Adrick Malekian
Vaibhav Garg

# Assignment 2 Report

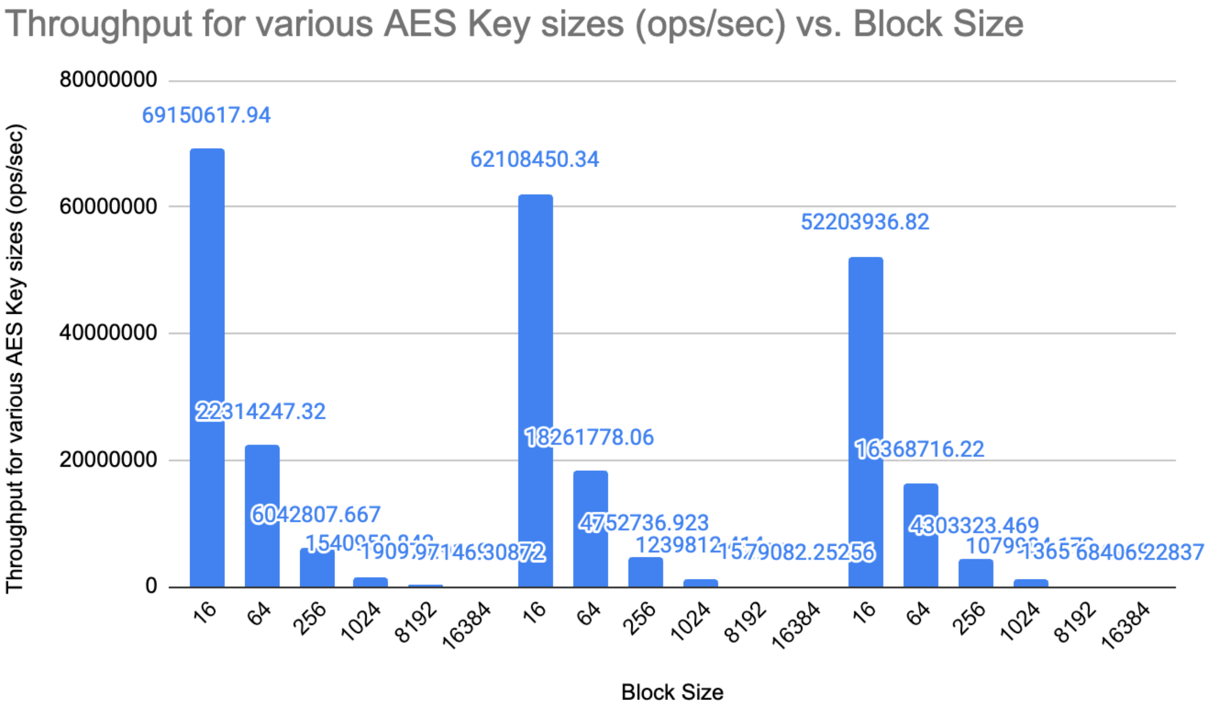
## Part 1:

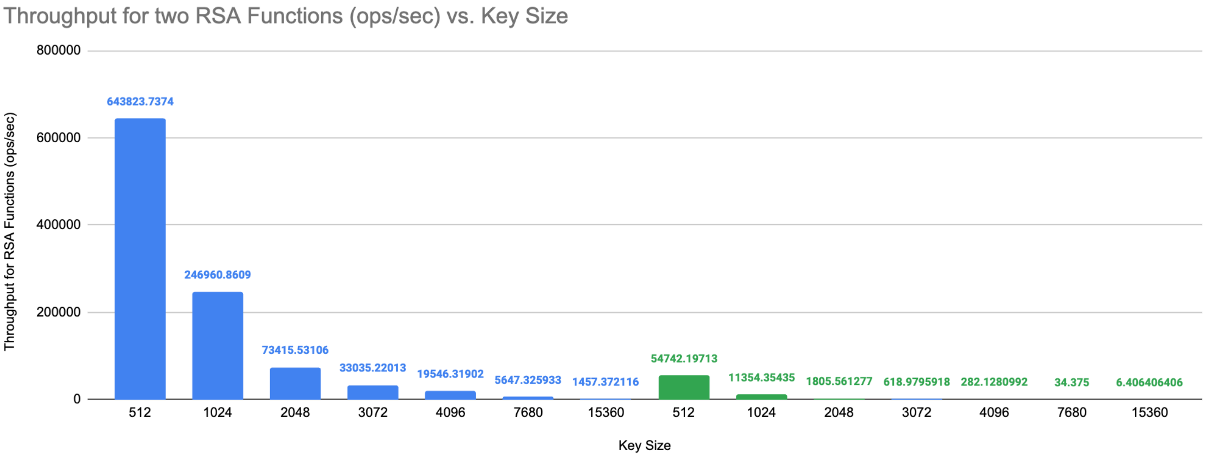
For this task, we were asked to implement the ECB and CBC encryption algorithms for encrypting bmp images. For ECB we went ahead and stripped the header from the data so that we can preserve that information especially for the images. Then once completed, we had the bmp section of the data holding the pixel info to encrypt. After padding using PKCS#7, we had the final data before encryption. To encrypt, first we needed to split the data into 128 bit blocks and using the AES-128-CBC cipher from the cryptographic library, we were able to build the encrypted text one block at a time appending the encrypted block to the resulting string. The header was then attached to the ciphertext and behold, our ciphertext. Decrypting is simply following these steps, but using the decrypt function instead of the encrypt. Once the ciphertext is decrypted, we accessed the last character in our decrypted text which will hold the padding length and using that value we were able to remove any excess padding. An interesting sight to see was that the encrypted image for ECB was very similar to the original since the ECB algorithm is repetitive for each block of data and thus forms a similar pattern to that of the original image.

## Part 2:

For this task, we were asked to implement a submit() and verify() function to help emulate a web server that uses cryptography to protect access to a site administration page. The submit() function encodes any symbols that the user provides and pads the final string before calling the encrypt functions. Then, the resulting ciphertext is sent to the verify() function that decrypts the ciphertext and looks for the pattern admin=true. Since we are encoding the “=” signs, this verify() function should never return true, since it won’t find the “=” in the admin=true, as it would instead be a %3D. To try and break this, we intercept the ciphertext after the submit() method encrypts, and forcefully look for an ‘admin=’ string. If we see it, we want to look at the block before it at that same byte, and we want to XOR that block with 1 at the right bit position so that the value can be changed for that block and will subsequently change that same bit in the next block since CBC does an XOR with the current encrypted string with the next blocks plaintext. This caveat was hard to discover but ultimately with some math and precise calculations, we were able to complete this.

## Part 3:





<u>Questions:</u>

1. For ECB I noticed that the resulting ciphertext was very similar to the original bmp however the color of the image was different and some of the pixels seem to appear blurry. This most likely occurred because ECB uses the same key to encrypt each block and thus we notice a very strong pattern between each block of encoded bytes. However for the CBC implementation, we can see the ciphertext is much more encrypted (as shown in the image below) to the point where we can't make it out. The reason is because

of the chained cipher blocks that use different keys. It is a slower process (because we lose the parallel encrypting power we get with ECB) but nonetheless more secure.

2. This attack is possible because we are able to exploit the predictable relationship between ciphertext and plaintext. We know the format of the plaintext beforehand, so we are able to count the number of bits from that for the ciphertext, flip the bits from there, and then return that during the interception. Having MACs can help prevent this as we are able to verify the integrity of some message to ensure it came from the actual user that made that request. Right now, the receiving oracle (verify() function) has no way of knowing if it came from the submit() function or from the flippng_bit attack function.
3. For the first graph above, it is showing us the throughput vs block size (bytes) for various AES key sizes. The first set of block sizes from left is using the AES-128 key encryption, the next is AES-192, and the last set of block sizes are from AES-256 key encryption. We can see that it's much faster to encrypt/verify information using smaller number of block sizes regardless of the key size. Most likely, the computer's CPU has an easier time doing the math and dealing with the data overhead for smaller chunks rather than larger chunks at a time. Essentially splitting up the tasks into smaller parts makes the machine more efficient at doing its encryption.

   The next graph below the first is the graph of two RSA functions (Public and Private RSA's) and their throughput for each key size. The blue columns are associated with the Public RSA function and the green columns are associated with the Private RSA function. We can see here that the public RSA function was able to compute significantly more RSA's per second than the private RSA function. This might have to do with the fact that private RSA keys have a larger exponent for the modular math and thus takes more time to encrypt or verify using each key. This is better since the computational complexity is higher for a private RSA key and thus will take longer for a malicious person to brute force their way into finding the private keys.

**<u>Our code:</u>**

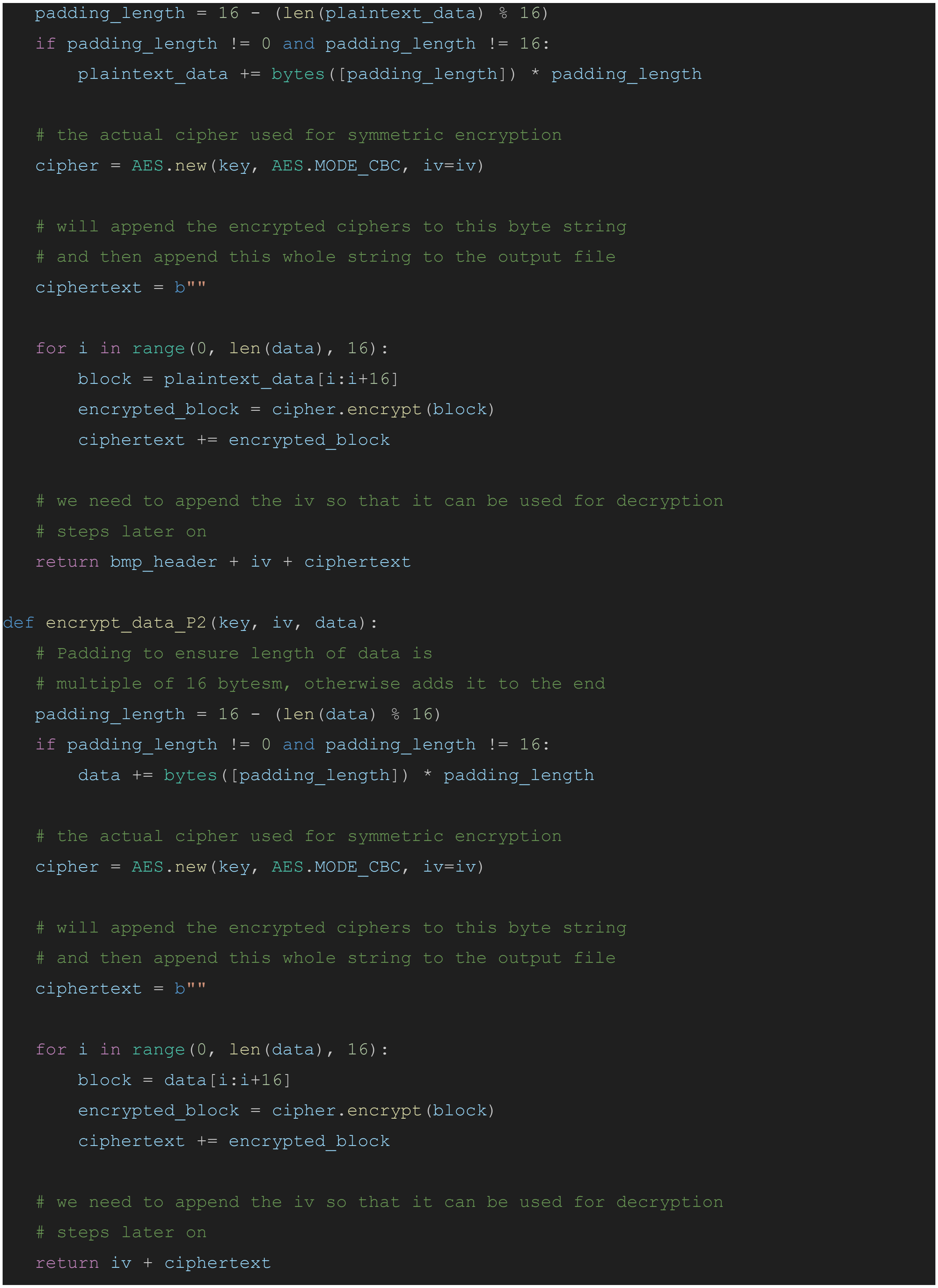
ECB

```
from Crypto.Cipher import AES
import random

def encrypt(key, data):
   header = data[:138]                                                     # The first 138
bytes of the data are the header
   data = data[138:]
   padding_len = 16 - (len(data) % 16)
   data += bytes([padding_len] * padding_len)
# Pad the data before encryption

   cipher = AES.new(key, AES.MODE_ECB)
   encrypted_data = b''
   for i in range(0, len(data), 16):                                      # Goes through 16
bytes of data at a time
       block = data[i:i+16]
       encrypted_data += cipher.encrypt(block)
   return header + encrypted_data

def decrypt(key, data):
   header = data[:138]
   data = data[138:]
   cipher = AES.new(key, AES.MODE_ECB)
   decrypted_data = b''
   for i in range(0, len(data), 16):
       block = data[i:i+16]
       decrypted_data += cipher.decrypt(block)
   padding_len = data[-1]
   decrypted_data[:-padding_len]                                 # Unpad the data after
decryption
   return header + decrypted_data

def generate_key():
   return bytes([random.randint(0, 255) for _ in range(16)])
```

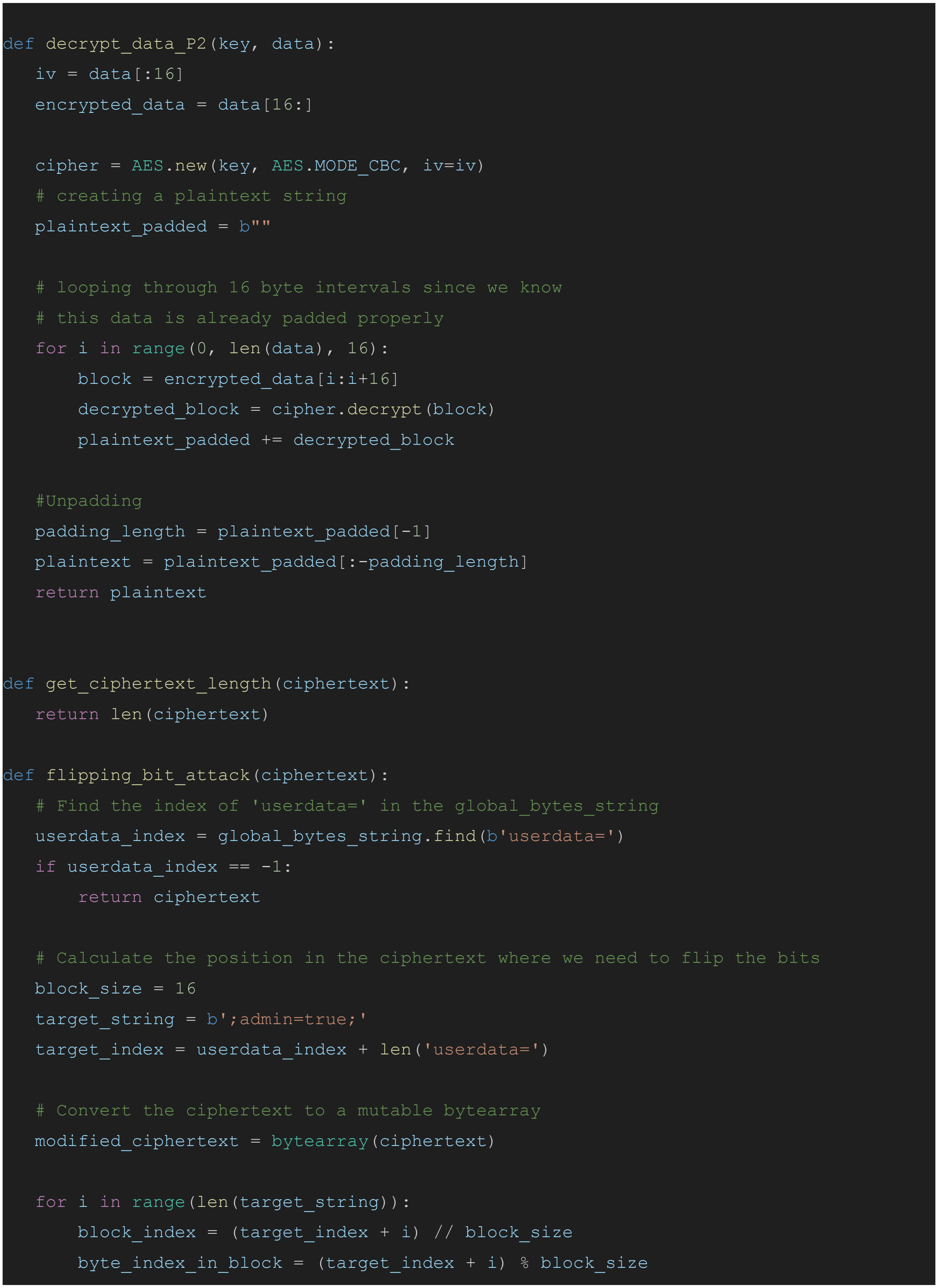
```
if __name__ == '__main__':
   with open('./cp-logo.bmp', 'rb') as f:
       plaintext = f.read()

   key = generate_key()                                            # This key must be
16 bytes long
   ciphertext = encrypt(key, plaintext)

   with open('./ciphertext.bmp', 'wb') as f:
       f.write(ciphertext)

   with open('./ciphertext.bmp', 'rb') as f:
       ciphertext = f.read()

   decrypted = decrypt(key, ciphertext)
   with open('./decrypted.bmp', 'wb') as f:
       f.write(decrypted)
```

CBC

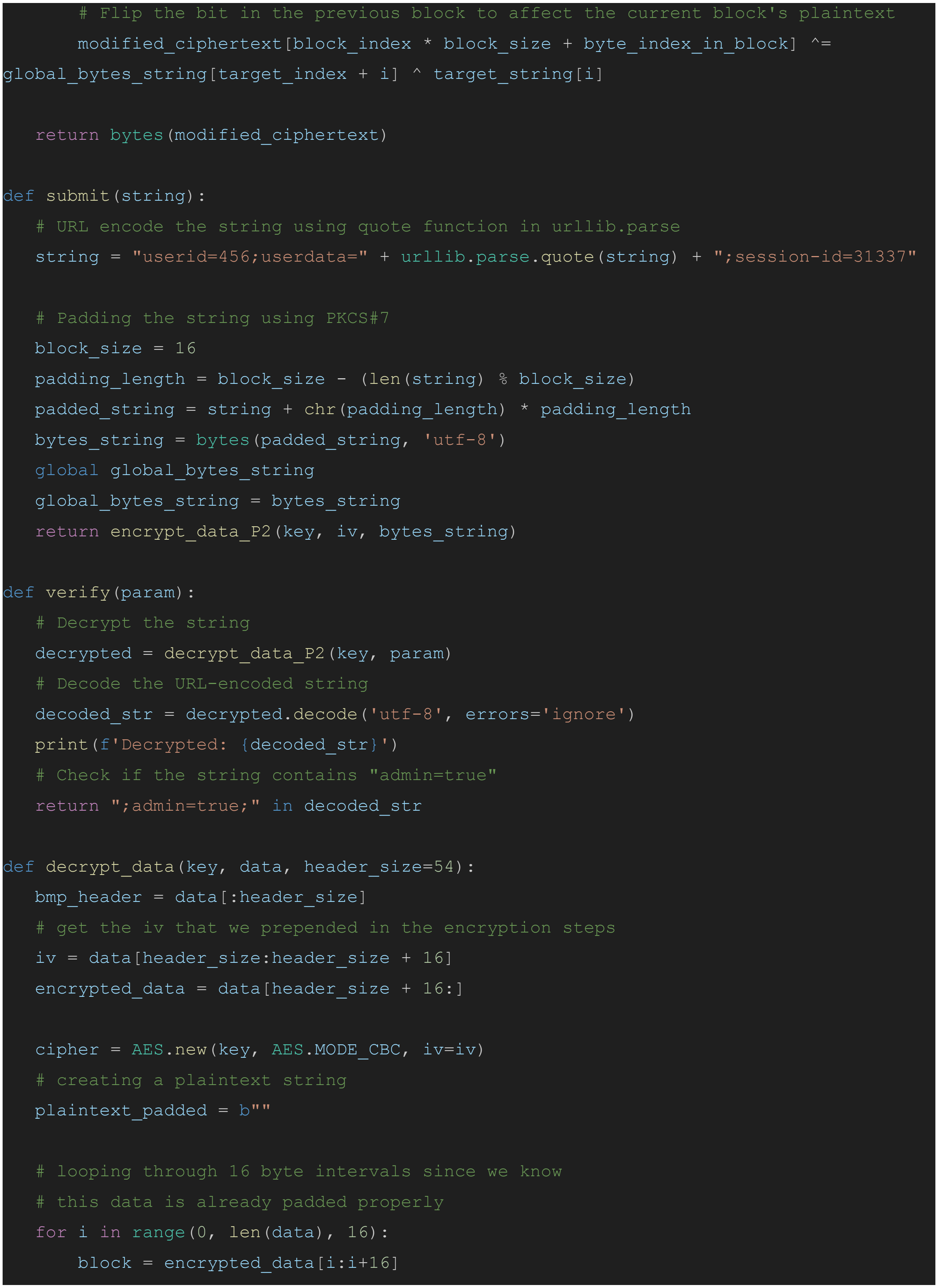
```
from Crypto.Cipher import AES
import os
import urllib.parse

key = None
iv = None
global_bytes_string = None

def generate_key():
   return os.urandom(16)

def generate_iv():
   return os.urandom(16)

def encrypt_data(key, iv, data, header_size=54):
   # we might want to change header_size to 138 depending
   bmp_header = data[:header_size]
   # contains the rest of the plaintext data
   plaintext_data = data[header_size:]

   # Padding to ensure length of data is
   # multiple of 16 bytesm, otherwise adds it to the end
```



```
    padding_length = 16 - (len(plaintext_data) % 16)
    if padding_length != 0 and padding_length != 16:
        plaintext_data += bytes([padding_length]) * padding_length

    # the actual cipher used for symmetric encryption
    cipher = AES.new(key, AES.MODE_CBC, iv=iv)

    # will append the encrypted ciphers to this byte string
    # and then append this whole string to the output file
    ciphertext = b""

    for i in range(0, len(data), 16):
        block = plaintext_data[i:i+16]
        encrypted_block = cipher.encrypt(block)
        ciphertext += encrypted_block

    # we need to append the iv so that it can be used for decryption
    # steps later on
    return bmp_header + iv + ciphertext

def encrypt_data_P2(key, iv, data):
    # Padding to ensure length of data is
    # multiple of 16 bytesm, otherwise adds it to the end
    padding_length = 16 - (len(data) % 16)
    if padding_length != 0 and padding_length != 16:
        data += bytes([padding_length]) * padding_length

    # the actual cipher used for symmetric encryption
    cipher = AES.new(key, AES.MODE_CBC, iv=iv)

    # will append the encrypted ciphers to this byte string
    # and then append this whole string to the output file
    ciphertext = b""

    for i in range(0, len(data), 16):
        block = data[i:i+16]
        encrypted_block = cipher.encrypt(block)
        ciphertext += encrypted_block

    # we need to append the iv so that it can be used for decryption
    # steps later on
    return iv + ciphertext
```



```
def decrypt_data_P2(key, data):
   iv = data[:16]
   encrypted_data = data[16:]

   cipher = AES.new(key, AES.MODE_CBC, iv=iv)
   # creating a plaintext string
   plaintext_padded = b""

   # looping through 16 byte intervals since we know
   # this data is already padded properly
   for i in range(0, len(data), 16):
       block = encrypted_data[i:i+16]
       decrypted_block = cipher.decrypt(block)
       plaintext_padded += decrypted_block

   #Unpadding
   padding_length = plaintext_padded[-1]
   plaintext = plaintext_padded[:-padding_length]
   return plaintext


def get_ciphertext_length(ciphertext):
   return len(ciphertext)

def flipping_bit_attack(ciphertext):
   # Find the index of 'userdata=' in the global_bytes_string
   userdata_index = global_bytes_string.find(b'userdata=')
   if userdata_index == -1:
       return ciphertext

   # Calculate the position in the ciphertext where we need to flip the bits
   block_size = 16
   target_string = b';admin=true;'
   target_index = userdata_index + len('userdata=')

   # Convert the ciphertext to a mutable bytearray
   modified_ciphertext = bytearray(ciphertext)

   for i in range(len(target_string)):
       block_index = (target_index + i) // block_size
       byte_index_in_block = (target_index + i) % block_size
```



```
        # Flip the bit in the previous block to affect the current block's plaintext
        modified_ciphertext[block_index * block_size + byte_index_in_block] ^=
global_bytes_string[target_index + i] ^ target_string[i]

    return bytes(modified_ciphertext)

def submit(string):
    # URL encode the string using quote function in urllib.parse
    string = "userid=456;userdata=" + urllib.parse.quote(string) + ";session-id=31337"

    # Padding the string using PKCS#7
    block_size = 16
    padding_length = block_size - (len(string) % block_size)
    padded_string = string + chr(padding_length) * padding_length
    bytes_string = bytes(padded_string, 'utf-8')
    global global_bytes_string
    global_bytes_string = bytes_string
    return encrypt_data_P2(key, iv, bytes_string)

def verify(param):
    # Decrypt the string
    decrypted = decrypt_data_P2(key, param)
    # Decode the URL-encoded string
    decoded_str = decrypted.decode('utf-8', errors='ignore')
    print(f'Decrypted: {decoded_str}')
    # Check if the string contains "admin=true"
    return ";admin=true;" in decoded_str

def decrypt_data(key, data, header_size=54):
    bmp_header = data[:header_size]
    # get the iv that we prepended in the encryption steps
    iv = data[header_size:header_size + 16]
    encrypted_data = data[header_size + 16:]

    cipher = AES.new(key, AES.MODE_CBC, iv=iv)
    # creating a plaintext string
    plaintext_padded = b""

    # looping through 16 byte intervals since we know
    # this data is already padded properly
    for i in range(0, len(data), 16):
        block = encrypted_data[i:i+16]
        decrypted_block = cipher.decrypt(block)
        plaintext_padded += decrypted_block

    #Unpadding
    padding_length = plaintext_padded[-1]
    plaintext = plaintext_padded[:-padding_length]

    return bmp_header + plaintext

if __name__ == '__main__':
    # part 1
    with open('./cp-logo.bmp', 'rb') as f:
        plaintext = f.read()

    key = generate_key()                                           # This key myst be
16 bytes long
    iv = generate_iv()

    ciphertext = encrypt_data(key, iv, plaintext)

    with open('./ciphertext.bmp', 'wb') as f:
        f.write(ciphertext)

    with open('./ciphertext.bmp', 'rb') as f:
        ciphertext = f.read()


    with open('./decrypted.bmp', 'wb') as f:
        decrypted = decrypt_data(key, ciphertext)
        f.write(decrypted)

    # part 2
    with open('./plaintextP2.txt', 'rb') as f:
        # Submit without attack, should always return false even if ;admin=true; is in
the plaintext
        plaintextP2 = f.read()
        ciphertext = submit(plaintextP2.decode('utf-8'))
        print(f'Verifying if admin=true: {verify(ciphertext)}\n')

        # Flipping bit attack
        hacked_cipher_text = flipping_bit_attack(ciphertext)
        print(f'Verifying if admin=true after flipping bit: 
{verify(hacked_cipher_text)}\n')
```