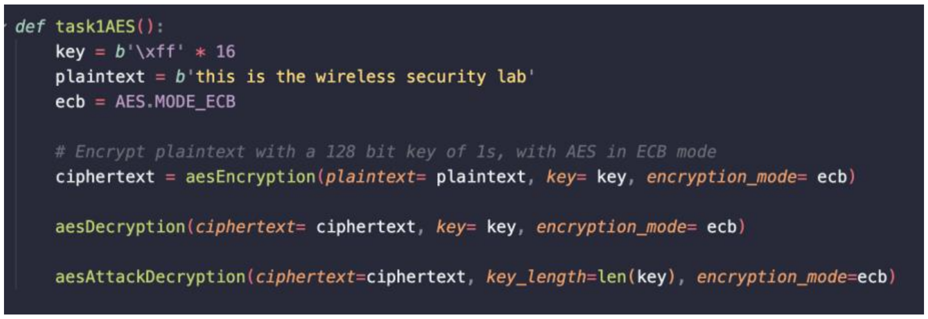
Salvador Gutierrez
CPE 425-01 | Lab 1
Sunday 22, 2023

Task 1

AES Encryption and Decryption Attack

For this part of Task 1, I decided to have a main function call the encryption of a message, “this is the wireless security lab”, with AES on ECB mode. The key needed to be 128-bits long and only 1s, so I represented that by multiply b‘\xff’ sixteen times. To see what the functions aesEncryption, aesDecryption, aesAttackDecryption do please refer to Code Segments 2 and 3.



```
def task1AES():
    key = b'\xff' * 16
    plaintext = b'this is the wireless security lab'
    ecb = AES.MODE_ECB

    # Encrypt plaintext with a 128 bit key of 1s, with AES in ECB mode
    ciphertext = aesEncryption(plaintext= plaintext, key= key, encryption_mode= ecb)

    aesDecryption(ciphertext= ciphertext, key= key, encryption_mode= ecb)

    aesAttackDecryption(ciphertext=ciphertext, key_length=len(key), encryption_mode=ecb)
```

Code Segment 1 - Main Function for AES Encryption, Decryption, & Attack

I decided to use the ECB mode because it is simplest mode on AES. This is because it does not consider the relationship between the plaintext and ciphertext [1]. Therefore, ECB is vulnerable to patterns still being present in the ciphertext. This is best visualized by this image below:

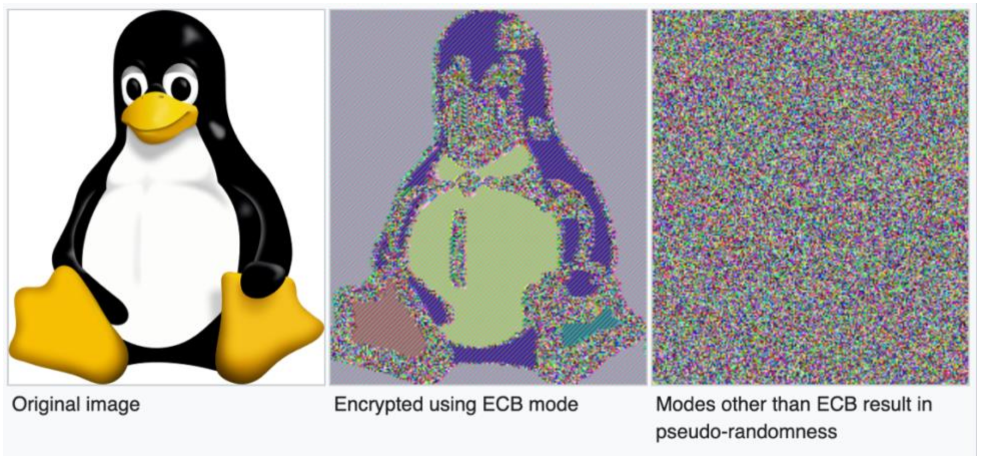
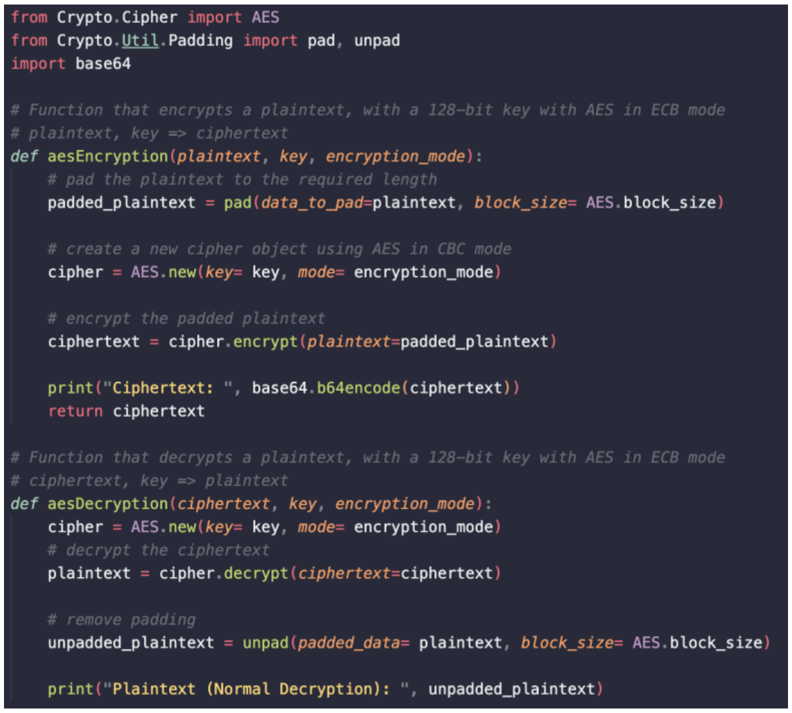


Image 1 - Graphic Example of Vulnerability of ECB [1]

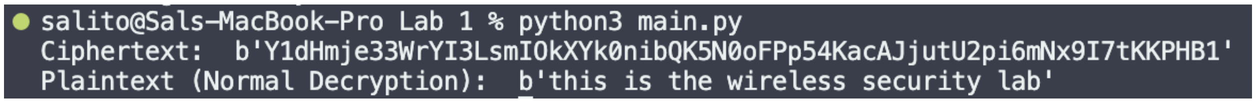
It is also important to note that because I am using ECB, it was necessary to pad the plaintext to fit the right dimensions of the blocks that ECB uses to encrypt. After padding, you then have to encrypt the plaintext. Once you want to decrypt, you should decrypt the ciphertext and then finally, unpad the text so you can make sense of the bytes. All this is shown in code segment 2 below.



```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import base64

# Function that encrypts a plaintext, with a 128-bit key with AES in ECB mode
# plaintext, key => ciphertext
def aesEncryption(plaintext, key, encryption_mode):
    # pad the plaintext to the required length
    padded_plaintext = pad(data_to_pad=plaintext, block_size= AES.block_size)

    # create a new cipher object using AES in CBC mode
    cipher = AES.new(key= key, mode= encryption_mode)

    # encrypt the padded plaintext
    ciphertext = cipher.encrypt(plaintext=padded_plaintext)

    print("Ciphertext: ", base64.b64encode(ciphertext))
    return ciphertext

# Function that decrypts a plaintext, with a 128-bit key with AES in ECB mode
# ciphertext, key => plaintext
def aesDecryption(ciphertext, key, encryption_mode):
    cipher = AES.new(key= key, mode= encryption_mode)
    # decrypt the ciphertext
    plaintext = cipher.decrypt(ciphertext=ciphertext)

    # remove padding
    unpadded_plaintext = unpad(padded_data= plaintext, block_size= AES.block_size)

    print("Plaintext (Normal Decryption): ", unpadded_plaintext)
```

Code Segment 2 - Encryption and Decryption functions

Running the above two functions alone, results in this output:

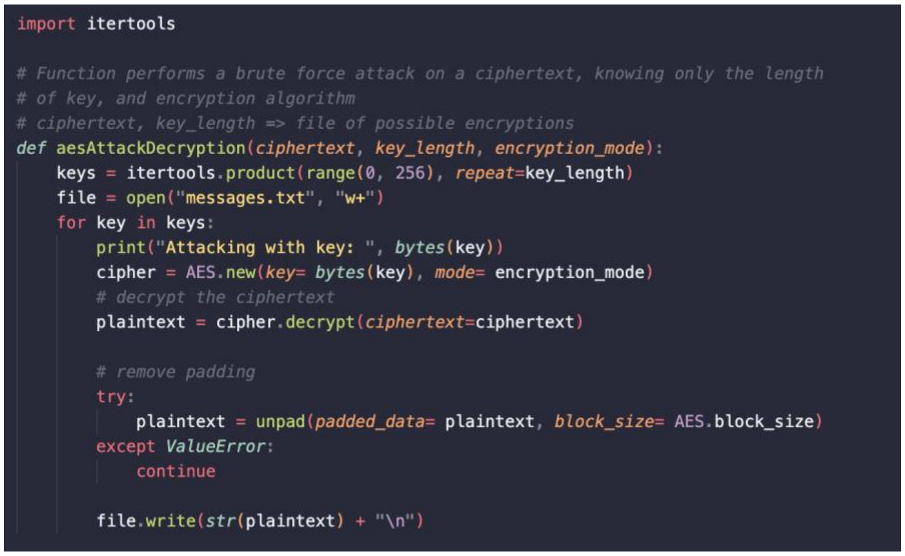


```
● salito@Sals-MacBook-Pro Lab 1 % python3 main.py
  Ciphertext:  b'Y1dHmje33WrYI3LsmIOkXYk0nibQK5N0oFPp54KacAJjutU2pi6mNx9I7tKKPHB1'
  Plaintext (Normal Decryption):  b'this is the wireless security lab'
```

Figure 1- Output from running functions in Code Segment 2

As specified by the instructions of the lab, we are only supposed to assume we know the length of the key, and encryption algorithm. That's why I only included the ciphertext, key_length, and encryption_mode parameters in this function. To crack the ciphertext, I decided to follow the brute force approach. First, I generate all possible keys that we could've used to encrypt the message with. To do this I used the itertools library and their product function. To mimic the way I constructed the original key before, I have a list of integers from 0 to 255, and then this is permutated 16 times, for a total of 16 bytes of integers, that when called with bytes() would result in a key of 128 bits.

After decrypting with the generated key, I want the plaintext to be written into a messages.txt file. I ran into some issues with unpad() as it was giving me some issues with length. So, I decided to not unpad the new plaintext if there was a ValueError and then add the padded or unpadded plaintext to the file. You can see this below in Code Segment 3.

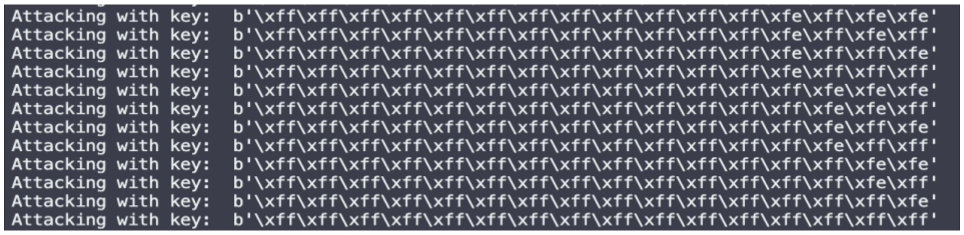


```python
import itertools

# Function performs a brute force attack on a ciphertext, knowing only the length
# of key, and encryption algorithm
# ciphertext, key_length => file of possible encryptions
def aesAttackDecryption(ciphertext, key_length, encryption_mode):
    keys = itertools.product(range(0, 256), repeat=key_length)
    file = open("messages.txt", "w+")
    for key in keys:
        print("Attacking with key: ", bytes(key))
        cipher = AES.new(key= bytes(key), mode= encryption_mode)
        # decrypt the ciphertext
        plaintext = cipher.decrypt(ciphertext=ciphertext)

        # remove padding
        try:
            plaintext = unpad(padded_data= plaintext, block_size= AES.block_size)
        except ValueError:
            continue

        file.write(str(plaintext) + "\n")
```

Code Segment 3 - Attack Function for Decrypting a ciphertext

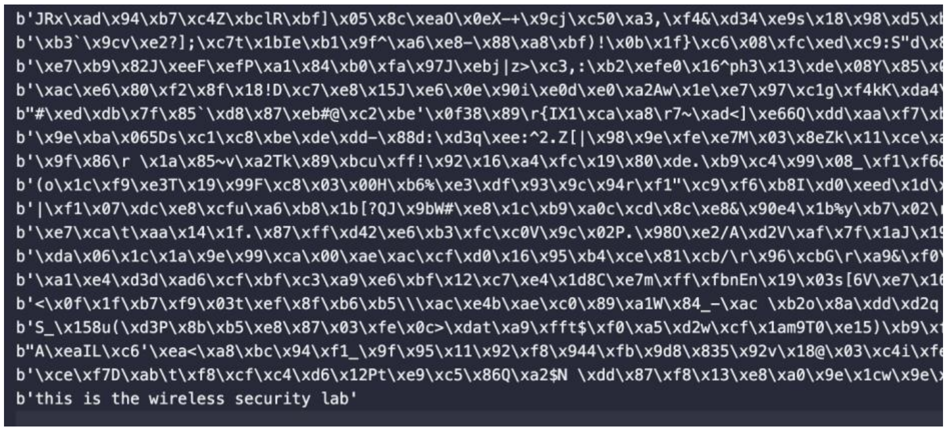
Given the long nature of the key used to encrypt the data, and that I am using the brute force approach as well, there will be 2^128 possible keys. This would take an insane amount of time and given the time constrains for this lab, and the computational limitations of my computer, running this from range(0, 256) would be next to impossible.

However, I can speed up the time it would take to brute force the decryption by changing the parameters of the range to range(254, 256). However, here I am assuming that the key was not found in any combinations before. Changing the code to reflect that new range results in this output:



```
Attacking with key:  b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xfe\xff\xfe\xfe'
Attacking with key:  b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xfe\xff\xfe\xff'
Attacking with key:  b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xfe\xff\xff\xfe'
Attacking with key:  b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xfe\xff\xff\xff'
Attacking with key:  b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xfe\xfe\xfe'
Attacking with key:  b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xfe\xfe\xff'
Attacking with key:  b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xfe\xff\xfe'
Attacking with key:  b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xfe\xff\xff'
Attacking with key:  b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xfe\xfe'
Attacking with key:  b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xfe\xff'
Attacking with key:  b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xfe'
Attacking with key:  b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff'
```

Figure 2 - Terminal output, Note the last line is the actual key we used to encrypt the message



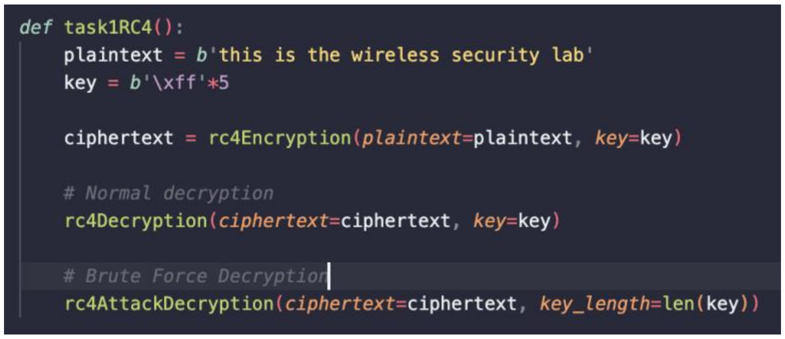
```
b'JRx\xad\x94\xb7\xc4Z\xbclR\xbf]\x05\x8c\xea0\x0eX-+\x9cj\xc50\xa3,\xf4&\xd34\xe9s\x18\x98\xd5\x
b'\xb3`\x9cv\xe2?];\xc7t\x1bIe\xb1\x9f^\xa6\xe8-\x88\xa8\xbf)!\x0b\x1f}\xc6\x08\xfc\xed\xc9:S"d\x
b'\xe7\xb9\x82J\xeeF\xefP\xa1\x84\xb0\xfa\x97J\xebj|z>\xc3,:\xb2\xefe0\x16^ph3\x13\xde\x08Y\x85\x
b'\xac\xe6\x80\xf2\x8f\x18!D\xc7\xe8\x15J\xe6\x0e\x90i\xe0d\xe0\xa2Aw\x1e\xe7\x97\xc1g\xf4kK\xda4
b"#\xed\xdb\x7f\x85`\xd8\x87\xeb#@\xc2\xbe'\x0f38\x89\r{IX1\xca\xa8\r7~\xad<]\xe66Q\xdd\xaa\xf7\x
b'\x9e\xba\x065Ds\xc1\xc8\xbe\xde\xdd-\x88d:\xd3q\xee:^2.Z[|\x98\x9e\xfe\xe7M\x03\x8eZk\x11\xce\x
b'\x9f\x86\r \x1a\x85~v\xa2Tk\x89\xbcu\xff!\x92\x16\xa4\xfc\x19\x80\xde.\xb9\xc4\x99\x08_\xf1\xf6
b'(o\x1c\xf9\xe3T\x19\x99F\xc8\x03\x00H\xb6%\xe3\xdf\x93\x9c\x94r\xf1"\xc9\xf6\xb8I\xd0\xeed\x1d\
b'|\xf1\x07\xdc\xe8\xcfu\xa6\xb8\x1b[?QJ\x9bW#\xe8\x1c\xb9\xa0c\xcd\x8c\xe8&\x90e4\x1b%y\xb7\x02\
b'\xe7\xca\t\xaa\x14\x1f.\x87\xff\xd42\xe6\xb3\xfc\xc0V\x9c\x02P.\x98O\xe2/A\xd2V\xaf\x7f\x1aJ\x1
b'\xda\x06\x1c\x1a\x9e\x99\xca\x00\xae\xac\xcf\xd0\x16\x95\xb4\xce\x81\xcb/\r\x96\xcbG\r\xa9&\xf0
b'\xa1\xe4\xd3d\xad6\xcf\xbf\xc3\xa9\xe6\xbf\x12\xc7\xe4\x1d8C\xe7m\xff\xfbnEn\x19\x03s[6V\xe7\x1
b'<\x0f\x1f\xb7\xf9\x03t\xef\x8f\xb6\xb5\\\xac\xe4b\xae\xc0\x89\xa1W\x84_-\xac \xb2o\x8a\xdd\xd2q
b'S_\x158u(\xd3P\x8b\xb5\xe8\x87\x03\xfe\x0c>\xdat\xa9\xfft$\xf0\xa5\xd2w\xcf\x1am9T0\xe15)\xb9\x
b"A\xeaIL\xc6'\xea<\xa8\xbc\x94\xf1_\x9f\x95\x11\x92\xf8\x944\xfb\x9d8\x835\x92v\x18@\x03\xc4i\xf
b'\xce\xf7D\xab\t\xf8\xcf\xc4\xd6\x12Pt\xe9\xc5\x86Q\xa2$N \xdd\x87\xf8\x13\xe8\xa0\x9e\x1cw\x9e\
b'this is the wireless security lab'
```

Figure 3 - messages.txt after running with range(254, 256)

You could assume that my brute force algorithm and key generation would've worked if given the time to generate the keys since it generated the original key used to encrypt the data b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff'.

RC4 Encryption and Decryption Attack

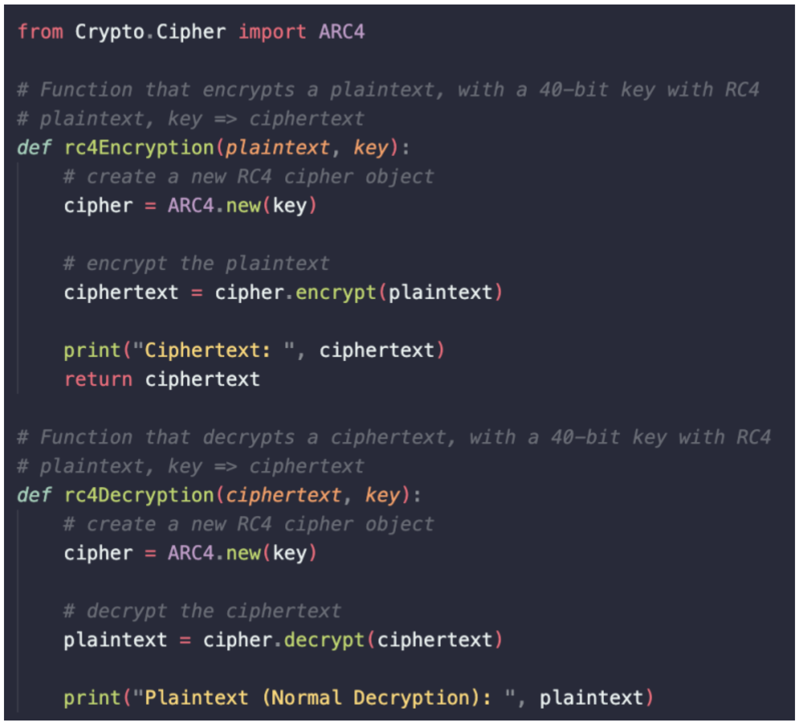
In similar fashion to Code Segment 1, I've created a task1RC4 function that handles the function calls for this part of the lab. Notice that this time, because the key is a 40-bit key of 1s, I've represented it with 5 'x\ff' bytes. No mode has to be declared for RC4, so the only required variables for this algorithm are plaintext and key.



```python
def task1RC4():
    plaintext = b'this is the wireless security lab'
    key = b'\xff'*5

    ciphertext = rc4Encryption(plaintext=plaintext, key=key)

    # Normal decryption
    rc4Decryption(ciphertext=ciphertext, key=key)

    # Brute Force Decryption
    rc4AttackDecryption(ciphertext=ciphertext, key_length=len(key))
```

Code Segment 4 - Setup and Function Calls for RC4 Encryption and Decryption.

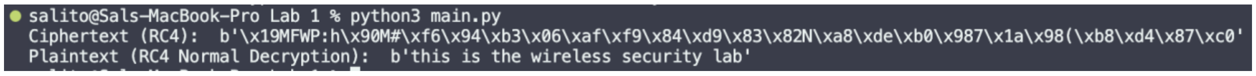
Please refer to Code Segments 5, and 6 for a detailed description of these functions.



```python
from Crypto.Cipher import ARC4

# Function that encrypts a plaintext, with a 40-bit key with RC4
# plaintext, key => ciphertext
def rc4Encryption(plaintext, key):
    # create a new RC4 cipher object
    cipher = ARC4.new(key)

    # encrypt the plaintext
    ciphertext = cipher.encrypt(plaintext)

    print("Ciphertext: ", ciphertext)
    return ciphertext

# Function that decrypts a ciphertext, with a 40-bit key with RC4
# plaintext, key => ciphertext
def rc4Decryption(ciphertext, key):
    # create a new RC4 cipher object
    cipher = ARC4.new(key)

    # decrypt the ciphertext
    plaintext = cipher.decrypt(ciphertext)

    print("Plaintext (Normal Decryption): ", plaintext)
```

Code Segment 5 - Encryption and Decryption in RC4 with the PyCryptoDome library

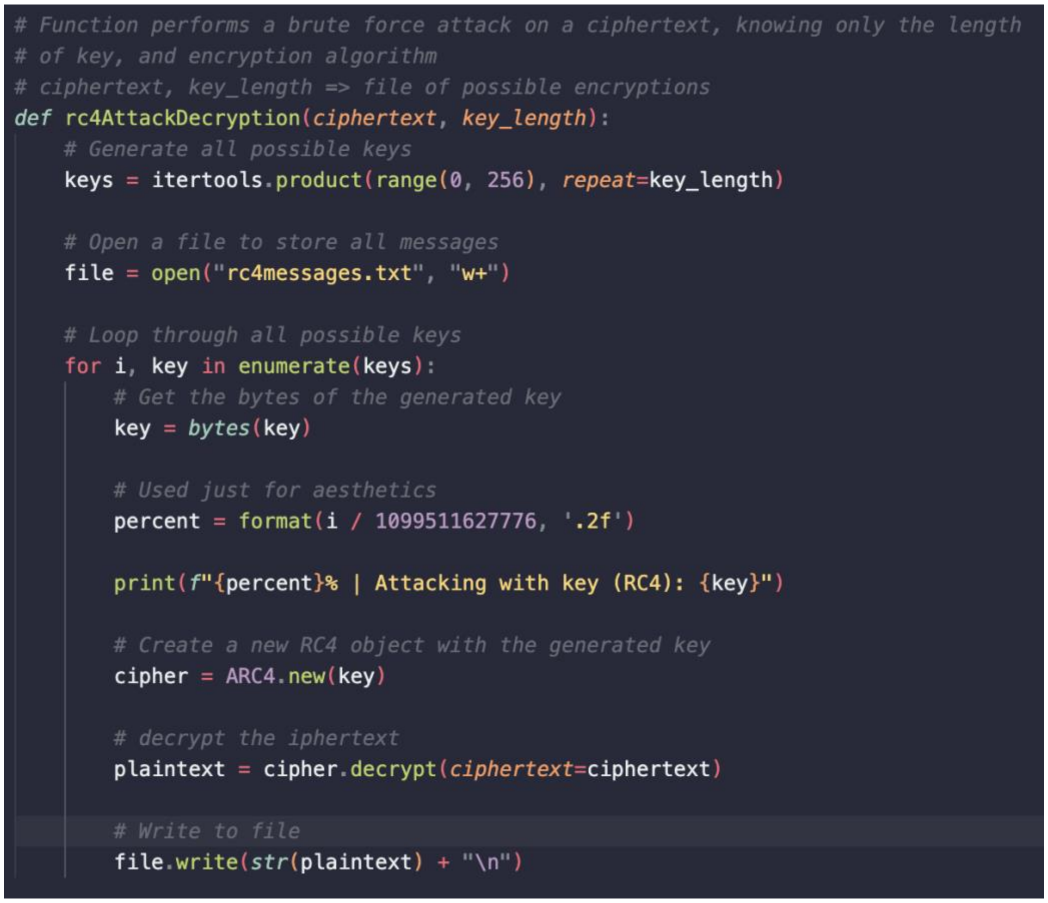
As seen in the Code Segments above, RC4's code is very similar to AES. However, the main differences are that the key can be of any size between 5 to 256 bits [2] and that there is no need to pad the plaintext before or after encrypting. If we run these functions together we get the output below:



```
● salito@Sals-MacBook-Pro Lab 1 % python3 main.py
  Ciphertext (RC4):  b'\x19MFWP:h\x90M#\xf6\x94\xb3\x06\xaf\xf9\x84\xd9\x83\x82N\xa8\xde\xb0\x987\x1a\x98(\xb8\xd4\x87\xc0'
  Plaintext (RC4 Normal Decryption):  b'this is the wireless security lab'
```

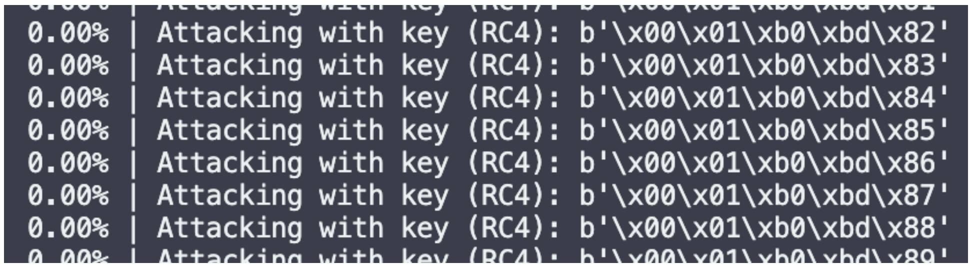
Figure 4 - Output from rc4Encryption(), and rc4Decryption()

Now, since the key we are working with is only of 40 bits of length, brute forcing will be a much easier and faster task than cracking AES. Below is the code for performing the attack:



```python
# Function performs a brute force attack on a ciphertext, knowing only the length
# of key, and encryption algorithm
# ciphertext, key_length => file of possible encryptions
def rc4AttackDecryption(ciphertext, key_length):
    # Generate all possible keys
    keys = itertools.product(range(0, 256), repeat=key_length)

    # Open a file to store all messages
    file = open("rc4messages.txt", "w+")

    # Loop through all possible keys
    for i, key in enumerate(keys):
        # Get the bytes of the generated key
        key = bytes(key)

        # Used just for aesthetics
        percent = format(i / 1099511627776, '.2f')

        print(f"{percent}% | Attacking with key (RC4): {key}")

        # Create a new RC4 object with the generated key
        cipher = ARC4.new(key)

        # decrypt the iphertext
        plaintext = cipher.decrypt(ciphertext=ciphertext)

        # Write to file
        file.write(str(plaintext) + "\n")
```

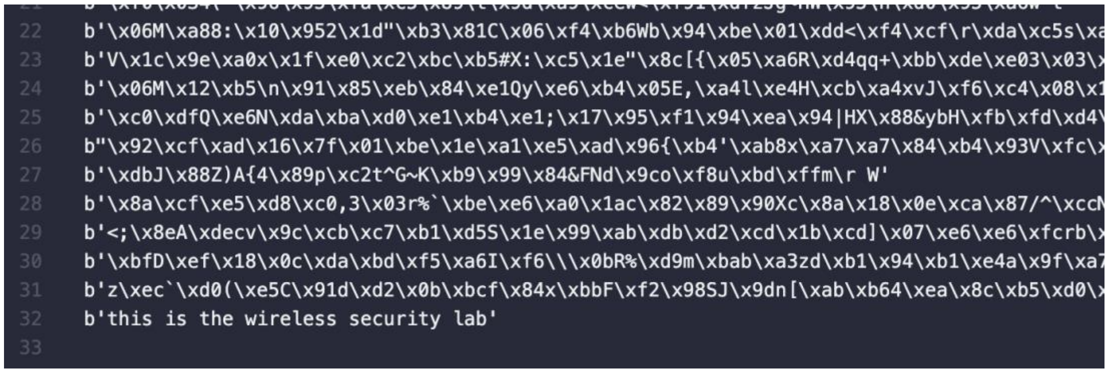
Code Segment 6 - Brute Force Attack on RC4 encrypted ciphertext



```
0.00% | Attacking with key (RC4): b'\x00\x01\xb0\xbd\x82'
0.00% | Attacking with key (RC4): b'\x00\x01\xb0\xbd\x83'
0.00% | Attacking with key (RC4): b'\x00\x01\xb0\xbd\x84'
0.00% | Attacking with key (RC4): b'\x00\x01\xb0\xbd\x85'
0.00% | Attacking with key (RC4): b'\x00\x01\xb0\xbd\x86'
0.00% | Attacking with key (RC4): b'\x00\x01\xb0\xbd\x87'
0.00% | Attacking with key (RC4): b'\x00\x01\xb0\xbd\x88'
0.00% | Attacking with key (RC4): b'\x00\x01\xb0\xbd\x89'
```

Figure 5 - Output from running Code Segment 6

Although the key size was significantly reduced to only 40 bits, it would still take a long time to execute the code in its entirety. This is because there are 2 ^ 40 = 1,099,511,627,776 possible keys. Thus, I did a similar thing to range as I did to the AES part of task 1. Changing my range from 0 to 256, to 254 to 256, my code produced the following rc4Messages.txt file:



```
b'\x06M\xa88:\x10\x952\x1d"\xb3\x81C\x06\xf4\xb6Wb\x94\xbe\x01\xdd<\xf4\xcf\r\xda\xc5s\xa
b'V\x1c\x9e\xa0x\x1f\xe0\xc2\xbc\xb5#X:\xc5\x1e"\x8c[{\x05\xa6R\xd4qq+\xbb\xde\xe03\x03\x
b'\x06M\x12\xb5\n\x91\x85\xeb\x84\xe1Qy\xe6\xb4\x05E,\xa4l\xe4H\xcb\xa4xvJ\xf6\xc4\x08\x1
b'\xc0\xdfQ\xe6N\xda\xba\xd0\xe1\xb4\xe1;\x17\x95\xf1\x94\xea\x94|HX\x88&ybH\xfb\xfd\xd4\
b"\x92\xcf\xad\x16\x7f\x01\xbe\x1e\xa1\xe5\xad\x96{\xb4'\xab8x\xa7\xa7\x84\xb4\x93V\xfc\x
b'\xdbJ\x88Z)A{4\x89p\xc2t^G~K\xb9\x99\x84&FNd\x9co\xf8u\xbd\xffm\r W'
b'\x8a\xcf\xe5\xd8\xc0,3\x03r%`\xbe\xe6\xa0\x1ac\x82\x89\x90Xc\x8a\x18\x0e\xca\x87/^\xccN
b'<;\x8eA\xdecv\x9c\xcb\xc7\xb1\xd5S\x1e\x99\xab\xdb\xd2\xcd\x1b\xcd]\x07\xe6\xe6\xfcrb\x
b'\xbfD\xef\x18\x0c\xda\xbd\xf5\xa6I\xf6\\\x0bR%\xd9m\xbab\xa3zd\xb1\x94\xb1\xe4a\x9f\xa7
b'z\xec`\xd0(\xe5C\x91d\xd2\x0b\xbcf\x84x\xbbF\xf2\x98SJ\x9dn[\xab\xb64\xea\x8c\xb5\xd0\x
b'this is the wireless security lab'
```

Figure 6 - rc4messages.txt file after running rc4AttackDecryption with range(254, 256)

As you can see, it is possible to brute force this algorithm. However, due to the long execution time, it is not a viable solution. Even if you thread the attack, it will still take a long time to execute. If we had more information about the key then this attack would be easier to perform.

Task 2

For task 2 of this lab, I will only be providing minimal Code Segments and include primarily terminal and file screenshots to keep the document short. The code to everything will be found in my GitHub[1].

Since for the task, we are working with 5 different AES modes, I first randomly generate a 128-bit key, a 128-bit initialization vector (iv), a 64-bit nonce (used for CTR mode), and a predetermined plaintext "My Name is Sal, nice to meet you :) ! Im almost done with lab!".

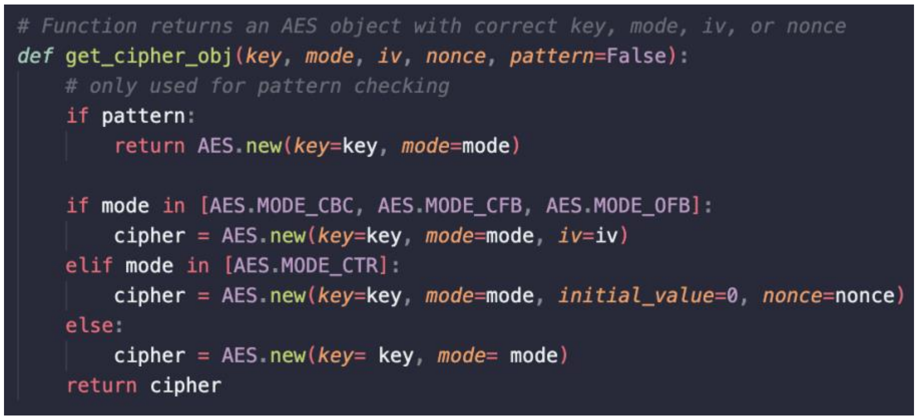
The way I approached this task is as follows:

1. Have a list of all AES modes to test
2. For each mode
    a. Test for Pattern Preservation
    b. Test for Error Propagation
    c. Couple the results and print them out to the console

Pattern Preservation and Error Propagation were tested by calling two different functions and passing in the plaintext, key, mode, iv, and nonce. Also, my encrypt and decrypt functions are very similar to Code Segment 2, but with little modifications to handle the padding for ECB, CBC, and CFB modes.

It is also noteworthy that I have a separate function get_cipher_obj(key, mode, iv, nonce), it creates an AES object based on the parameters passed. See below

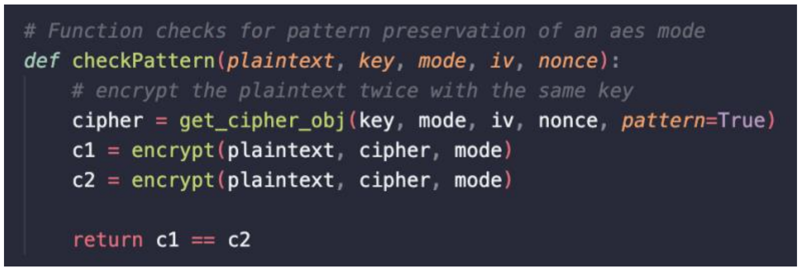
[1] GitHub link: https://github.com/SalGuti0608/CPE-425/tree/main/Lab%201



```python
# Function returns an AES object with correct key, mode, iv, or nonce
def get_cipher_obj(key, mode, iv, nonce, pattern=False):
    # only used for pattern checking
    if pattern:
        return AES.new(key=key, mode=mode)

    if mode in [AES.MODE_CBC, AES.MODE_CFB, AES.MODE_OFB]:
        cipher = AES.new(key=key, mode=mode, iv=iv)
    elif mode in [AES.MODE_CTR]:
        cipher = AES.new(key=key, mode=mode, initial_value=0, nonce=nonce)
    else:
        cipher = AES.new(key= key, mode= mode)
    return cipher
```

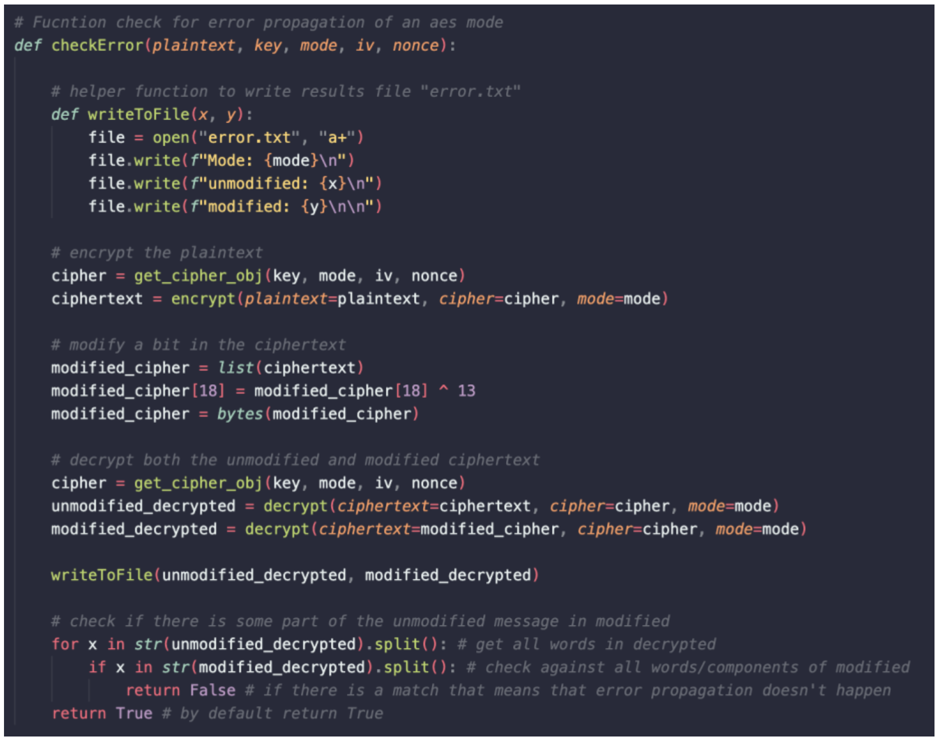
Code Segment 7 - Helper function that returns an AES object based on AES mode being used

This helps lower the amount of code I needed to write for this lab. Below is the code for my Pattern checking and Error checking functions:



```python
# Function checks for pattern preservation of an aes mode
def checkPattern(plaintext, key, mode, iv, nonce):
    # encrypt the plaintext twice with the same key
    cipher = get_cipher_obj(key, mode, iv, nonce, pattern=True)
    c1 = encrypt(plaintext, cipher, mode)
    c2 = encrypt(plaintext, cipher, mode)

    return c1 == c2
```

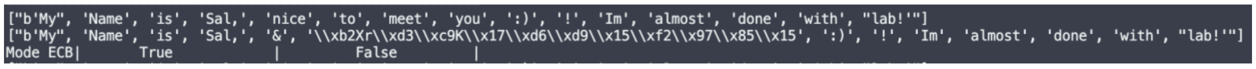
Code Segment 8 - Code that checks for Pattern Preservation of an encryption



```python
# Fucntion check for error propagation of an aes mode
def checkError(plaintext, key, mode, iv, nonce):

    # helper function to write results file "error.txt"
    def writeToFile(x, y):
        file = open("error.txt", "a+")
        file.write(f"Mode: {mode}\n")
        file.write(f"unmodified: {x}\n")
        file.write(f"modified: {y}\n\n")

    # encrypt the plaintext
    cipher = get_cipher_obj(key, mode, iv, nonce)
    ciphertext = encrypt(plaintext=plaintext, cipher=cipher, mode=mode)

    # modify a bit in the ciphertext
    modified_cipher = list(ciphertext)
    modified_cipher[18] = modified_cipher[18] ^ 13
    modified_cipher = bytes(modified_cipher)

    # decrypt both the unmodified and modified ciphertext
    cipher = get_cipher_obj(key, mode, iv, nonce)
    unmodified_decrypted = decrypt(ciphertext=ciphertext, cipher=cipher, mode=mode)
    modified_decrypted = decrypt(ciphertext=modified_cipher, cipher=cipher, mode=mode)

    writeToFile(unmodified_decrypted, modified_decrypted)

    # check if there is some part of the unmodified message in modified
    for x in str(unmodified_decrypted).split(): # get all words in decrypted
        if x in str(modified_decrypted).split(): # check against all words/components of modified
            return False # if there is a match that means that error propagation doesn't happen
    return True # by default return True
```

Code Segment 9 - Code that checks for Error Propagation in encryptions and decryptions

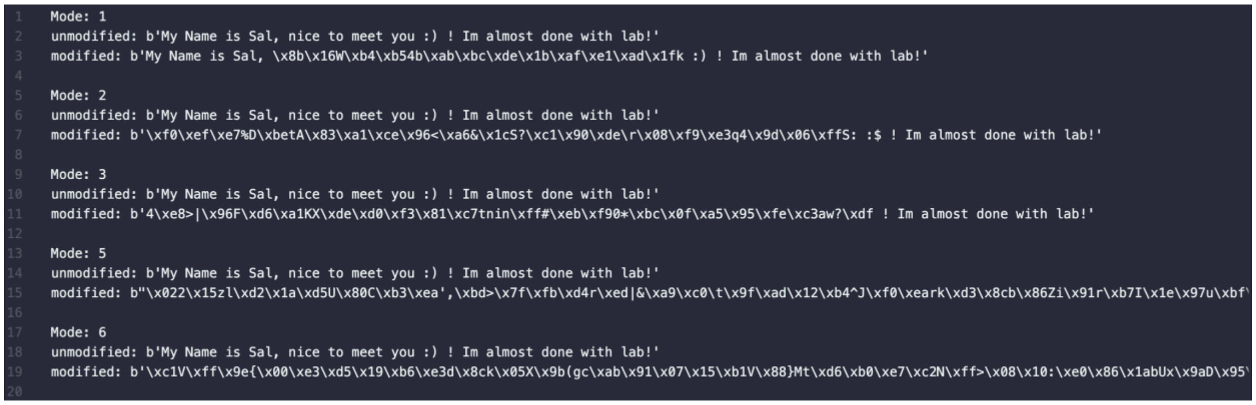
When checking for Error Propagation I am doing so by making a list of elements by splitting the string value of the unmodified and modified decrypted ciphertexts. Then I check if any of the elements in unmodified_decrypted are in modified_decrypted. If true, then that means there is no error propagation on at least one block of the ciphertext. If false, then there was error propagation along all the blocks of the ciphertext. To help visualize these lists of elements, I have printed the corresponding lists for when ECB is being tested. This can be seen in figure 7 below.



```
["b'My", 'Name', 'is', 'Sal,', 'nice', 'to', 'meet', 'you', ':)', '!', 'Im', 'almost', 'done', 'with', "lab!'"]
["b'My", 'Name', 'is', 'Sal,', '&', '\\xb2Xr\\xd3\\xc9K\\x17\\xd6\\xd9\\x15\\xf2\\x97\\x85\\x15', ':)', '!', 'Im', 'almost', 'done', 'with', "lab!'"]
Mode ECB|       True            |          False          |
```

*Figure 7 - Printed **str(unmodified_decrypted).split()** and **str(modified_decrypted).split()** respectively*

Furthermore, as an extra effort to double check my findings for error propagation, I have outputted the unmodified and modified decrypted ciphertexts in a file called errors.txt. You can see them below in figure 8.

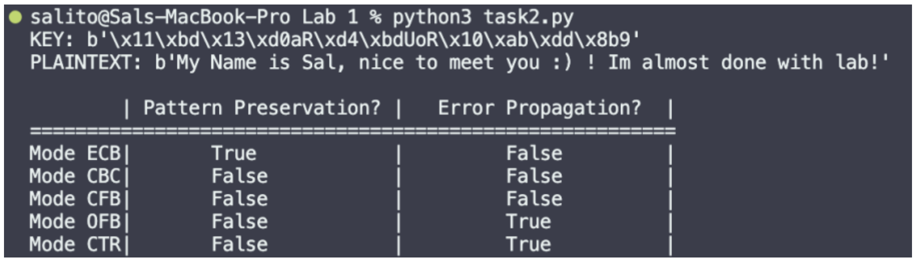


```
Mode: 1
unmodified: b'My Name is Sal, nice to meet you :) ! Im almost done with lab!'
modified: b'My Name is Sal, \x8b\x16W\xb4\xb54b\xab\xbc\xde\x1b\xaf\xe1\xad\x1fk :) ! Im almost done with lab!'

Mode: 2
unmodified: b'My Name is Sal, nice to meet you :) ! Im almost done with lab!'
modified: b'\xf0\xef\xe7%D\xbetA\x83\xa1\xce\x96<\xa6&\x1cS?\xc1\x90\xde\r\x08\xf9\xe3q4\x9d\x06\xffS: :$ ! Im almost done with lab!'

Mode: 3
unmodified: b'My Name is Sal, nice to meet you :) ! Im almost done with lab!'
modified: b'4\xe8>|\x96F\xd6\xa1KX\xde\xd0\xf3\x81\xc7tnin\xff#\xeb\xf90*\xbc\x0f\xa5\x95\xfe\xc3aw?\xdf ! Im almost done with lab!'

Mode: 5
unmodified: b'My Name is Sal, nice to meet you :) ! Im almost done with lab!'
modified: b"\x022\x15zl\xd2\x1a\xd5U\x80C\xb3\xea',\xbd>\x7f\xfb\xd4r\xed|&\xa9\xc0\t\x9f\xad\x12\xb4^J\xf0\xeark\xd3\x8cb\x86Zi\x91r\xb7I\x1e\x97u\xbf

Mode: 6
unmodified: b'My Name is Sal, nice to meet you :) ! Im almost done with lab!'
modified: b'\xc1V\xff\x9e{\x00\xe3\xd5\x19\xb6\xe3d\x8ck\x05X\x9b(gc\xab\x91\x07\x15\xb1V\x88}Mt\xd6\xb0\xe7\xc2N\xff>\x08\x10:\xe0\x86\x1abUx\x9aD\x95
```

Figure 8 - errors.txt file after running task2.py

I observed that for modes 1, 2, and 3, or modes ECB, CBC, and CFB respectively, error propagation only happened in the blocks where the byte was changed. The rest of the blocks were unaffected. For modes 5 and 6, or OFB and CTR, respectively, all the blocks resulted in total gibberish, proving that there was error propagation when a byte was modified. It could be said that for CBC and CFB error propagation does exist but only for the block where the byte was modified. For ECB, error propagation is more interesting as it is limited to after the byte was modified. For example, you can still read "My name is Sal, "but "nice to meet you :)" was lost.

Finally, when you run the code: python3 task2.py, the following output will be populated to the console.



```
salito@Sals-MacBook-Pro Lab 1 % python3 task2.py
KEY: b'\x11\xbd\x13\xd0aR\xd4\xbdUoR\x10\xab\xdd\x8b9'
PLAINTEXT: b'My Name is Sal, nice to meet you :) ! Im almost done with lab!'

        | Pattern Preservation? |   Error Propagation?  |
=========================================================
Mode ECB|       True            |          False        |
Mode CBC|       False           |          False        |
Mode CFB|       False           |          False        |
Mode OFB|       False           |          True         |
Mode CTR|       False           |          True         |
```

Figure 9 - Output for task2.py

In a similar fashion, below is the required table with the information for task 2:

	ECB	CBC	CFB	OFB	CTR
Pattern Preservation	Yes	No	No	No	No
Error Propagation	No	No	No	Yes	Yes

Questions

1. What are the advantages and disadvantages of AES and RC4?

	AES	RC4
Advantage	- Supports 128-, 192-, and 256-bit keys - Larger block size so you can encrypt more data - Symmetric Key Encryption - Brute-Force, Linear, and Differential attack resistant	- Faster algorithm, allows for less capable systems to encrypt data - Smaller key requirements so it might be easier to implement
Disadvantages	- Slightly slower than RC4 - Every block is encrypted the same way - Harder to implement in a large number of languages	- Considered obsolete - Authentication is not possible [3] - Less secure than AES

2. Which one should you choose for file encryption? Why?

   In my opinion, you should use AES for your encryptions. This is because it counts with various modes that fit different specifications and use cases. It is also very well researched, compared to RC4, and is the standard for encryption in the industry. Because of this, there is a lot more documentation and libraries that allow you to implement AES in a larger set of languages.

   AES also allows you to do very strong encryptions with longer keys that might take years for a computer to decrypt.

3. Discuss your experience during the crack implementations.

   At first, I was tempted to reverse engineer their algorithm. However, I soon realized that this wouldn't be possible because the key information we have is very limited. I then proceeded with a brute force approach. I knew that this would take a lot of time, so I implemented my code so that I would be able to modify key generation to be a little faster and test if my algorithm would eventually work. This proved to be a good thing as brute-forcing both AES and RC4 would be next to impossible, and I would already have kids and a house when the algorithm was done (maybe I would be 6-feet under the ground by then).

There must exist a better way to crack these algorithms like pattern analysis, and other threading the process. Regardless, the experience was interesting and also fun but I do wish I had to time implement a better algorithm and analysis to maybe possible cracking RC4.

4. Discuss the features of different mode of operations and their applications.

	Features	Applications
ECB	- Simpler mode of AES - Does not mix around the plaintext before encrypting (does not diffuse) - Each block is encrypted independently from the others - Vulnerable to a block repetition attack	Used for smaller amounts of data where security isn't a real concern
CBC	- Diffuses by mixing plaintext blocks before encrypting - Encrypts each plaintext by XORing it with a previous ciphertext block (this is before encryption happens) - Therefore an Initialization vector is needed, to encrypt the first block of plaintext	Used in networks, disk encryption, and document exchange over the internet.
CFB	- An initialization vector is needed - The IV is XORed with the first block of plaintext to generate a keystream - The rest of the plaintext is encrypted in a stream with the keystream in <u>blocks</u> - Sometimes accompanied by a message authentication code	Used for encrypting real time data streams. For example, secure communications
OFP	- An initialization vector is needed - The IV is XORed with the first block of plaintext to generate a keystream - The rest of the plaintext is encrypted in a stream with the keystream <u>one bit at a time</u>	Used for encrypting real time data streams. For example, secure communications. Similar to CFB but might be more helpful with smaller data.
CTR	- Requires a nonce (never repeated value), and a counter to generate a keystream. - Its maintains integrity and confidentiality as it prevents unauthorized modifications to the plaintext	Used when confidentiality and integrity are needed. It is used to encrypt documents, secure communications, and hard disks.