Assignment Group 9

Arian Houshmand
Lucas Summers

Module 2 Assignment - Block Ciphers

Task 1:

```
def pkcs7_pad(data, block_size=16):
    padding_len = block_size - (len(data) % block_size)
    return data + bytes([padding_len] * padding_len)

def pkcs7_unpad(data):
    padding_len = data[-1]
    return data[:-padding_len]
```

These functions deal with PKCS7 padding of data, given a block size in bytes. If the given data needs to be padded, bytes representing the byte length of the entire padding are added to the end of the data. Unpadding takes the last byte, which must hold the representation of the overall padding length, and uses this value to return the data without the padding.

```
def aes_ecb_encrypt(plaintext, key):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = b""
    for i in range(0, len(plaintext), 16):
        block = plaintext[i:i+16]
        ciphertext += cipher.encrypt(block)
    return ciphertext
```

This function encrypts the provided plaintext with the provided key using AES and ECB mode of encryption. AES.new establishes a new AES cipher object using the PyCryptodome package. Then we loop through 16 byte blocks of the provided plaintext, AES encrypting the block and appending it to our ciphertext.

```
def aes_cbc_encrypt(plaintext, key, iv):
    cipher = AES.new(key, AES.MODE_ECB)
    ciphertext = b""
    prev_block = iv
    for i in range(0, len(plaintext), 16):
```

```
        block = plaintext[i:i+16]
        block = bytes(a ^ b for a, b in zip(block, prev_block))
        encrypted_block = cipher.encrypt(block)
        ciphertext += encrypted_block
        prev_block = encrypted_block
    return iv + ciphertext
```

This function encrypts the provided plaintext with the provided keys using AES and CBC mode of encryption. We establish our first block as the IV (initialization vector) by setting **prev_block = IV.** Then we loop through 16 byte blocks of the provided plaintext, XOR each byte of the block with the corresponding byte of the **prev_block** (which is first set to the IV). We then AES encrypt the result of this and append it to the ciphertext. Last, we set this encrypted block as the new **prev_block** so that is can be XOR'ed with the next block. Note, we must prepend the IV to the ciphertext so that the data can be decrypted later.

```
def encrypt_aes(key, data, mode):
    padded_data = pkcs7_pad(data)

    if mode == 'ecb':
        encrypted_data = aes_ecb_encrypt(padded_data, key)
    elif mode == 'cbc':
        iv = get_random_bytes(16)
        encrypted_data = aes_cbc_encrypt(padded_data, key, iv)
    else:
        raise ValueError("Not a valid mode")
    return encrypted_data
```

This function is a helper that deals with encrypting data. First, it pads the data using our pkcs7_pad functions. Then, based on the provided mode of encryption, it calls the right function to encrypt the padded data and returns the encrypted data. Note, a random 16 byte IV is generated for CBC mode.

```
def decrypt_aes(key, data, mode):
    if mode == 'ecb':
        cipher = AES.new(key, AES.MODE_ECB)
        plaintext = pkcs7_unpad(cipher.decrypt(data), AES.block_size)
    elif mode == 'cbc':
        iv = data[:16]
        ciphertext = data[16:]
        cipher = AES.new(key, AES.MODE_CBC, iv)
        plaintext = pkcs7_unpad(cipher.decrypt(ciphertext), AES.block_size)
```

```
    else:
        raise ValueError("Not a valid mode")
    return plaintext
```

This is also a helper function that deal with decrypting data for task 2. It uses the built-in methods provided by the PyCryptodome package for both CBC and ECB modes to decrypt. Then, it unpads the data using our pkcs7_unpad function, returning the plaintext. Note, the IV for CBC mode is pulled out of the data as the first 16 bytes.

```
def encrypt_bmp_file(infile, outfile, mode):
    with open(infile, 'rb') as f:
        header = f.read(54)  # Adjust if the header is 138 bytes
        data = f.read()

    key = get_random_bytes(16)
    encrypted_data = encrypt_aes(key, data, mode)

    with open(outfile, 'wb') as f:
        f.write(header + encrypted_data)

    return key
```

This function handle encrypting a provided bmp file. First, it opens the file at the file path **infile** to read bytes and reads the first 54 bytes as the bmp header. Then it reads the rest of the bytes as the raw binary data. It then generates a random 16 byte key, calling our **encrypt_aes** function with the provided mode. Finally, the encrypted data is written to the provided output file located at the file path **outfile**, with the unencrypted header prepended to it. Note, the generated encryption key is also returned, so the user can securely save it and decrypt the data when needed.

Task 2:

```
def cbc_decrypt(encrypted_string, key, iv):
    cipher = AES.new(key, AES.MODE_ECB)
    decrypted_data = b''
    prev_block = iv
    for i in range(0, len(encrypted_string), 16):
        encrypted_block = encrypted_string[i:i+16]
        decrypted_block = cipher.decrypt(encrypted_block)
        decrypted_block = bytes(
            x ^ y for x, y in zip(decrypted_block, prev_block))
        decrypted_data += decrypted_block
```

```
        prev_block = encrypted_block
    decrypted_data = pkcs7_unpad(decrypted_data)
    return decrypted_data
```

This function decrypts data that was encrypted using the CBC. It initializes an AES cipher in ECB mode and processes the encrypted string in 16-byte blocks. For each block, it decrypts using the AES cipher and XORs the result with the previous ciphertext block (it uses IV for the first block) to reconstruct the plaintext. Finally, it removes any padding added in order to return it to the original data.

```
def submit(user_string):
    encoded_string = user_string.replace(";", "%3B")
    encoded_string = encoded_string.replace("=", "%3D")
    encoded_plaintext =
f"userid=456;userdata={encoded_string};session-id=31337".encode()
    encoded_plaintext = pkcs7_pad(encoded_plaintext)
    ciphertext, key, iv = cbc_encrypt(encoded_plaintext)
    return ciphertext, key, iv
```

The submit function prepares a user-provided string for encryption by encoding characters ; and = and formatting the input into a specific structure with “userid” in the front and the “session-id” at the end of the text. We then pad and encrypt the text using the functions we created. The function returns the encrypted text along with the encryption key and IV for use in the future.

```
def verify(encrypted_string, key, iv):


    decrypted_data = cbc_decrypt(encrypted_string, key, iv)
    print("decrypted data:",decrypted_data)
    admin_encoded = ";admin=true;".encode()
    return admin_encoded in decrypted_data
```

This function is our “authenticator”. We use it to decrypt a given ciphertext and verify whether it contains the ";admin=true;" substring in the plaintext. It uses the cbc_decrypt function from earlier to recover the plaintext from the encrypted string. After decryption, it checks for the presence of ";admin=true;" and returns True if the substring is found, otherwise False.

Currently, our verify function will never return True because of the way submit is set up. If you look at submit’s implementation, you will realize that no matter what the user inputs, our string could never purposely include “;admin=true;”, thus always being False.

But, CBC has its vulnerabilities. Let us simulate an attack by using bit flipping

```
def flip_the_bits(ciphertext):
   ciphertext_blocks = [ciphertext[i:i+16] for i in range(0, len(ciphertext), 16)]
   target_block = ciphertext_blocks[0]
   xor1 = ord('3') ^ ord(';')
   xor2 = ord('9') ^ ord('=')
   mod_cipher_block = target_block[:4] + bytes([target_block[4] ^ xor1]) +
target_block[5:10] + bytes([target_block[10] ^ xor2]) + target_block[11:15] +
bytes([target_block[15] ^ xor1])
   print("modified cipher block:", mod_cipher_block)
   ciphertext_blocks[0] = bytes(mod_cipher_block)
   modified_ciphertext = b''.join(ciphertext_blocks)
   return modified_ciphertext
```

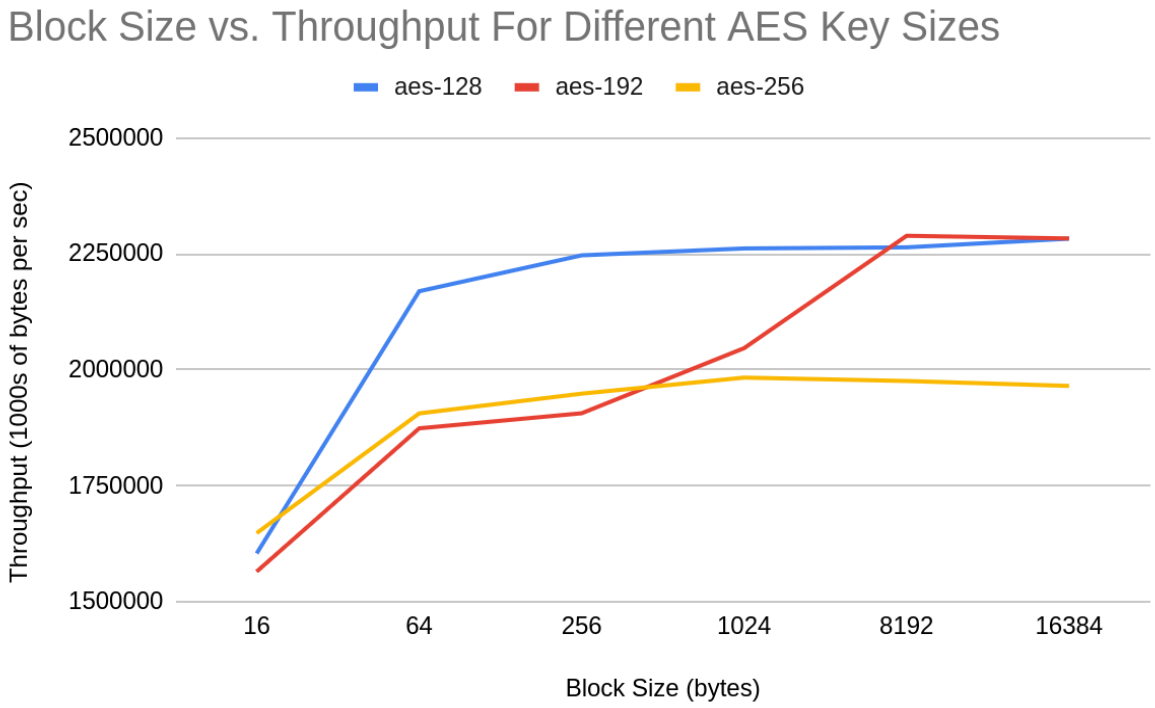
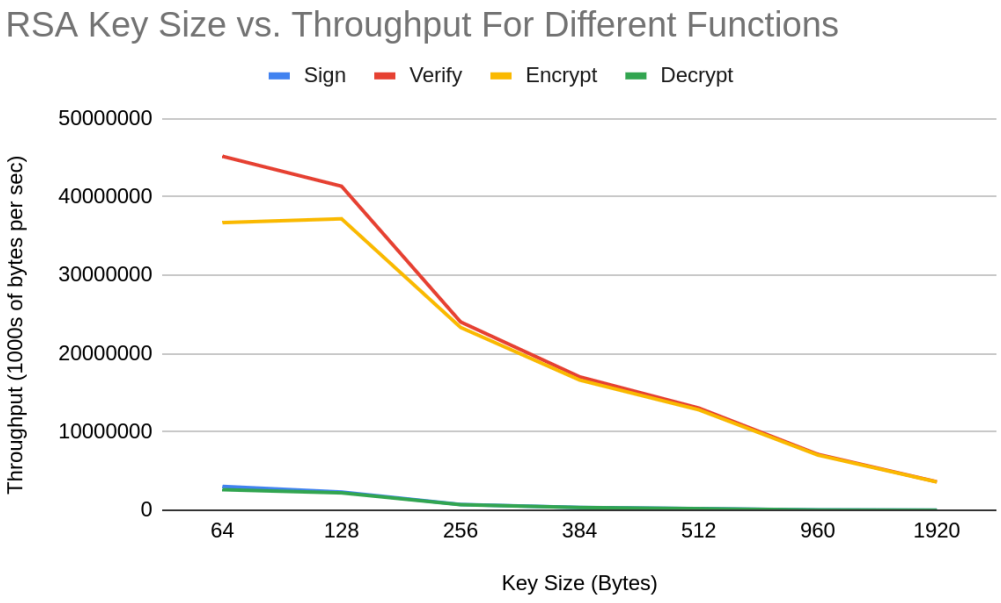
This function breaks the ciphertext into 16-byte blocks. For the attack, we crafted the user string as "3admin9true3," which gets placed in the second block of the ciphertext since the first block contains "userid=456;userd" and the second block becomes "ata=3admin9true3." To modify the second block, where the target string is located, we adjust the first block (our target block) so that, when XORed with "3" and "9," it produces ";" and "=" respectively. The values xor1 and xor2 are calculated to ensure this transformation happens. These calculated values are then used to replace specific bytes in the first block. Finally, the modified first block is swapped into the original ciphertext, completing the manipulation

```
def main():
   user_attack = "3admin9true3"
   cipher, key, iv = submit(user_attack)
   mod_cipher = flip_the_bits(cipher)
   val = verify(mod_cipher, key, iv)
   print(val)
```

Finally we pass the input string “3admint9true3” into submit. We then use our flip the bits function to attack the CBC method. And then using verify we can see that we were successful per the following output.

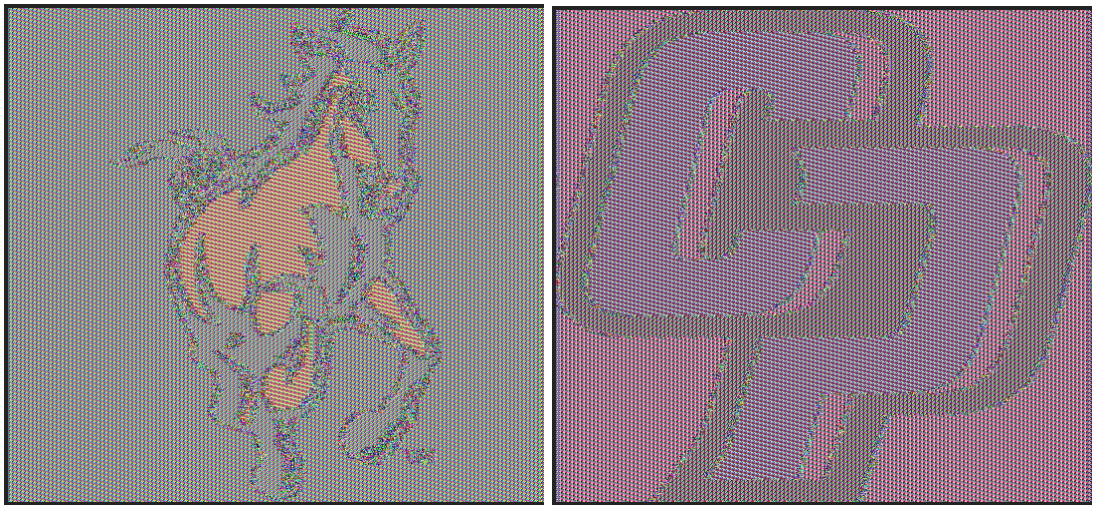
OUTPUT

```
modified cipher:
b'(\x1dQF\xb1#\x04\xc7_\xa6l\xfeW\xdc8D\x12\xfe\xa5\x1b\xe4N\xfb\x0e\x0b\xc5Wp\xd5\xb2~\
xa8T\xc2\xb8u\x83\xca\x15\xc0\x88\xec\xb8{\xb4#\x13\xbc\xc7\xe6v\x0b\xbf\xe3:\xb7\xf9B\x9b\
x1a\x16A\x9f\x16hb\x8d\xec\x90\x03=\x9cV`\xd6\\\xb8\x8f\xea\xbb'
decrypted data:
b'\x9a\t\xe0\x1e\x84)\x04\xc8\xc2\xdaT\x15G&\xd8Gata=;admin=true;;session-id=31337\x0f\x0f\
x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f\x0f'
```

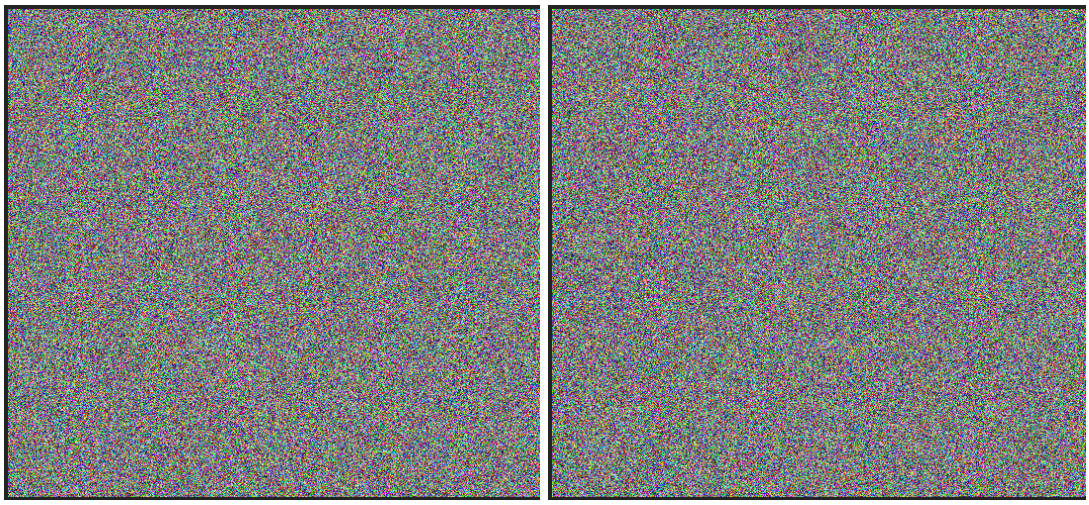
True
```

## Task 3:







**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?**

   We observed that the ciphertexts created using the ECB mode of encryption were still able to recognize the images, even though the colors changed. This is because the ECB mode changes that data in a uniform way. Here's what the images encrypted with ECB mode look like:



   For contrast, the images created with CBC mode of encryption looked completely scrambled due to the added XOR operation and random IV:



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

The attack is possible because in CBC mode, flipping a bit in a ciphertext block affects the corresponding plaintext and the next block during decryption. This allows an attacker to inject malicious data (e.g., ";admin=true;") by manipulating the ciphertext. However, it is important to note that this attack is also only possible because we know what the desired value, ";admin=true;" is. If we didn't know that the verify() function looks for this string within the text, we would not be able to find values that we would need to bit flip. To prevent such attacks, we can use a Message Authentication Code (MAC) or an authenticated encryption mode, which combines encryption and integrity verification. These measures ensure that any tampering with the ciphertext is detectable, effectively stopping such exploits.

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

AES Performance:

- The throughput increases as the block size grows, peaking at larger block sizes for all three key sizes
- AES-128 consistently outperforms AES-192 and AES-256, as it requires fewer computational operations due to its smaller key size.
- For all key sizes, the throughput levels off at around 2,250,000 KB/sec for block sizes of 8192 bytes or larger, showcasing AES's efficiency for high-throughput encryption.

RSA Performance:

- Throughput decreases as the key size increases, reflecting the greater computational overhead of larger RSA keys.
- Among the RSA functions, verification has the highest throughput, followed by encryption, decryption, and signing, indicating that verification is less computationally expensive than other RSA operations.
- At larger key sizes (1920 bytes), the throughput drops significantly, highlighting RSA's inefficiency for large-scale data encryption compared to AES.

AES is significantly faster and more suitable for encrypting large amounts of data, while RSA is slower and better suited for secure key exchange or small data operations. AES's throughput is higher than RSA's for equivalent block sizes, underscoring its efficiency for symmetric encryption tasks.