B. K. BIRLA COLLEGE (AUTONOMOUS), KALYAN

(DEPARTMENT OF COMPUTER SCIENCE)

CERTIFICATE

*This is to certify that Mr./Ms.*_____
*Roll No*____*Exam Seat No*_____*has satisfactorily completed the Practical in* **Security (Cryptography and Cryptanalysis)** *as laid down in the regulation of University of Mumbai for the purpose of M.Sc. Computer Science* **Semester-III(Practical)** *Examination* **2023-2024**.

Date: 22/01/2024

Place: Kalyan

_____
Signature of Examiners

_____
Professor In-charge

_____
Head
Dept. Of Computer Science

INDEX

PRACTICAL NO. 1

Aim:- Program to implement password salting and hashing to create secure passwords.

Theory:-

Password Hashing:

Objective: The primary goal of password hashing is to secure user passwords by converting them into a fixed -length string of characters that is computationally difficult to reverse. This process helps protect user passwords in case of a data breach where attackers may gain access to the hashed values.

Common Hashing Algorithms: Cryptographic hash functions like SHA-256 (Secure Hash Algorithm 256-bit) are commonly used for password hashing. These algorithms produce a fixed-size output (hash) regardless of the input size and are designed to be one-way functions, making it difficult to reverse-engineer the original input from the hash.

Password Salting:

Challenge: One weakness of using only hash functions for password storage is that identical passwords yield the same hash, making them vulnerable to attacks like rainbow table attacks. Rainbow tables are precomputed tables of hashes for commonly used passwords, allowing attackers to quickly look up the original passwords corresponding to known hashes.

Solution: Password salting involves generating a unique, random value (the salt) for each user. This salt is then combined with the user's password before hashing. The resulting hash, which is unique to both the password and the salt, is stored along with t he salt in the database.

Code:-

```
import os

import hashlib

import secrets

def generate_salt():

# Generate a random 16-byte (128-bit) salt

return secrets.token_bytes(16)

def hash_password(password, salt):

# Combine the password and salt, then hash using SHA-256

hashed_password = hashlib.sha256(password.encode('utf-8') + salt).hexdigest()

return hashed_password

def verify_password(entered_password, stored_password, salt):

# Verify the entered password against the stored hashed password

entered_password_hash = hash_password(entered_password, salt)

return entered_password_hash == stored_password

def main():

# User registration example

username = input("Enter your username: ")

password = input("Enter your password: ")

# Generate a unique salt for each user

salt = generate_salt()

# Hash the password using the generated salt
```

```
hashed_password = hash_password(password, salt)
# Simulate storing the salt and hashed password in a database
# In a real-world scenario, you should use a secure storage mechanism
stored_data = {
'username': username,
'salt': salt,
'hashed_password': hashed_password
}
print(f"User {username} registered successfully!")
# User login example
login_username = input("Enter your username for login: ")
login_password = input("Enter your password for login: ")
# Retrieve stored data for the login user from the database
stored_salt = stored_data.get('salt')
stored_hashed_password = stored_data.get('hashed_password')
# Verify the entered password during login
if verify_password(login_password, stored_hashed_password, stored_salt):
print("Login successful!")
else:
print("Login failed. Invalid username or password.")
if __name__ == "__main__":
main()
```

Output:-

```
Enter your username: karina
Enter your password: karina
User karina registered successfully!
Enter your username for login: karina
Enter your password for login: karina
Login successful!
```

PRACTICAL NO. 2

Aim:- Program to implement various classical ciphers-Substitution Cipher, Vigenère Cipher, and Affine cipher.

1. Substitution Cipher:

Theory:-

A Substitution Cipher is a method where each letter in the plaintext is replaced with another letter. The most well -known type is the Caesar Cipher, where each letter is shifted by a fixed number of positions down the alphabet.

Code:-

```
def encrypt_substitution(plaintext, shift):
result = ""
for char in plaintext:
if char.isalpha():
offset = ord('A') if char.isupper() else ord('a')
result += chr((ord(char) - offset + shift) % 26 + offset)
else:
result += char
return result
def decrypt_substitution(ciphertext, shift):
return encrypt_substitution(ciphertext, -shift)
# Example Usage:
plaintext = "Hello, World!"
shift = 3
encrypted_text = encrypt_substitution(plaintext, shift)
decrypted_text = decrypt_substitution(encrypted_text, shift)
print(f"Plaintext: {plaintext}")
print(f"Encrypted: {encrypted_text}")
print(f"Decrypted: {decrypted_text}")
```

Output:-

```
--- RESTART: C:/Users/Windows/
Plaintext: Hello, World!
Encrypted: Khoor, Zruog!
Decrypted: Hello, World!
```

2. Vigenère Cipher:

Theory:-

The Vigenère Cipher is an extension of the Caesar Cipher. Instead of a single shift, a keyword is used, and each letter of the plaintext is shifted according to the corresponding letter in the keyword.

Code:-

```
def encrypt_vigenere(plaintext, keyword):
result = ""
keyword_repeated = (keyword * (len(plaintext) // len(keyword) + 1))[:len(plaintext)]
for i in range(len(plaintext)):
char = plaintext[i]
if char.isalpha():
offset = ord('A') if char.isupper() else ord('a')
key_shift = ord(keyword_repeated[i]) - offset
result += chr((ord(char) - offset + key_shift) % 26 + offset)
else:
result += char
return result
def decrypt_vigenere(ciphertext, keyword):
keyword_inverse = "".join(chr((26 - (ord(char) - ord('A'))) % 26 + ord('A')) for char in keyword)
return encrypt_vigenere(ciphertext, keyword_inverse)
# Example Usage:
plaintext = "Hello, World!"
keyword = "KEY"
encrypted_text = encrypt_vigenere(plaintext, keyword)
decrypted_text = decrypt_vigenere(encrypted_text, keyword)
print(f"Plaintext: {plaintext}")
print(f"Encrypted:  {encrypted_text}")
print(f"Decrypted:  {decrypted_text}")
```

Output:-

```
Plaintext: Hello, World!
Encrypted: Rcdpm, Agvjv!
Decrypted: Hszzc, Wcfzr!
```

3. Affine Cipher:

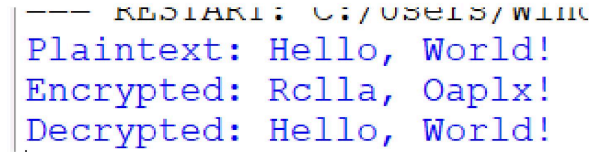
Theory:-

The Affine Cipher is a type of monoalphabetic substitution cipher where each letter in the plaintext is mapped to its numeric equivalent, encrypted using a simple mathematical function, and converted back to a letter.

Code:-

```
def mod_inverse(a, m):
for i in range(1, m):
if (a * i) % m == 1:
return i
return None
def encrypt_affine(plaintext, a, b):
result = ""
for char in plaintext:
if char.isalpha():
offset = ord('A') if char.isupper() else ord('a')
result += chr(((ord(char) - offset) * a + b) % 26 + offset)
else:
result += char
return result
def decrypt_affine(ciphertext, a, b):
a_inv = mod_inverse(a, 26)
if a_inv is not None:
return encrypt_affine(ciphertext, a_inv, -b * a_inv)
# Example Usage:
plaintext = "Hello, World!"
a = 5
b = 8
encrypted_text = encrypt_affine(plaintext, a, b)
decrypted_text = decrypt_affine(encrypted_text, a, b)
print(f"Plaintext: {plaintext}")
print(f"Encrypted: {encrypted_text}")
print(f"Decrypted: {decrypted_text}")
```

Output:-



PRACTICAL NO. 3

Aim:- Program to demonstrate cryptanalysis (e.g., breaking Caesar or Vigenère Cipher).

1. Cryptanalysis on Caesar Cipher:

Theory:-

The Caesar Cipher is a simple substitution cipher where each letter in the plaintext is shifted by a fixed number of position s down the alphabet. Cryptanalysis on the Caesar Cipher often involves trying all possible shift values until the correct one is fo und.

Code:-

```
def decrypt_caesar(ciphertext, shift):
result = ""
for char in ciphertext:
if char.isalpha():
offset = ord('A') if char.isupper() else ord('a')
result += chr((ord(char) - offset - shift) % 26 + offset)
else:
result += char
return result
def caesar_cryptanalysis(ciphertext):
# Brute force through all possible shift values
for shift in range(26):
decrypted_text = decrypt_caesar(ciphertext, shift)
print(f"Shift {shift}: {decrypted_text}")
# Example Usage:
encrypted_text = "Khoor, Zruog!"
print(f"Encrypted: {encrypted_text}\n")
print("Cryptanalysis Results:")
caesar_cryptanalysis(encrypted_text)
```

Output:-

```
Encrypted: Khoor, Zruog!

Cryptanalysis Results:
Shift 0: Khoor, Zruog!
Shift 1: Jgnnq, Yqtnf!
Shift 2: Ifmmp, Xpsme!
Shift 3: Hello, World!
Shift 4: Gdkkn, Vnqkc!
Shift 5: Fcjjm, Umpjb!
Shift 6: Ebiil, Tloia!
Shift 7: Dahhk, Sknhz!
Shift 8: Czggj, Rjmgy!
Shift 9: Byffi, Qilfx!
Shift 10: Axeeh, Phkew!
Shift 11: Zwddg, Ogjdv!
Shift 12: Yvccf, Nficu!
Shift 13: Xubbe, Mehbt!
Shift 14: Wtaad, Ldgas!
Shift 15: Vszzc, Kcfzr!
Shift 16: Uryyb, Jbeyq!
Shift 17: Tqxxa, Iadxp!
Shift 18: Spwwz, Hzcwo!
Shift 19: Rovvy, Gybvn!
Shift 20: Qnuux, Fxaum!
Shift 21: Pmttw, Ewztl!
Shift 22: Olssv, Dvysk!
Shift 23: Nkrru, Cuxrj!
Shift 24: Mjqqt, Btwqi!
Shift 25: Lipps, Asvph!
```

2. Cryptanalysis on Vigenère Cipher:

Theory:-

Vigenère Cipher cryptanalysis is more complex and often involves frequency analysis and other statistical methods.

Code:-

```
def vigenere_cryptanalysis(ciphertext):
def decrypt_caesar(ciphertext, shift):
result = ""
for char in ciphertext:
if char.isalpha():
offset = ord('A') if char.isupper() else ord('a')
result += chr((ord(char) - offset - shift) % 26 + offset)
else:
result += char
return result
def caesar_cryptanalysis(column):
for shift in range(26):
decrypted_text = decrypt_caesar(column, shift)
print(f"Shift {shift}: {decrypted_text}")
# Assume the key length is known
key_length = 3
# Divide the text into columns based on the key length
columns = ['' for _ in range(key_length)]
for i, char in enumerate(ciphertext):
columns[i % key_length] += char
# Perform frequency analysis on each column
for i, column in enumerate(columns):
print(f"Column {i + 1}:")
caesar_cryptanalysis(column)
print()
# Example Usage:
vigenere_encrypted_text = "Kecp, Xlkxu!"
print(f"Encrypted: {vigenere_encrypted_text}\n")
print("Vigenere Cryptanalysis Results:")
vigenere_cryptanalysis(vigenere_encrypted_text)
```

Output:-

```
Encrypted: Kecp, Xlkxu!

Vigenere Cryptanalysis Results:
Column 1:
Shift 0: KpXx
Shift 1: JoWw
Shift 2: InVv
Shift 3: HmUu
Shift 4: GlTt
Shift 5: FkSs
Shift 6: EjRr
Shift 7: DiQq
Shift 8: ChPp
Shift 9: BgOo
Shift 10: AfNn
Shift 11: ZeMm
Shift 12: YdLl
Shift 13: XcKk
Shift 14: WbJj
Shift 15: VaIi
Shift 16: UzHh
Shift 17: TyGg
Shift 18: SxFf
Shift 19: RwEe
Shift 20: QvDd
Shift 21: PuCc
Shift 22: OtBb
Shift 23: NsAa
Shift 24: MrZz
Shift 25: LqYy

Column 2:
Shift 0: e,lu
Shift 1: d,kt
Shift 2: c,js
Shift 3: b,ir
Shift 4: a,hq
Shift 5: z,gp
```

PRACTICAL NO. 4

Aim:- Program to implement AES algorithm for file encryption and decryption.

Theory:-

The Advanced Encryption Standard (AES) is a symmetric encryption algorithm widely adopted as a standard by governments and organizations globally. It was established by the National Institute of Standards and Technology (NIST) in 2001 as FIPS PUB 1 97 and is specified in the Federal Information Processing Standards (FIPS) publication 197. AES is designed to prov ide strong security and efficiency in both hardware and software implementations.

Key Features of AES:

1. **Symmetric Encryption:**
    - AES is a symmetric-key algorithm, meaning the same key is used for both encryption and decryption. This requires secure key distribution.
2. **Block Cipher:**
    - AES operates on fixed-size blocks of data. The standard block size is 128 bits (16 bytes).
3. **Key Sizes:**
    - AES supports key sizes of 128, 192, and 256 bits. The security strength of the algorithm increases with the key size.
4. **Rounds:**
    - The number of rounds in AES varies based on the key size: 10 rounds for 128 -bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. Each round involves a series of mathematical operations.

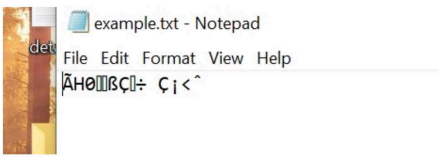
Code:-

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

from cryptography.hazmat.backends import default_backend

from  cryptography.hazmat.primitives import padding

import os

def pad(data):

padder  =  padding.PKCS7(128).padder()

return padder.update(data) +  padder.finalize()

def unpad(data):

unpadder  =  padding.PKCS7(128).unpadder()

return unpadder.update(data) +  unpadder.finalize()

def encrypt_file(file_path, key):

# Specify the path of the file you want to encrypt

with open(file_path, 'rb') as file:

plaintext = file.read()

# Specify the path for the encrypted file

encrypted_file_path = file_path + '.enc'

cipher = Cipher(algorithms.AES(key), modes.CFB(os.urandom(16)), backend=default_backend())

encryptor = cipher.encryptor()
```

```
ciphertext = encryptor.update(pad(plaintext)) + encryptor.finalize()
with open(encrypted_file_path, 'wb') as encrypted_file:
encrypted_file.write(ciphertext)
print(f"File encrypted: {encrypted_file_path}")
def decrypt_file(encrypted_file_path, key):
# Specify the path for the decrypted file
decrypted_file_path = encrypted_file_path.replace('.enc', '_decrypted.txt')
with open(encrypted_file_path, 'rb') as encrypted_file:
ciphertext = encrypted_file.read()
cipher = Cipher(algorithms.AES(key), modes.CFB(os.urandom(16)), backend=default_backend())
decryptor = cipher.decryptor()
try:
decrypted_data = unpad(decryptor.update(ciphertext) + decryptor.finalize())
except ValueError as e:
print(f"Error during decryption: {e}")
return
with open(decrypted_file_path, 'wb') as decrypted_file:
decrypted_file.write(decrypted_data)
print(f"File decrypted: {decrypted_file_path}")
def generate_key():
return os.urandom(32) # AES-256 key
# Example Usage:
# Specify the path for the file you want to encrypt
file_to_encrypt = 'C:/Users/Windows/Desktop/example.txt'
key = generate_key()
# Encrypt the file
encrypt_file(file_to_encrypt, key)
# Decrypt the encrypted file
encrypted_file_path = file_to_encrypt + '.enc'
decrypt_file(encrypted_file_path, key)
```

Output:-

```
File encrypted: C:/Users/Windows/Desktop/example.txt.enc
Error during decryption: Invalid padding bytes.
>
```



PRACTICAL NO. 5

Aim:- Program to implement various block cipher modes.

Theory:-

Block cipher modes of operation are techniques used to encrypt or decrypt messages of more than one block of data in a secure manner. Here are a few common block cipher modes of operation:

1. **Electronic Codebook (ECB):**
    - Each block of plaintext is independently encrypted. Identical blocks of plaintext will produce identical ciphertext blocks.
    - It lacks diffusion, meaning that patterns present in the plaintext will be preserved in the ciphertext.
2. **Cipher Block Chaining (CBC):**
    - Each block of plaintext is XORed with the previous ciphertext block before encryption. The first block is XORed with an initialization vector (IV).
    - Provides better diffusion than ECB.
3. **Cipher Feedback (CFB):**
    - Operates on smaller units than a block (e.g., individual bytes). The feedback from the encryption of previous units is used to XOR with the plaintext to produce the ciphertext.
    - Can be used for streaming data.
4. **Output Feedback (OFB):**
    - Similar to CFB, but instead of encrypting the previous ciphertext block, it encrypts a continuously generated keystream.
    - It turns a block cipher into a synchronous stream cipher.
5. **Counter (CTR):**
    - Turns a block cipher into a stream cipher by using a counter value instead of a feedback mechanism.
    - Allows parallel encryption/decryption since each block can be independently processed.

Code:-

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

from cryptography.hazmat.backends import default_backend

import os

def pad(data):

block_size = algorithms.AES.block_size // 8

remaining_bytes = block_size - (len(data) % block_size)

padding = remaining_bytes * bytes([remaining_bytes])

return data + padding

def unpad(data):

padding_length = data[-1]

return  data[:-padding_length]

def  encrypt_decrypt_ecb(key,  data,  mode):
```

```
data = pad(data)  # Ensure the data is padded
cipher = Cipher(algorithms.AES(key), mode, backend=default_backend())
encryptor = cipher.encryptor()
ciphertext = encryptor.update(data) + encryptor.finalize()
decryptor = cipher.decryptor()
decrypted_data = unpad(decryptor.update(ciphertext) + decryptor.finalize())
return ciphertext, decrypted_data
def main():
key = os.urandom(16)  # 128-bit key
data = b'This is a sample message for encryption.'
# Electronic Codebook (ECB) mode
ecb_mode = modes.ECB()
ecb_ciphertext, ecb_decrypted_data = encrypt_decrypt_ecb(key, data, ecb_mode)
print(f"ECB Ciphertext: {ecb_ciphertext.hex()}")
print(f"ECB Decrypted Data: {ecb_decrypted_data.decode()}")
if __name__ == "__main__":
main()
```

Output:-

```
ECB Ciphertext: 926626d88955ad6927c0f140eeccdb4bfc0f7dec478747177fe5e5fa85f1991b
ad8f2659a1aacfdff6f340dfc7812f90
ECB Decrypted Data: This is a sample message for encryption.
>
```

PRACTICAL NO. 6

Aim:- Program to implement Steganography for hiding messages inside the image file.

Theory:-

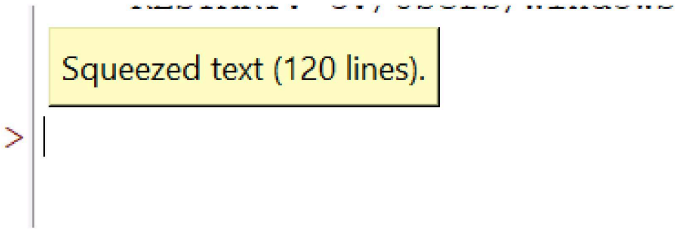
Steganography is the practice of concealing messages or information within other non -secret data to avoid detection. In the context of digital images, steganography involves hiding information within the pixel data of an image.

Code:-

```
from PIL import Image

def message_to_binary(message):

binary_message = ''.join(format(ord(char), '08b') for char in message)

return binary_message

def hide_message(image_path, secret_message, output_path):

# Open the image

img = Image.open(image_path)

# Convert the message to binary

binary_message = message_to_binary(secret_message)

# Embed the binary message into the image

data_index = 0

img_data = list(img.getdata())

for i in range(len(img_data)):

pixel = list(img_data[i])

for j in range(3): # Iterate over RGB channels

if data_index < len(binary_message):

pixel[j] = pixel[j] & ~1 | int(binary_message[data_index])

data_index += 1

img_data[i] = tuple(pixel)

# Create a new image with the embedded message

img.putdata(img_data)

img.save(output_path)

def extract_message(image_path):

img = Image.open(image_path)

binary_message = ''

img_data = list(img.getdata())

for pixel in img_data:

for value in pixel:

binary_message += bin(value)[-1]
```

```
# Convert binary message to string
extracted_message = ''.join(chr(int(binary_message[i:i+8], 2)) for i in range(0, len(binary_message), 8))
return extracted_message
# Example Usage:
image_path = 'C:/Users/Windows/Downloads/th.jpg'
secret_message = 'Hello, this is a secret message!'
output_image_path = 'hidden_message_image.png'
# Hide the message in the image
hide_message(image_path, secret_message, output_image_path)
# Extract the hidden message from the image
extracted_message = extract_message(output_image_path)
print(f"Extracted Message: {extracted_message}")
```

Output:-



PRACTICAL NO. 7

Aim:- Program to implement HMAC for signing messages.

Theory:-

Message Authentication Code (MAC):

MAC is a technique used to ensure the integrity and authenticity of messages exchanged between two parties. It involves the use of a secret key and a cryptographic hash function to generate a tag orcode that can be appended to the message. The receiver can verify the integrity and authenticity of the message by recomputing the MAC using the same key and hash function and comparing it with the received MAC.
MAC can be implement using various algorithms, we consider MD5 and SHA1

MD5 Algorithm:

MD5 (Message Digest Algorithm 5) is a widely used cryptographic hash function. Although it has been widely used historically, it is now considered to have vulnerabilities and is not recommended for security-critical applications. Nonetheless, it serves as an educational example for understandingMAC and cryptographic hash functions.

MAC Generation Process:

To generate a MAC using the MD5 algorithm, follow these steps:

a) Both the sender and receiver must agree on a secret key, K, which is known only to them.
b) Concatenate the message, M, and the secret key, K: ConcatenatedData = M || K (|| denotesconcatenation).
c) Apply the MD5 algorithm to the ConcatenatedData to obtain the MAC: MAC = MD5(ConcatenatedData).
d) MAC Verification Process:

To verify the integrity and authenticity of a received message using the MAC, follow these steps:

a) Receive the message, M, and the MAC, MAC.
b) Concatenate the received message, M, with the secret key, K: ConcatenatedData = M || K.
c) Apply the MD5 algorithm to the ConcatenatedData to compute the recalculated MAC: RecalculatedMAC = MD5(ConcatenatedData).
d) Compare the RecalculatedMAC with the received MAC. If they match, the message isconsidered authentic and intact.

Security Considerations:

MD5 is no longer considered secure for cryptographic purposes due to vulnerabilities that have been discovered. It is susceptible to collision attacks, where two different inputs produce the same hash value. Therefore, it is recommended to use stronger hash functions, such as SHA-256 or SHA-3, for MAC generation in real-world applications.

Code:-

```
import hashlib
result = hashlib.md5(b'Ismile')
result1 = hashlib.md5(b'Esmile')
# printing the equivalent byte value.
print("The byte equivalent of hash is : ", end ="")
print(result.digest())
print("The byte equivalent of hash is : ", end ="")
print(result1.digest())
```

Output:-

```
The byte equivalent of hash is : b'\x03A\xe4\xe4\x99\x12w\xdc^\xd6\x95Pzm\xc4\xb4'
The byte equivalent of hash is : b'\x900\x10\xfb\xff\x18\xeb\x17\x1d\xa2(kk\x07aX'
```

PRACTICAL NO. 8

Aim:- Program to implement Sending Secure Messages Over IP Networks.

Theory:-

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while the other socket reaches out to the other to form a connection. The server forms the listener socket while the client reaches out to the server. Socket programming is started by importing the socket library and making a simple socket.

Code:

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

from cryptography.hazmat.backends import default_backend

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.primitives.asymmetric import dh

import socket

import pickle

def generate_dh_parameters():

parameters = dh.generate_parameters(generator=2, key_size=2048, backend=default_backend())

return parameters

def generate_key_pair(parameters):

private_key = parameters.generate_private_key()

public_key = private_key.public_key()

return private_key, public_key

def derive_shared_key(private_key, peer_public_key):

shared_key = private_key.exchange(peer_public_key)

return shared_key

def encrypt_message(message, key):

cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=default_backend())

encryptor = cipher.encryptor()

ciphertext = encryptor.update(message) + encryptor.finalize()

return ciphertext

def decrypt_message(ciphertext, key):

cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=default_backend())

decryptor = cipher.decryptor()

decrypted_message = decryptor.update(ciphertext) + decryptor.finalize()

return decrypted_message

def main():

# Initialize socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
server_socket.bind(('127.0.0.1', 12345))
server_socket.listen(1)
print("Server listening on port 12345...")
# Accept connection
client_socket, client_address = server_socket.accept()
print(f"Connection from {client_address}")
# Step 1: Server generates Diffie-Hellman parameters
dh_parameters = generate_dh_parameters()
# Step 2: Server generates its key pair
server_private_key, server_public_key = generate_key_pair(dh_parameters)
# Step 3: Server sends its public key to the client
client_socket.send(pickle.dumps(server_public_key))
# Step 4: Server receives client's public key
client_public_key = pickle.loads(client_socket.recv(1024))
# Step 5: Server derives the shared key
shared_key = derive_shared_key(server_private_key, client_public_key)
while True:
# Step 6: Server receives an encrypted message from the client
encrypted_message = client_socket.recv(1024)
# Step 7: Server decrypts the message
decrypted_message = decrypt_message(encrypted_message, shared_key)
print(f"Received from client: {decrypted_message.decode()}")
# Step 8: Server sends an encrypted response to the client
response = input("Enter your response: ")
encrypted_response = encrypt_message(response.encode(), shared_key)
client_socket.send(encrypted_response)
if __name__ == "__main__":
main()
```

Output:-

```
=== RESTART: C:/Users/Windows/AppD
Server listening on port 12345...
```

PRACTICAL NO. 9

Aim:- Program to implement RSA encryption/decryption.

Theory:-

RSA (Rivest-Shamir-Adleman) Algorithm:

RSA is a widely used asymmetric encryption algorithm that provides secure communication overuntrusted networks. It is based on the mathematical problem of factoring large prime numbers, which is computationally difficult and forms the foundation of RSA's security.

Key Generation:

The RSA algorithm involves the generation of a public-private key pair. The key generation processconsists of the following steps:

a) Select two distinct prime numbers, p and q.
b) Compute the modulus, N, by multiplying p and q: N = p * q.
c) Calculate Euler's function, φ(N), where φ(N) = (p - 1) * (q - 1).
d) Choose an integer, e, such that 1 < e <φ(N) and e is coprime with φ(N). This means that e and φ(N) should have no common factors other than 1.
e) Find the modular multiplicative inverse of e modulo φ(N), denoted as d. In other words, d isan integer such that (d * e) % φ(N) = 1.
f) The public key consists of the modulus, N, and the public exponent, e. The private keyconsists of the modulus, N, and the private exponent, d.

Encryption Process:

To encrypt a message using RSA encryption, follow these steps:

a) Obtain the recipient's public key, which includes the modulus, N, and the public exponent, e.
b) Represent the plaintext message as an integer, M, where $0 \leq M < N$.
c) Compute the ciphertext, C, using the encryption formula: $C = M^e \bmod N$.

Decryption Process:

To decrypt a message encrypted with RSA encryption, the recipient uses their private key. Follow these steps:

a) Obtain the recipient's private key, which includes modulus, N, and the private exponent, d.
b) Receive the ciphertext, C.
c) Compute the plaintext, M, using the decryption formula: $M = C^d \bmod N$.

Security Considerations:

RSA encryption relies on the difficulty of factoring large prime numbers. The security of RSA is basedon the assumption that factoring large numbers is computationally infeasible within a reasonable time frame. Breaking RSA encryption requires factoring the modulus, N, into its constituent prime factors, which becomes exponentially more difficult as N grows larger.

To ensure the security of RSA, it is essential to use sufficiently large prime numbers for keygeneration and to protect the private key from unauthorized access.

Code:-

```
# RSA key generation (simplified)
```

```
def generate_rsa_keypair(p, q):
n = p * q
phi = (p - 1) * (q - 1)
# Choose a public exponent (e)
e = 65537  # Commonly used value

# Calculate the private exponent (d)
d = mod_inverse(e, phi)
return ((n, e), (n, d))
# Extended Euclidean Algorithm to calculate modular multiplicative inverse
def mod_inverse(a, m):
m0, x0, x1 = m, 0, 1
while a > 1:
q = a // m
m, a = a % m, m
x0, x1 = x1 - q * x0, x0
return x1 + m0 if x1 < 0 else x1
# RSA encryption
def rsa_encrypt(plain_text, public_key):
n, e = public_key
encrypted_text = [pow(ord(char), e, n) for char in plain_text]
return encrypted_text
# RSA decryption
def rsa_decrypt(encrypted_text, private_key):
n, d = private_key
decrypted_text = [chr(pow(char, d, n)) for char in encrypted_text]
return ''.join(decrypted_text)
# Main program
if __name__ == "__main__":
p = 61 # Prime number 1
q = 53 # Prime number 2
public_key, private_key = generate_rsa_keypair(p, q)
message = "Karina!"
print("Original Message:", message)
encrypted_message = rsa_encrypt(message, public_key)
print("Encrypted Message:", encrypted_message)
decrypted_message = rsa_decrypt(encrypted_message, private_key)
print("Decrypted Message:", decrypted_message)
```

Output:-

```
Original Message: Karina!
Encrypted Message: [597, 1632, 2412, 3179, 2235, 1632, 1853]
Decrypted Message: Karina!
```

PRACTICAL NO. 10

Aim:- Program to implement Digital Signatures.

Theory:-

Digital Signature:

Digital signatures provide a means of ensuring message integrity and authenticity in secure communication. A digital signature is a cryptographic technique that uses asymmetric encryption algorithms, such as RSA (Rivest-Shamir-Adleman), to bind the identity of the signer with the contentof a message. It allows the recipient to verify the integrity of the message and authenticate the signer's identity.

RSA Algorithm:

RSA (Rivest-Shamir-Adleman) is an asymmetric encryption algorithm widely used for secure communication. It is based on the mathematical problem of factoring large prime numbers, which iscomputationally difficult. RSA consists of a key pair: a public key for encryption and a private key fordecryption and digital signing.

Digital Signature Generation Process:

To generate a digital signature using RSA, follow these steps:

a) The signer generates a key pair: a private key (d) and a public key (e, N).
b) The signer computes the hash value of the message using a cryptographic hash function,such as SHA-256, to ensure data integrity.
c) The signer applies a mathematical function to the hash value using their private key
d) togenerate the digital signature.

Code:-

```
from Crypto.Signature import pkcs1_15

from Crypto.Hash import SHA256

from Crypto.PublicKey import RSA

from Crypto import Random

def generate_signature(private_key, message):

key = RSA.import_key(private_key)

h = SHA256.new(message.encode('utf-8'))

signature = pkcs1_15.new(key).sign(h)

return signature

def verify_signature(public_key, message, signature):

key = RSA.import_key(public_key)

h = SHA256.new(message.encode('utf-8'))

try:

pkcs1_15.new(key).verify(h, signature)

return True

except (ValueError, TypeError):
```

```
return False

random_generator = Random.new().read

key_pair = RSA.generate(2048, random_generator)

public_key = key_pair.publickey().export_key()

private_key = key_pair.export_key()

message = "Hello, World!"

signature = generate_signature(private_key, message)

print("Generated Signature:", signature)

is_valid = verify_signature(public_key, message, signature)

print("Signature Verification Result:", is_valid)
```

Output:-

Generated Signature: b'\xc8\xdbI-\xd4DO\xc0DrzY\x13\x94\x97\x06L\x0b\xd5\xfc\xd4c\xd2\xdc\xad*"&Erb\xe0\x7fzE\xbd\xf6
N\x8fhF\x1aBj\x10\xb3P\xa8k\x1a\x01\xee*\xe6\xaeH\xc8\x1a\xe7v\xff\x14\xfe\xaa\xe0\xc8w\x89\xe6\xd3\x82\xb3\x9e\xcd\xd
9\x11\x8eX=\xa1\xa2\xb5,\xcd\xab=\x8d^\xfb\x7f\xe7\xb53\xdc@\xb2s\xaa\x9f\xccD\xd4.\xaf\xb3\x15\xa8\xc17\x13}!\xb6\xc8\
xd9x\xa7\x85\x90\xf6\xb6Oc\xbaU\xd5\x0f\x98\xbb\xb3\x96m\x90y\xe8\xa5,f*~\xe1\xbd)f,T\x97\x9c\xe0\xe4\x90\x9d:}\x8f\x8
6-\x9f\xb6y\xda\xd6\xbc\tn\x87\x90@T.:2d\xf5`\xd6\x8el#\x11\xd0\x15\xe6\x0c\xd3\x1b\xcd\xf21\xf3=\x93\xc4\xd1`\xb1\xa4J\
xb1\xcf\x8b\xa9\xa1\xfb\\p\x12\x13\x8c\x96\x13\xbd\xa2\xb6\xf23b\x86t\x1e\x99\xac\x9a\xd8/[x\xf44i\xc7M\x86\xc6\xb2\xc7t\
x84\xdc\xfe\xc9'
Signature Verification Result: True