Cyber Security

Lab-3

Mruganshi Gohel

B20CS014

1. **Use DES and AES with modes like ECB, CBC, CFB, and OFB provided in Symmetric block**
   ciphers to Encrypt and Decrypt a Message. The Key used for encryption and decryption will be your roll number + first name (if the length is insufficient, pad the key with 0's).

   roll number + first name = B20CS014GOHEL

- **DES with ECB**

  DES has a fixed key size of 8 bytes (64 bits), while my key has more than 8 bytes, that's why I used a technique called triple DES (3DES) to use a longer key size effectively. 3DES uses three iterations of DES to create a longer key size. There are two keying options for 3DES: two-key and three-key. Two-key 3DES uses two keys, each 8 bytes long, for a total key size of 16 bytes (128 bits). I first define a DES key and plaintext to be encrypted. Then, I initialize the `DES`
  cipher with ECB mode and the key. I encrypt the padded plaintext using the `encrypt` method of the `DES` cipher object.

  To decrypt the ciphertext, I have used `decrypt` method of the `DES` cipher object. We unpad the decrypted plaintext by removing the padding bytes added during encryption.

  code is shown below:

```
from Crypto.Cipher import DES3
import base64
plaintext = b'This is a secret message'
key = b'B20CS014GOHEL000'
print('Original plaintext:', plaintext)
cipher = DES3.new(key, DES3.MODE_ECB)
ciphertext = cipher.encrypt(plaintext)
print('ciphertext:', ciphertext)
```

```
decrypted_padded_plaintext = cipher.decrypt(ciphertext)
print('Decrypted plaintext:', decrypted_padded_plaintext)
```

Output

```
PS D:\Cyber Security\lab-2> & C:/Users/HP/AppData/Local/Microsoft/WindowsApps/python3.10.exe "d:/
Original plaintext: b'This is a secret message'
ciphertext: b'\xdf\x1c}\xf8md\x9c%\xf7\xe0q\xd3\xb8\x9a\xee\x7fTQ6x\x8e\xcfGZ'
Decrypted plaintext: b'This is a secret message'
```

- **DES with CBC**

  As I mentioned earlier, the DES cipher is not designed to work with key sizes larger than 8 bytes. However, if you need to use DES with a larger key, you could consider using the Triple DES (3DES) algorithm. The initialization vector (IV) is defined and passed to the cipher constructor. Note that the IV must be the same length as the block size, which for DES is 8 bytes. the plaintext is padded to make it 16 bytes. The padded plaintext is then encrypted using the `encrypt` method of the cipher object. To decrypt the ciphertext, we create a new instance of the 3DES cipher with the same key and IV, and then call the `decrypt` method on the new object and print decrypted text.

  code is shown below:

```
from Crypto.Cipher import DES3
import base64
plaintext = b'This is a secret message'
key = b'B20CS014GOHEL000'
iv = b'ThisisIV'
print('Original plaintext:', plaintext)
cipher = DES3.new(key, DES3.MODE_CBC, iv=iv)
ciphertext = cipher.encrypt(plaintext)
print("Ciphertext: ",ciphertext)
cipher2 = DES3.new(key, DES3.MODE_CBC, iv=iv)
decrypted_padded_plaintext = cipher2.decrypt(ciphertext)
print('Decrypted plaintext:', decrypted_padded_plaintext)
```

Output

```
PS D:\Cyber Security\lab-2> & C:/Users/HP/AppData/Local/Microsoft/WindowsApps/python3.10.ex
Original plaintext: b'This is a secret message'
Ciphertext:  b'`\xe2w[p\x9b\x9a\xac/\x19\x18\xa5\xc9?\x9a\x0b\xca\xd2a\xec\xd3V\x8c\x12'
Decrypted plaintext: b'This is a secret message'
PS D:\Cyber Security\lab-2>
```

- **DES with CFB**

  CFB stands for Cipher Feedback. In CFB mode, the message is encrypted one block at a time, and the encryption of each block is dependent on the previous ciphertext block. It is similar to CBC mode, but instead of XORing the plaintext with the previous ciphertext block, it XORs the plaintext with the output of the encryption function applied to the previous ciphertext block. This makes CFB mode more suitable for streaming data, as it can begin encrypting data before the entire message is available.

  In order to use DES with CFB mode, I have to modify the code provided for DES with CBC mode by changing the mode to `DES3.MODE_CFB`. The rest of the code should remain the same.

  code is shown below:

```
from Crypto.Cipher import DES3
plaintext = b'This is a secret message'
key = b'B20CS014GOHEL000'
iv = b'Thisisan'
print('Original plaintext:', plaintext)
cipher = DES3.new(key, DES3.MODE_CFB, iv=iv)
ciphertext = cipher.encrypt(plaintext)
print("Ciphertext: ",ciphertext)
cipher2 = DES3.new(key, DES3.MODE_CFB, iv=iv)
decrypted_padded_plaintext = cipher2.decrypt(ciphertext)
print('Decrypted plaintext:', decrypted_padded_plaintext)
```

Output

```
PS D:\Cyber Security\lab-2> & C:/Users/HP/AppData/Local/Microsoft/WindowsApps/python3.10.exe
Original plaintext: b'This is a secret message'
Ciphertext:  b'D\xbflY\x9em\xd5\xe79gS\xcfl@e\xc3\x1b\xaa\xe5\xba\xa3H\xda\xed'
Decrypted plaintext: b'This is a secret message'
PS D:\Cyber Security\lab-2>
```

- **DES with OFB**

  OFB stands for Output Feedback. In OFB mode, the encryption function is applied to an initialization vector (IV), and the output is XORed with the plaintext to create the ciphertext. The encryption function is then applied to the IV again, and the output XORed with the previous ciphertext block to create the next ciphertext block. This process is repeated for the entire message.

  In order to use DES with OFB mode, you can modify the code provided for DES with CFB mode by changing the mode to `DES3.MODE_OFB`. The rest of the code should remain the same.

  code is shown below:

```
from Crypto.Cipher import DES3
plaintext = b'This is a secret message'
key = b'B20CS014GOHEL000'
iv = b'Thisisan'
print('Original plaintext:', plaintext)
cipher = DES3.new(key, DES3.MODE_OFB, iv=iv)
ciphertext = cipher.encrypt(plaintext)
print("Ciphertext: ",ciphertext)
cipher2 = DES3.new(key, DES3.MODE_OFB, iv=iv)
decrypted_padded_plaintext = cipher2.decrypt(ciphertext)
print('Decrypted plaintext:', decrypted_padded_plaintext)
```

Output

```
PS D:\Cyber Security\lab-2> & C:/Users/HP/AppData/Local/Microsoft/WindowsApps/python3.
Original plaintext: b'This is a secret message'
Ciphertext:  b'Df4#\xef\x893\x1b\xfauS\xb3\xb6[I\xf1\xee\xb9\xfe[\x90[g\n'
Decrypted plaintext: b'This is a secret message'
```

- **AES with ECB**

  I first define the AES key as a bytes object. I then created an instance of the AES cipher using the `new` method from `Crypto.Cipher.AES` module specifies the mode as `MODE_ECB` and pass `es` the key as an argument. I then encrypted some plaintext using the `encrypt`

method of the cipher object. To decrypt the ciphertext, I created a new instance of the cipher object using the same key and ECB mode, and then call the `decrypt` method on the cipher object, passing in the ciphertext.

If the key is correct, the decrypted plaintext should match the original plaintext. the key length must be 16 bytes (128 bits), 24 bytes (192 bits), or 32 bytes (256 bits) for AES.

code is shown below:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
key = b'B20CS014GOHEL000'
cipher = AES.new(key, AES.MODE_ECB)
plaintext = b'This is a secret message'
print("Original message: ",plaintext)
padded_plaintext = pad(plaintext, AES.block_size)
ciphertext = cipher.encrypt(padded_plaintext)
print("Ciphertext: ",ciphertext)
cipher = AES.new(key, AES.MODE_ECB)
decrypted_plaintext = cipher.decrypt(ciphertext)
unpadded_plaintext = unpad(decrypted_plaintext, AES.block_size)
print("Decrypted message: ",unpadded_plaintext)
```

Output

```
PS D:\Cyber Security\lab-2> & C:/Users/HP/AppData/Local/Microsoft/WindowsApps/python3.10.exe "d:/Cyber
Original message:  b'This is a secret message'
Ciphertext:  b".\x8e\xe8VM\xf6l\xc5,)y\x81\xffSnn\xad\xc0\xeb\xa9f\xdc^\xe1\xfb\xc2'\x1a\x9aV\xa9\xbe"
Decrypted message:  b'This is a secret message'
PS D:\Cyber Security\lab-2>
```

- **AES with CBC**

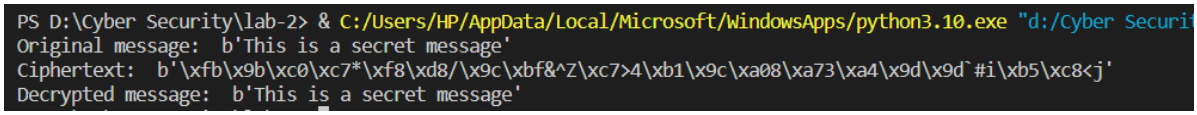
  To use AES with CBC mode, the code for AES with ECB mode can be modified by changing the mode to **AES.MODE_CBC**. The initialization vector (IV) is defined and passed to the cipher constructor. Note that the IV must be the same length as the block size, which for AES is 16 bytes. The plaintext is padded to make it a multiple of 16 bytes. The padded plaintext is then encrypted using the encrypt method of the cipher object. To decrypt the ciphertext, we create a new instance of the AES cipher

with the same key and IV, and then call the decrypt method on the new object and print decrypted text.

code is shown below:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import os
key = b'B20CS014GOHEL000'
iv = os.urandom(16)
cipher = AES.new(key, AES.MODE_CBC, iv)
plaintext = b'This is a secret message'
print("Original message: ",plaintext)
padded_plaintext = pad(plaintext, AES.block_size)
ciphertext = cipher.encrypt(padded_plaintext)
print("Ciphertext: ",ciphertext)
cipher = AES.new(key, AES.MODE_CBC, iv)
decrypted_plaintext = cipher.decrypt(ciphertext)
unpadded_plaintext = unpad(decrypted_plaintext, AES.block_size)
print("Decrypted message: ",unpadded_plaintext)
```

Output



- **AES with CFB**

  To use AES with CFB mode, the code for AES with CBC mode can be modified by changing the mode to **AES.MODE_CFB**. The initialization vector (IV) is defined and passed to the cipher constructor. Note that the IV must be the same length as the block size, which for AES is 16 bytes. The plaintext is padded to make it a multiple of 16 bytes. The padded plaintext is then encrypted using the encrypt method of the cipher object. To decrypt the ciphertext, we create a new instance of the AES cipher with the same key and IV, and then call the decrypt method on the new object and print decrypted text.

  code is shown below:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import os
key = b'B20CS014GOHEL000'
iv = os.urandom(16)
cipher = AES.new(key, AES.MODE_CFB, iv)
plaintext = b'This is a secret message'
print("Original message: ",plaintext)
padded_plaintext = pad(plaintext, AES.block_size)
ciphertext = cipher.encrypt(padded_plaintext)
print("Ciphertext: ",ciphertext)
cipher = AES.new(key, AES.MODE_CFB, iv)
decrypted_plaintext = cipher.decrypt(ciphertext)
unpadded_plaintext = unpad(decrypted_plaintext, AES.block_size)
print("Decrypted message: ",unpadded_plaintext)
```

Output

```
PS D:\Cyber Security\lab-2> python -u "d:\Cyber Security\lab-2\lab-3\q-1_aes_with_cfb.py"
Original message:  b'This is a secret message'
Ciphertext:  b'%m9>B\x01\xc8fb\xe2\x01\xd7\xd1L\xc6\x8e\xcf\xd9g2\xc8\xe8\x01\x97{\x9caw\xd1(\xfd\x17'
Decrypted message:  b'This is a secret message'
PS D:\Cyber Security\lab-2>
```

- **AES with OFB**

  To use AES with OFB mode, the code for AES with CFB mode can be modified by changing the mode to **AES.MODE_OFB**. The initialization vector (IV) is defined and passed to the cipher constructor. Note that the IV must be the same length as the block size, which for AES is 16 bytes. The plaintext is padded to make it a multiple of 16 bytes. The padded plaintext is then encrypted using the encrypt method of the cipher object. To decrypt the ciphertext, we create a new instance of the AES cipher with the same key and IV, and then call the decrypt method on the new object and print decrypted text.

  code is shown below:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import os
key = b'B20CS014GOHEL000'
```

```
iv = os.urandom(16)
cipher = AES.new(key, AES.MODE_OFB, iv)
plaintext = b'This is a secret message'
print("Original message: ",plaintext)
padded_plaintext = pad(plaintext, AES.block_size)
ciphertext = cipher.encrypt(padded_plaintext)
print("Ciphertext: ",ciphertext)
cipher = AES.new(key, AES.MODE_OFB, iv)
decrypted_plaintext = cipher.decrypt(ciphertext)
unpadded_plaintext = unpad(decrypted_plaintext, AES.block_size)
print("Decrypted message: ",unpadded_plaintext)
```

Output

```
PS D:\Cyber Security\lab-2> python -u "d:\Cyber Security\lab-2\lab-3\q-1_aes_with_ofb.py"
Original message:  b'This is a secret message'
Ciphertext:  b'\xe0p[\x0e\xef\xedW\x94gN\xa2\x8eD\x10:w:\x88\xfd\xa3\x96\x9f\xf9\xbc\x0f\n\x00\xc3\x01\xfb\xcd\xdc'
Decrypted message:  b'This is a secret message'
PS D:\Cyber Security\lab-2>
```

2. **Perform Diffie-Hellman key exchange between a client and server to share a secret key. Using this secret key, encrypt the message on the server side and send it to the client. Decrypt the message on the client side using the same key.**

   The Diffie Hellman Algorithm is being used to establish a shared secret that can be used for secret communications while exchanging data over a public network. In the below program, the client will share the value of p, q, and public key A. Whereas, the server will accept the values and calculate its public key B and send it to the client. Both the Client and Server will calculate the secret key for symmetric encryption by using the public key.

   - **Server code**

```
import socket
HOST = 'localhost'
PORT = 80

b = 6

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((HOST, PORT))
server_socket.listen(1)
print(f"Waiting for client on port {PORT}...")
```

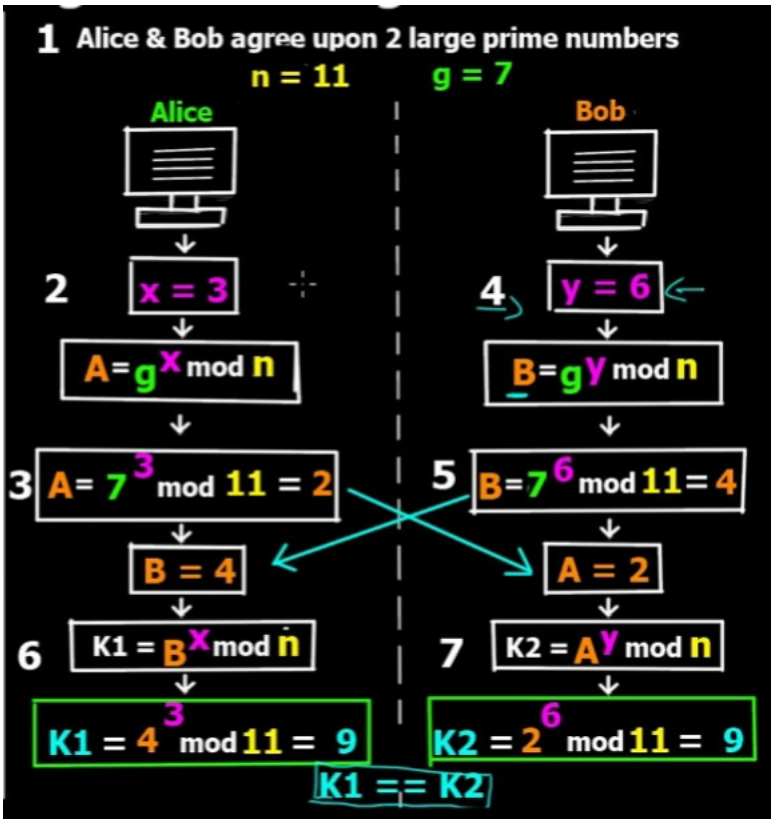
```
client_socket, address = server_socket.accept()
print(f"Just connected to {address}")

print(f"From Server : Private Key = {b}")
data = client_socket.recv(1024).decode()
clientP = int(data)

print(f"From Client : P = {clientP}")
data = client_socket.recv(1024).decode()
clientG = int(data)

print(f"From Client : G = {clientG}")
data = client_socket.recv(1024).decode()
clientA = int(data)

print(f"From Client : Public Key = {clientA}")
B = pow(clientG, b, clientP)
Bstr = str(B)

client_socket.sendall(Bstr.encode())
Bdash = pow(clientA, b, clientP)
print(f"Secret Key to perform Symmetric Encryption = {Bdash}")
client_socket.close()
```

- **client code**

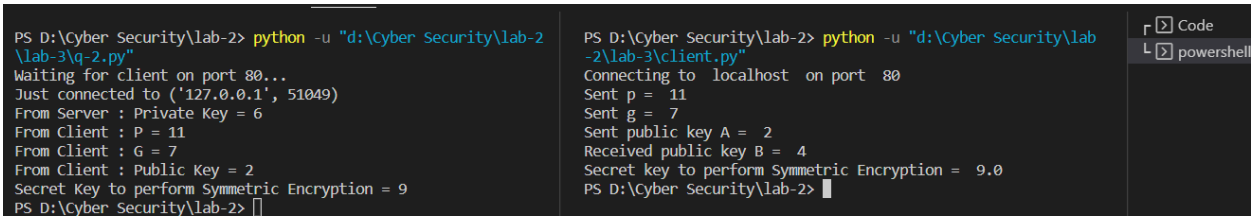
```
import socket
HOST = 'localhost'
PORT = 80

p = 11
g = 7
a = 3

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    print("Connecting to ", HOST, " on port ", PORT)

    s.sendall(str(p).encode())
    print("Sent p = ", p)

    s.sendall(str(g).encode())
    print("Sent g = ", g)

    A = (g ** a) % p
    s.sendall(str(A).encode())
    print("Sent public key A = ", A)

    B = s.recv(1024)
    print("Received public key B = ", B.decode())
```

```
    B = float(B.decode())
    Adash = (B ** a) % p
    print("Secret key to perform Symmetric Encryption = ", Adash)
```

process :



Output



here we can see that output of the code matches with the answer we get after the calculation.

3. **Perform Encryption and Decryption of the provided Image using any two modes of AES. Also**
compare the encrypted image in both cases (any type of comparison will suffice).

I have performed encryption and decryption of an image using the Advanced Encryption Standard (AES) algorithm in two modes of operation, namely Electronic Codebook (ECB) and Cipher Block Chaining (CBC) mode. Here is a step-by-step explanation of the code:

- The code imports necessary libraries including "Crypto.Cipher" for AES encryption, "Crypto.Util.Padding" for padding and unpadding the data, "PIL" for reading and saving images, "skimage.io" for image processing, and "os" for generating random numbers.
- The code reads an image file named "Lab3_image.jpg" and converts it into bytes using the "tobytes()" method. This will be the data to be encrypted.
- The code generates a random 16-byte key for AES encryption using the "os.urandom(16)" function.
- The code performs AES encryption in Electronic Codebook (ECB) mode using the generated key and the "cipher_ecb.encrypt()" method from the "Crypto.Cipher" library. The data is padded using the "pad()" function from the "Crypto.Util.Padding" library before encryption. The encrypted data is saved as an image.
- The code performs AES encryption in Cipher Block Chaining (CBC) mode using the same key and an Initialization Vector (IV) generated using "os.urandom(16)" and "AES.MODE_CBC" as the mode of operation. The data is padded using the "pad()" function before encryption. The encrypted data is saved as an image.
- The code reads the two encrypted image files and compares them byte by byte using the "with open()" statement. If the two files are identical, the code prints "The two encrypted image files are the same." Otherwise, it prints "The two encrypted image files are different."
- The code decrypts the encrypted data in ECB mode and CBC mode using the corresponding keys and the "cipher_ecb.decrypt()" method from the "Crypto.Cipher" library. The data is unpadded using the "unpad()" function before being saved as an image.

- The code reads the two decrypted image files and compares them byte by byte using the "with open()" statement. If the two files are identical, the code prints "The two decrypted image files are the same." Otherwise, it prints "The two decrypted image files are different."

below code is showed:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from PIL import Image
from skimage import io
import os
img = Image.open('D:\Cyber Security\lab-2\Lab3_image.jpg')
img_data = img.tobytes()

# Generate a secret key for AES encryption
key = os.urandom(16)

# ECB mode encryption
cipher_ecb = AES.new(key, AES.MODE_ECB)
encrypted_data_ecb = cipher_ecb.encrypt(pad(img_data, AES.block_size))
encrypted_img_ecb = Image.frombytes(img.mode, img.size, encrypted_data_ecb)
encrypted_img_ecb.save('encrypted_image_ecb.png')

# CBC mode encryption
iv = os.urandom(16)
cipher_cbc = AES.new(key, AES.MODE_CBC, iv)
encrypted_data_cbc = cipher_cbc.encrypt(pad(img_data, AES.block_size))
encrypted_img_cbc = Image.frombytes(img.mode, img.size, encrypted_data_cbc)
encrypted_img_cbc.save('encrypted_image_cbc.png')

# Compare the two encrypted image files
with open('encrypted_image_ecb.png', 'rb') as f1,
open('encrypted_image_cbc.png', 'rb') as f2:
    if f1.read() == f2.read():
        print("The two encrypted image files are the same.")
    else:
        print("The two encrypted image files are different.")

# Decryption using ECB mode
cipher_ecb = AES.new(key, AES.MODE_ECB)
decrypted_data_ecb = unpad(cipher_ecb.decrypt(encrypted_data_ecb), AES.block_size)
decrypted_img_ecb = Image.frombytes(img.mode, img.size, decrypted_data_ecb)
decrypted_img_ecb.save('decrypted_image_ecb.png')

# Decryption using CBC mode
cipher_cbc = AES.new(key, AES.MODE_CBC, iv)
decrypted_data_cbc = unpad(cipher_cbc.decrypt(encrypted_data_cbc), AES.block_size)
decrypted_img_cbc = Image.frombytes(img.mode, img.size, decrypted_data_cbc)
decrypted_img_cbc.save('decrypted_image_cbc.png')
```

```
with open('decrypted_image_ecb.png', 'rb') as f1,
open('decrypted_image_cbc.png', 'rb') as f2:
    if f1.read() == f2.read():
        print("The two decrypted image files are the same.")
    else:
        print("The two decrypted image files are different.")
```

Output

below is the **terminal output:**

```
PS D:\Cyber Security\lab-2> python -u "d:\Cyber Security\lab-2\lab-3\q-3_aes_ecb.py"
The two encrypted image files are different.
The two decrypted image files are the same.
PS D:\Cyber Security\lab-2>
```

here two encrypted image files are the different means of encryption done by CBC and EBC are different. and two decrypted image files are the same means after decryption in both modes output is the original image means two decrypted image files are the same.

below image is encrypted image after **AES with CBC mode** encryption

below image is encrypted image after **AES with ECB mode** encryption

below image is decrypted image after **AES with CBC mode** encryption, which is the same as the original image.

The below image is decrypted image after **AES with ECB mode** encryption, which is the same as the original image.