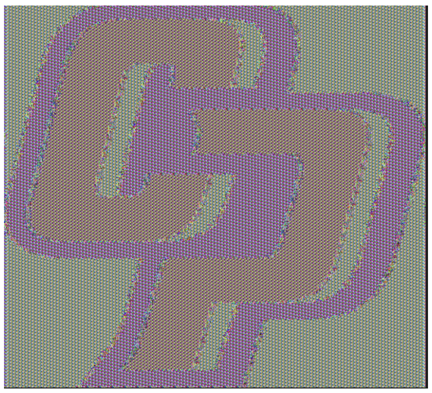
CSC 321, Lab 2
Hank Tsai

Task I:

What I did:
I utilized Crypto.Cipher library's AES to encrypt a Cal Poly Logo image.
For ecb mode, I initialized an AES with its mode set to ECB. I read and stored the first 54 bits as header. Then I read 16 bits at a time because that is the size of a block cipher. Encrypt each block using aes.encrypt method. For the last chunk, I padded it with \x00. While encrypting I wrote bytes to an output file called ecb.bmp
For cbc mode, I also initialized an AES with its mode set to ECB. Most of the operations are similar to ecb mode, except we used an IV (Initialization Vector). Performing IV XOR first block and take the resulting cipher to XOR with the next block. Handle padding and write bytes to cbc.bmp
What I observed in the resulting ciphertext is that they are in bytes and very random. I was able to derive information from the ECB image. I was able to see the original CP logo with its color being modified, which means the information of the original image leaked in the ECB mode. The cause was the same plaintext is always generated to the same ciphertext, which is very predictable. However, ECB mode's output image is very random because its ciphertext always changes in each block by using IV and ciphertext from the last block.

```
#CSC 321, Lab 2 (part 1), Hank Tsai
#Modes of Operation

from hashlib import new
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

def cbc_encrypt():
    key = get_random_bytes(16)
    IV = get_random_bytes(16)
    cbc = open("cbc.bmp", "wb")
    #change cbc to ecb
    aes = AES.new(key, AES.MODE_ECB)
    with open("cp-logo.bmp", "rb") as image_file:
        prev = IV
        header = image_file.read(54)
        cbc.write(header)
        body = image_file.read(16)
        while body:
            if len(body) < 16:
                body = body + b"\x00"*(16-len(body)%16)
            xoring = bytes([a ^ b for a, b in zip(body, prev)])
            new_data = aes.encrypt(xoring)
            prev = new_data
            cbc.write(new_data)
            print(new_data)
            body = image_file.read(16)
    cbc.close()

def ecb_encrypt():
    key = get_random_bytes(16)
    ecb = open("ecb.bmp", "wb")
    aes = AES.new(key, AES.MODE_ECB)
    with open("cp-logo.bmp", "rb") as image_file:
        header = image_file.read(54)
        ecb.write(header)
        body = image_file.read(16)
        while body:
            if len(body) < 16:
                body = body + b"\x00"*(16-len(body)%16)
            new_data = aes.encrypt(body)
            print(new_data)
            # ecb.write(new_data)
            body = image_file.read(16)
    ecb.close()

def main():
    ecb_encrypt()
    cbc_encrypt()


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



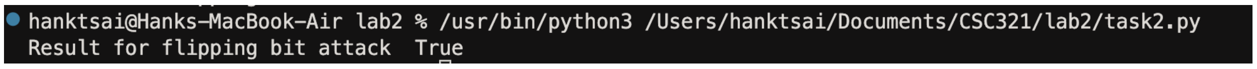
Task II:

What I did:
Generated random key and IV, and passed user input to the submit() method. In submit(), I url encoded ; and =, concatenated user input, padded the string, encrypted it, bit flipped some of the bits and passed back the cipher text (or attacked cipher text). And we passed this cipher text to the verify() method. Verify() method decrypted it, unpadded it, and checked if ';admin=true;' is in the string (return true if it is). I basically used 'Hank Tsai p2 AadminAtrueA' as a user input string, and since I knew the position for those 3 A's, I flipped these 3 bits by calculating the ASCII difference between current character and desired character, then passed the attacked ciphertext to the verify() test. A very interesting thing to observe is that if we flip a bit in mi, other bits in the mi block will get scrambled (turns to some random bits). So this block became 'garbage' as we called it in class.

This attack was possible because if we know the pattern of important information (such as setting admin to true in our case), the position of those targeted bits, and the mechanism of CBC decryption (the last ciphertext will be used in the decryption of the next ciphertext in XOR operation), an attacker can alter bits to desired bit and cause final change in the decrypted plaintext. Essentially this attack was possible because the integrity of the data is compromised . Therefore, this scheme would need an authentication method to authenticate and protect the integrity of the data. It is mentioned online that a MAC (message authentication code) could be used in this case. And below is my code for lab2 task II.

```
#CSC 321, Lab 2 (part 2), Hank Tsai
#Modes of Operation

from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

def modify_bits(cipherArr):
    cipherArr[17] = cipherArr[17] ^ 122
    cipherArr[23] = cipherArr[23] ^ 124
    cipherArr[28] = cipherArr[28] ^ 122
    return bytes(cipherArr)

def submit(input, key, IV):
    input = input.replace(';', '%3B')
    input = input.replace('=', '%3D')
    modified = "userid=456;userdata=" + input + ";session-id=31337"
    padded = pad(modified.encode('utf-8'), AES.block_size)
    aes = AES.new(key, AES.MODE_CBC, IV)
    ciphertext = aes.encrypt(padded)
    attacked = modify_bits(list(ciphertext))
    return attacked

def verify(string, key, IV):
    aes = AES.new(key, AES.MODE_CBC, IV)
    decrypted = aes.decrypt(string)
    unpadded = unpad(decrypted, AES.block_size)
    return ';admin=true;' in str(unpadded)

def main():
    key = get_random_bytes(16)
    IV = get_random_bytes(16)
    text = submit('Hank Tsai p2 AadminAtrueA', key, IV)
    result = verify(text, key, IV)
    print('Result for flipping bit attack ', result)

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

Program output:



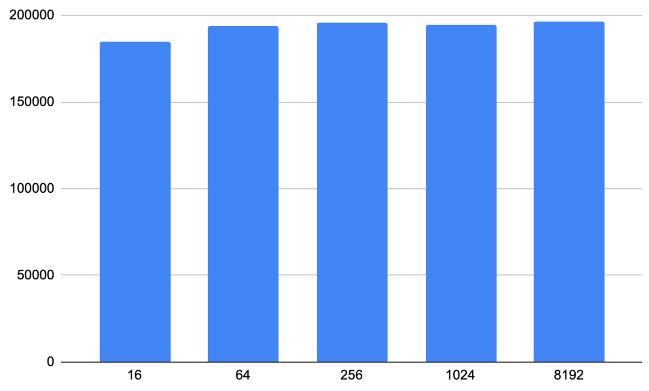
```
hanktsai@Hanks-MacBook-Air lab2 % /usr/bin/python3 /Users/hanktsai/Documents/CSC321/lab2/task2.py
Result for flipping bit attack  True
```

Task III

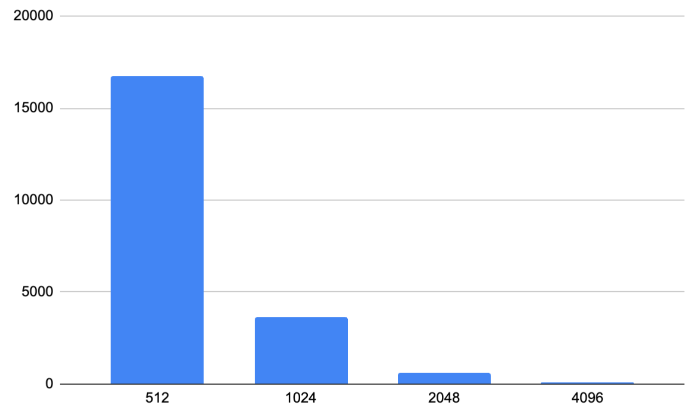
What I did

I ran those two commands given in the assignment in the terminal. And input needed data into spreadsheet and generated graphs for both operations.

block size vs. throughput for the various AES modes of operations



RSA key size vs. throughput for each RSA operation



Those results are pretty interesting to see. So basically for AES modes of operations, the size of the block does not really affect operations per time or signatures per second of this mechanism, which is a sign of stability and reliability I feel like. On the other hand, the signatures per second decreased dramatically for RSA operation which means it cannot handle very big key size.