Assignment 1- Symmetric Key Cryptography

Seamus O'Malley and Ian Loo

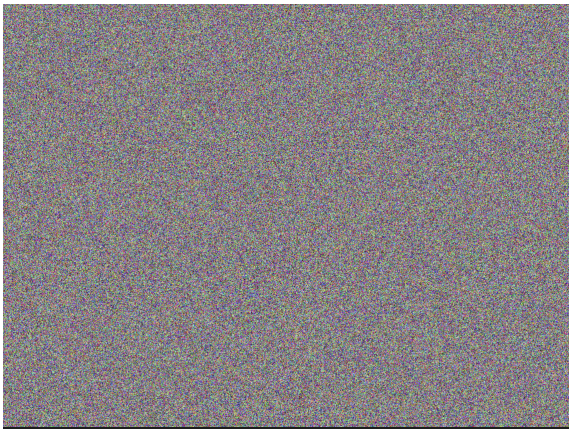
Questions:

1. For task 1, looking at the resulting ciphertexts, what do you observe? Are you able to derive any useful information about either of the encrypted images? What are the causes for what you observe?

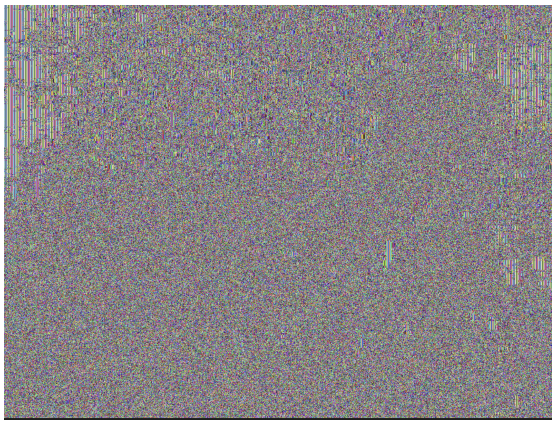
After encrypting an image of a robot with CBC, the generated bmp file looks extremely random and it looks like a lot of noise with no noticeable patterns. The cause of this is that CBC uses the previous block's ciphertext to generate the subsequent block of ciphertext meaning that repeated bytes of plaintext will be encrypted differently. This output is similar to the output of encryption with EBC, however, there were some patterns of repetition near the upper parts of the image which correlated with the black background of the setting. One can also see the subtle outline of the robot's back in the encrypted file due to this repetition of encryption. The cause of this is because EBC applies the same key to encrypt the plaintext for each block and the next block doesn't depend on the previous block so encrypting the same bytes in different blocks will result in the same ciphertext which explains the repetition in the resulting encryption of the robot image.

Original Picture

CBC Encryption



EBC Encryption



* notice the outline of the robot on the top right

2. For task 2, why is this attack possible? What would this scheme need in order to prevent such attacks?

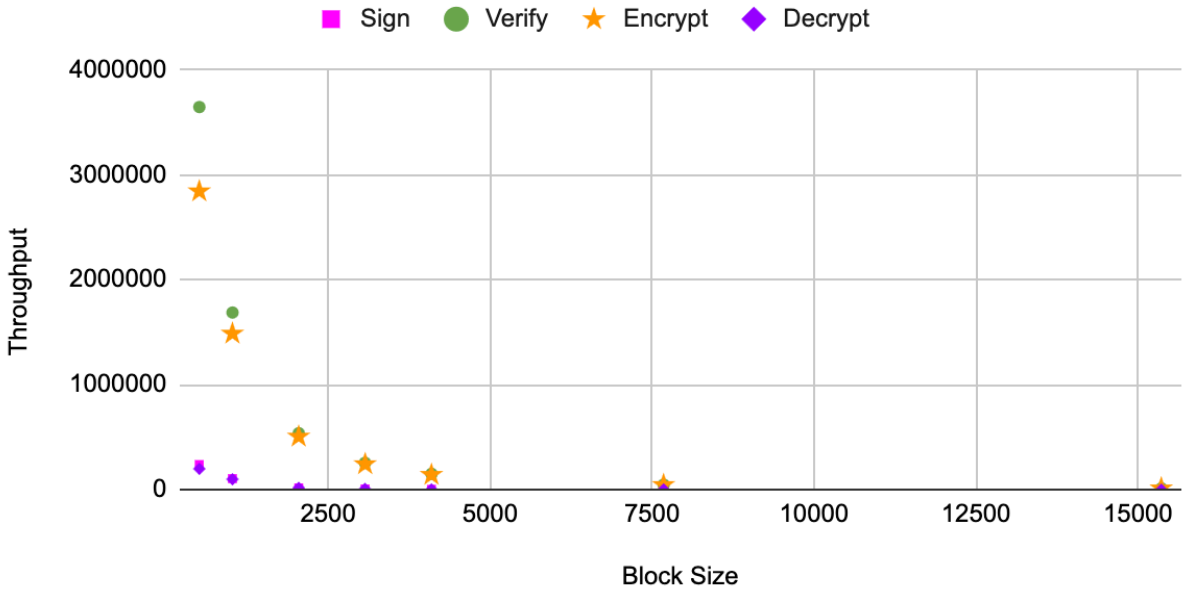
This bit flipping attack is possible for CBC because of the way plaintext is decrypted where each block depends on the previous ciphertext block. During decryption, the previous ciphertext block is XORed with the current block cipher decryption to generate the original plaintext; due to this process, an attacker can modify the previous ciphertext block to, in turn, modify the resulting plaintext block that is decrypted. By knowing the position of the byte the attacker wants to manipulate, as well as the value at that position, the attacker can use XOR operations to change the value at that position to whatever they want. In order to prevent this attack, the process of encrypting using CBC would need to be unique for each block meaning that the current encryption for a block wouldn't depend on the previous ciphertext. This could be achieved by using a different random key or iv to perform encryption on each block.

3. For task 3, how do the results compare?

The most notable thing initially noticeable is how much more throughput can be done with the lowest viable AES option vs the lowest viable RSA option. It must be noted that block sizes are different in tests. However, even with the same block sizes (1024 bits), AES is 4 times faster than the fastest RSA function (verify). This makes it seem, to us, that AES is a no-brainer for large payloads of data if it's more secure and quicker. Another interesting thing to note is just how drastically throughput drops off when larger key sizes are used. And it’s not even a close ordeal. As seen for example with RSA sign, which goes from close to a quarter million throughput with 512-bit block size to 51 throughput with 15360-bit block size, larger blocks greatly affect the throughput, as the throughput to block size relationships in both cases follow a negative exponential distribution. This brings up an interesting dilemma in cybersecurity: Yes, it would be ideal if we used the larger block sizes for most of our encryption, as they are inherently more secure (attackers must work harder to break larger blocks). This was seen with some of our exploration into DES/ 3DES, where their 64 bit blocks are just too small to be secure. But we also must consider the efficiency and practicality of using the largest available block sizes, and find a compromise between the two, while at the same time not compromising much of either. To us, it would make sense that we would find a minimum baseline for what we consider secure enough and then optimize speed around that, as security is a higher priority than speed.

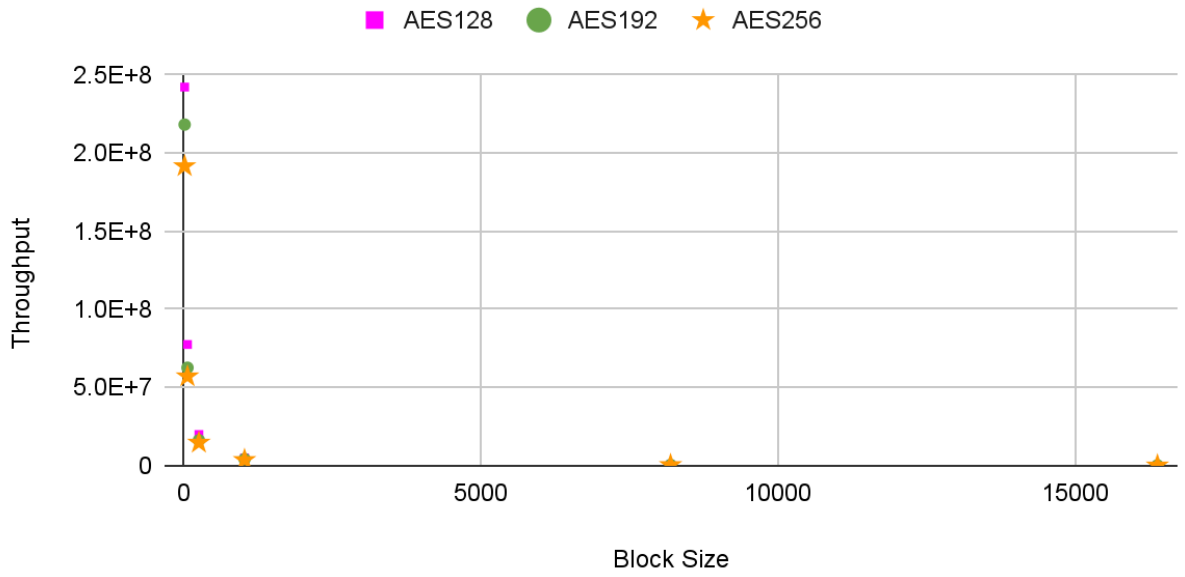
Block Size vs Throughput

RSA Encryption



Block Size vs Throughput

AES Encryption



Code Explanation:

ECB Encryption function:

```
def encode_text_ecb(content, file_path):
    # create a 128-bit (16-byte) key
    key = secrets.token_bytes(16)
    print(f"key is {key.hex()}")

    content = add_padding(content, len(content))

    # init the cipher using key, and init the byte array to hold encrypted content
    # each block encrypted separately with the same key
    ecb_cipher = AES.new(key, AES.MODE_ECB)
    encrypted_byte_array = b""

    header = content[0:54]

    # loop through 128 bits (16 bytes) at a time and
    # encrypt each 128 bit block with the generated key
    for i in range(54, len(content), 16):
        block = content[i:i + 16]
        encrypted_byte_array += ecb_cipher.encrypt(block)

    encrypted_byte_array = header + encrypted_byte_array
    write_to_file(encrypted_byte_array)
    return encrypted_byte_array
```

To encrypt a plaintext file using the ECB method of encryption, we first generated a random, 16 byte key, and subsequently added padding to the content of the file so that it was evenly divisible by 16 bytes. We then generated an ECB cipher using the AES.new function by feeding it the key we created. Next, we initialized a byte array to store our cipher text; we also stored the header (first 54 bytes) so that we could securely append it to our generated cipher text. To encrypt each 16 byte block, we looped through the content, 16 bytes at a time, starting at byte 54 to skip the header. We stored the 16 byte block in a variable and then encrypted it using the .encrypt function that is built into the generated ECB cipher and we appended the generated cipher text to the byte array storing our entire ciphertext. Finally, we prepended the header to the ciphertext, wrote the ciphertext to a file and returned the encrypted byte array (cipher text).

CBC Encryption function:

```
def encode_text_cbc(content, file_path=None, given_key=None, given_iv=None,
starting=54):
    # generate the iv and key
    if given_key is None and given_iv is None:
        iv = secrets.token_bytes(16)
        key = secrets.token_bytes(16)
    else:
        iv = given_iv
        key = given_key

    # find the file size in bytes
    if file_path is not None:
        file_size = os.stat(file_path).st_size
    else:
        file_size = len(content)

    encrypted_byte_array = b""
    # generate the cipher
    cbc_cipher = AES.new(key, AES.MODE_CBC, iv)
    # add padding to plaintext
    content = add_padding(content, file_size, starting)

    for i in range(starting, len(content), 16):
        # encrypt 16 bytes at a time
        cipher_text = cbc_cipher.encrypt(content[i:i + 16])
        encrypted_byte_array += cipher_text

    encrypted_byte_array = content[:starting] + encrypted_byte_array
    write_to_file(encrypted_byte_array)
    return encrypted_byte_array
```

To encrypt a plaintext file using the CBC method of encryption, we first generated the random key and initialization vector. Since this function needed to be compatible with a string (for the bit flip attack portion of the assignment), we included optional parameters for a key, initialization vector and starting point of encryption. We also had an optional parameter for a file_path since the bit flip attack methods don't utilize a file. We needed to obtain the size of the content which was done by either stat-ing the file or using the built in len function on the content if there was no file. We then initialize an empty byte array to store our cipher text, create the cipher using the key and initialization vector, and add padding to the content. Next we, once again, looped through the content from the starting variable (0 (for string) or 54 (for file with header)) to the length of the content, by 16 bytes at a time. In the loop, we once again encrypted in 16 byte

blocks and appended the resulting cipher text to the byte array. Finally, we prepended the header (if there was one) to the byte array, wrote the resulting cipher text to a file, and returned the cipher text.

Padding function:

```
def add_padding(content, file_size, starting=54):
    if (file_size - starting) % 16 != 0:
        padding_size = 16 - ((file_size - starting) % 16)
        print("padding_size: ", padding_size)
        padding = bytes([padding_size] * padding_size)

        return content + padding

    return content
```

To add padding to plaintext content, we followed PKCS#7 padding guidelines where we first checked if the content (barring the header of bmp files) was evenly divisible by 16. If it wasn't, we calculated the additional bytes needed to make the content divisible by 16 bytes and we added the number of bytes needed. Each additional byte was represented by the literal number of bytes to pad. Finally, we returned the padded content.

Remove padding function:

```
def remove_padding(padded_data):
    # the last byte indicates the padding length
    padding_len = padded_data[-1]
    if len(set(padded_data[-padding_len:])) == 1:
        return padded_data[:-padding_len]
    return padded_data
```

To remove padding, we retrieved the last byte (which would be the number of bytes we added for padding), and used the number to splice the padded data to take out the padding. We also checked if the last x bytes of the padded_data matched up with the padding length before we removed the padding, if not, we simply returned the padded data.

Submit function:

```
prepend = "userid=456;userdata="
append = ";session-id=31337"

def submit(string, key, iv):
   string = quote(string)
   new_string = prepend + string + append

   encrypt = encode_text_cbc(new_string.encode("utf-8"), given_key=key, given_iv=iv,
starting=0)

   return encrypt
```

For the submit function we first created two global variables, the first to be prepended to the user inputted string, and the second to be appended. We then use the imported quote function to URL encode the user inputted string, and prepend and append the two global variables to it. Finally, we call the CBC encryption function and pass in an already created, random key and initialization vector, as well as a starting point of 0 (since there is no header), and return the generated encryption.

Verify function:

```
def verify(string):
   # generating the constant key and iv
   key = secrets.token_bytes(16)
   iv = secrets.token_bytes(16)

   # getting the encrypted string
   encrypted_string = submit(string, key, iv)

   # attacking the encrypted string
   encrypted_string = bit_flip_attack(encrypted_string)

   # decrypting the encrypted string
   decrypted_string = cbc_decrypt(encrypted_string, key, iv)
```

```
    if b";admin=true;" in decrypted_string:
        print("Admin is true?? (you've been attacked)")
        return True
    else:
        return False
```

For the verify function, we first generate the constant key and initialization vector and pass in the user inputted string as well as the key and IV to the submit function which returns cipher text. Next, we call the bit flip attack function with the ciphertext which manipulates the given ciphertext. Finally, we decrypt the ciphertext using the stored key and initialization vector and we run a check to see if ";admin=true;" is in the decrypted string, if so, we return true, else we return false.

Bit flip attack function:

```
def bit_flip_attack(ciphertext):
    # string we want to inject into the plaintext
    ciphertext = bytearray(ciphertext)
    bit = ord('.') ^ ord('=')
    bit2 = ord('=') ^ ord(';')

    # flipping period to equal sign
    ciphertext[9] = ciphertext[9] ^ bit
    # flipping equal sign to semicolon
    ciphertext[3] = ciphertext[3] ^ bit2
    return bytes(ciphertext)
```

To do a bit flip attack on the ciphertext, we first converted the ciphertext into a bytearray so that it is indexable. Next we assume that the user inputted string was "admin.true" which comes right after "userdata=". Our goal here is to flip the bits so that we have "userdata;admin=true;" meaning we need to flip the "=" sign in "userdata=" to a ";" and the "." in "admin.true" to a "=". To do this we needed to know at what index the aforementioned characters were at, in the plaintext. Since the ciphertext would be the same length as the plaintext (with padding), and because we know that each ciphertext block is created by XOR'ing with the previous ciphertext block we found the associated indexes and manipulated the bytes in the previous block at those same indexes. Since we knew that the "." was at index 25, this means that it was in the second block so the associated byte position in block 1 would be 9 (25 - 16). The same method was used to flip the "=" to a ";". In order to correctly flip the bytes, we first had to XOR the original character with the character we wanted, and finally we could XOR the result of that operation with the current byte in the ciphertext at their respective positions in the previous block. Finally, we returned the manipulated ciphertext as bytes.

CBC Decryption function:

```
def cbc_decrypt(ciphertext, key, iv):
    cbc_cipher = AES.new(key, AES.MODE_CBC, iv)
    # Decrypt the ciphertext
    decrypted_data = remove_padding(cbc_cipher.decrypt(ciphertext))
    return decrypted_data
```

To decrypt ciphertext encrypted through CBC, we first needed the key and initialization vector. Then we could create a cipher with those values and decrypt the ciphertext. Finally, we removed the padding and returned the decrypted plaintext.