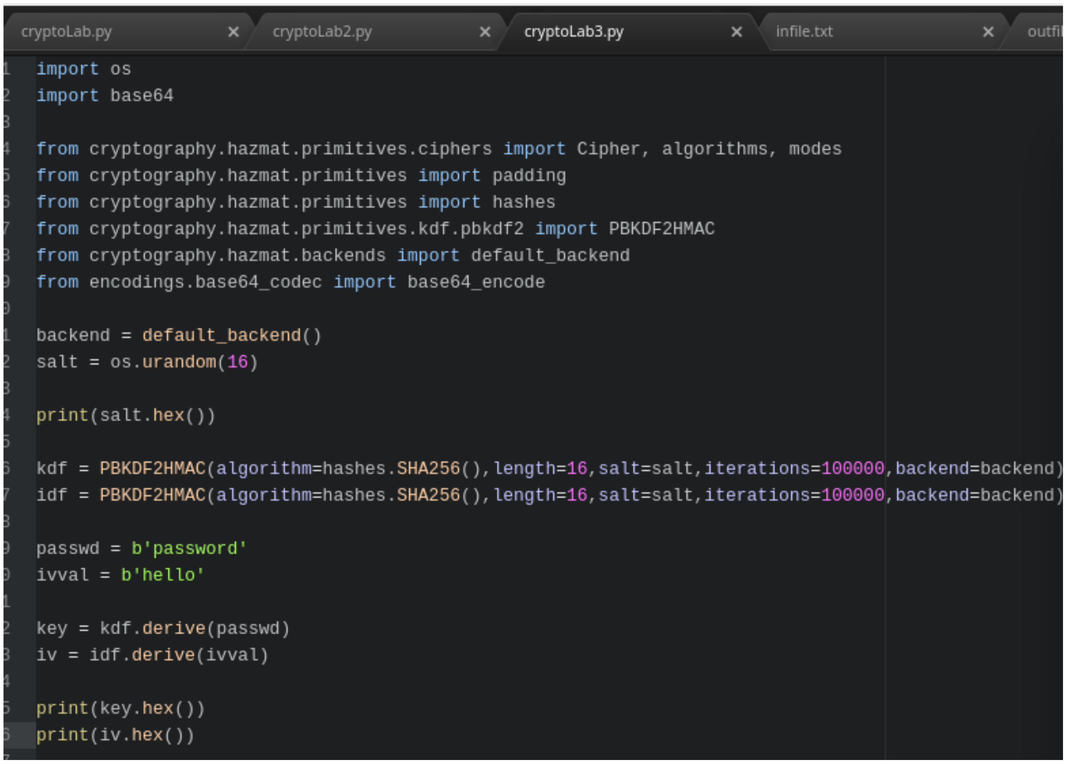
Figure 6: Output of cryptoLab2.py

0.0.2 Task 2: Create and use a symmetric key block cipher in Python using the cryptography package

Program: CryptoLab3.py

0.0.2.1 Task 2.1: Use a KDF to generate the keys we need

For this task, I had to create a random number generator and seed values, which are passwords entered by the user to create a pseudo-random sequence generator that will then be used to create a secret key for a cipher to be used for encryption and decryption. To make this, I had to make a python file with the following code:

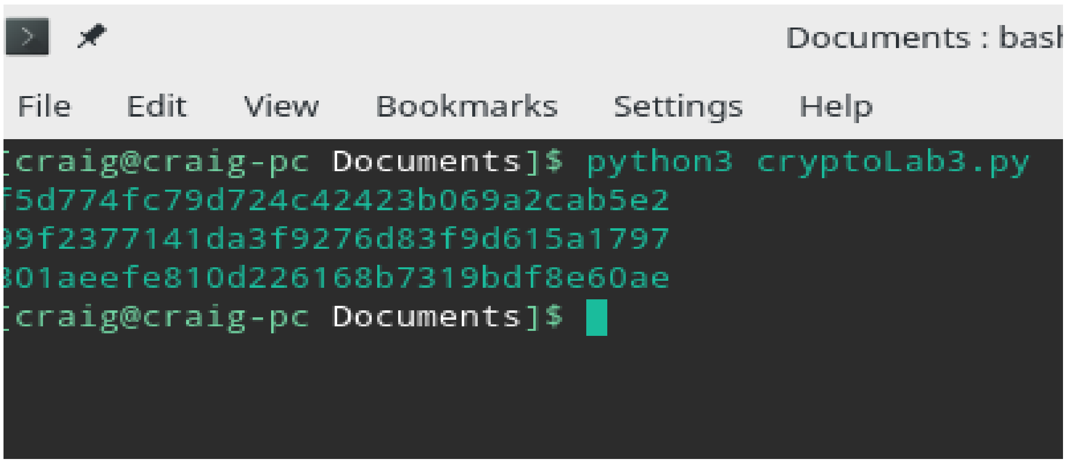


```
cryptoLab.py   cryptoLab2.py   cryptoLab3.py   infile.txt   outfi
```

```python
import os
import base64

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.backends import default_backend
from encodings.base64_codec import base64_encode

backend = default_backend()
salt = os.urandom(16)

print(salt.hex())

kdf = PBKDF2HMAC(algorithm=hashes.SHA256(),length=16,salt=salt,iterations=100000,backend=backend)
idf = PBKDF2HMAC(algorithm=hashes.SHA256(),length=16,salt=salt,iterations=100000,backend=backend)

passwd = b'password'
ivval = b'hello'

key = kdf.derive(passwd)
iv = idf.derive(ivval)

print(key.hex())
print(iv.hex())
```

Figure 7: Code for CryptoLab3.py - section task 2.1

After running the program, the result was the following:



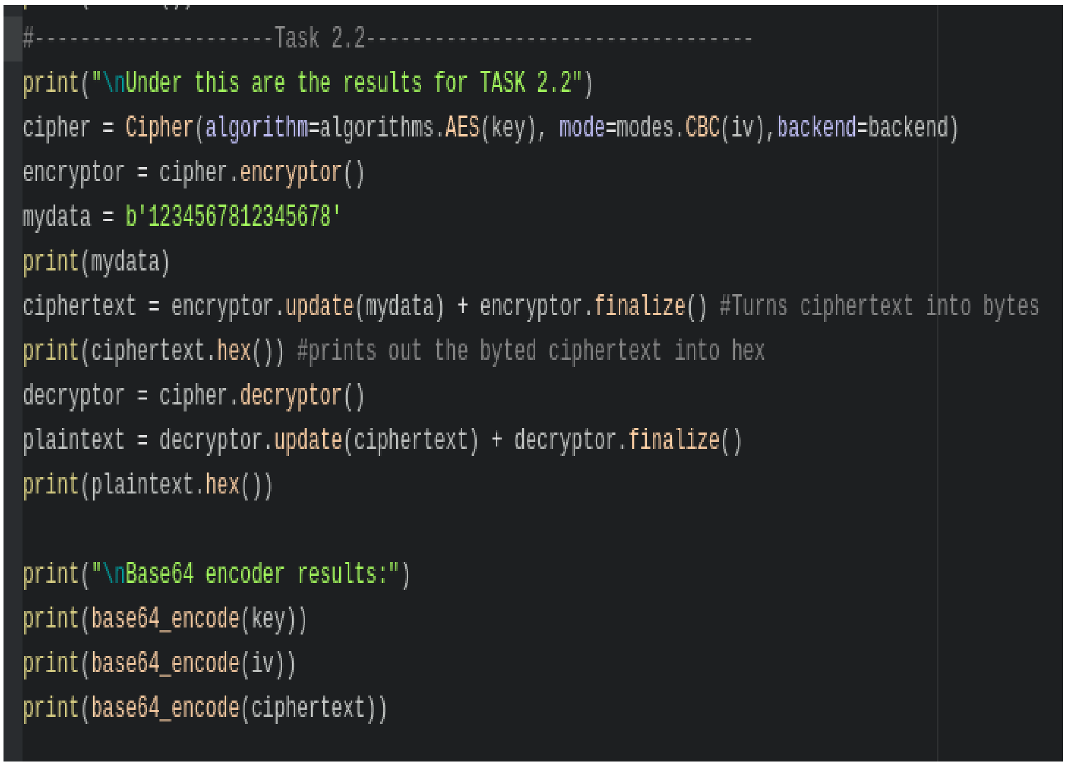
```
[craig@craig-pc Documents]$ python3 cryptoLab3.py
f5d774fc79d724c42423b069a2cab5e2
99f2377141da3f9276d83f9d615a1797
301aeefe810d226168b7319bdf8e60ae
[craig@craig-pc Documents]$ 
```

Figure 8: Output for CryptoLab3.py - section task 2.1

In the terminal result, the first line printed is the salt, followed by the key and the iv. All of them are printed in hex format. The key and the salt "constructions" are the same. Both use the same hashing algorithm, are the same length, and have the same number of iterations and backend. The only difference is the value of the password. The password for the key is 'password' and the password for the iv is 'hello'. Because of this, the two of them have different values. In fact, they will always have the same value unless given the same password. In practice, this is ideal because one should not have the same value for the iv adn the password. It is insecure.

0.0.2.2 Task 2.2: The Cipher

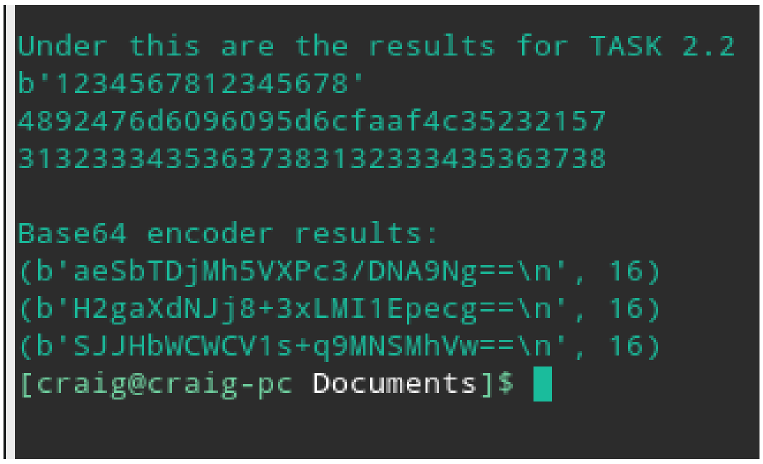
Because I made some material for the key, I was able to create the cipher and use it to do some encryption. Following the steps in the lab manual, I came up with the following block of code:



```
#--------------------Task 2.2--------------------------------
print("\nUnder this are the results for TASK 2.2")
cipher = Cipher(algorithm=algorithms.AES(key), mode=modes.CBC(iv),backend=backend)
encryptor = cipher.encryptor()
mydata = b'1234567812345678'
print(mydata)
ciphertext = encryptor.update(mydata) + encryptor.finalize() #Turns ciphertext into bytes
print(ciphertext.hex()) #prints out the byted ciphertext into hex
decryptor = cipher.decryptor()
plaintext = decryptor.update(ciphertext) + decryptor.finalize()
print(plaintext.hex())

print("\nBase64 encoder results:")
print(base64_encode(key))
print(base64_encode(iv))
print(base64_encode(ciphertext))
```

Figure 9: Code for CryptoLab3.py - section 2.2

Note that the block of code shown was added on to the entire program that was shown previously. This was not its own file. Thus, after running the program, the result was the following:



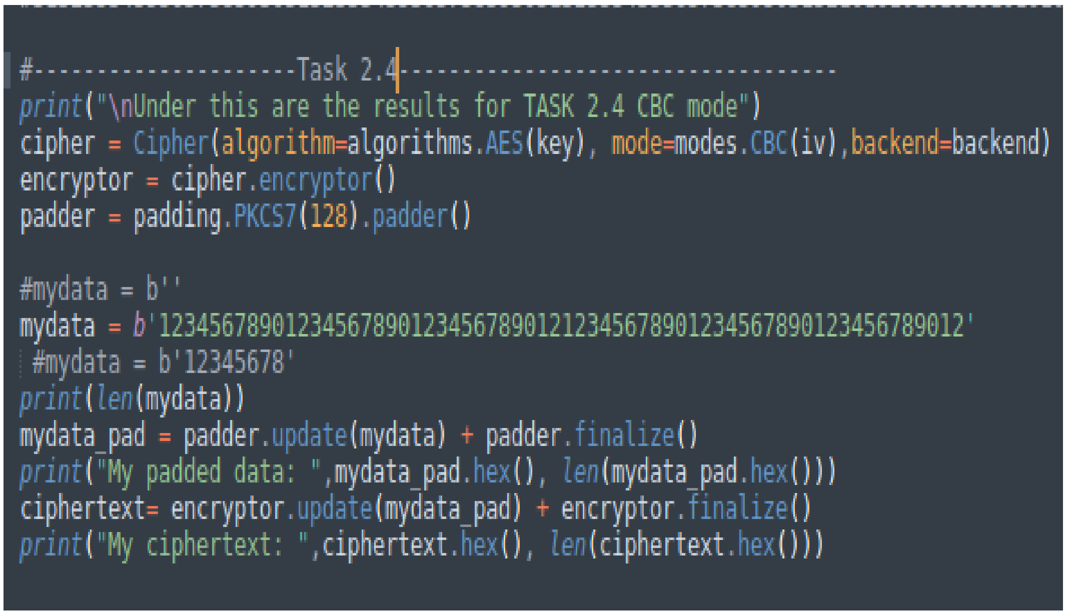
```
Under this are the results for TASK 2.2
b'1234567812345678'
4892476d6096095d6cfaaf4c35232157
31323334353637383132333435363738

Base64 encoder results:
(b'aeSbTDjMh5VXPc3/DNA9Ng==\n', 16)
(b'H2gaXdNJj8+3xLMI1Epecg==\n', 16)
(b'SJJHbWCWCV1s+q9MNSMhVw==\n', 16)
[craig@craig-pc Documents]$
```

Figure 10: Output for CryptoLab2 - section 2.2

In the terminal, it shows the results for the task in hex format and base64 encode format. Essentially, Base64 is a binary to text encoding scheme that is generally used to transfer content-based messages over the Internet. Every three bits of binary data is divided into six units. This new data is represented in a 64-radix numerical system and as 7-bit ASCII text. Since each bit is divided into two bits, the new converted data is $\frac{1}{3}$ larger than the original.

0.0.2.3 Task 2.3: Effect of Padding

For this task, I had to try to add some padding to the data so that the program can handle data of any size. This is done by creating the padder, padding the data, and printing the padded data and encrypting it. The code that was added to the program is the following:



```
#--------------------Task 2.4--------------------------------
print("\nUnder this are the results for TASK 2.4 CBC mode")
cipher = Cipher(algorithm=algorithms.AES(key), mode=modes.CBC(iv),backend=backend)
encryptor = cipher.encryptor()
padder = padding.PKCS7(128).padder()

#mydata = b''
mydata = b'1234567890123456789012345678901212345678901234567890123456789012'
 #mydata = b'12345678'
print(len(mydata))
mydata_pad = padder.update(mydata) + padder.finalize()
print("My padded data: ",mydata_pad.hex(), len(mydata_pad.hex()))
ciphertext= encryptor.update(mydata_pad) + encryptor.finalize()
print("My ciphertext: ",ciphertext.hex(), len(ciphertext.hex()))
```

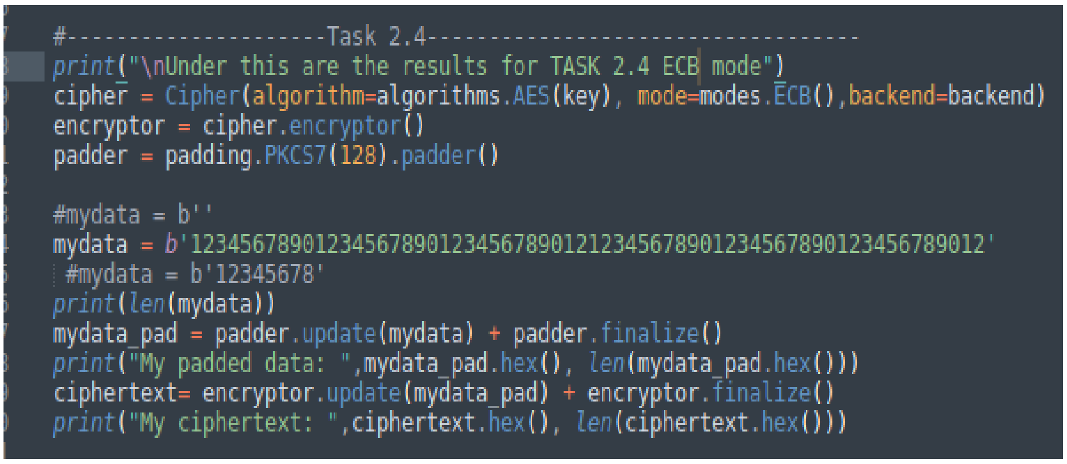
Figure 11: Code for CryptoLab3.py - section 2.3

Please note that although the comment says it is for task 2.4, this is the same code that was used for this task. It includes the padder that allows for padding the text. Some observations are the following:

- If the length of the message is a multiple, so like 64, then it still adds a length of 32 at the end. THEN it pads 64 so the total length is 160.
- If the length of the message if not a multiple, so like 67, the size to pad is 64. HOWEVER, the code adds a padding of 32 MINUS the amount after 64. In this case, it is 32 - 3, so it adds 29. THEN it adds a padding of 64. So total length is still 160.
- If the length of message is 96, it will add 32 times at the end of the message. In addition, it will add a padding of 96 so that the padding is the same size of the message.
- So whether or not the message is a not a multiple, it will add to the end until it gets to the next multiple of 32. So if message is size 17, it will add 32 - (17-16) to the message. And THEN add a padding of 32. In general, if not a multiple, adds 32 - (n-(n-1)), where n is the length of message. So if length is 65, then adds 32 - (65-64) = 31. So full "message length" is 96 + 64 to match the message size.

0.0.2.4 Task 2.4: Using AES in other modes

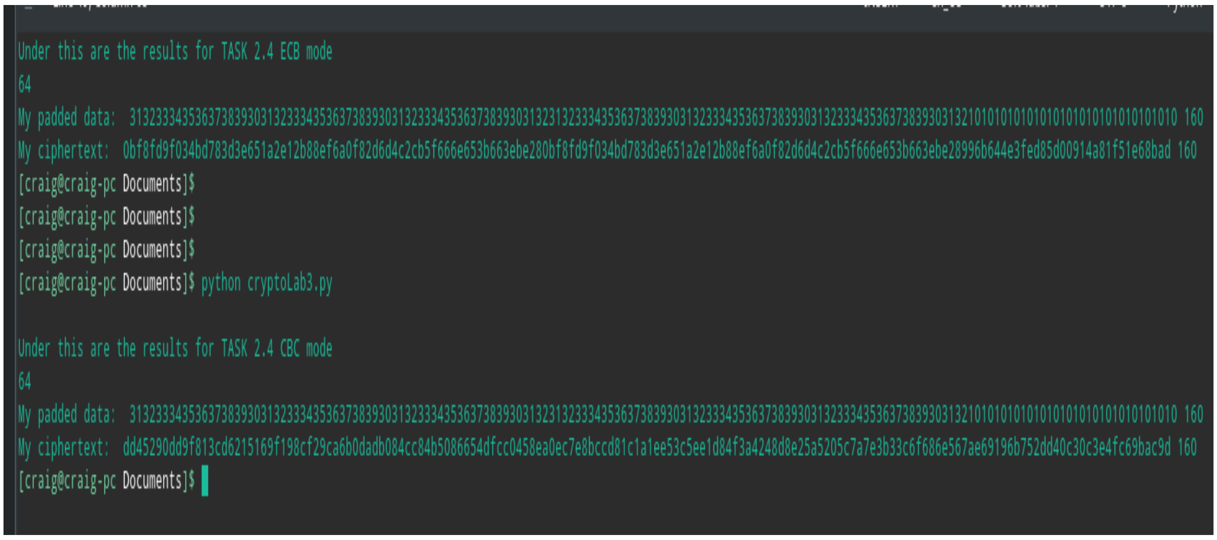
For this task, I just had to try out another mode of encryption. The first mode I tried was shown in the previous task, which was CBC mode. Another mode I did was ECB mode, whose code is shown below:



```python
#--------------------Task 2.4----------------------------------
print("\nUnder this are the results for TASK 2.4 ECB mode")
cipher = Cipher(algorithm=algorithms.AES(key), mode=modes.ECB(),backend=backend)
encryptor = cipher.encryptor()
padder = padding.PKCS7(128).padder()

#mydata = b''
mydata = b'1234567890123456789012345678901212345678901234567890123456789012'
 #mydata = b'12345678'
print(len(mydata))
mydata_pad = padder.update(mydata) + padder.finalize()
print("My padded data: ",mydata_pad.hex(), len(mydata_pad.hex()))
ciphertext= encryptor.update(mydata_pad) + encryptor.finalize()
print("My ciphertext: ",ciphertext.hex(), len(ciphertext.hex()))
```

Figure 12: Code for CryptoLab3.py - section 2.4

After running both codes, the results were the following:



```
Under this are the results for TASK 2.4 ECB mode
64
My padded data:  31323334353637383930313233343536373839303132333435363738393031323132333435363738393031323334353637383930313233343536373839303132101010101010101010101010101010 160
My ciphertext:  0bf8fd9f034bd783d3e651a2e12b88ef6a0f82d6d4c2cb5f666e653b663ebe280bf8fd9f034bd783d3e651a2e12b88ef6a0f82d6d4c2cb5f666e653b663ebe28996b644e3fed85d00914a81f51e68bad 160
[craig@craig-pc Documents]$
[craig@craig-pc Documents]$
[craig@craig-pc Documents]$
[craig@craig-pc Documents]$ python cryptoLab3.py

Under this are the results for TASK 2.4 CBC mode
64
My padded data:  31323334353637383930313233343536373839303132333435363738393031323132333435363738393031323334353637383930313233343536373839303132101010101010101010101010101010 160
My ciphertext:  dd45290dd9f813cd6215169f198cf29ca6b0dadb084cc84b5086654dfcc0458ea0ec7e8bccd81c1a1ee53c5ee1d84f3a4248d8e25a5205c7a7e3b33c6f686e567ae69196b752dd40c30c3e4fc69bac9d 160
[craig@craig-pc Documents]$
```

Figure 13: Output for CryptoLab3.py

The padding principles are the same as described in Task 2.3. The only difference is what the cipher-text is. The plain-text is still received, padded, and encrypted. However, the resulting cipher-text is the only thing that is different. This is because the IV is not used in the ECB mode like the CBC mode.

0.0.3 Task 3: Encrypting a file with AES

Program: encryption_decryption.py

For this task, I had to make a program, combining all the previous sections to encrypt a text file, and then be able to decrypt it. Note that both results are written out to a new file. The code is too big to be shown in a single screenshot, but it is added to the submission as a python file. The way the code works is explained below:

- First, the text file that I wanted to encrypt had to be relatively long. So I chose a section of the Gettysburg Address and copied the contents into a file called "File To Encrypt".
- Once the user runs the program, a message appears asking to either type 'e' to encrypt, or 'd' to decrypt. The way the program is set up is to encrypt the Gettysburg Address and decrypt its cipher-text. However, to encrypt and decrypt files, one would simply have to change the names that are in the code to their choosing.
- If encrypting, the code goes to the encrypt function. Note that the password in generated ONLY in the encrypt function, so it is not known on the outside. The program pads the data, encrypts it, and returns the hex value(s). In addition, the password is added to the first 16 characters of the cipher-text to use for decryption. Then, the output is written to a file called "Encryption Result"
- If decrypting, the code goes to the decrypt function. Note that the password is first retrieved by making note of the first 16 characters of the cipher-text. Then the 16 characters are deleted from the cipher-text. Now it is ready to decrypt. It first decrypts, then unpads the data, and writes the output to a file called "Decryption Result".

- It is important to note that the password is NEVER in the open. The IV is the only thing in the open, but if needed, it is possible to also append it to either the beginning of the cipher-text as well or at the end to use for decrypting. This way, both are not in the open.
- The end result for decryption is the same as the original plain-text. So it is safe to say that the program works. Thus, the encryption works as well.
- To see the program, simply download the .rar file that is included in the submission and run the "encryption_decryption.py" file. It is important to ensure that all the text files are in the same directory as the python file so that the program works properly. If you have your own text file you want to encrypt, simply modify the code to encrypt that file. Afterwards, you can decrypt it to make sure that it is done correctly.

Please note that the python code is too big to be included in one snapshot. So to see the PDF of the code click here

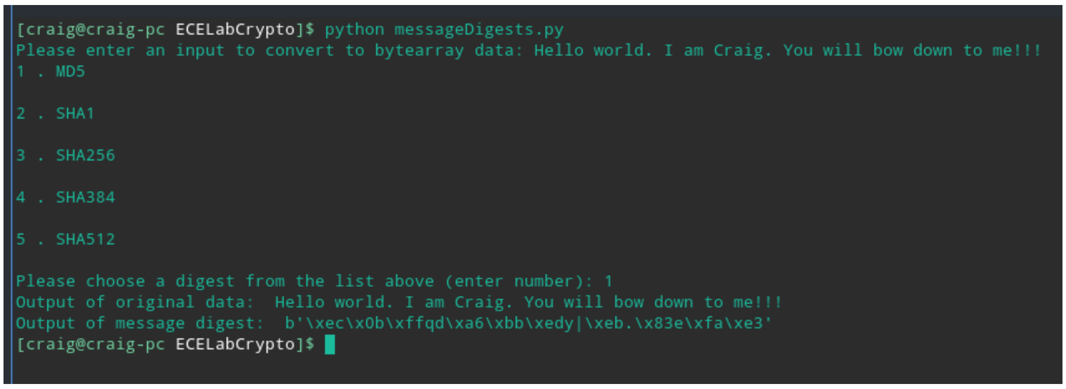
CryptoLab2

0.0.4 Task 1: Create and use Message Digests (Hashes)

Program: messageDigests.py

"Message Digests are basic hashes. These are created in much the same way as the Cipher objects. They do not have keys associated with them, so they only provide based integrity checks, but are easily bypassed"

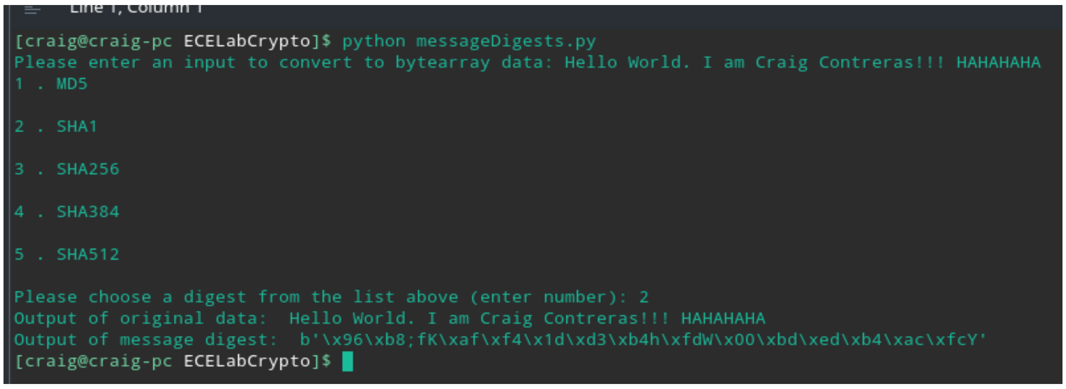
The program shown combines task 1 and task 1.2 was to modify our program so that it can use different message digests, such as MD5, SHA1, SHA256, SHA-384, SHA-512. The way I made the program was to have it so that it "interacts" with the user. The first thing that happens when the program is run is ask you to type a message that you want to digest/hash. The user is then free to type in any kind of message that feels appropriate. Then, the user is prompted to choose which digest they would like to use. There is a list of 1 - 5 which a digest for each of them. The user has to type in one of those numbers and the message will be "digested". The first example shows the user being prompted to type a message. Then, the MD5 digest is chosen and the original data with the digested message is outputted.



```
[craig@craig-pc ECELabCrypto]$ python messageDigests.py
Please enter an input to convert to bytearray data: Hello world. I am Craig. You will bow down to me!!!
1 . MD5

2 . SHA1

3 . SHA256

4 . SHA384

5 . SHA512

Please choose a digest from the list above (enter number): 1
Output of original data:  Hello world. I am Craig. You will bow down to me!!!
Output of message digest:  b'\xec\x0b\xffqd\xa6\xbb\xedy|\xeb.\x83e\xfa\xe3'
[craig@craig-pc ECELabCrypto]$
```

Figure 14: Output for messageDigests.py - picking MD5 Digest

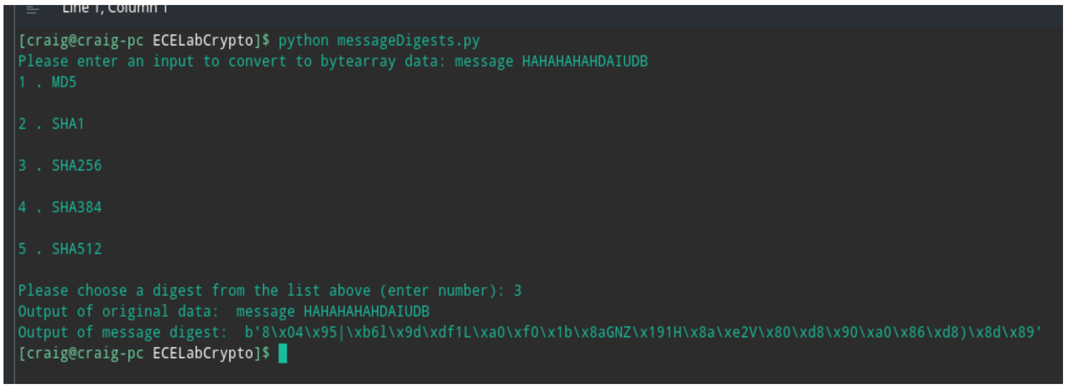
The next picture shows the user picking the SHA1 digest.



```
Line 1, Column 1
[craig@craig-pc ECELabCrypto]$ python messageDigests.py
Please enter an input to convert to bytearray data: Hello World. I am Craig Contreras!!! HAHAHAHA
1 . MD5

2 . SHA1

3 . SHA256

4 . SHA384

5 . SHA512

Please choose a digest from the list above (enter number): 2
Output of original data:  Hello World. I am Craig Contreras!!! HAHAHAHA
Output of message digest:  b'\x96\xb8;fK\xaf\xf4\x1d\xd3\xb4h\xfdW\x00\xbd\xed\xb4\xac\xfcY'
[craig@craig-pc ECELabCrypto]$
```

Figure 15: Output for messageDigests.py - picking SHA1 Digest

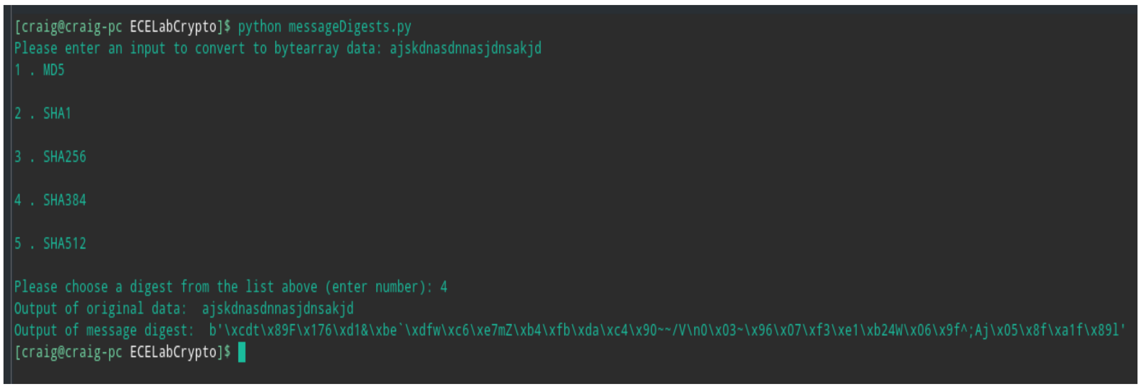
Next, the user picked the SHA256 digest.



```
Line 1, Column 1
[craig@craig-pc ECELabCrypto]$ python messageDigests.py
Please enter an input to convert to bytearray data: message HAHAHAHAHDAIUDB
1 . MD5

2 . SHA1

3 . SHA256

4 . SHA384

5 . SHA512

Please choose a digest from the list above (enter number): 3
Output of original data:  message HAHAHAHAHDAIUDB
Output of message digest:  b'8\x04\x95|\xb6l\x9d\xdf1L\xa0\xf0\x1b\x8aGNZ\x191H\x8a\xe2V\x80\xd8\x90\xa0\x86\xd8)\x8d\x89'
[craig@craig-pc ECELabCrypto]$
```

Figure 16: Output for messageDigests.py - picking SHA256 Digest

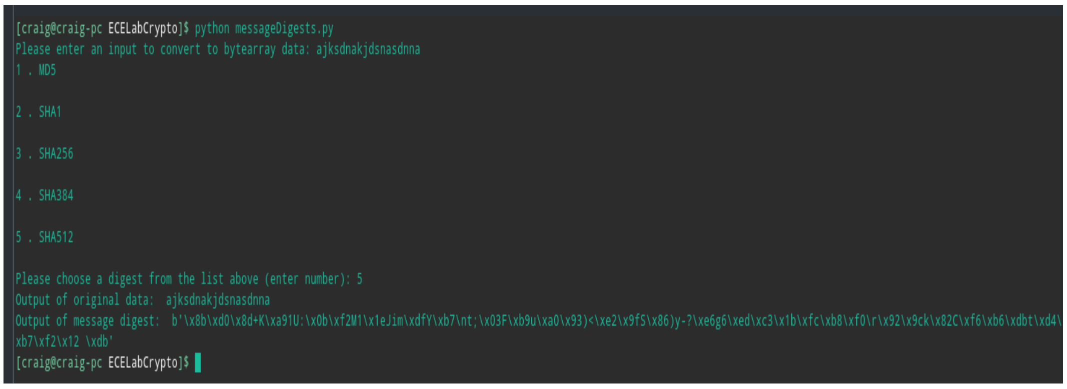
This is followed by the user picking the SHA384 digest.



```
[craig@craig-pc ECELabCrypto]$ python messageDigests.py
Please enter an input to convert to bytearray data: ajskdnasdnnasjdnsakjd
1 . MD5

2 . SHA1

3 . SHA256

4 . SHA384

5 . SHA512

Please choose a digest from the list above (enter number): 4
Output of original data:  ajskdnasdnnasjdnsakjd
Output of message digest:  b'\xcdt\x89F\x176\xd1&\xbe`\xdfw\xc6\xe7mZ\xb4\xfb\xda\xc4\x90~~/V\n0\x03~\x96\x07\xf3\xe1\xb24W\x06\x9f^;Aj\x05\x8f\xa1f\x89l'
[craig@craig-pc ECELabCrypto]$
```

Figure 17: Output for messageDigests.py - picking SHA384 Digest

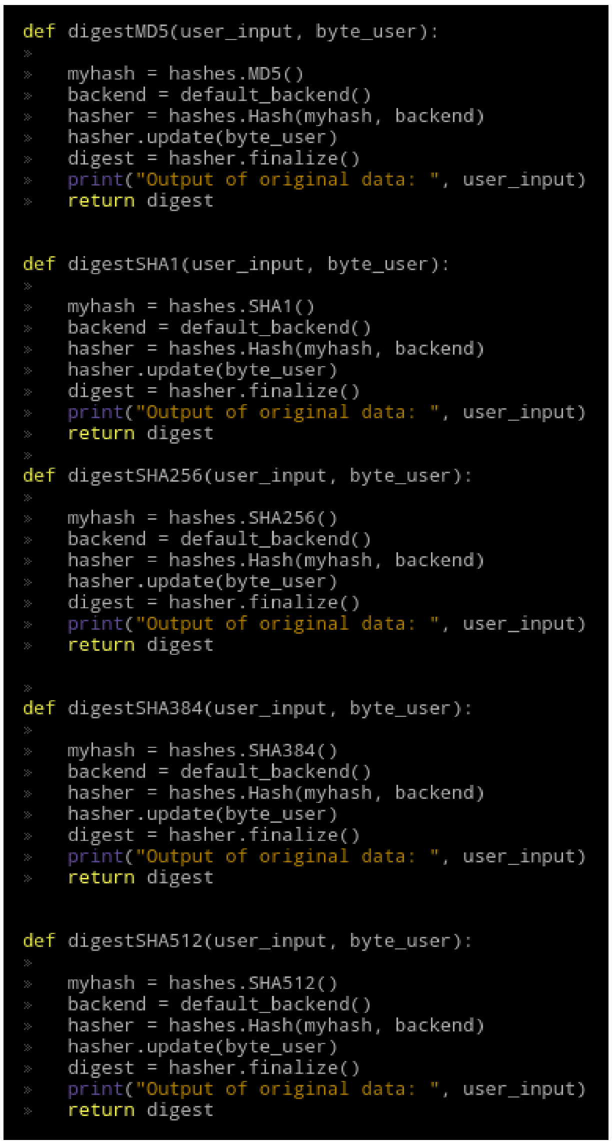
Finally, the SHA512 digest.



```
[craig@craig-pc ECELabCrypto]$ python messageDigests.py
Please enter an input to convert to bytearray data: ajksdnakjdsnasdnna
1 . MD5

2 . SHA1

3 . SHA256

4 . SHA384

5 . SHA512

Please choose a digest from the list above (enter number): 5
Output of original data:  ajksdnakjdsnasdnna
Output of message digest:  b'\x8b\xd0\x8d+K\xa91U:\x0b\xf2M1\x1eJim\xdfY\xb7\nt;\x03F\xb9u\xa0\x93)<\xe2\x9fS\x86)y-?\xe6g6\xed\xc3\x1b\xfc\xb8\xf0\r\x92\x9ck\x82C\xf6\xb6\xdbt\xd4\
xb7\xf2\x12 \xdb'
[craig@craig-pc ECELabCrypto]$
```

Figure 18: Output for messageDigests.py - picking SHA512 Digest

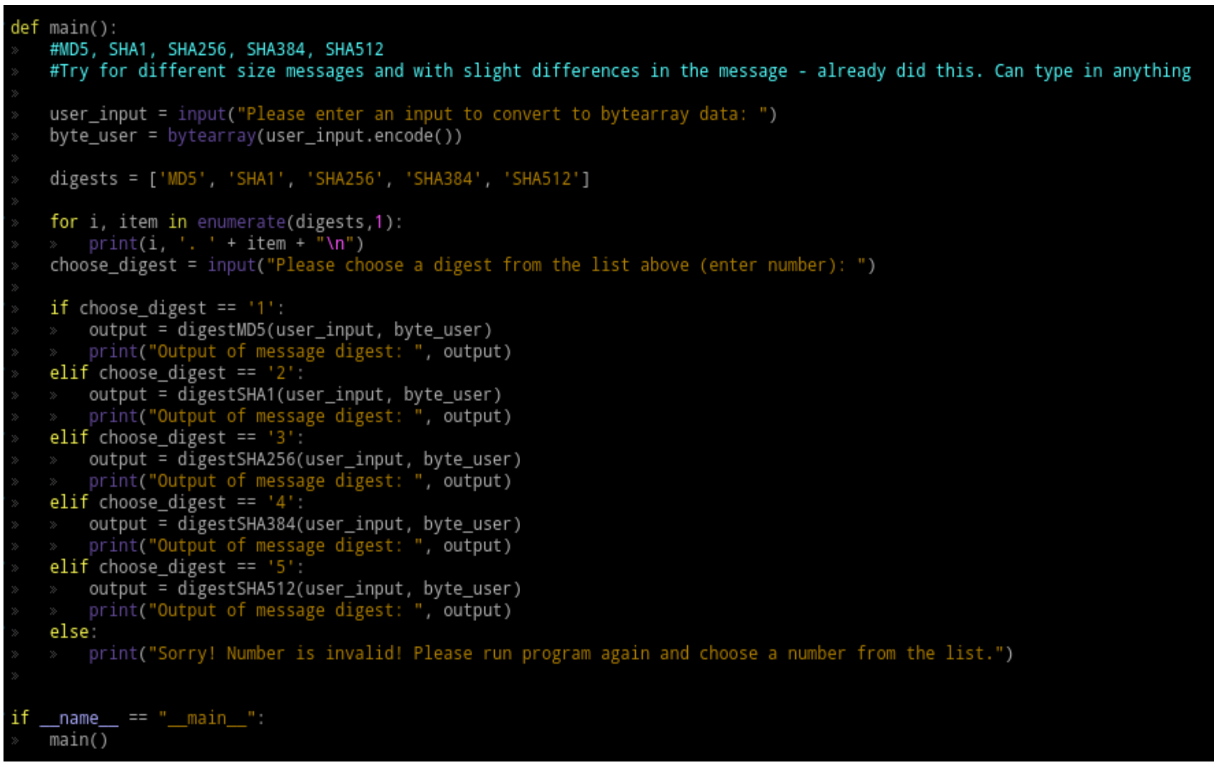
For each of the message digests, I made a separate function. This is because it allowed for the program to be deemed as "interactive". It did NOT do all of the digests sequentially. In addition, the program could be modified so that it reads from a text file or could already be given a string variable that has the different digests done on. The program is flexible, but for the purpose of this lab, the user simply had to input a message. A screenshot of how the different digests were coded is shown below.



```
def digestMD5(user_input, byte_user):

    myhash = hashes.MD5()
    backend = default_backend()
    hasher = hashes.Hash(myhash, backend)
    hasher.update(byte_user)
    digest = hasher.finalize()
    print("Output of original data: ", user_input)
    return digest


def digestSHA1(user_input, byte_user):

    myhash = hashes.SHA1()
    backend = default_backend()
    hasher = hashes.Hash(myhash, backend)
    hasher.update(byte_user)
    digest = hasher.finalize()
    print("Output of original data: ", user_input)
    return digest

def digestSHA256(user_input, byte_user):

    myhash = hashes.SHA256()
    backend = default_backend()
    hasher = hashes.Hash(myhash, backend)
    hasher.update(byte_user)
    digest = hasher.finalize()
    print("Output of original data: ", user_input)
    return digest


def digestSHA384(user_input, byte_user):

    myhash = hashes.SHA384()
    backend = default_backend()
    hasher = hashes.Hash(myhash, backend)
    hasher.update(byte_user)
    digest = hasher.finalize()
    print("Output of original data: ", user_input)
    return digest


def digestSHA512(user_input, byte_user):

    myhash = hashes.SHA512()
    backend = default_backend()
    hasher = hashes.Hash(myhash, backend)
    hasher.update(byte_user)
    digest = hasher.finalize()
    print("Output of original data: ", user_input)
    return digest
```

Figure 19: Functions for all the digest methods

In addition, a screenshot of the the "main" function was coded is also shown below. The main function shows what would happen if the user types either number between 1 - 5 and what would happen if the user typed anything else besides that.



```python
def main():
    #MD5, SHA1, SHA256, SHA384, SHA512
    #Try for different size messages and with slight differences in the message - already did this. Can type in anything

    user_input = input("Please enter an input to convert to bytearray data: ")
    byte_user = bytearray(user_input.encode())

    digests = ['MD5', 'SHA1', 'SHA256', 'SHA384', 'SHA512']

    for i, item in enumerate(digests,1):
        print(i, '. ' + item + "\n")
    choose_digest = input("Please choose a digest from the list above (enter number): ")

    if choose_digest == '1':
        output = digestMD5(user_input, byte_user)
        print("Output of message digest: ", output)
    elif choose_digest == '2':
        output = digestSHA1(user_input, byte_user)
        print("Output of message digest: ", output)
    elif choose_digest == '3':
        output = digestSHA256(user_input, byte_user)
        print("Output of message digest: ", output)
    elif choose_digest == '4':
        output = digestSHA384(user_input, byte_user)
        print("Output of message digest: ", output)
    elif choose_digest == '5':
        output = digestSHA512(user_input, byte_user)
        print("Output of message digest: ", output)
    else:
        print("Sorry! Number is invalid! Please run program again and choose a number from the list.")


if __name__ == "__main__":
    main()
```

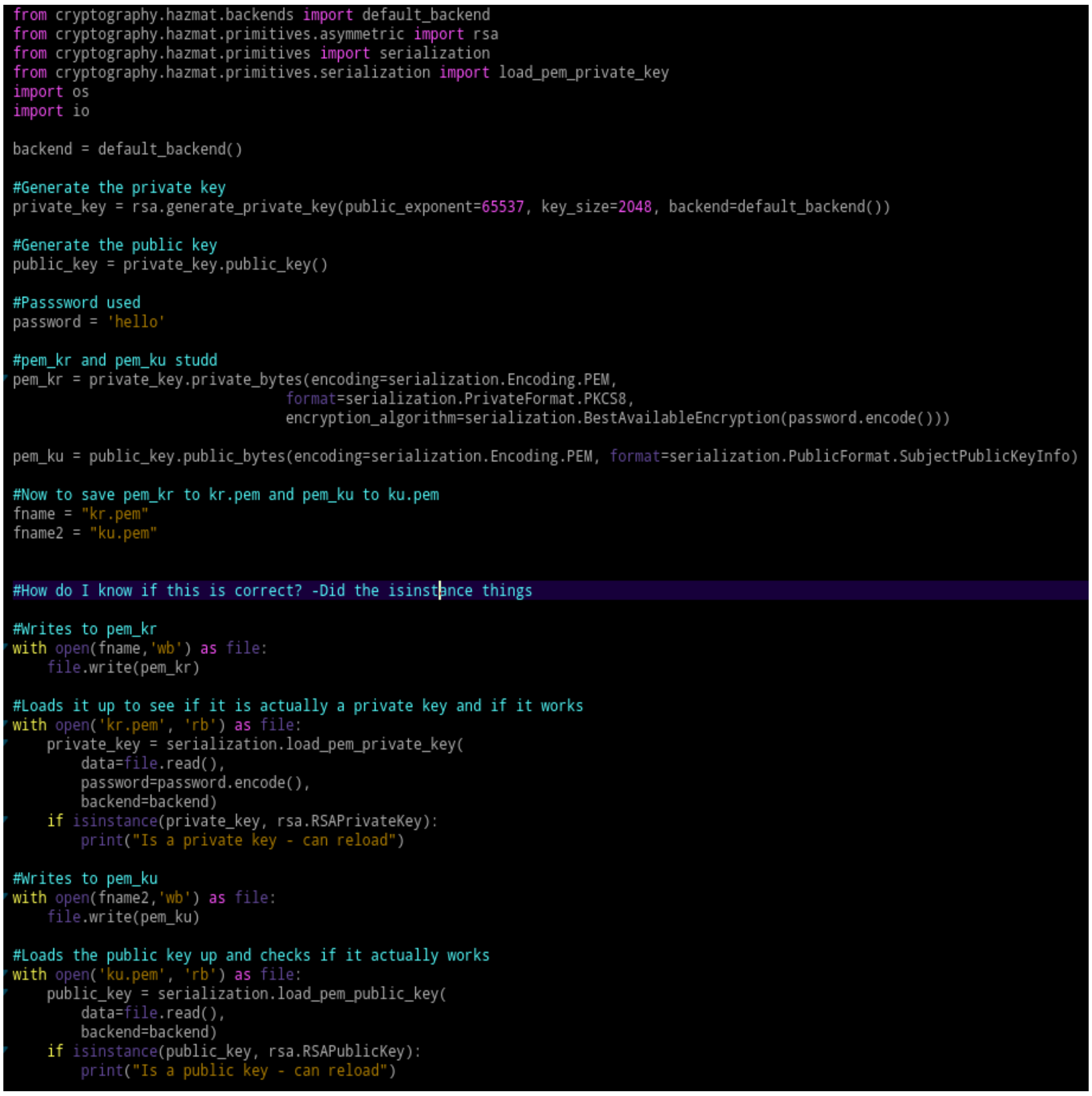
Figure 20: Main function for messageDigests.py

0.0.5 Task 2: Create and use RSA public/private key pair

Program: RSA_keypair.py

"Now you will create a RSA key pair"

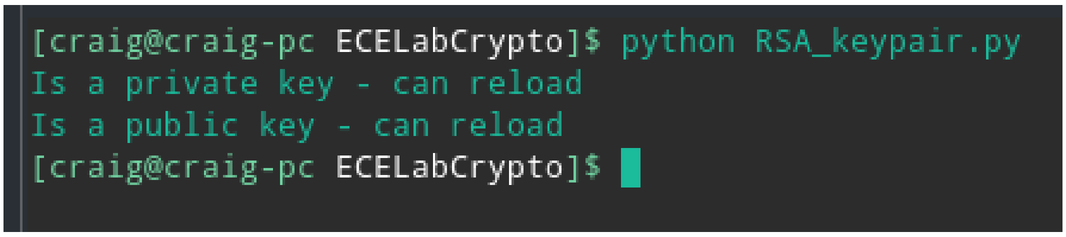
For this task, there wasn't much output to show. The entire program's code can fit in one screenshot, so that will be shown later. It is important to know that in order to check as to whether or the the private and public keys were actuallu public and private, respectively and if they actually worked, I did the "isinstance" command. I check to see whether my public/private key is an rsa.RSAPrivateKey and rsa.RSAPublicKey. If they are, then I print out, "is private/public key - can reload". That ensures that I can reload the keys. The first picture that is showed will be the code for the program.



```python
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.serialization import load_pem_private_key
import os
import io

backend = default_backend()

#Generate the private key
private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048, backend=default_backend())

#Generate the public key
public_key = private_key.public_key()

#Passsword used
password = 'hello'

#pem_kr and pem_ku studd
pem_kr = private_key.private_bytes(encoding=serialization.Encoding.PEM,
                                format=serialization.PrivateFormat.PKCS8,
                                encryption_algorithm=serialization.BestAvailableEncryption(password.encode()))

pem_ku = public_key.public_bytes(encoding=serialization.Encoding.PEM, format=serialization.PublicFormat.SubjectPublicKeyInfo)

#Now to save pem_kr to kr.pem and pem_ku to ku.pem
fname = "kr.pem"
fname2 = "ku.pem"


#How do I know if this is correct? -Did the isinstance things

#Writes to pem_kr
with open(fname,'wb') as file:
    file.write(pem_kr)

#Loads it up to see if it is actually a private key and if it works
with open('kr.pem', 'rb') as file:
    private_key = serialization.load_pem_private_key(
        data=file.read(),
        password=password.encode(),
        backend=backend)
    if isinstance(private_key, rsa.RSAPrivateKey):
        print("Is a private key - can reload")

#Writes to pem_ku
with open(fname2,'wb') as file:
    file.write(pem_ku)

#Loads the public key up and checks if it actually works
with open('ku.pem', 'rb') as file:
    public_key = serialization.load_pem_public_key(
        data=file.read(),
        backend=backend)
    if isinstance(public_key, rsa.RSAPublicKey):
        print("Is a public key - can reload")
```

Figure 21: Code used for RSA_keypair.py

In order to make the program run, one simply has to run the python file and the output should be:



```
[craig@craig-pc ECELabCrypto]$ python RSA_keypair.py
Is a private key - can reload
Is a public key - can reload
[craig@craig-pc ECELabCrypto]$
```

Figure 22: Output for RSA_keypair.py

Thus, the task was successful and the keys that were created and saved can be reloaded.

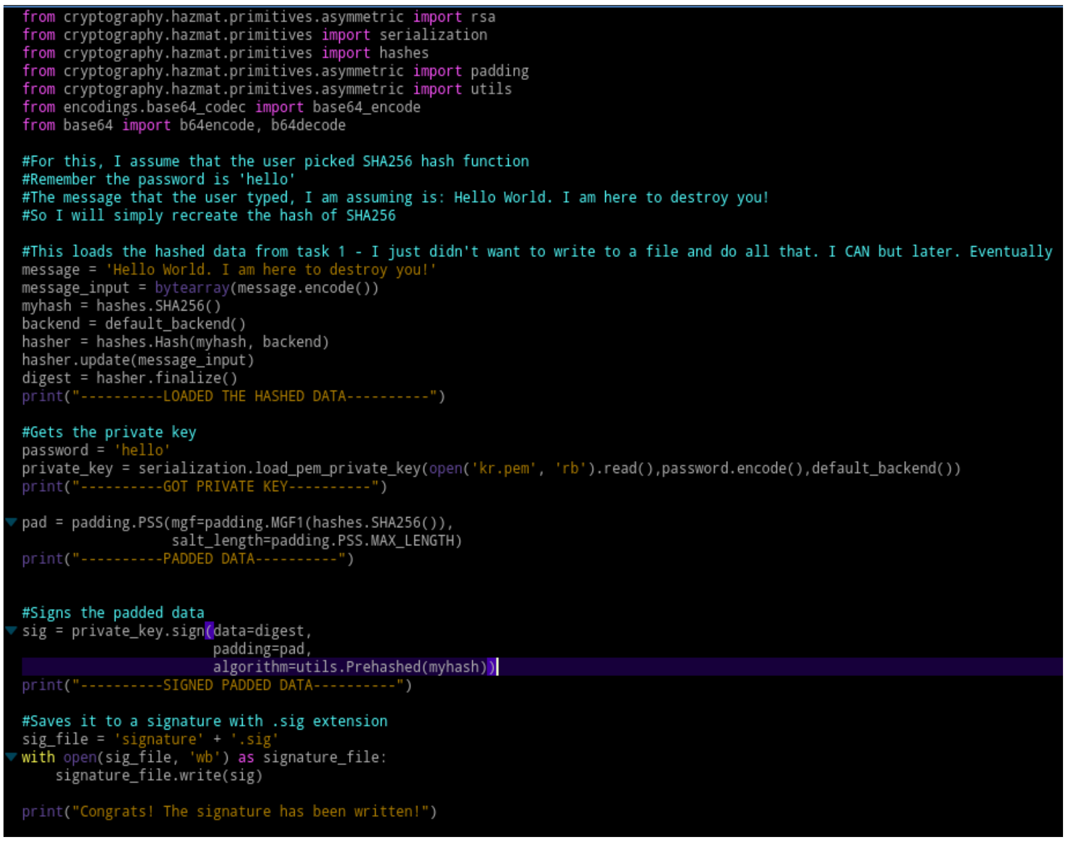
0.0.6 Task 3: Combine the message digest with the signature to create a signed document

"Combine the code from Task 1 and Task 2 to create a signed message digest that acts as a signature."

0.0.6.1 Task 3.1

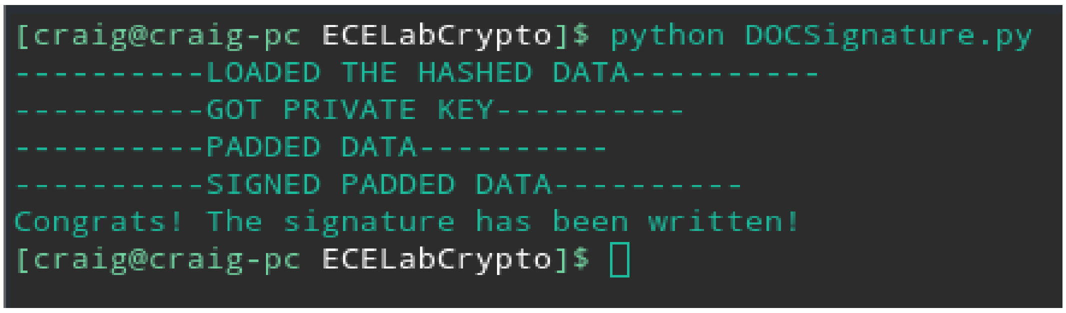
Program: DOCSignature.py

For this task, I had to load and hash the data from Task 1. For this, I kept the data with a digest of SHA256. Instead of saving that data onto a file and then reading it, I just re-created it on the program. A pre-determined message of "Hello World. I am here to destroy you!" was used. In addition, the private key was read from the "kr.pem" file. Afterwards, the padded digested data is signed and the signature written to a .sig file with PEM format. The output is shown in a snapshot below after the snapshot of the code.



```python
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives.asymmetric import utils
from encodings.base64_codec import base64_encode
from base64 import b64encode, b64decode

#For this, I assume that the user picked SHA256 hash function
#Remember the password is 'hello'
#The message that the user typed, I am assuming is: Hello World. I am here to destroy you!
#So I will simply recreate the hash of SHA256

#This loads the hashed data from task 1 - I just didn't want to write to a file and do all that. I CAN but later. Eventually
message = 'Hello World. I am here to destroy you!'
message_input = bytearray(message.encode())
myhash = hashes.SHA256()
backend = default_backend()
hasher = hashes.Hash(myhash, backend)
hasher.update(message_input)
digest = hasher.finalize()
print("----------LOADED THE HASHED DATA----------")

#Gets the private key
password = 'hello'
private_key = serialization.load_pem_private_key(open('kr.pem', 'rb').read(),password.encode(),default_backend())
print("----------GOT PRIVATE KEY----------")

pad = padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
                  salt_length=padding.PSS.MAX_LENGTH)
print("----------PADDED DATA----------")


#Signs the padded data
sig = private_key.sign(data=digest,
                       padding=pad,
                       algorithm=utils.Prehashed(myhash))
print("----------SIGNED PADDED DATA----------")

#Saves it to a signature with .sig extension
sig_file = 'signature' + '.sig'
with open(sig_file, 'wb') as signature_file:
    signature_file.write(sig)

print("Congrats! The signature has been written!")
```

Figure 23: Code used for DOCSignature.py



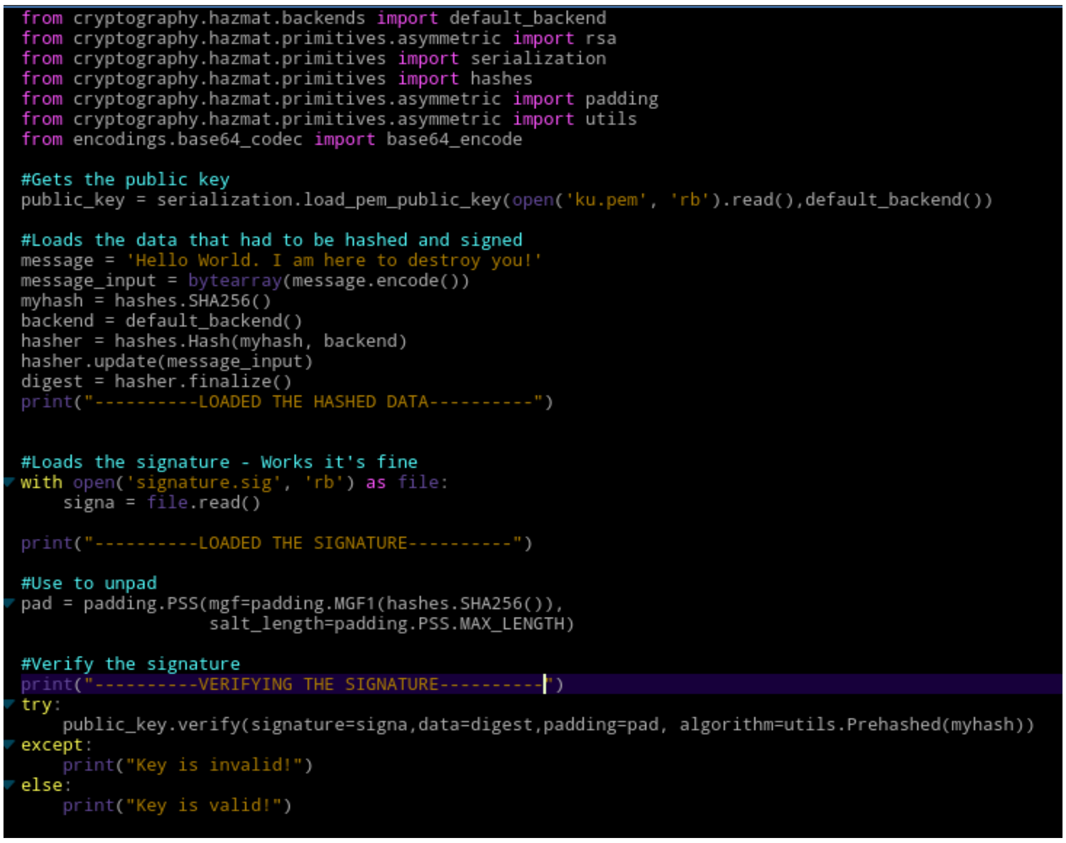
```
[craig@craig-pc ECELabCrypto]$ python DOCSignature.py
----------LOADED THE HASHED DATA----------
----------GOT PRIVATE KEY----------
----------PADDED DATA----------
----------SIGNED PADDED DATA----------
Congrats! The signature has been written!
[craig@craig-pc ECELabCrypto]$ 
```

Figure 24: Output for DOCSignature.py

0.0.6.2 Task 3.2

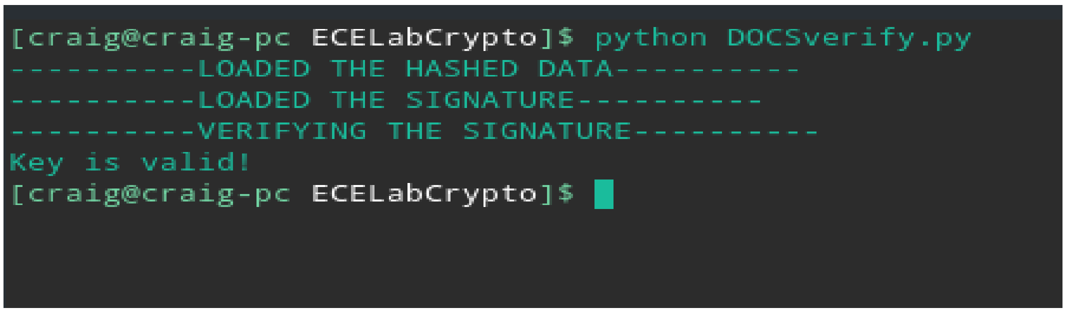
Program: DOCSverify.py

For this task, I had to verify the signature. First, I had to load the hashed data to be signed. Like task 3.1, I just re-created it on the program. Then, I loaded the public key followed by the signature that was made on task 3.1. To verify that the signature is correct, or deemed as valid, I used a try-except condition that tries to run the public key verify command as indicated in the lab manual. If it gives off an error, or doesn't work for some reason, it prints out "The signature is invalid". If it is valid, the output is "The signature is valid". The code and the output of the program is shown below. One just has to run the program through the terminal.



```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives.asymmetric import utils
from encodings.base64_codec import base64_encode

#Gets the public key
public_key = serialization.load_pem_public_key(open('ku.pem', 'rb').read(),default_backend())

#Loads the data that had to be hashed and signed
message = 'Hello World. I am here to destroy you!'
message_input = bytearray(message.encode())
myhash = hashes.SHA256()
backend = default_backend()
hasher = hashes.Hash(myhash, backend)
hasher.update(message_input)
digest = hasher.finalize()
print("----------LOADED THE HASHED DATA----------")


#Loads the signature - Works it's fine
with open('signature.sig', 'rb') as file:
    signa = file.read()

print("----------LOADED THE SIGNATURE----------")

#Use to unpad
pad = padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
                  salt_length=padding.PSS.MAX_LENGTH)

#Verify the signature
print("----------VERIFYING THE SIGNATURE----------")
try:
    public_key.verify(signature=signa,data=digest,padding=pad, algorithm=utils.Prehashed(myhash))
except:
    print("Key is invalid!")
else:
    print("Key is valid!")
```

Figure 25: Code used for DOCSverify.py



```
[craig@craig-pc ECELabCrypto]$ python DOCSverify.py
----------LOADED THE HASHED DATA----------
----------LOADED THE SIGNATURE----------
----------VERIFYING THE SIGNATURE----------
Key is valid!
[craig@craig-pc ECELabCrypto]$
```

Figure 26: Output for DOCSverify.py

0.0.7 Task 4

Program: crypt_and_sign.py

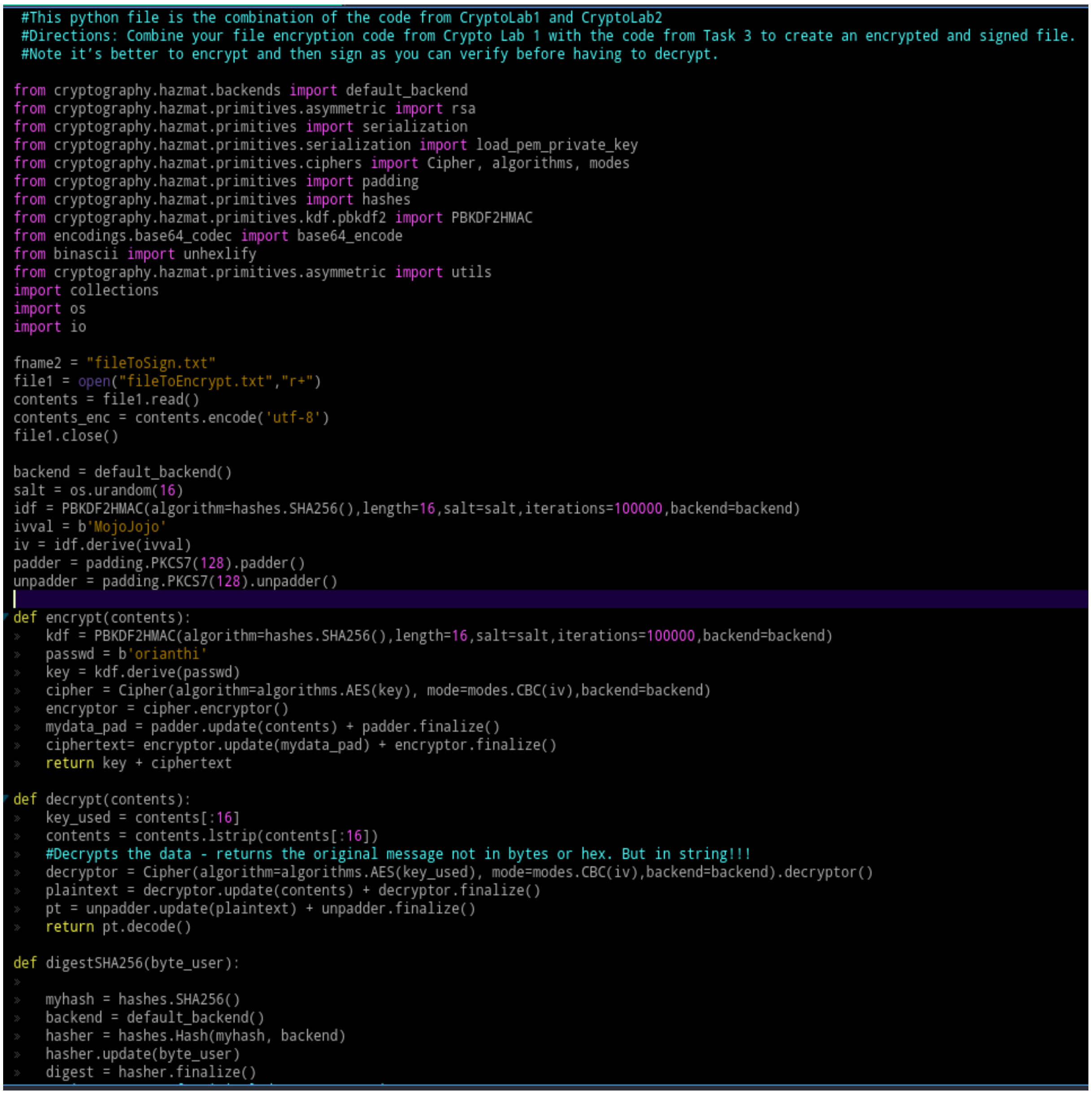
"Combine your file encryption code from Crypto Lab 1 with the code from Task 3 to create an encrypted and signed file. Note it's better to encrypt and then sign as you can verify before having to decrypt."

For this task, the first thing that had to be done was essentially combine the encryption/decryption file that was done in Crypto Lab 1 with out code from task 3 in order to create an encrypted and signed file. A snapshot of the code and the output I had it do will be shown at the end of the explanation.

First, the user is told to encrypt a file (note the file can be changed by modifying the code). My program is made so that the user has to at least have a file to encrypt for the program to run. In this case, the file that is being encrypted is the Gettysburg Address. The encryption result is written to a file called "fileToSign.txt". Afterwards, that file is opened and read and its contents are saved onto a variable.

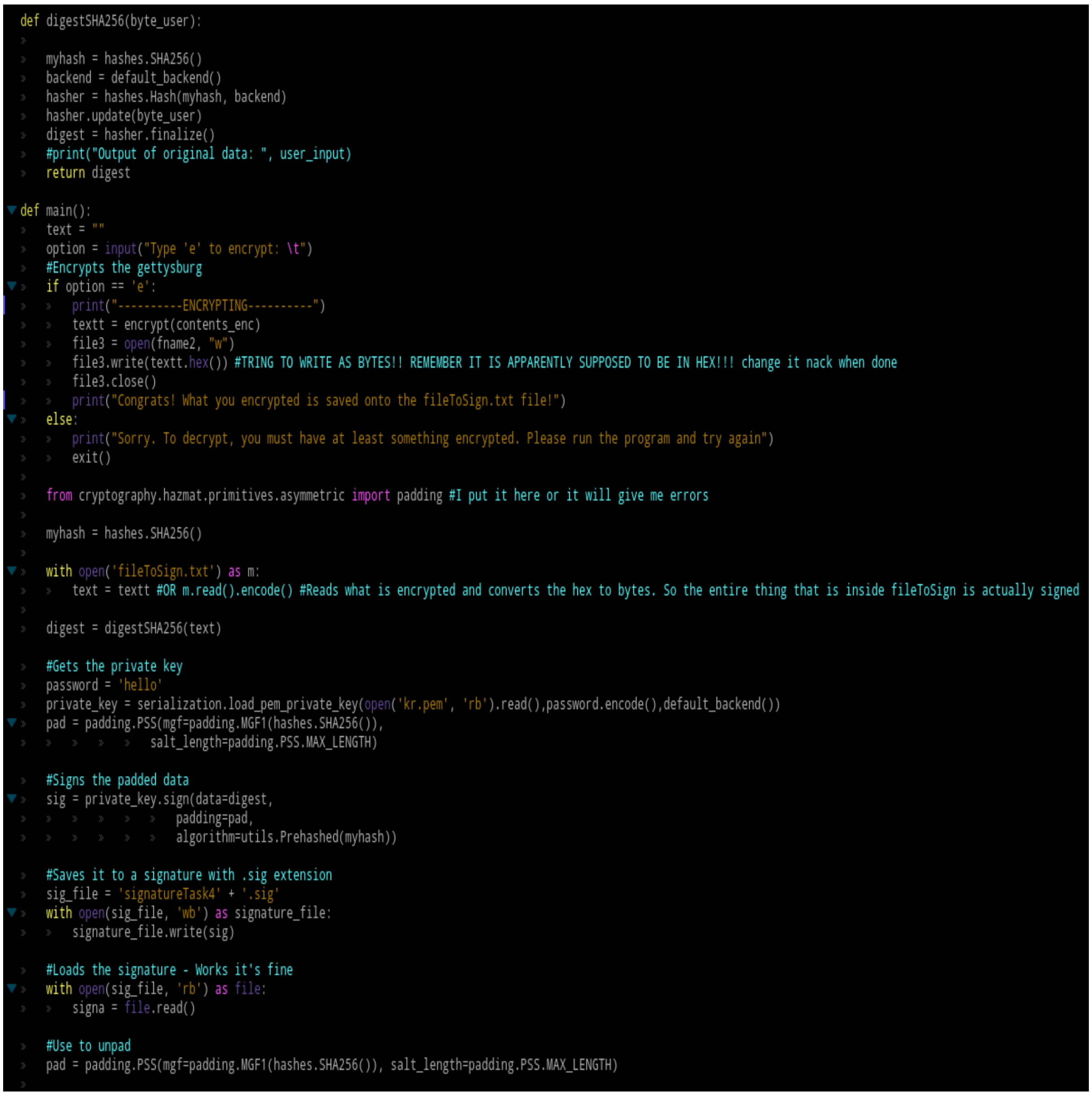
The default digest that I did was the SHA256 digest. Any digest can be done, but for the lab I had it run the SHA256 digest. Please note that all the digests presented in the lab work, as shown in the previous tasks/lab. The private key was loaded and then the padded data was signed. The data that was read and signed was the encryption of the Gettysburg Address in BYTES, not hex. The reason I kept it in bytes is because I did not want to convert the hex to bytes; it wasn't entirely accurate. The encryption was only written in hex, but digested as bytes.

To ensure the signature works/loads, I had it saved to a file and loaded up. Afterwards, I loaded up the public key and the variable used to unpad the data. Lastly, to verify the signature, I used a try-except-else expression. The verification was a success since it gave off no errors. So, the data was successfully encrypted, signed, and verified. The last task was to decrypt the encryption, which involved using the "decrypt" function. Those contents were written to a file called "decryptedSignedFile.txt". In addition, I had a print statement indicating that the decryption was a success.



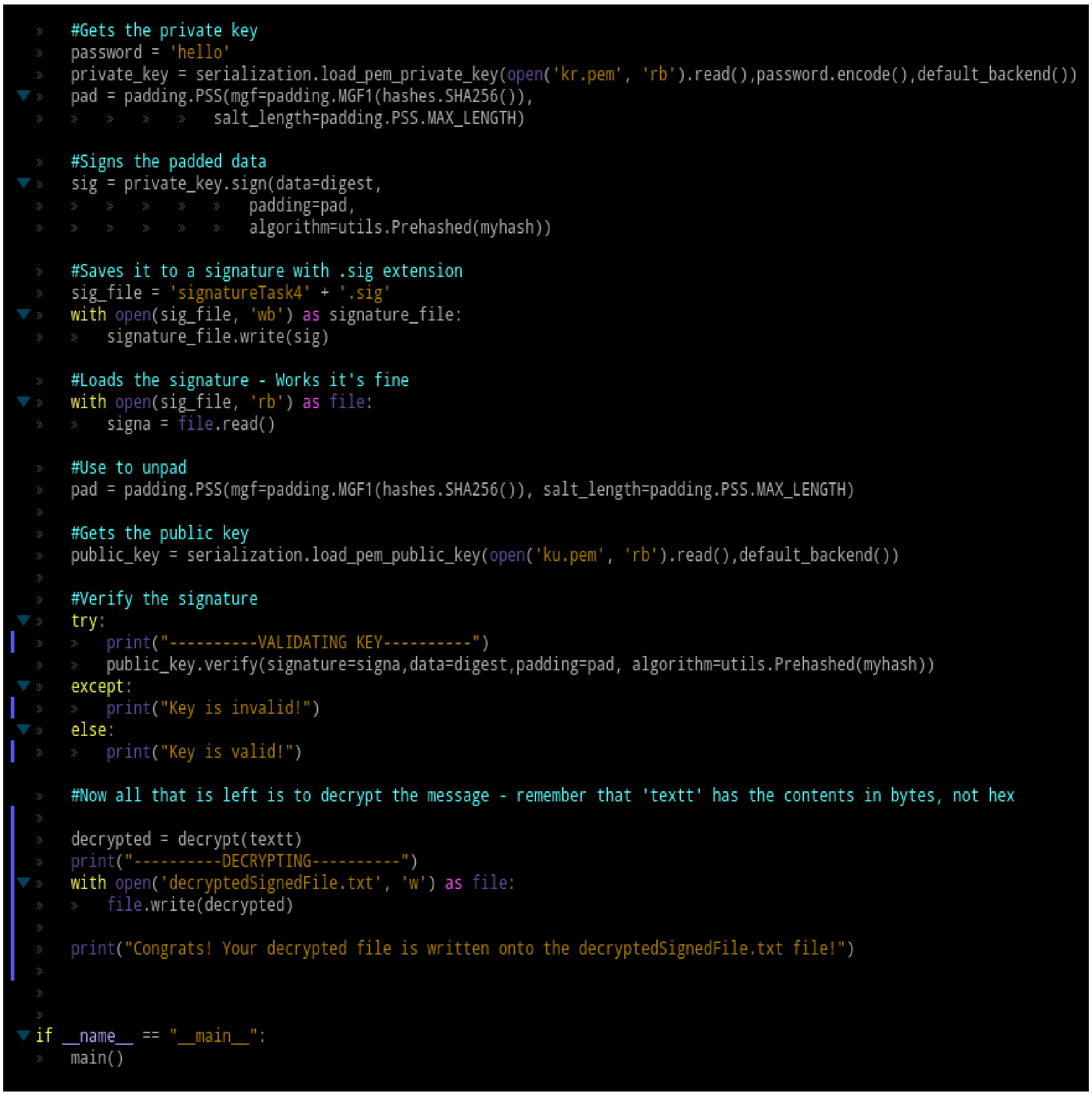
```python
 #This python file is the combination of the code from CryptoLab1 and CryptoLab2
 #Directions: Combine your file encryption code from Crypto Lab 1 with the code from Task 3 to create an encrypted and signed file.
 #Note it's better to encrypt and then sign as you can verify before having to decrypt.

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.serialization import load_pem_private_key
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from encodings.base64_codec import base64_encode
from binascii import unhexlify
from cryptography.hazmat.primitives.asymmetric import utils
import collections
import os
import io

fname2 = "fileToSign.txt"
file1 = open("fileToEncrypt.txt","r+")
contents = file1.read()
contents_enc = contents.encode('utf-8')
file1.close()

backend = default_backend()
salt = os.urandom(16)
idf = PBKDF2HMAC(algorithm=hashes.SHA256(),length=16,salt=salt,iterations=100000,backend=backend)
ivval = b'MojoJojo'
iv = idf.derive(ivval)
padder = padding.PKCS7(128).padder()
unpadder = padding.PKCS7(128).unpadder()

def encrypt(contents):
    kdf = PBKDF2HMAC(algorithm=hashes.SHA256(),length=16,salt=salt,iterations=100000,backend=backend)
    passwd = b'orianthi'
    key = kdf.derive(passwd)
    cipher = Cipher(algorithm=algorithms.AES(key), mode=modes.CBC(iv),backend=backend)
    encryptor = cipher.encryptor()
    mydata_pad = padder.update(contents) + padder.finalize()
    ciphertext= encryptor.update(mydata_pad) + encryptor.finalize()
    return key + ciphertext

def decrypt(contents):
    key_used = contents[:16]
    contents = contents.lstrip(contents[:16])
    #Decrypts the data - returns the original message not in bytes or hex. But in string!!!
    decryptor = Cipher(algorithm=algorithms.AES(key_used), mode=modes.CBC(iv),backend=backend).decryptor()
    plaintext = decryptor.update(contents) + decryptor.finalize()
    pt = unpadder.update(plaintext) + unpadder.finalize()
    return pt.decode()

def digestSHA256(byte_user):

    myhash = hashes.SHA256()
    backend = default_backend()
    hasher = hashes.Hash(myhash, backend)
    hasher.update(byte_user)
    digest = hasher.finalize()
```

Figure 27: Code used for crypt_sign.py - part 1



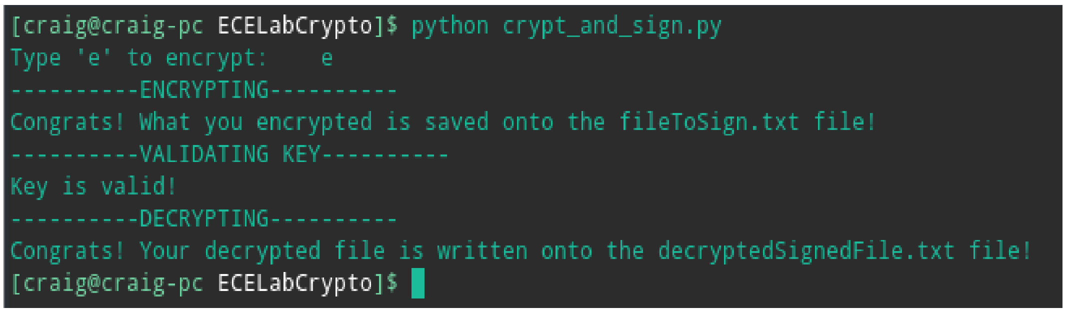
```
def digestSHA256(byte_user):

    myhash = hashes.SHA256()
    backend = default_backend()
    hasher = hashes.Hash(myhash, backend)
    hasher.update(byte_user)
    digest = hasher.finalize()
    #print("Output of original data: ", user_input)
    return digest

def main():
    text = ""
    option = input("Type 'e' to encrypt: \t")
    #Encrypts the gettysburg
    if option == 'e':
        print("----------ENCRYPTING----------")
        textt = encrypt(contents_enc)
        file3 = open(fname2, "w")
        file3.write(textt.hex()) #TRING TO WRITE AS BYTES!! REMEMBER IT IS APPARENTLY SUPPOSED TO BE IN HEX!!! change it nack when done
        file3.close()
        print("Congrats! What you encrypted is saved onto the fileToSign.txt file!")
    else:
        print("Sorry. To decrypt, you must have at least something encrypted. Please run the program and try again")
        exit()

    from cryptography.hazmat.primitives.asymmetric import padding #I put it here or it will give me errors

    myhash = hashes.SHA256()

    with open('fileToSign.txt') as m:
        text = textt #OR m.read().encode() #Reads what is encrypted and converts the hex to bytes. So the entire thing that is inside fileToSign is actually signed

    digest = digestSHA256(text)

    #Gets the private key
    password = 'hello'
    private_key = serialization.load_pem_private_key(open('kr.pem', 'rb').read(),password.encode(),default_backend())
    pad = padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
                    salt_length=padding.PSS.MAX_LENGTH)

    #Signs the padded data
    sig = private_key.sign(data=digest,
                        padding=pad,
                        algorithm=utils.Prehashed(myhash))

    #Saves it to a signature with .sig extension
    sig_file = 'signatureTask4' + '.sig'
    with open(sig_file, 'wb') as signature_file:
        signature_file.write(sig)

    #Loads the signature - Works it's fine
    with open(sig_file, 'rb') as file:
        signa = file.read()

    #Use to unpad
    pad = padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=padding.PSS.MAX_LENGTH)
```

Figure 28: Code used for crypt_sign.py - part 2



```
    #Gets the private key
    password = 'hello'
    private_key = serialization.load_pem_private_key(open('kr.pem', 'rb').read(),password.encode(),default_backend())
    pad = padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
                    salt_length=padding.PSS.MAX_LENGTH)

    #Signs the padded data
    sig = private_key.sign(data=digest,
                        padding=pad,
                        algorithm=utils.Prehashed(myhash))

    #Saves it to a signature with .sig extension
    sig_file = 'signatureTask4' + '.sig'
    with open(sig_file, 'wb') as signature_file:
        signature_file.write(sig)

    #Loads the signature - Works it's fine
    with open(sig_file, 'rb') as file:
        signa = file.read()

    #Use to unpad
    pad = padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=padding.PSS.MAX_LENGTH)

    #Gets the public key
    public_key = serialization.load_pem_public_key(open('ku.pem', 'rb').read(),default_backend())

    #Verify the signature
    try:
        print("----------VALIDATING KEY----------")
        public_key.verify(signature=signa,data=digest,padding=pad, algorithm=utils.Prehashed(myhash))
    except:
        print("Key is invalid!")
    else:
        print("Key is valid!")

    #Now all that is left is to decrypt the message - remember that 'textt' has the contents in bytes, not hex

    decrypted = decrypt(textt)
    print("----------DECRYPTING----------")
    with open('decryptedSignedFile.txt', 'w') as file:
        file.write(decrypted)

    print("Congrats! Your decrypted file is written onto the decryptedSignedFile.txt file!")



if __name__ == "__main__":
    main()
```

Figure 29: Code used for crypt_sign.py - part 3



```
[craig@craig-pc ECELabCrypto]$ python crypt_and_sign.py
Type 'e' to encrypt:    e
----------ENCRYPTING----------
Congrats! What you encrypted is saved onto the fileToSign.txt file!
----------VALIDATING KEY----------
Key is valid!
----------DECRYPTING----------
Congrats! Your decrypted file is written onto the decryptedSignedFile.txt file!
[craig@craig-pc ECELabCrypto]$
```

Figure 30: Output for crypt_sign.py - part 4

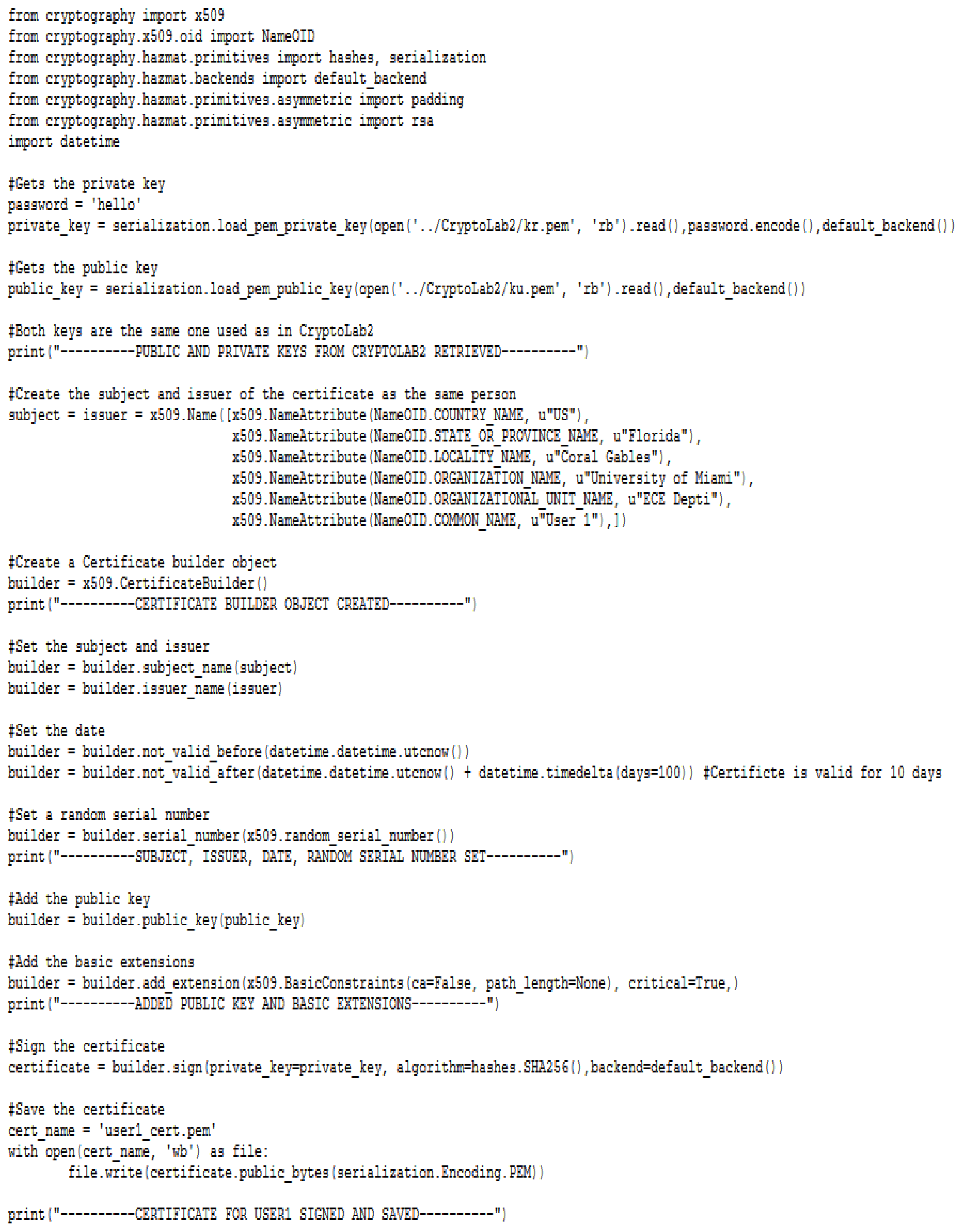
CryptoLab3

0.0.8 Task 1: Certificates

Program: certificates.py

0.0.8.1 Task 1.1: Create a self-signed certificate for User1

For this task, I first had to regenerate the RSA private/public key pair that was done in CryptoLab2. Next, the subject and the issuer of the certificate used was declared as the same person. The certificate builder object was then made, followed by setting the subject and the issuer. In order to set the date, I had to change the wording from the lab manual because I kept getting errors. I make it so that my certificate is valid for 100 days from the run of the program. The random serial number is then set and the public key is added. Simple basic extensions are added before the certificate is signed and saved onto "user1_cert.pem".



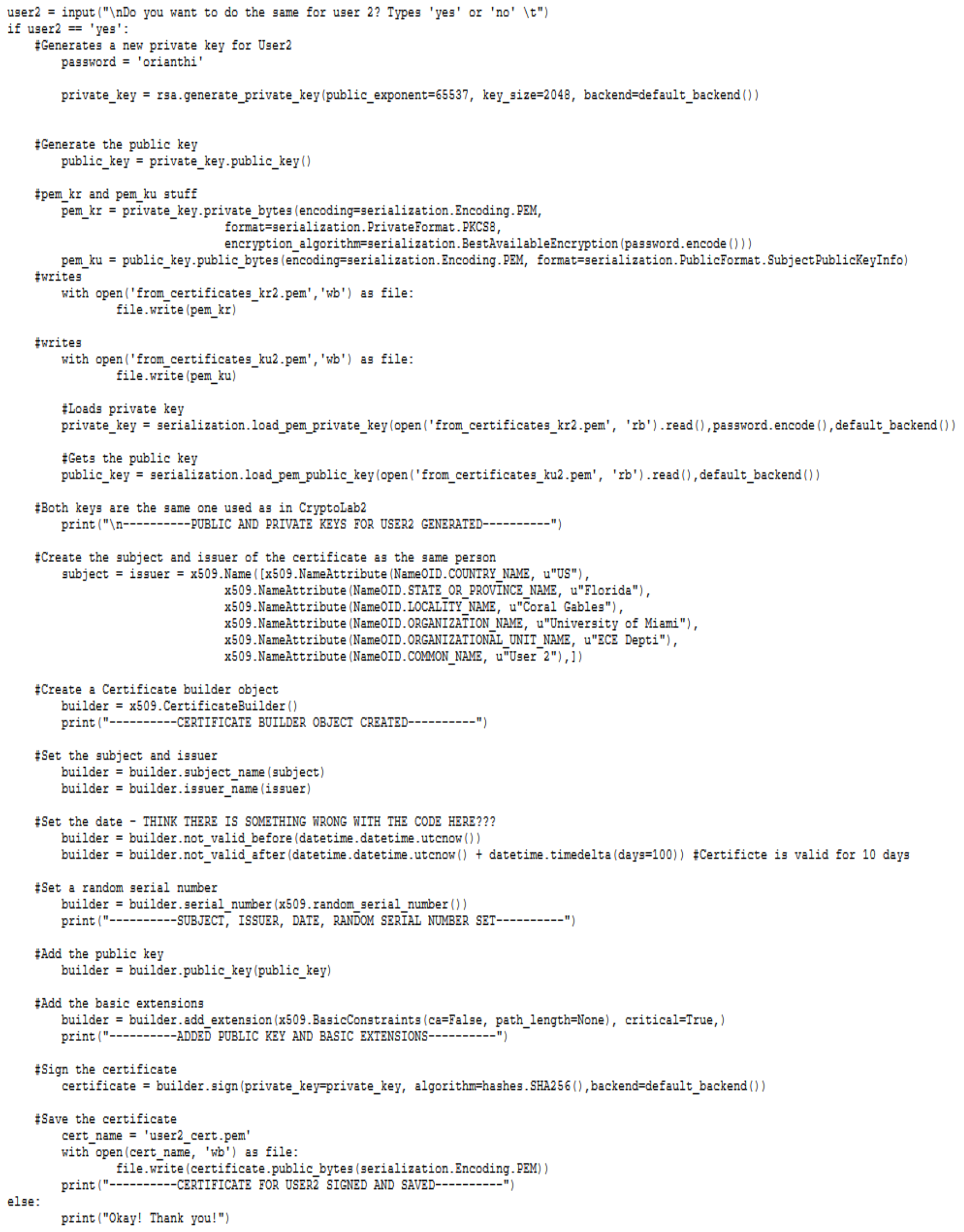
```
from cryptography import x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives.asymmetric import rsa
import datetime

#Gets the private key
password = 'hello'
private_key = serialization.load_pem_private_key(open('../CryptoLab2/kr.pem', 'rb').read(),password.encode(),default_backend())

#Gets the public key
public_key = serialization.load_pem_public_key(open('../CryptoLab2/ku.pem', 'rb').read(),default_backend())

#Both keys are the same one used as in CryptoLab2
print("----------PUBLIC AND PRIVATE KEYS FROM CRYPTOLAB2 RETRIEVED----------")

#Create the subject and issuer of the certificate as the same person
subject = issuer = x509.Name([x509.NameAttribute(NameOID.COUNTRY_NAME, u"US"),
                              x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"Florida"),
                              x509.NameAttribute(NameOID.LOCALITY_NAME, u"Coral Gables"),
                              x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"University of Miami"),
                              x509.NameAttribute(NameOID.ORGANIZATIONAL_UNIT_NAME, u"ECE Depti"),
                              x509.NameAttribute(NameOID.COMMON_NAME, u"User 1"),])

#Create a Certificate builder object
builder = x509.CertificateBuilder()
print("----------CERTIFICATE BUILDER OBJECT CREATED----------")

#Set the subject and issuer
builder = builder.subject_name(subject)
builder = builder.issuer_name(issuer)

#Set the date
builder = builder.not_valid_before(datetime.datetime.utcnow())
builder = builder.not_valid_after(datetime.datetime.utcnow() + datetime.timedelta(days=100)) #Certificte is valid for 10 days

#Set a random serial number
builder = builder.serial_number(x509.random_serial_number())
print("----------SUBJECT, ISSUER, DATE, RANDOM SERIAL NUMBER SET----------")

#Add the public key
builder = builder.public_key(public_key)

#Add the basic extensions
builder = builder.add_extension(x509.BasicConstraints(ca=False, path_length=None), critical=True,)
print("----------ADDED PUBLIC KEY AND BASIC EXTENSIONS----------")

#Sign the certificate
certificate = builder.sign(private_key=private_key, algorithm=hashes.SHA256(),backend=default_backend())

#Save the certificate
cert_name = 'user1_cert.pem'
with open(cert_name, 'wb') as file:
        file.write(certificate.public_bytes(serialization.Encoding.PEM))

print("----------CERTIFICATE FOR USER1 SIGNED AND SAVED----------")
```

Figure 31: Code for creating self-signed certificate for User1

Thus, making a self-signed certificate for User1 was a success.

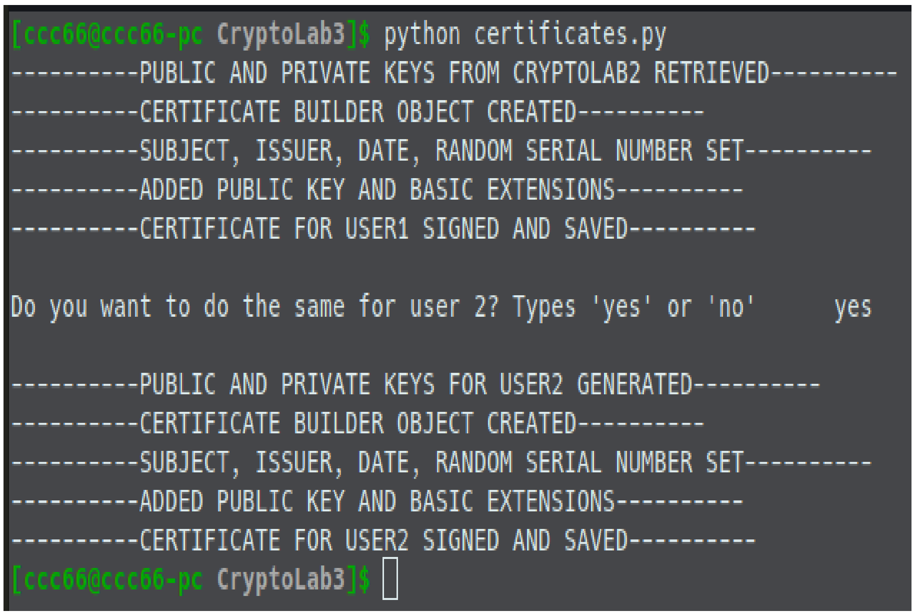
0.0.8.2 Task 1.2: Create a self-signed certificate for User2

The next task involved creating another self-signed certificate, but for a second user. In order to do this, the code is kept exactly the same except for a few differences. The COMMON_NAME was changed to "User 2" and new private and public keys were generated. The new key pair was saved onto "kr2.pem" and "ku2.pem". It is also important to note that task 1.1 and task 1.2 were included in the same program. I make the program ask the user if he/she wants to create another self-signed certificate for User2. All the user has to do is type ’yes’ or ’no’. An output showing the terminal output for both inputs is also shown below.



```
user2 = input("\nDo you want to do the same for user 2? Types 'yes' or 'no' \t")
if user2 == 'yes':
    #Generates a new private key for User2
        password = 'orianthi'

        private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048, backend=default_backend())


    #Generate the public key
        public_key = private_key.public_key()

    #pem_kr and pem_ku stuff
        pem_kr = private_key.private_bytes(encoding=serialization.Encoding.PEM,
                                format=serialization.PrivateFormat.PKCS8,
                                encryption_algorithm=serialization.BestAvailableEncryption(password.encode()))
        pem_ku = public_key.public_bytes(encoding=serialization.Encoding.PEM, format=serialization.PublicFormat.SubjectPublicKeyInfo)
    #writes
        with open('from_certificates_kr2.pem','wb') as file:
                file.write(pem_kr)

    #writes
        with open('from_certificates_ku2.pem','wb') as file:
                file.write(pem_ku)

        #Loads private key
        private_key = serialization.load_pem_private_key(open('from_certificates_kr2.pem', 'rb').read(),password.encode(),default_backend())

        #Gets the public key
        public_key = serialization.load_pem_public_key(open('from_certificates_ku2.pem', 'rb').read(),default_backend())

    #Both keys are the same one used as in CryptoLab2
        print("\n----------PUBLIC AND PRIVATE KEYS FOR USER2 GENERATED----------")

    #Create the subject and issuer of the certificate as the same person
        subject = issuer = x509.Name([x509.NameAttribute(NameOID.COUNTRY_NAME, u"US"),
                                x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"Florida"),
                                x509.NameAttribute(NameOID.LOCALITY_NAME, u"Coral Gables"),
                                x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"University of Miami"),
                                x509.NameAttribute(NameOID.ORGANIZATIONAL_UNIT_NAME, u"ECE Depti"),
                                x509.NameAttribute(NameOID.COMMON_NAME, u"User 2"),])

    #Create a Certificate builder object
        builder = x509.CertificateBuilder()
        print("----------CERTIFICATE BUILDER OBJECT CREATED----------")

    #Set the subject and issuer
        builder = builder.subject_name(subject)
        builder = builder.issuer_name(issuer)

    #Set the date - THINK THERE IS SOMETHING WRONG WITH THE CODE HERE???
        builder = builder.not_valid_before(datetime.datetime.utcnow())
        builder = builder.not_valid_after(datetime.datetime.utcnow() + datetime.timedelta(days=100)) #Certificte is valid for 10 days

    #Set a random serial number
        builder = builder.serial_number(x509.random_serial_number())
        print("----------SUBJECT, ISSUER, DATE, RANDOM SERIAL NUMBER SET----------")

    #Add the public key
        builder = builder.public_key(public_key)

    #Add the basic extensions
        builder = builder.add_extension(x509.BasicConstraints(ca=False, path_length=None), critical=True,)
        print("----------ADDED PUBLIC KEY AND BASIC EXTENSIONS----------")

    #Sign the certificate
        certificate = builder.sign(private_key=private_key, algorithm=hashes.SHA256(),backend=default_backend())

    #Save the certificate
        cert_name = 'user2_cert.pem'
        with open(cert_name, 'wb') as file:
                file.write(certificate.public_bytes(serialization.Encoding.PEM))
        print("----------CERTIFICATE FOR USER2 SIGNED AND SAVED----------")
else:
        print("Okay! Thank you!")
```

Figure 32: Code for creating self-signed certificate for User2



```
[ccc66@ccc66-pc CryptoLab3]$ python certificates.py
----------PUBLIC AND PRIVATE KEYS FROM CRYPTOLAB2 RETRIEVED----------
----------CERTIFICATE BUILDER OBJECT CREATED----------
----------SUBJECT, ISSUER, DATE, RANDOM SERIAL NUMBER SET----------
----------ADDED PUBLIC KEY AND BASIC EXTENSIONS----------
----------CERTIFICATE FOR USER1 SIGNED AND SAVED----------

Do you want to do the same for user 2? Types 'yes' or 'no'      yes

----------PUBLIC AND PRIVATE KEYS FOR USER2 GENERATED----------
----------CERTIFICATE BUILDER OBJECT CREATED----------
----------SUBJECT, ISSUER, DATE, RANDOM SERIAL NUMBER SET----------
----------ADDED PUBLIC KEY AND BASIC EXTENSIONS----------
----------CERTIFICATE FOR USER2 SIGNED AND SAVED----------
[ccc66@ccc66-pc CryptoLab3]$
```

Figure 33: Terminal output for certificates.py

0.0.9 Task 2: Signing and Verifying

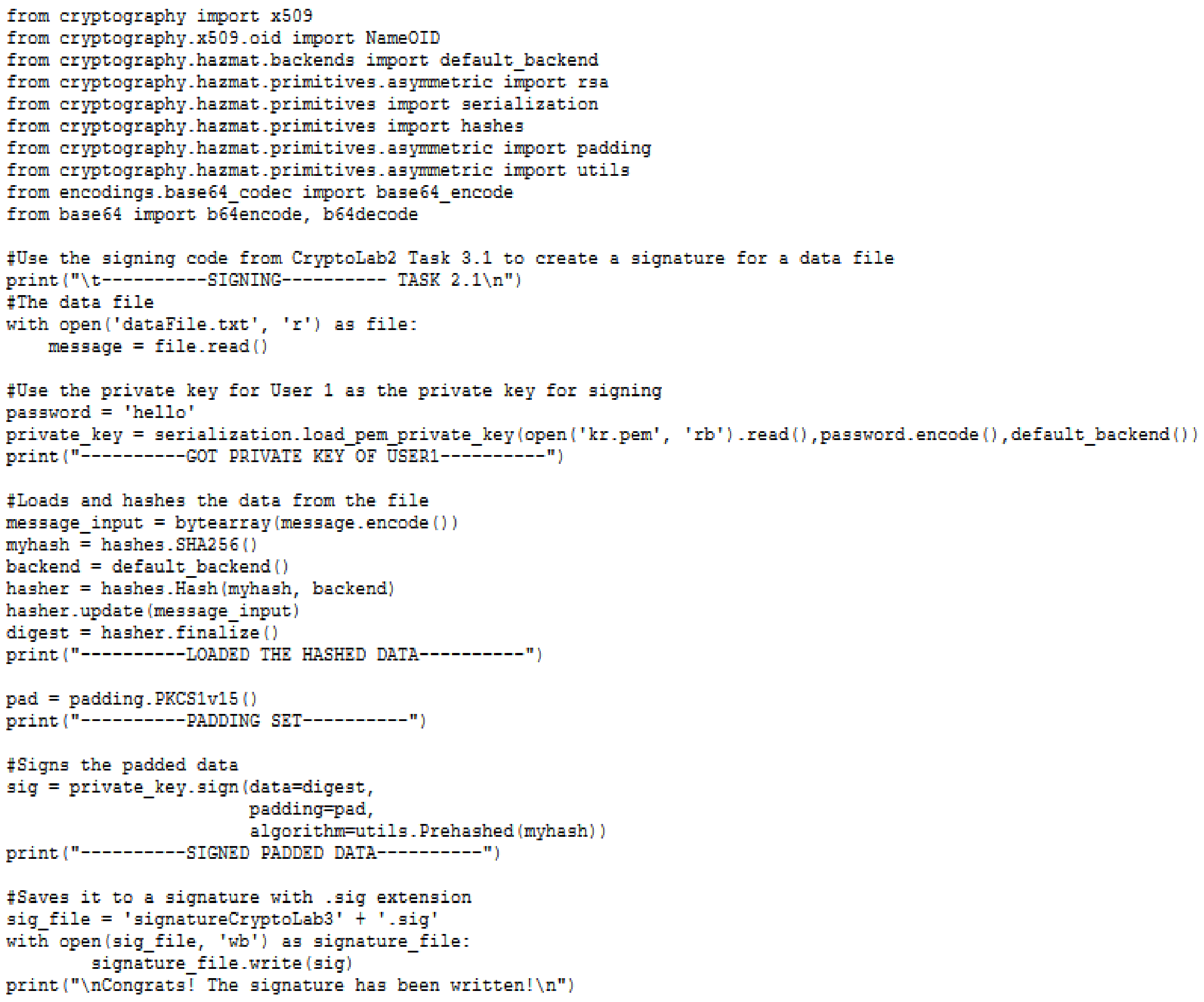
Program: signing_and_verifying.py

For tasks 2.1 - 2.3, I had the codes for each task be written in one program. It made sense to me because it re-uses some variables and their contents. To distinguish between the tasks, I inserted comments and the output indicates which task is which. As a result, the screenshots of the code for each task are continuations of one big program. Lastly, the output for each task is shown in one screenshot.

0.0.9.1 Task 2.1: Signing a file

For this task, I first had to use the code that I used to sign from CryptoLab2 in order to create a signature for the file. The private key for User1 was used as the private key for signing. Next, I had to change the padding to "padding.PKCS1v15()" to simplify future operations.

Thus, the general steps that I took was to load and hash the data from "dataFile.txt". The contents of that file were "Lorem Ipsum" phrases. Note that is used the SHA256 hash. The private key used was the key that was written to 'kr.pem' with its password as 'hello'. Then, the padded data was signed with the key and the signature was saved onto "signatureCryptoLab3.sig". The terminal outputs are shown below. The outputs just indicate that each step was performed successfully.

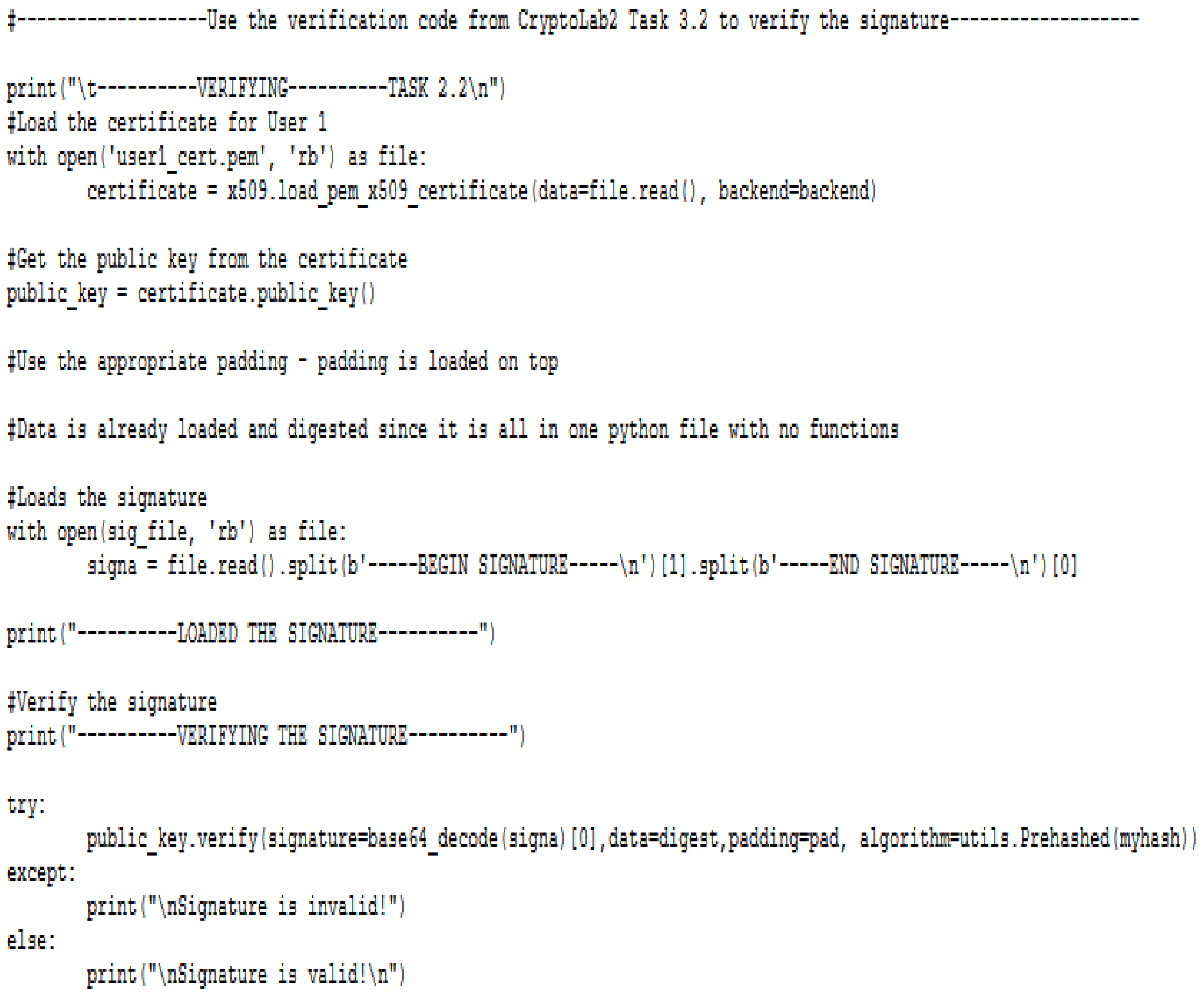


```
from cryptography import x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives.asymmetric import utils
from encodings.base64_codec import base64_encode
from base64 import b64encode, b64decode

#Use the signing code from CryptoLab2 Task 3.1 to create a signature for a data file
print("\t----------SIGNING---------- TASK 2.1\n")
#The data file
with open('dataFile.txt', 'r') as file:
    message = file.read()

#Use the private key for User 1 as the private key for signing
password = 'hello'
private_key = serialization.load_pem_private_key(open('kr.pem', 'rb').read(),password.encode(),default_backend())
print("----------GOT PRIVATE KEY OF USER1----------")

#Loads and hashes the data from the file
message_input = bytearray(message.encode())
myhash = hashes.SHA256()
backend = default_backend()
hasher = hashes.Hash(myhash, backend)
hasher.update(message_input)
digest = hasher.finalize()
print("----------LOADED THE HASHED DATA----------")

pad = padding.PKCS1v15()
print("----------PADDING SET----------")

#Signs the padded data
sig = private_key.sign(data=digest,
                       padding=pad,
                       algorithm=utils.Prehashed(myhash))
print("----------SIGNED PADDED DATA----------")

#Saves it to a signature with .sig extension
sig_file = 'signatureCryptoLab3' + '.sig'
with open(sig_file, 'wb') as signature_file:
        signature_file.write(sig)
print("\nCongrats! The signature has been written!\n")
```

Figure 34: Code for signing a file in task 2.1

0.0.9.2 Task 2.2: Verifying a signature

For this task, I first had to load the certificate for User1 ('user1_cert.pem') and save that onto the "certificate" variable using the x.509_load_pem_x509_certificate function. To get the public key from the certificate, I used "certificate.public_key()". The appropriate padding to be used was already loaded from task 2.1 since its all in the same python file. The same goes for the data to be loaded. The signature from task 2.1 is then loaded and verified with the public key from the certificate by using the public_key.verify() function with a try-except-else statement. The outputs in the terminal just indicate that each step was performed successfully.

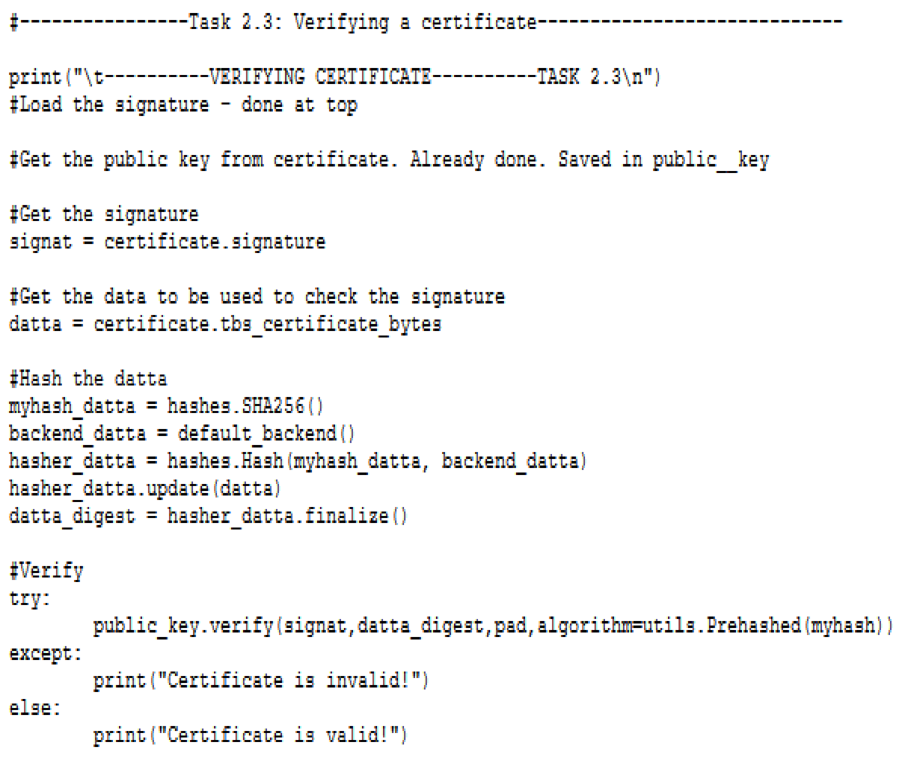


```
#------------------Use the verification code from CryptoLab2 Task 3.2 to verify the signature------------------

print("\t----------VERIFYING----------TASK 2.2\n")
#Load the certificate for User 1
with open('user1_cert.pem', 'rb') as file:
        certificate = x509.load_pem_x509_certificate(data=file.read(), backend=backend)

#Get the public key from the certificate
public_key = certificate.public_key()

#Use the appropriate padding - padding is loaded on top

#Data is already loaded and digested since it is all in one python file with no functions

#Loads the signature
with open(sig_file, 'rb') as file:
        signa = file.read().split(b'-----BEGIN SIGNATURE-----\n')[1].split(b'-----END SIGNATURE-----\n')[0]

print("----------LOADED THE SIGNATURE----------")

#Verify the signature
print("----------VERIFYING THE SIGNATURE----------")

try:
        public_key.verify(signature=base64_decode(signa)[0],data=digest,padding=pad, algorithm=utils.Prehashed(myhash))
except:
        print("\nSignature is invalid!")
else:
        print("\nSignature is valid!\n")
```

Figure 35: Code for verifying a signature in task 2.2

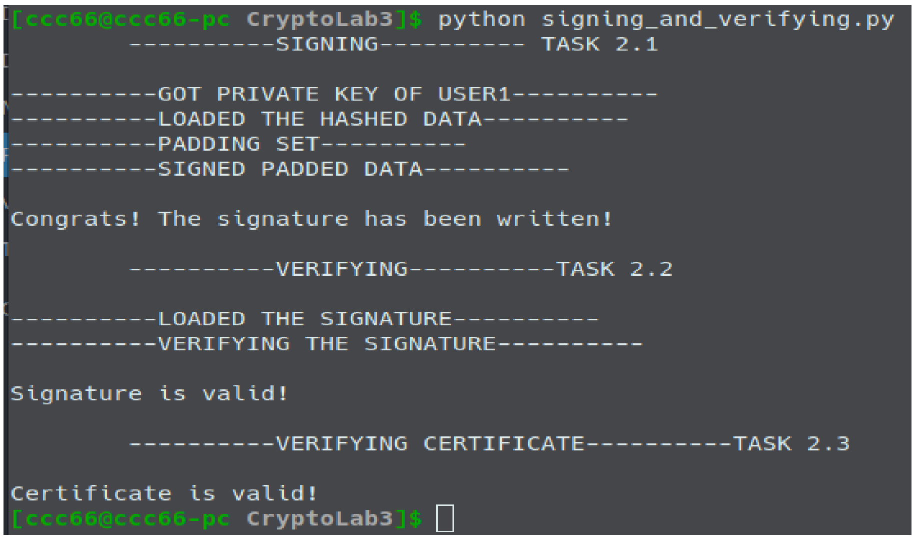
0.0.9.3 Task 2.3: Verifying a certificate

For this task, I had to load the certificate that was written for user1. Then the public key was received from the certificate using the certificate.public_key() function. In addition, the signature was received from the certificate using the certificate.signature function. The data used to check the signature was retrieved and hashed. To verify the signature with the public key, the same try-exception-else statement was used. The code for the task is shown below along with the terminal output.



```
#----------------Task 2.3: Verifying a certificate----------------------------

print("\t----------VERIFYING CERTIFICATE---------TASK 2.3\n")
#Load the signature - done at top

#Get the public key from certificate. Already done. Saved in public__key

#Get the signature
signat = certificate.signature

#Get the data to be used to check the signature
datta = certificate.tbs_certificate_bytes

#Hash the datta
myhash_datta = hashes.SHA256()
backend_datta = default_backend()
hasher_datta = hashes.Hash(myhash_datta, backend_datta)
hasher_datta.update(datta)
datta_digest = hasher_datta.finalize()

#Verify
try:
        public_key.verify(signat,datta_digest,pad,algorithm=utils.Prehashed(myhash))
except:
        print("Certificate is invalid!")
else:
        print("Certificate is valid!")
```

Figure 36: Code for verifying a certificate in task 2.3



```
[ccc66@ccc66-pc CryptoLab3]$ python signing_and_verifying.py
        ----------SIGNING---------- TASK 2.1

----------GOT PRIVATE KEY OF USER1----------
----------LOADED THE HASHED DATA----------
----------PADDING SET----------
----------SIGNED PADDED DATA----------

Congrats! The signature has been written!

        ----------VERIFYING----------TASK 2.2

----------LOADED THE SIGNATURE----------
----------VERIFYING THE SIGNATURE----------

Signature is valid!

        ----------VERIFYING CERTIFICATE----------TASK 2.3

Certificate is valid!
[ccc66@ccc66-pc CryptoLab3]$ 
```

Figure 37: Terminal ouptut for signing_and_verifying.py

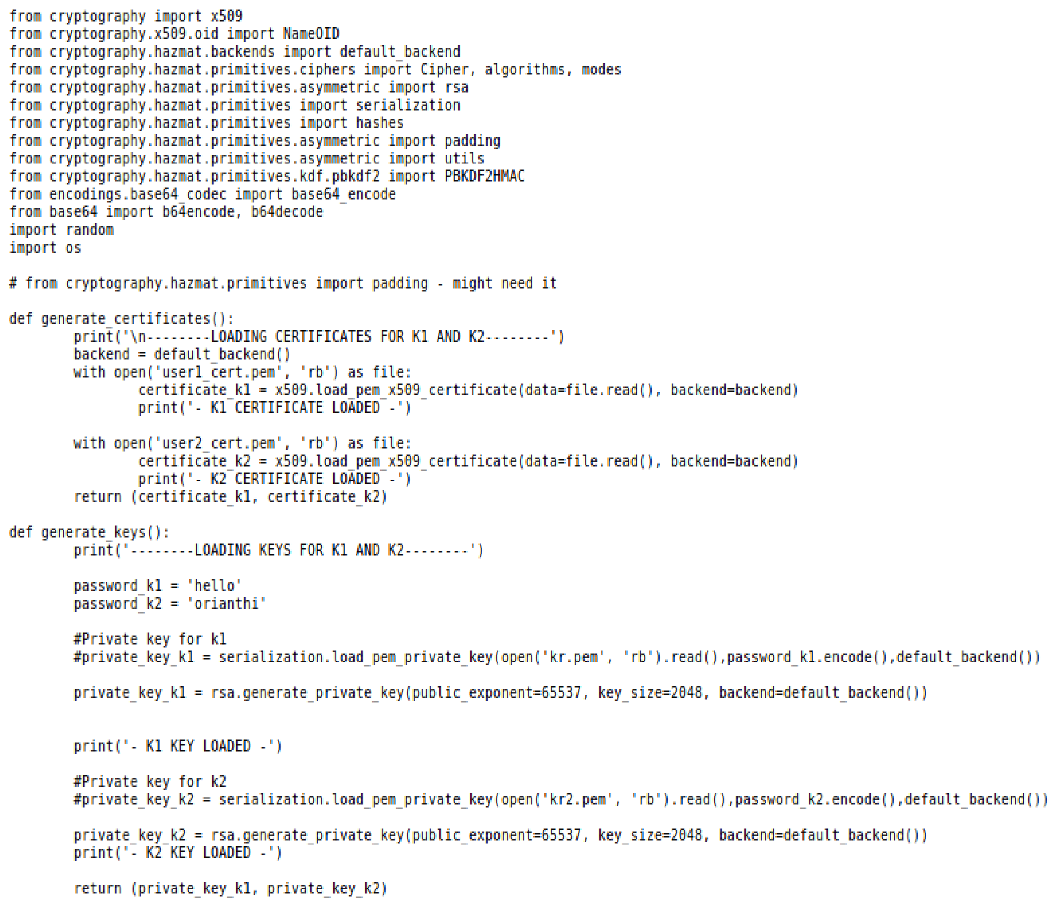
0.0.10 Task 3: Incorporating to create a complete file exchange

Programs: complete_file_exchange.py, encryptionUSER1.py, decryptverify.pyUSER2

0.0.10.1 Steps 1 - 2

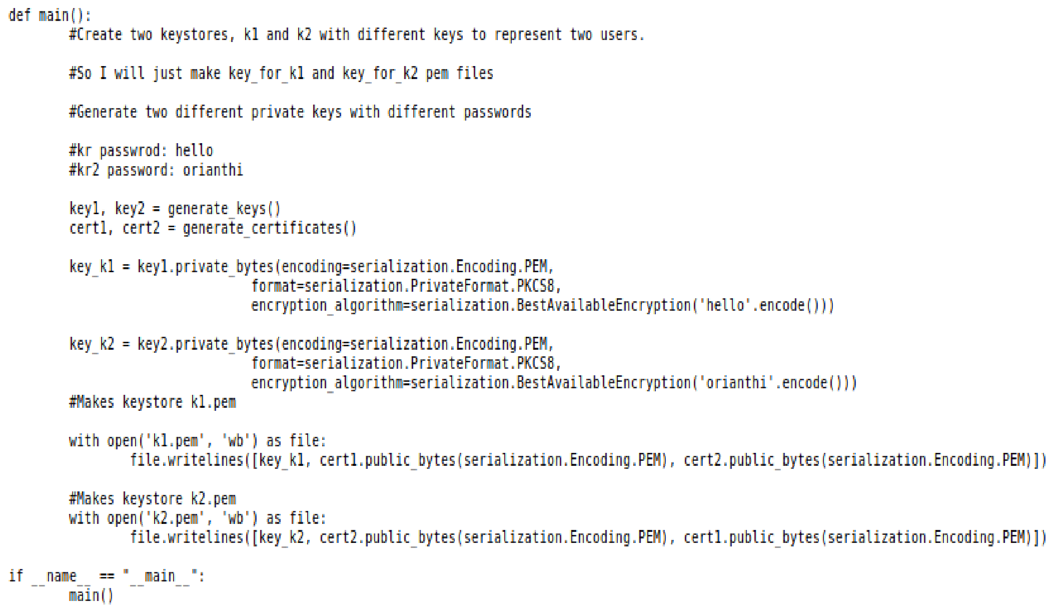
Program: complete_file_exchange.py

The program indicated above was used to complete the first three steps. The task was to create two keystores, k1 and k2 with different keys to represent two users. First, I simply generated two new keys because I did not want to rely on keys used in previous tasks. The password for the first key was 'hello' while the password for the second key was 'orianthi'. The certificates used were the certificates created by 'certificates.py'. They were just loaded. Once I got the certificates for both users and the pair of keys, I was able to create the keystores k1 and k2 by writing them with the writelines() function. So keystore k1 contained the key for user1, the certificate for user1 and the certificate for user2, in that order. Keystore k2 contained the key for user2, certificate for user2 and the certificate for user1, in that order. The code for this program was too big to show in a screenshot. So it was split up into 2.



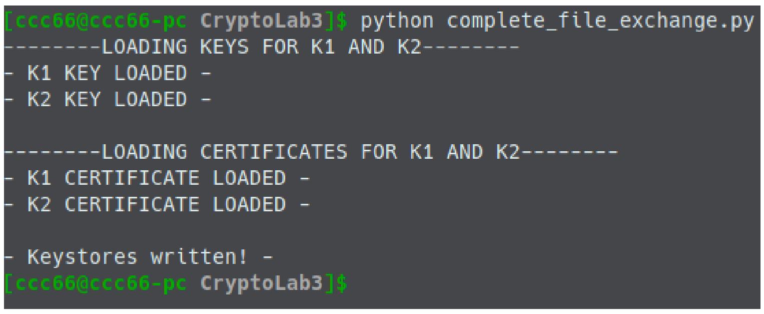
```
from cryptography import x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives.asymmetric import utils
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from encodings.base64_codec import base64_encode
from base64 import b64encode, b64decode
import random
import os

# from cryptography.hazmat.primitives import padding - might need it

def generate_certificates():
        print('\n--------LOADING CERTIFICATES FOR K1 AND K2--------')
        backend = default_backend()
        with open('user1_cert.pem', 'rb') as file:
                certificate_k1 = x509.load_pem_x509_certificate(data=file.read(), backend=backend)
                print('- K1 CERTIFICATE LOADED -')

        with open('user2_cert.pem', 'rb') as file:
                certificate_k2 = x509.load_pem_x509_certificate(data=file.read(), backend=backend)
                print('- K2 CERTIFICATE LOADED -')
        return (certificate_k1, certificate_k2)

def generate_keys():
        print('--------LOADING KEYS FOR K1 AND K2--------')

        password_k1 = 'hello'
        password_k2 = 'orianthi'

        #Private key for k1
        #private_key_k1 = serialization.load_pem_private_key(open('kr.pem', 'rb').read(),password_k1.encode(),default_backend())

        private_key_k1 = rsa.generate_private_key(public_exponent=65537, key_size=2048, backend=default_backend())


        print('- K1 KEY LOADED -')

        #Private key for k2
        #private_key_k2 = serialization.load_pem_private_key(open('kr2.pem', 'rb').read(),password_k2.encode(),default_backend())

        private_key_k2 = rsa.generate_private_key(public_exponent=65537, key_size=2048, backend=default_backend())
        print('- K2 KEY LOADED -')

        return (private_key_k1, private_key_k2)
```

Figure 38: Code for complete_file_exchange.py ($part1$)



```
def main():
        #Create two keystores, k1 and k2 with different keys to represent two users.

        #So I will just make key_for_k1 and key_for_k2 pem files

        #Generate two different private keys with different passwords

        #kr passwrod: hello
        #kr2 password: orianthi

        key1, key2 = generate_keys()
        cert1, cert2 = generate_certificates()

        key_k1 = key1.private_bytes(encoding=serialization.Encoding.PEM,
                                format=serialization.PrivateFormat.PKCS8,
                                encryption_algorithm=serialization.BestAvailableEncryption('hello'.encode()))

        key_k2 = key2.private_bytes(encoding=serialization.Encoding.PEM,
                                format=serialization.PrivateFormat.PKCS8,
                                encryption_algorithm=serialization.BestAvailableEncryption('orianthi'.encode()))
        #Makes keystore k1.pem

        with open('k1.pem', 'wb') as file:
                file.writelines([key_k1, cert1.public_bytes(serialization.Encoding.PEM), cert2.public_bytes(serialization.Encoding.PEM)])

        #Makes keystore k2.pem
        with open('k2.pem', 'wb') as file:
                file.writelines([key_k2, cert2.public_bytes(serialization.Encoding.PEM), cert1.public_bytes(serialization.Encoding.PEM)])

if __name__ == "__main__":
        main()
```

Figure 39: Code for complete_file_exchange.py ($part2$)



```
[ccc66@ccc66-pc CryptoLab3]$ python complete_file_exchange.py
--------LOADING KEYS FOR K1 AND K2--------
- K1 KEY LOADED -
- K2 KEY LOADED -

--------LOADING CERTIFICATES FOR K1 AND K2--------
- K1 CERTIFICATE LOADED -
- K2 CERTIFICATE LOADED -

- Keystores written! -
[ccc66@ccc66-pc CryptoLab3]$
```

Figure 40: Terminal output after running complete_file_exchange.py

0.0.10.2 Step 3 - as user1 in 'user1' folder

Program: encryptionUSER1.py

The first thing I had to do was to create an encrypted file with a randomly generated key. Symmetric encryption, AES-128 needed to be used, so I used the OFB mode of encryption. OFB is symmetric and uses PKCS7 padding, with a given IV. In addition, no padding is required, so I did not need to pad.

First, I had a text file to encrypt that I read and encrypted, writing the ciphertext onto a ciphertext .txt file. Note that the key and IV were both randomly generated. So, in order to save the key and IV for later use, I simply appended the IV THEN the key to the beginning of the ciphertext. That way, once the key and IV are needed for decryption, I can just strip off the first 16 characters to get the key, and the second 16 characters to get the IV.

In fact, the secret key used for the file encryption was needed, so appending the key to the beginning of the file was useful. I made a 'get key used' function that read the first 16 characters of the ciphertext to get the string. Next, the key was stripped off. To get the public key of user2, I read keystore k1 onto a variable. I made a 'split' function that took in that variable, the text/line that I wanted to target, and its occurrence number. Since the certificate of user2 in k1 was the last certificate that was in k1, I called the split function to target the second occurrence of '—–BEGIN CERTIFICATE—–'. The output would be a tuple where the first index is everything before that target and the second index is everything after that target. I just saved the second index since that is the desired result.

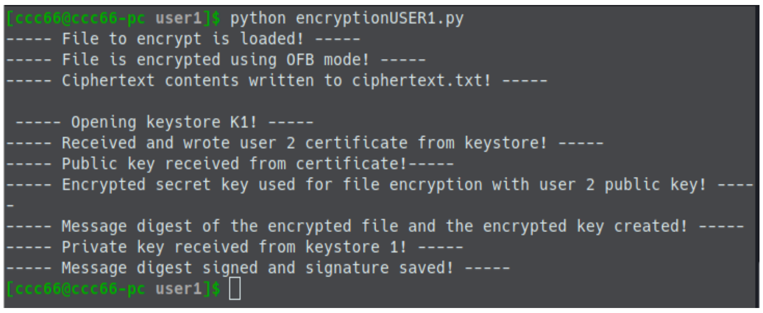
The only downside is that it also deleted that target when it needed to be included. So, I just appended the deleted target to the front and it worked perfectly. Then, I wrote that certificate to a 'user2cert.pem' file so I can retrieve the certificate and get the public key using the certificate.public_key function.

Finally, to encrypt the secret key used for the file encryption with the public key of user 2, I used public_key_user2.encrypt(key, pad) where pad = padding.PKCS1v15(). Then, the encryption of the secret key was saved onto a .pem file.

To create a message digest of the encrypted file and the encrypted key, I just combined the encrypted file with the encrypted key by using the '+' sign. The encrypted key was appended to the end of the encrypted file. Then, as done in the messageDigest.py program, I used the SHA256() digest and made its function.

Next, I had to sign the message digest with the private key of user 1. So, I used the contents of the variable used to load keystore k1 and ran the serialization.load_pem_private_key() function. Since there was only one key in the keystore, the function was able to retrieve the key without any errors. The message digest was then signed using the private key and then saved onto a .pem file. In addition, the signature was also written to a .sig file with PEM format.

Note that the code is also too big to be included in one screenshot. So if you want to see a PDF of the code click here



```
[ccc66@ccc66-pc user1]$ python encryptionUSER1.py
----- File to encrypt is loaded! -----
----- File is encrypted using OFB mode! -----
----- Ciphertext contents written to ciphertext.txt! -----

 ----- Opening keystore K1! -----
----- Received and wrote user 2 certificate from keystore! -----
----- Public key received from certificate!-----
----- Encrypted secret key used for file encryption with user 2 public key! ----
-
----- Message digest of the encrypted file and the encrypted key created! -----
----- Private key received from keystore 1! -----
----- Message digest signed and signature saved! -----
[ccc66@ccc66-pc user1]$ 
```

Figure 41: Terminal output after running encryptionUSER1.py

0.0.10.3 Step 4 - as user2 in 'user2' folder

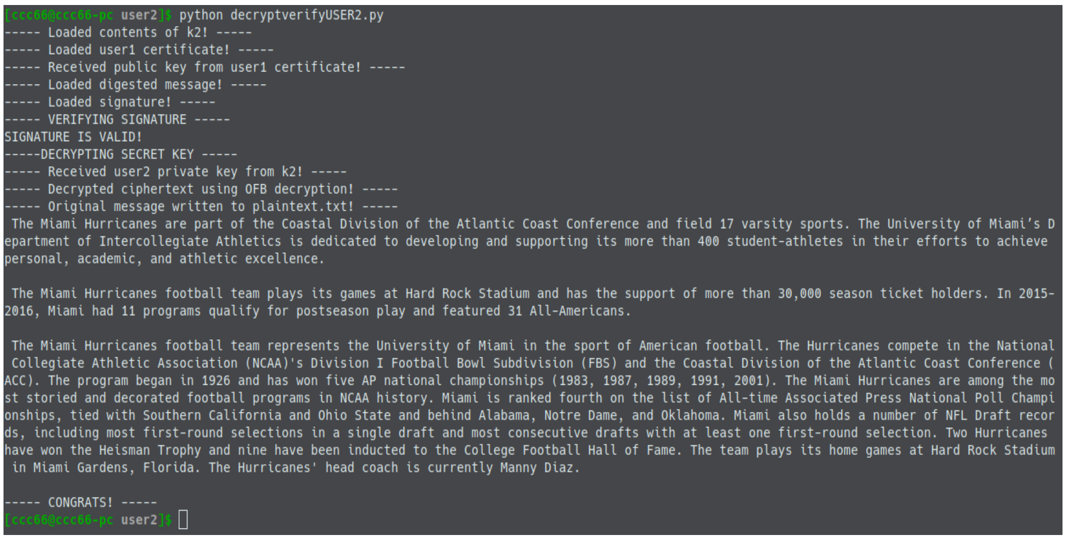
Program: decryptionUSER2.py

What I first had to do was copy all the file I had written to the folder for user2. Then, the task was to verify the signature done on the message digest using the public key for user 1. In order to get the public key, I had to load the certificate for user1 from the keystore k2. This followed the same steps as before. I used the 'split()' function to target the first line of the certificate and receive a tuple. Since the certificate for user1 was the second index of the tuple, I saved only that index and appended the '—–BEGIN CERTIFICATE—–' to the start of it. I saved the result to a user1_cert.pem file to have it. Then, I got the public key from it by running the 'certificate.public_key()' function. With that, I verified the signature done on the digest, which used the try-exception-else statement. The end result was successful and the signature was verified.

Then, I had to decrypt the secret key using the private key for user 2. Since the secret key was encrypted using the public key for user 2, it made sense to decrypt it using the private key. The private key was in the keystore k2. Since it was the only key in the keystore, I simply ran the serialization.load_pem_private_key() and was able to retrieve the key successfully.

To decrypt the ciphertext with the secret key, I had to make the decryptOFB() function with two parameters: the key used and the ciphertext. Note that the ciphertext still includes the IV that was used to encrypt. So the IV had to be stored and removed from the ciphertext. Once that was done, the decryption was able to proceed as normal. The plaintext was written to the 'plaintext.txt' file and was also outputted in the terminal. Lastly, the final output just says 'CONGRATS!' indicating that the last step was a success.

Again, the python code is too big to be included in one snapshot. So to see the PDF of the code click here



```
[ccc66@ccc66-pc user2]$ python decryptverifyUSER2.py
----- Loaded contents of k2! -----
----- Loaded user1 certificate! -----
----- Received public key from user1 certificate! -----
----- Loaded digested message! -----
----- Loaded signature! -----
----- VERIFYING SIGNATURE -----
SIGNATURE IS VALID!
-----DECRYPTING SECRET KEY -----
----- Received user2 private key from k2! -----
----- Decrypted ciphertext using OFB decryption! -----
----- Original message written to plaintext.txt! -----
 The Miami Hurricanes are part of the Coastal Division of the Atlantic Coast Conference and field 17 varsity sports. The University of Miami’s D
epartment of Intercollegiate Athletics is dedicated to developing and supporting its more than 400 student-athletes in their efforts to achieve
personal, academic, and athletic excellence.

 The Miami Hurricanes football team plays its games at Hard Rock Stadium and has the support of more than 30,000 season ticket holders. In 2015-
2016, Miami had 11 programs qualify for postseason play and featured 31 All-Americans.

 The Miami Hurricanes football team represents the University of Miami in the sport of American football. The Hurricanes compete in the National
 Collegiate Athletic Association (NCAA)'s Division I Football Bowl Subdivision (FBS) and the Coastal Division of the Atlantic Coast Conference (
ACC). The program began in 1926 and has won five AP national championships (1983, 1987, 1989, 1991, 2001). The Miami Hurricanes are among the mo
st storied and decorated football programs in NCAA history. Miami is ranked fourth on the list of All-time Associated Press National Poll Champi
onships, tied with Southern California and Ohio State and behind Alabama, Notre Dame, and Oklahoma. Miami also holds a number of NFL Draft recor
ds, including most first-round selections in a single draft and most consecutive drafts with at least one first-round selection. Two Hurricanes
have won the Heisman Trophy and nine have been inducted to the College Football Hall of Fame. The team plays its home games at Hard Rock Stadium
 in Miami Gardens, Florida. The Hurricanes' head coach is currently Manny Diaz.

----- CONGRATS! -----
[ccc66@ccc66-pc user2]$
```

Figure 42: Terminal output after running decryptverifyUSER2.py

Please note that although I copied all the files from user1 to user2, I DO NOT call upon those files when decrypting. This is because every time that the program in user1 is run, all the files change. So instead of copying all the files over every time the user1 program is run, I just reference those files from the user2 program. The task says to copy the files over, which I do, but it seems tedious to have to copy the files every time the program for user1 is run. The program for user2 can be changed to decrypt the files that were copied over easily. It works either way.

Appendix

Please note that the links are only for the programs' codes, NOT the contents of any created/modified .txt or .pem files. To see those, please refer to the Github link provided at the end. The link leads to the repository containing both the codes and the created/modified files.

0.0.10.4 Programs for CryptoLab1

CryptoLab.py
CryptoLab2.py
CryptoLab3.py
encryption_decryption.py

0.0.10.5 Programs for CryptoLab2

messageDigests.py
RSA_keypair.py
DOCSignature.py
DOCSverify.py
crypt_and_sign.py

0.0.10.6 Programs for CryptoLab3

certificates.py
signing_and_verifying.py
complete_file_exchange.py
encryptionUSER1.py
decryptverifyUSER2.py

0.0.10.7 Github Repository

ECELabCrypto