Indian Institute of Technology Jodhpur

CSL6010 – Cyber Security
Report – Lab 3

Name- Aman Srivastava
Roll No.- B20CS100

Date-26th Feb 2023

- **PROBLEM STATEMENT 2:**

  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 B20CS100AMAN0000.

  1. **DES algorithm with ECB mode:**

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad

# encryption function
def des_ecb_encrypt(key, message):
    cipher = DES.new(key.encode(), DES.MODE_ECB)
    padded_message = pad(message.encode(), DES.block_size)
    ciphertext = cipher.encrypt(padded_message)
    return ciphertext
# decryption function
def des_ecb_decrypt(key, ciphertext):
    cipher = DES.new(key.encode(), DES.MODE_ECB)
    padded_message = cipher.decrypt(ciphertext)
    plaintext = unpad(padded_message, DES.block_size)
    return plaintext.decode()
# sample message to be encrypted
message = "This is a secret message that needs to be encrypted using DES with
ECB mode."
# encryption with padded key
key = 'B20CS100AMAN0000'
key = key.encode()
key = key[:8] # truncate to 8 bytes
key = key.ljust(8, b'\0') # pad with zeroes if needed
key = key.decode()
ciphertext = des_ecb_encrypt(key, message)
print("Encrypted message (DES ECB):", ciphertext)
# decryption with padded key
plaintext = des_ecb_decrypt(key, ciphertext)
print("Decrypted message (DES ECB):", plaintext)
```

Output:

```
Encrypted message (DES ECB): b'\xac/\xe1A\xd6\xfb\xab\xa5\x86\xb1\xbf\xe8\xe2\xdfW\xe7\xf2)3sG\n\x02\x90
\xf6\xba\xd2\xf5\x853i\xbf\x1e\xcd\xd4\x1e\x96x\x8f/\xca\xba,hu^\xbd\x99WgX\x96I\xda\x7f\xcc\xe3\x07\xee
\xe1\xbd\x0c\x14\xd0\xac\xa1\x82\xf3oC\xba\xb0$X\xaf\x8f#C\x98\xf0'
Decrypted message (DES ECB): This is a secret message that needs to be encrypted using DES with ECB mode
```

2. **DES algorithm with CBC mode:**

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

# Function to encrypt plaintext message using DES in CBC mode
def des_cbc_encrypt(key, message):
    # Generate a random IV (Initialization Vector)
    iv = get_random_bytes(DES.block_size)
    padded_key = key[:8].ljust(8, '\0')
    cipher = DES.new(padded_key.encode(), DES.MODE_CBC, iv=iv)
    padded_message = pad(message.encode(), DES.block_size)
    ciphertext = cipher.encrypt(padded_message)
    return iv + ciphertext


# Function to decrypt ciphertext message using DES in CBC mode
def des_cbc_decrypt(key, ciphertext):
    # Extract the IV (Initialization Vector) from the ciphertext
    iv = ciphertext[:DES.block_size]
    padded_key = key[:8].ljust(8, '\0')
    cipher = DES.new(padded_key.encode(), DES.MODE_CBC, iv=iv)
    decrypted_message = cipher.decrypt(ciphertext[DES.block_size:])
    plaintext = unpad(decrypted_message, DES.block_size).decode()
    return plaintext


# Sample usage of the functions
key = 'B20CS100AMAN0000'
message = 'This is a secret message that needs to be encrypted using DES with
CBC mode.'

# Encrypt the message using DES in CBC mode
ciphertext = des_cbc_encrypt(key, message)
print('Encrypted message (DES CBC):', ciphertext)

# Decrypt the ciphertext using DES in CBC mode
plaintext = des_cbc_decrypt(key, ciphertext)
print('Decrypted message (DES CBC):', plaintext)
```

Output:

```
Encrypted message (DES CBC): b't\x94\xa3q\x01\x19 \x86\x1dS\xf2\xb559\xd7\xd7&\x1c\x1a\x8aT\n\xe8\x7fD\x8
9C\xaa|\xc9|\xf1\x17\x92=\xef\xc2\xa9f\xcdB\xd6\x0b\xb9\x1a\x81\x95h\xfeQ?\x07\tq\x0cL*\x99\xba\xc4Z\x1b\
xd4p@Ut\xa9\xccl\x86\x9d,\x9c\x1d\x89\xdd\x12\xf4\xd0qL\xa2\x9a\xdb\xea\x80\xd2'
Decrypted message (DES CBC): This is a secret message that needs to be encrypted using DES with CBC mode.
```

3. DES algorithm with CFB mode:

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

# Function to encrypt plaintext message using DES in CFB mode
def des_cfb_encrypt(key, message):
    # Generate a random IV (Initialization Vector)
    iv = get_random_bytes(DES.block_size)
    padded_key = key[:8].ljust(8, '\0')
    cipher = DES.new(padded_key.encode(), DES.MODE_CFB, iv=iv)
    ciphertext = cipher.encrypt(message.encode())
    return iv + ciphertext


# Function to decrypt ciphertext message using DES in CFB mode
def des_cfb_decrypt(key, ciphertext):
    # Extract the IV (Initialization Vector) from the ciphertext
    iv = ciphertext[:DES.block_size]
    padded_key = key[:8].ljust(8, '\0')
    cipher = DES.new(padded_key.encode(), DES.MODE_CFB, iv=iv)
    decrypted_message = cipher.decrypt(ciphertext[DES.block_size:])
    return decrypted_message.decode()

# Sample usage of the functions
key = 'B20CS100AMAN0000'
message = 'This is a secret message that needs to be encrypted using DES with
CFB mode.'

# Encrypt the message using DES in CFB mode
ciphertext = des_cfb_encrypt(key, message)
print('Encrypted message (DES CFB):', ciphertext)

# Decrypt the ciphertext using DES in CFB mode
plaintext = des_cfb_decrypt(key, ciphertext)
print('Decrypted message(DES CFB):', plaintext)
```

Output:

```
Encrypted message (DES CFB): b'\xab\xabA\xc9?Q\x90\ro\xc2L\xc8\xbf\xa1>\xe8\xb8|g~\xa7\xb8\x9f;\xd3\xdao
\xf2]\x13\xcc{\xde\xfb\x11\n\xe765\xfe\x82;\x90\x1e\x8b\x8dm\x90S\xa1\xb6\x13fOc\xdcN\xfeN|\n\xf9\xc8\x9
d|\xa4p#\xc8\x8d\x17\xd3HL\xab\x17\xfe\r\x9c\xb1\xb6\x0b\xd5\x90'
Decrypted message(DES CFB): This is a secret message that needs to be encrypted using DES with CFB mode.
```

4. **DES algorithm with OFB mode:**

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

# Function to encrypt plaintext message using DES in OFB mode
def des_ofb_encrypt(key, message):
    # Generate a random IV (Initialization Vector)
    iv = get_random_bytes(DES.block_size)
    padded_key = key[:8].ljust(8, '\0')
    cipher = DES.new(padded_key.encode(), DES.MODE_OFB, iv=iv)
    ciphertext = cipher.encrypt(message.encode())
    return iv + ciphertext


# Function to decrypt ciphertext message using DES in OFB mode
def des_ofb_decrypt(key, ciphertext):
    # Extract the IV (Initialization Vector) from the ciphertext
    iv = ciphertext[:DES.block_size]
    padded_key = key[:8].ljust(8, '\0')
    cipher = DES.new(padded_key.encode(), DES.MODE_OFB, iv=iv)
    decrypted_message = cipher.decrypt(ciphertext[DES.block_size:])
    return decrypted_message.decode()


# Sample usage of the functions
key = 'B20CS100AMAN0000'
message = 'This is a secret message that needs to be encrypted using DES with
OFB mode.'

# Encrypt the message using DES in OFB mode
ciphertext = des_ofb_encrypt(key, message)
print('Encrypted message (DES OFB):', ciphertext)

# Decrypt the ciphertext using DES in OFB mode
plaintext = des_ofb_decrypt(key, ciphertext)
print('Decrypted message (DES OFB):', plaintext)
```

Output:

```
Encrypted message (DES OFB): b'\xec\xf6\xa4\xc6\xc0aW\xdb\xe5}\x16\xfaF\xebj\xa95Zj|w\xab\x9a\xe7\x9ed\x
b2t\x07\x12\xc8\xaa/\xfb\x9e@\x9e\x12H\x18}\n1\x10\xad\x13\xcb\xcc6\x9d\x0c\x04\xcf\xbf\xecC\xb7\x88%\xc
56T<\x04\xc3\xdd261{R\xe7\xbc\xdb\xc3\xfd\xcen\xe4\xf1\x84?m\xb1'
Decrypted message (DES OFB): This is a secret message that needs to be encrypted using DES with OFB mode
```

5. **AES algorithm with ECB mode:**

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

# Function to encrypt plaintext message using AES in ECB mode
def aes_ecb_encrypt(key, message):
    # Pad the key to a multiple of 16 bytes
    padded_key = key[:16].ljust(16, '\0')
    cipher = AES.new(padded_key.encode(), AES.MODE_ECB)
    padded_message = pad(message.encode(), AES.block_size)
    ciphertext = cipher.encrypt(padded_message)
    return ciphertext

# Function to decrypt ciphertext message using AES in ECB mode
def aes_ecb_decrypt(key, ciphertext):
    # Pad the key to a multiple of 16 bytes
    padded_key = key[:16].ljust(16, '\0')
    cipher = AES.new(padded_key.encode(), AES.MODE_ECB)
    decrypted_message = cipher.decrypt(ciphertext)
    return unpad(decrypted_message, AES.block_size).decode()

# Sample usage of the functions
key = 'B20CS100AMAN0000'
message = 'This is a secret message that needs to be encrypted using aes with
ECB mode.'

# Encrypt the message using AES in ECB mode
ciphertext = aes_ecb_encrypt(key, message)
print('Encrypted message (AES ECB):', ciphertext)

# Decrypt the ciphertext using AES in ECB mode
plaintext = aes_ecb_decrypt(key, ciphertext)
print('Decrypted message (AES ECB):', plaintext)
```

Output:

```
Encrypted message (AES ECB): b'\x04\xff\xdau\xb4\xf5'%2\x0e\xbb\x0b\xc0I\xf7iz)\xc1\x1e5\x85\x88\x07\xf5=
C\x84\xed\xcfjP\xa0\xd4e\t\x88\xa9uQt\x15*\xb2\xf5\x89\x14\xfd\x82mq\x8bc\xc7\x00@\xce\xe0\x92\x9c~\xd4\x
82\xa1\xf0N>BB6q\xd0s\xf2\x81\xe8q{\xd2c'
Decrypted message (AES ECB): This is a secret message that needs to be encrypted using aes with ECB mode.
```

6. AES algorithm with CBC mode:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

# Function to encrypt plaintext message using AES in CBC mode
def aes_cbc_encrypt(key, message):
    # Pad the key to a multiple of 16 bytes
    padded_key = key[:16].ljust(16, '\0')
    iv = get_random_bytes(AES.block_size)
    cipher = AES.new(padded_key.encode(), AES.MODE_CBC, iv)
    padded_message = pad(message.encode(), AES.block_size)
    ciphertext = cipher.encrypt(padded_message)
    return iv + ciphertext

# Function to decrypt ciphertext message using AES in CBC mode
def aes_cbc_decrypt(key, ciphertext):
    # Pad the key to a multiple of 16 bytes
    padded_key = key[:16].ljust(16, '\0')
    iv = ciphertext[:AES.block_size]
    cipher = AES.new(padded_key.encode(), AES.MODE_CBC, iv)
    decrypted_message = cipher.decrypt(ciphertext[AES.block_size:])
    return unpad(decrypted_message, AES.block_size).decode()

# Sample usage of the functions
key = 'B20CS100AMAN0000'
message = 'This is a secret message that needs to be encrypted using AES with
CBC mode.'

# Encrypt the message using AES in CBC mode
ciphertext = aes_cbc_encrypt(key, message)
print('Encrypted message (AES CBC):', ciphertext)

# Decrypt the ciphertext using AES in CBC mode
plaintext = aes_cbc_decrypt(key, ciphertext)
print('Decrypted message (AES CBC):', plaintext)
```

Output:

```
Encrypted message (AES CBC): b'Go1\xfe\x1f\xcf\x10\xedy\x82L4\xcfA\x98T\xa6w\x93X\xc0\xb4r\xb5\xd0\xb7\x1
8\xd4\x93+\xcb\xbf\xb2\xbd\xce\x83TH\x80C\x05)\x00\xeb\xa3Q\xca\x97\xd0\xc4\x8d\x1dPt\x87\x05\xd9\x99\xbe
\x91\x16\xc3\x1d\xb5\xf2\xd0\xc8\xf6_5U\xc9>\xc3\xc9z\x90\x98q\xcb\xaa\xda\xd4\x82i\xc4\x96\xfa\xc6\x1b\x
b4k\xb3\x94\xdd\x98'
Decrypted message (AES CBC): This is a secret message that needs to be encrypted using AES with CBC mode.
```

7. **AES algorithm with CFB mode:**

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

# Function to encrypt plaintext message using AES in CFB mode
def aes_cfb_encrypt(key, message):
    # Pad the key to a multiple of 16 bytes
    padded_key = key[:16].ljust(16, '\0')
    iv = get_random_bytes(AES.block_size)
    cipher = AES.new(padded_key.encode(), AES.MODE_CFB, iv)
    ciphertext = cipher.encrypt(message.encode())
    return iv + ciphertext

# Function to decrypt ciphertext message using AES in CFB mode
def aes_cfb_decrypt(key, ciphertext):
    # Pad the key to a multiple of 16 bytes
    padded_key = key[:16].ljust(16, '\0')
    iv = ciphertext[:AES.block_size]
    cipher = AES.new(padded_key.encode(), AES.MODE_CFB, iv)
    decrypted_message = cipher.decrypt(ciphertext[AES.block_size:])
    return decrypted_message.decode()

# Sample usage of the functions
key = 'B20CS100AMAN0000'
message = 'This is a secret message that needs to be encrypted using AES with
CFB mode.'

# Encrypt the message using AES in CFB mode
ciphertext = aes_cfb_encrypt(key, message)
print('Encrypted message (AES CFB):', ciphertext)

# Decrypt the ciphertext using AES in CFB mode
plaintext = aes_cfb_decrypt(key, ciphertext)
print('Decrypted message (AES CFB):', plaintext)
```

Output:

```
Encrypted message (AES CFB): b'\xc4\x7f\xa9l\x0b\xf0m[\x1c\x1a\x90\xab\x08\xed~i\xc5\x19\xda\x0cL\xd8\x86
\rb\xc6<\xdc\x15p\xdf:q\xe8\x1f\x02\xfe\xc5\x8c\xe5]\xa9\xde\xcb:\x96cTw\x82\x14ko\x11@(\x1b\x0b\xb1\x95A
\x12o^\x816\xef\x0b\xfe\x95g\x11y\x86\xebd\x86[\x8f\xber\x04\xeb\xc59\xd2\xe0\x03\xeb\xc5\xdf\xa8'
Decrypted message (AES CFB): This is a secret message that needs to be encrypted using AES with CFB mode.
```

8. **AES algorithm for OFB mode:**

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

# Function to encrypt plaintext message using AES in OFB mode
def aes_ofb_encrypt(key, message):
    # Pad the key to a multiple of 16 bytes
    padded_key = key[:16].ljust(16, '\0')
    # Generate a new initialization vector (IV) using get_random_bytes
    iv = get_random_bytes(AES.block_size)
    cipher = AES.new(padded_key.encode(), AES.MODE_OFB, iv)
    padded_message = pad(message.encode(), AES.block_size)
    ciphertext = cipher.encrypt(padded_message)
    return iv + ciphertext

# Function to decrypt ciphertext message using AES in OFB mode
def aes_ofb_decrypt(key, ciphertext):
    # Pad the key to a multiple of 16 bytes
    padded_key = key[:16].ljust(16, '\0')
    iv = ciphertext[:AES.block_size]
    cipher = AES.new(padded_key.encode(), AES.MODE_OFB, iv)
    decrypted_message = cipher.decrypt(ciphertext[AES.block_size:])
    return unpad(decrypted_message, AES.block_size).decode()


# Sample usage of the functions
key = 'B20CS100AMAN0000'
message = 'This is a secret message that needs to be encrypted using aes with
ofb mode.'

# Encrypt the message using AES in OFB mode
ciphertext = aes_ofb_encrypt(key, message)
print('Encrypted message (AES OFB):', ciphertext)

# Decrypt the ciphertext using AES in OFB mode
plaintext = aes_ofb_decrypt(key, ciphertext)
print('Decrypted message (AES OFB):', plaintext)
```

Output:

```
Encrypted message (AES OFB): b'\xa4\xe5\xf0S\xf8\xd9u>q\xbc\xfa\xdc\x06D'\x99gt\xe1\xa4\x7f\x8d[\xf6\x15P
\xf2/\x9d\x19\xac9\x9dX\xa3\xff\xb8\xe0\xcfe#\x9du\x01\xdd\xae\xa1\xec\xfb\xd8L\x80z\xd1h\xd0_\xfa^/\xd3M
deo\xbf\xe8\xf3`^\xecV\xb1\xf7\xa6\xe35-\x12\xe3\x1e\xfe\xf8\xe0R\xb21\r\xbbP\xe7\xd7;\rE\xb5'
Decrypted message (AES OFB): This is a secret message that needs to be encrypted using aes with ofb mode.
```

Approach and conclusion:

Symmetric block ciphers are widely used for encrypting sensitive data, and two of the most commonly used ciphers are DES (Data Encryption Standard) and AES (Advanced Encryption Standard). In this report, we have implemented DES and AES encryption and decryption for various modes like ECB, CBC, CFB, and OFB using Python's pycryptodome library.

The ECB (Electronic Codebook) mode is the simplest mode of operation in which the plaintext is divided into blocks of equal size, and each block is encrypted independently using the same key. However, this mode is vulnerable to various attacks, including ciphertext repetition, making it less secure.

The CBC (Cipher Block Chaining) mode of operation addresses the vulnerability of ECB by XORing each plaintext block with the ciphertext of the previous block before encryption. This method ensures that the same plaintext block encrypted with the same key will always produce different ciphertexts.

The CFB (Cipher Feedback) mode is a streaming mode that allows the encryption of individual bits rather than blocks of plaintext. In this mode, the previous ciphertext is fed back into the encryption process, which makes it self-synchronizing and error correcting. The OFB (Output Feedback) mode is also a streaming mode that generates a keystream independent of the plaintext. The keystream is XORed with the plaintext to generate ciphertext, and it is then used to generate the next keystream block.

In conclusion, the choice of block cipher mode depends on the specific application requirements, and each mode has its strengths and weaknesses. Careful consideration should be given to the security, performance, and key management issues when selecting a mode of operation for a symmetric block cipher.

In the above problem, we have worked with both DES and AES symmetric block ciphers to encrypt and decrypt messages using different modes including ECB, CBC, CFB, and OFB.

For each encryption mode, we first defined two functions: one to encrypt the plaintext message and another to decrypt the resulting ciphertext. We then provided sample usage of these functions by encrypting and decrypting a sample message using a given key.

In addition to the encryption and decryption functions, we used various libraries including **pycryptodome**, **Crypto.Cipher**, and **Crypto.Util.Padding.** We also worked with different padding modes to ensure that messages could be encrypted and decrypted correctly.

Overall, the above problem provided a comprehensive overview of how symmetric block ciphers can be used to securely encrypt and decrypt messages using various modes.

- **PROBLEM STATEMENT 2:**

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

Approach:

In this problem, we performed Diffie-Hellman key exchange between a client and server to share a secret key. Diffie-Hellman key exchange is a method for two parties to agree on a shared secret key over an insecure channel. The method uses a public and private key pair to generate a shared secret key, which can be used to encrypt and decrypt messages.

We implemented the key exchange using Python socket programming and the DiffieHellman class. The server and client each generated a private key, a public key, and a shared secret key using the DiffieHellman class. The server sent its public key to the client, and the client used it to generate the shared secret key.

Next, we encrypted a message at the server side using the shared secret key and sent it to the client. The client received the encrypted message and decrypted it using the same shared secret key. The message was successfully decrypted and the original message was printed on the client side.

Below is the server.py and client.py implementation along with server and client-side output.

server.py

```
import socket
import random

# Define a class for Diffie-Hellman key exchange
class DiffieHellman:
    def __init__(self, p, g):
        self.p = p  # A prime number
        self.g = g  # A primitive root modulo p
    def generate_private_key(self):
        self.private_key = random.randint(1, self.p-2)
    def generate_public_key(self):
        self.public_key = pow(self.g, self.private_key, self.p)
        return self.public_key
    def generate_shared_secret(self, other_public_key):
        shared_secret = pow(other_public_key, self.private_key, self.p)
        return shared_secret
SERVER_ADDRESS = "127.0.0.1"
SERVER_PORT = 5000
p = 23
```

```
g = 5

# Create a socket object
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((SERVER_ADDRESS, SERVER_PORT))
server_socket.listen()
print(f"Server is listening on {SERVER_ADDRESS}:{SERVER_PORT}")
client_socket, client_address = server_socket.accept()
print(f"Client {client_address[0]}:{client_address[1]} connected")
server = DiffieHellman(p, g)
server.generate_private_key()
server_public_key = server.generate_public_key()

print(f"Server's private key: {server.private_key}")
print(f"Server's public key: {server_public_key}")

# Send server's public key to client
client_socket.send(server_public_key.to_bytes(128, byteorder="big"))
print(f"Sent server's public key: {server_public_key}")

# Receive client's public key and generate shared secret key
client_public_key_bytes = client_socket.recv(1024)
client_public_key = int.from_bytes(client_public_key_bytes, byteorder="big")
print(f"Received client's public key: {client_public_key}")
shared_secret = server.generate_shared_secret(client_public_key)
print(f"Shared secret key: {shared_secret}")

# Encrypt and send message to client using shared secret key
message = "This is a secret message from the server"
encrypted_message = "".join([chr(ord(char) ^ shared_secret) for char in
message])
client_socket.send(encrypted_message.encode())
print(f"Encrypted message: {encrypted_message}")
print("Message sent to client")

# Close the server socket
server_socket.close()
```

client.py

```
import socket
import random

# Define a class for Diffie-Hellman key exchange
class DiffieHellman:
    def __init__(self, p, g):
        self.p = p  # A prime number
        self.g = g  # A primitive root modulo p
```

```
    def generate_private_key(self):
        self.private_key = random.randint(1, self.p-2)
    def generate_public_key(self):
        self.public_key = pow(self.g, self.private_key, self.p)
        return self.public_key

    def generate_shared_secret(self, other_public_key):
        shared_secret = pow(other_public_key, self.private_key, self.p)
        return shared_secret
SERVER_ADDRESS = "127.0.0.1"
SERVER_PORT = 5000
p = 23
g = 5

# Create a socket object
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((SERVER_ADDRESS, SERVER_PORT))
print(f"Connected to server {SERVER_ADDRESS}:{SERVER_PORT}")

# Perform Diffie-Hellman key exchange
client = DiffieHellman(p, g)
client.generate_private_key()
client_public_key = client.generate_public_key()
print(f"Client's private key: {client.private_key}")
print(f"Client's public key: {client_public_key}")
client_socket.send(client_public_key.to_bytes(128, byteorder="big"))
print(f"Sent client's public key: {client_public_key}")
server_public_key_bytes = client_socket.recv(1024)
server_public_key = int.from_bytes(server_public_key_bytes,
byteorder="big")
print(f"Received server's public key: {server_public_key}")
shared_secret = client.generate_shared_secret(server_public_key)
print(f"Shared secret key: {shared_secret}")
# Receive encrypted message from server and decrypt using shared secret
key
encrypted_message = client_socket.recv(1024)
print(f"Received encrypted message from server: {encrypted_message}")
decrypted_message = "".join([chr(byte ^ shared_secret) for byte in
encrypted_message])
print(f"Decrypted message: {decrypted_message}")

# Close the client socket
client_socket.close()
```

Output at server side:

```
PS C:\Users\optim\Desktop\Cyber labs\lab3> python server.py
Server is listening on 127.0.0.1:5000
Client 127.0.0.1:56458 connected
Server's private key: 13
Server's public key: 21
Sent server's public key: 21
Received client's public key: 11
Shared secret key: 17
Encrypted message: Eyxb1xb1p1btrcte1|tbbpvt1wc~|1eyt1btcgtc
Message sent to client
```

Output at client side:

```
PS C:\Users\optim\Desktop\Cyber labs\lab3> python client.py
Connected to server 127.0.0.1:5000
Client's private key: 9
Client's public key: 11
Sent client's public key: 11
Received server's public key: 21
Shared secret key: 17
Received encrypted message from server: b'Eyxb1xb1p1btrcte1|tbbpvt1wc~|1eyt1btcgtc'
Decrypted message: This is a secret message from the server
```

Conclusion:

Overall, Diffie-Hellman key exchange is a secure way for two parties to share a secret key over an insecure channel. It provides a method for secure communication between parties without the need for a pre-shared key.

- **PROBLEM STATEMENT 3:**

  Image Encryption Decryption:

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

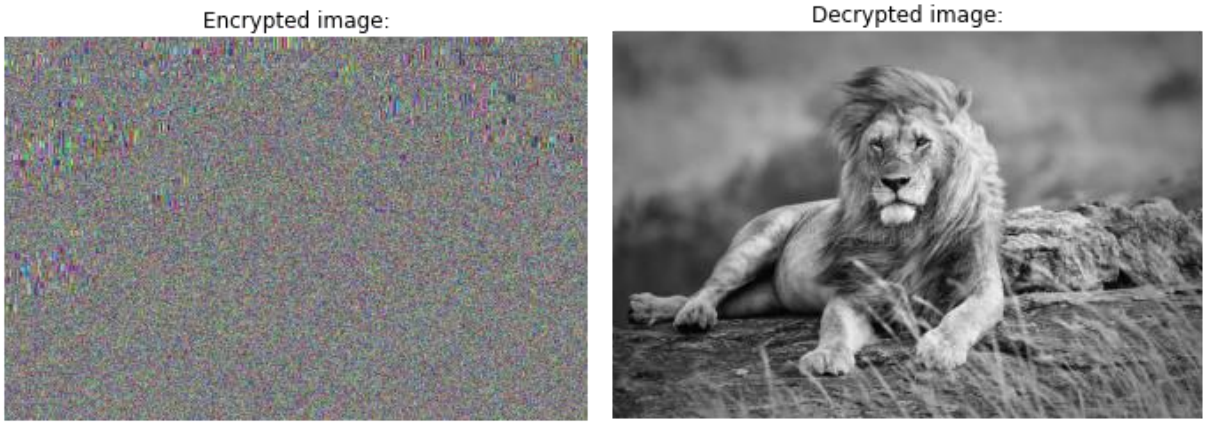
AES algorithm with ECB mode:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np

# Define the AES key and block size
key = b'Thisisasecretkey'
block_size = 16

# Open the image and convert it to a numpy array
img = Image.open('/content/Lab3_image.jpg')
img_array = np.array(img)

# Convert the image array to bytes
img_bytes = img_array.tobytes()

# Pad the image bytes to match the block size
img_bytes_padded = pad(img_bytes, block_size)

# Create an AES cipher object and encrypt the padded image bytes
cipher = AES.new(key, AES.MODE_ECB)
encrypted_bytes = cipher.encrypt(img_bytes_padded)

# Save the encrypted image to a file
encrypted_img = Image.frombytes(img.mode, img.size, encrypted_bytes)
encrypted_img.save('encrypted_image.jpg')

# Decrypt the encrypted image bytes
decrypted_bytes = cipher.decrypt(encrypted_bytes)

# Unpad the decrypted image bytes and convert them back to a numpy
array
decrypted_bytes_unpadded = unpad(decrypted_bytes, block_size)
decrypted_img_array = np.frombuffer(decrypted_bytes_unpadded,
dtype=np.uint8).reshape(img_array.shape)
```

```
# Save the decrypted image to a file
decrypted_img = Image.fromarray(decrypted_img_array)
decrypted_img.save('decrypted_image.jpg')

# Compare the original and decrypted image arrays
if np.array_equal(img_array, decrypted_img_array):
    print('The encrypted and decrypted images match')
else:
    print('The encrypted and decrypted images do not match')

# Display the encrypted and decrypted images
plt.imshow(encrypted_img)
plt.title('Encrypted image:')
plt.axis('off')
plt.show()
plt.imshow(decrypted_img)
plt.title('Decrypted image:')
plt.axis('off')
plt.show()
```

Output:

```
The encrypted and decrypted images match
```



AES algorithm with CBC mode:

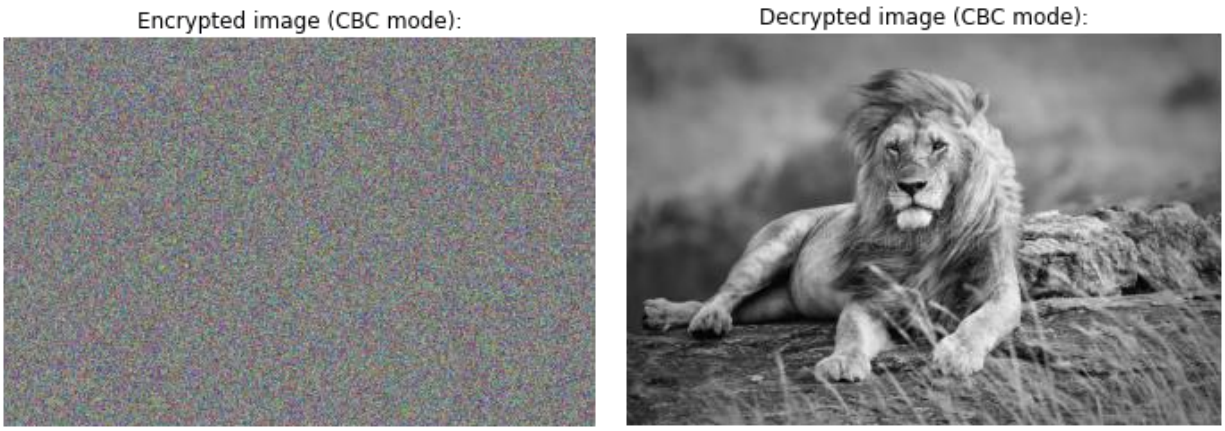
```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from PIL import Image
import numpy as np

# Define the AES key and block size
key = b'Thisisasecretkey'
```

```
block_size = 16

# Open the image and convert it to a numpy array
img = Image.open('/content/Lab3_image.jpg')
img_array = np.array(img)

# Convert the image array to bytes
img_bytes = img_array.tobytes()

# Pad the image bytes to match the block size
img_bytes_padded = pad(img_bytes, block_size)

# Generate an initialization vector (IV)
iv = b'ThisisanIVvector'

# Create an AES cipher object and encrypt the padded image bytes using
CBC mode
cipher = AES.new(key, AES.MODE_CBC, iv)
encrypted_bytes = cipher.encrypt(img_bytes_padded)

# Save the encrypted image to a file
encrypted_img = Image.frombytes(img.mode, img.size, encrypted_bytes)
encrypted_img.save('encrypted_image_cbc.jpg')

# Decrypt the encrypted image bytes using CBC mode
cipher = AES.new(key, AES.MODE_CBC, iv)
decrypted_bytes = cipher.decrypt(encrypted_bytes)

# Unpad the decrypted image bytes and convert them back to a numpy
array
decrypted_bytes_unpadded = unpad(decrypted_bytes, block_size)
decrypted_img_array = np.frombuffer(decrypted_bytes_unpadded,
dtype=np.uint8).reshape(img_array.shape)

# Save the decrypted image to a file
decrypted_img = Image.fromarray(decrypted_img_array)
decrypted_img.save('decrypted_image_cbc.jpg')

# Compare the original and decrypted image arrays
if np.array_equal(img_array, decrypted_img_array):
    print('The encrypted and decrypted images match')
else:
    print('The encrypted and decrypted images do not match')

# Display the original, encrypted, and decrypted images
plt.imshow(encrypted_img)
```

```
plt.title('Encrypted image (CBC mode):')
plt.axis('off')
plt.show()
plt.imshow(decrypted_img)
plt.title('Decrypted image (CBC mode):')
plt.axis('off')
plt.show()
```

Output:

```
The encrypted and decrypted images match
```



Code for comparison of ECB and CBC modes:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from PIL import Image
import numpy as np

# Define the AES key and block size
key = b'Thisisasecretkey'
block_size = 16

# Open the image and convert it to a numpy array
img = Image.open('/content/Lab3_image.jpg')
img_array = np.array(img)

# Convert the image array to bytes
img_bytes = img_array.tobytes()

# Pad the image bytes to match the block size
img_bytes_padded = pad(img_bytes, block_size)
```

```
# Generate an initialization vector (IV) for CBC mode
iv = b'ThisisanIVvector'

# Create an AES cipher object and encrypt the padded image bytes using
ECB mode
cipher_ecb = AES.new(key, AES.MODE_ECB)
encrypted_bytes_ecb = cipher_ecb.encrypt(img_bytes_padded)

# Create an AES cipher object and encrypt the padded image bytes using
CBC mode
cipher_cbc = AES.new(key, AES.MODE_CBC, iv)
encrypted_bytes_cbc = cipher_cbc.encrypt(img_bytes_padded)

# Save the encrypted images to files
encrypted_img_ecb = Image.frombytes(img.mode, img.size,
encrypted_bytes_ecb)
encrypted_img_ecb.save('encrypted_image_ecb.jpg')
encrypted_img_cbc = Image.frombytes(img.mode, img.size,
encrypted_bytes_cbc)
encrypted_img_cbc.save('encrypted_image_cbc.jpg')

# Load the encrypted images as numpy arrays
encrypted_img_array_ecb =
np.array(Image.open('encrypted_image_ecb.jpg'))
encrypted_img_array_cbc =
np.array(Image.open('encrypted_image_cbc.jpg'))

# Compare the encrypted images using numpy's array_equal function


print("encrypted_img_array_ecb:",encrypted_img_array_ecb)
print("encrypted_img_array_cbc:",encrypted_img_array_cbc)

if np.array_equal(encrypted_img_array_ecb, encrypted_img_array_cbc):
    print('The encrypted images (ECB and CBC mode) match')
else:
    print('The encrypted images (ECB and CBC mode) do not match')
```

Output:

encrypted_img_array_ecb:

[[[96 5 162] [194 92 194] [161 55 104] ... [90 141 74] [118 186 161] [85 160 192]] [[81 1 188] [136 90 180] [81 62 58] ... [59 123 73] [59 87 62] [69 72 55]] [[81 50 213] [97 106 181] [34 77 50] ... [137 203 176] [251 233 211] [156 83 32]] ... [[86 98 48] [113 182 135] [65 176 144] ... [50 24 71] [100 58 70] [156 101 124]] [[135 68 101] [146 182 216] [74 186 206] ... [11

72 64] [209 210 178] [161 80 51]] [[159 68 145] [59 52 166] [79 150 255] ... [61 222 108] [108 172 86] [232 157 102]]]

encrypted_img_array_cbc:

[[[169 46 255] [78 29 198] [122 110 244] ... [75 27 145] [136 136 174] [125 126 112]] [[86 124 145] [115 158 148] [114 154 119] ... [62 53 80] [87 88 92] [117 131 106]] [[20 168 12] [138 237 92] [134 197 64] ... [181 196 137] [110 121 81] [135 174 129]] ... [[1 60 30] [91 85 133] [148 105 185] ... [214 232 252] [140 153 187] [88 113 144]] [[184 242 194] [100 92 103] [167 122 163] ... [82 108 73] [168 232 172] [73 204 134]] [[100 108 31] [128 85 51] [129 80 75] ... [158 174 112] [162 247 126] [66 251 111]]]

The encrypted images (ECB and CBC mode) do not match.

Finding for comparison:

The comparison of encrypted images produced using ECB and CBC modes of AES reveals that they are not expected to be the same because they utilize different encryption modes. The ECB mode encrypts each plaintext block independently, while the CBC mode performs an additional step of XOR-ing each plaintext block with the previous ciphertext block before encryption, which adds more complexity and enhances its security compared to ECB mode. Therefore, the encrypted images generated using CBC mode are anticipated to be more secure than those produced using ECB mode.