Task 1:

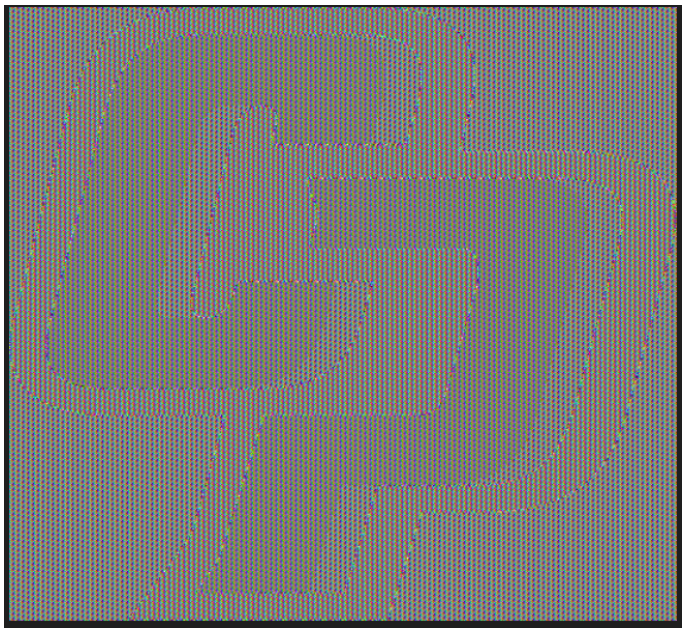
```
import random

blockSizeBytes = 16

def padByteArray(array):
    if(len(array) % blockSizeBytes != 0):
        newLen = (len(array)//blockSizeBytes + 1) * blockSizeBytes
        addedBytes = newLen - len(array)
        padding = bytearray(b'\x06') * addedBytes
        # print(padding)
        # print(array)
        return array + padding
    else:
        return array

def generateRandomByteKey(length):
    return [random.randint(0, 255) for _ in range(length)]

def xorByteArray(arr1, arr2):
    if len(arr1) != len(arr2):
        print("ERROR: arrays are not of equal length")
        return
    newArray = bytearray()
    for i in range(len(arr1)):
        newArray.append(arr1[i] ^ arr2[i])
    return newArray

def ECBEncryptBinary(key, plaintext):
    if len(key) != blockSizeBytes:
        print("ERROR: key is not of length " + blockSizeBytes)
        return

    plaintext = padByteArray(plaintext)

    cipherText = bytearray()

    for i in range(0, len(plaintext)//blockSizeBytes):
        cipherText += xorByteArray(plaintext[i*blockSizeBytes:
i*blockSizeBytes+blockSizeBytes], key)

    return cipherText

def CBCEncryptBinary(key, initVector, plaintext):
```

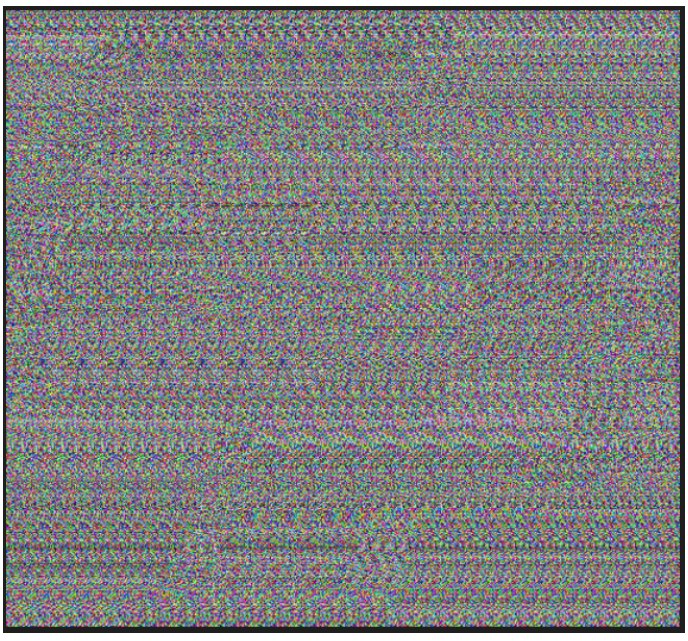
```
    if len(key) != blockSizeBytes:
        print("ERROR: key is not of length " + blockSizeBytes)
        return
    if len(initVector) != blockSizeBytes:
        print("ERROR: key is not of length " + blockSizeBytes)
        return

    # print(plaintext)
    plaintext = padByteArray(plaintext)

    cipherText = bytearray()

    lastBlock = initVector

    for i in range(0, len(plaintext)//blockSizeBytes):
        preCipherText = xorByteArray(plaintext[i*blockSizeBytes:
i*blockSizeBytes+blockSizeBytes], lastBlock)
        newCipherText = xorByteArray(preCipherText, key)
        lastBlock = newCipherText
        cipherText += lastBlock

    return cipherText

def ECBDecryptBinary(key, cipherText):
    if len(key) != blockSizeBytes:
        print("ERROR: key is not of length " + blockSizeBytes)
        return

    plaintext = bytearray()

    for i in range(0, len(cipherText)//blockSizeBytes):
        plaintext += xorByteArray(cipherText[i*blockSizeBytes:
i*blockSizeBytes+blockSizeBytes], key)

    return plaintext

def CBCDecryptBinary(key, initVector, cipherText):
    if len(key) != blockSizeBytes:
        print("ERROR: key is not of length " + blockSizeBytes)
        return
    if len(initVector) != blockSizeBytes:
        print("ERROR: key is not of length " + blockSizeBytes)
        return
```

```
    plainText = bytearray()

    lastBlock = initVector

    for i in range(0, len(cipherText)//blockSizeBytes):
        prePlaintext = xorByteArray(cipherText[i*blockSizeBytes:
i*blockSizeBytes+blockSizeBytes], key)
        newPlaintext = xorByteArray(prePlaintext, lastBlock)
        lastBlock = cipherText[i*blockSizeBytes: i*blockSizeBytes+blockSizeBytes]
        plainText += newPlaintext

    return plainText

def encryptBMP(filepath):
    key1 = generateRandomByteKey(blockSizeBytes)
    key2 = generateRandomByteKey(blockSizeBytes)
    initVector = generateRandomByteKey(blockSizeBytes)

    with open(filepath, 'rb') as ecbInput:
        txt = bytearray(ecbInput.read())

    # print(txt)
    header = txt[0:54]
    ecbEncryptedMessage = ECBEncryptBinary(key1, txt)
    ecbEncryptedMessage = header + ecbEncryptedMessage[54:]

    with open('ECB.bmp', 'wb+') as file:
        file.write(ecbEncryptedMessage)

    # print(txt)
    header = txt[0:54]
    cbcEncryptedMessage = CBCEncryptBinary(key2, initVector, txt)
    cbcEncryptedMessage = header + cbcEncryptedMessage[54:]

    with open('CBC.bmp', 'wb+') as file:
        file.write(cbcEncryptedMessage)

def encryptFile(filepath):
    key1 = generateRandomByteKey(blockSizeBytes)
    key2 = generateRandomByteKey(blockSizeBytes)
    initVector = generateRandomByteKey(blockSizeBytes)

    with open(filepath, 'rb') as ecbInput:
        txt = bytearray(ecbInput.read())
```

```
    ecbEncryptedMessage = ECBEncryptBinary(key1, txt)
    with open('ECB', 'wb+') as file:
        file.write(ecbEncryptedMessage)

    cbcEncryptedMessage = CBCEncryptBinary(key2, initVector, txt)
    with open('CBC', 'wb+') as file:
        file.write(cbcEncryptedMessage)

    ecbDecryptedMessage = ECBDecryptBinary(key1, ecbEncryptedMessage)
    cbcDecryptedMessage = CBCDecryptBinary(key2, initVector, cbcEncryptedMessage)

    print(ecbDecryptedMessage == txt)
    print(cbcDecryptedMessage == txt)


encryptFile('testInput')
encryptBMP('cp-logo.bmp')

# print(padPlaintext("ABCDEF").encode('ascii'))
# tmp = ECBEncrypt("aaaaaaaaaaaaaaaa", "b"*16+"c"*8)
# print(tmp.encode('ascii'))
```

In this task I wrote 3 helper functions: generateRandomByteKey(), xorByteArray(), and padByteArray(). They were all used in the encrypt/decrypt methods below them. generateRandomByteKey() uses a random number generator to get a random string of bytes of a given length. xorByteArray() is used to encrypt one string with another. padByteArray() performs PKCS #7 padding to ensure the plaintext length is divisible by 16. I tested each function one at a time so when I finally wrote the ECB and CBC encrypt methods, it was easy. Below are the ecb encrypted cal poly logo, and the cbc encrypted one.



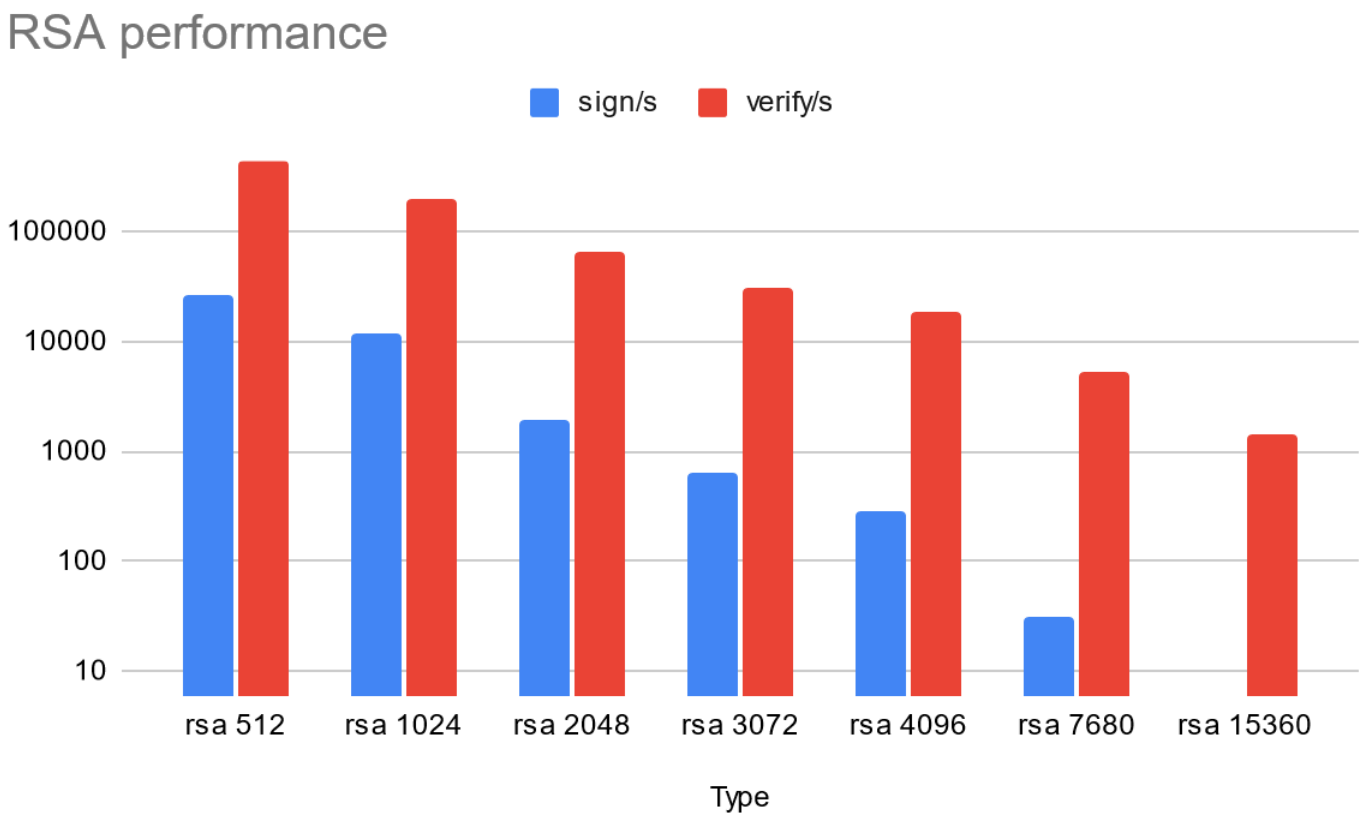


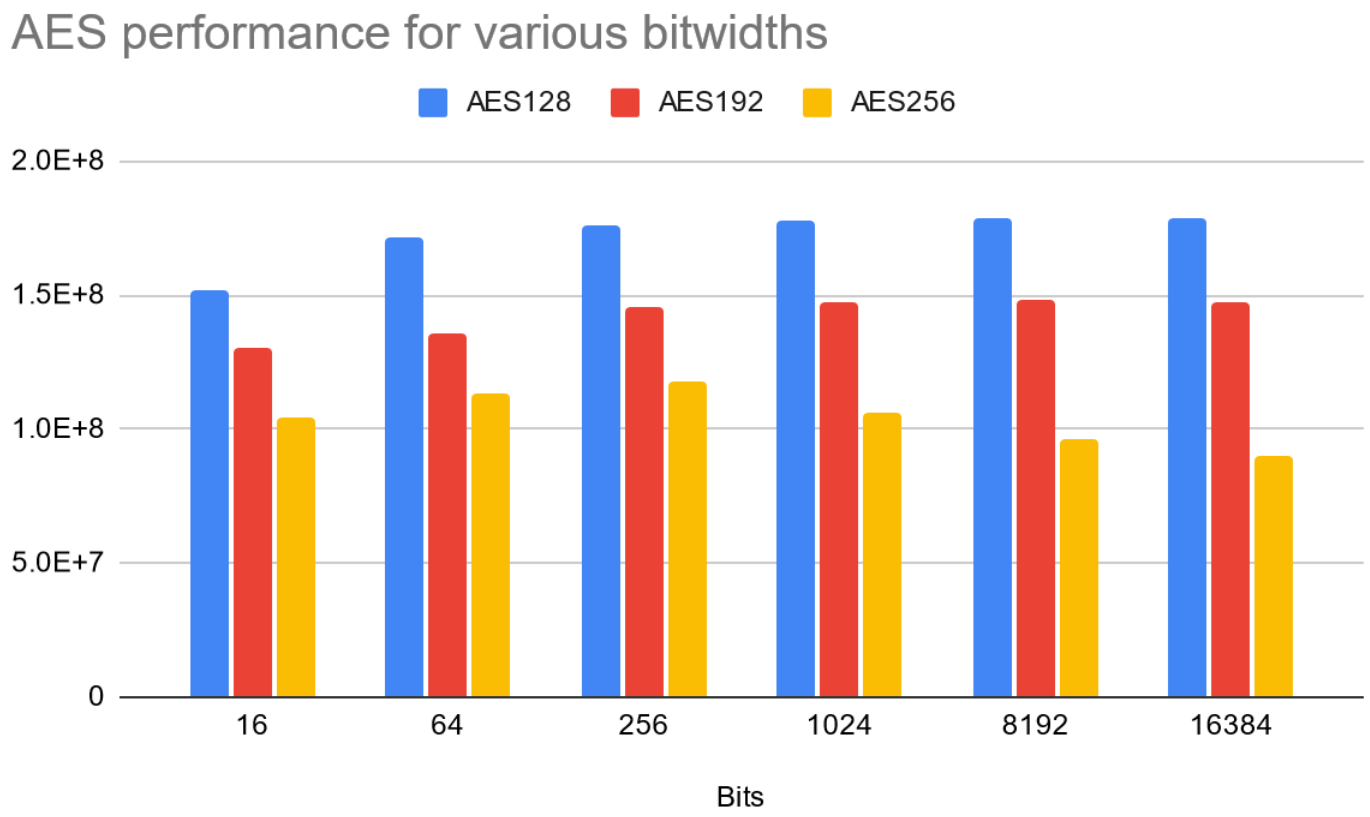
Task 2:

```
from blockEncryptions import generateRandomByteKey
from blockEncryptions import padByteArray
from blockEncryptions import CBCEncryptBinary
from blockEncryptions import CBCDecryptBinary

numUsers = 456
numSessions = 31337
blockSizeBytes = 16

def urlEncodeString(string:str):
    string = string.replace(';', '%3B')
    string = string.replace('=', '%3D')
    return string

def CDCattack(targetString, encryptedMsg, decryptedMsg):
    newStr = bytearray()
    for i, char in enumerate(targetString.encode()):
        decChar = encryptedMsg[i] ^ decryptedMsg[i+ blockSizeBytes]
        newChar = decChar ^ char
        newStr.append(newChar)
    return newStr + encryptedMsg[len(targetString):]

def submit(userData):
    key = generateRandomByteKey(blockSizeBytes)
    iv = generateRandomByteKey(blockSizeBytes)
    userData = urlEncodeString(userData)
    userString =
"userid="+str(numUsers)+';userdata='+userData+';session-id='+str(numSessions)
    userString = padByteArray(userString.encode())
    cipherText = CBCEncryptBinary(key, iv, userString)

    return (key, iv, cipherText)

def verify(key, iv, cipherText):
    plaintext = CBCDecryptBinary(key, iv, cipherText)
    return b';admin=true;' in plaintext


result = submit("asdf;alksdjf;alkjfasdlfkjf")
decryptedStr = CBCDecryptBinary(result[0], result[1], result[2])
encryptedAttack = CDCattack(";admin=true;", result[2], decryptedStr)
print(verify(result[0], result[1], encryptedAttack))
```

In this task, I reused my functions from the previous one and added the submit, verify, and CDCattack methods. The last 4 lines show how I implemented the byte change attack to trick the verify method. submit() returns the key, iv, and ciphertext as a tuple, which I indexed into to get a decrypted message, and create the encrypted attack.

Task 3:





Questions

1. In the bmp encrypted with ECB mode, you could still see the outline of the CP logo, although the colors are a bit off. I think this is due to the lack of randomness in the xoring. I was unable to discern the original image from the CBC encrypted bmp but it feels like there is a pattern to the randomness. I still can't see any remnants from the original image though.
2. This attack is possible due to the fact that the operations you are performing are relative to the ciphertext; you don't need the key or iv to trick the decryptor into returning a specific string. I think it would still be possible with a string longer than 16 character, but it would be a more complicated algorithm. To avoid the problem altogether there could be some form of two-factor authentication, requiring a verifiable ID as well as the CBC encryption. Thus, if an attack like this was performed, there would be an additional layer of defense.
3. The most interesting part of the AES chart is that the number of computations per second peaked at 256 bits for AES 256 while AES128 and AES192 increased as bitwidth increased. For RSA, there was a exponential decrease in performance as bitwidth increased.