DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering) Shahbad
Daulatpur, Bawana Road, Delhi 110042

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INFORMATION AND NETWORK SECURITY (CO - 427)
LAB FILE

November, 2022

SUBMITTED BY:

NAME: ZISHNENDU SARKER
ROLL: 2K19/CO/450
7th Semester, 2022

SUBMITTED TO:

Prof. SHAILENDER KUMAR
INFORMATION AND NETWORK SECURITY
DELHI TECHNOLOGICAL UNIVERSITY

INDEX

EXPERIMENT NO. 01

Aim:

To implement a program to show encryption and decryption through the Caesar cipher

Theory for Ceaser Cipher:

The Caesar cipher is the simplest and oldest method of cryptography. The Caesar cipher method is based on a mono-alphabetic cipher and is also called a shift cipher or additive cipher. Julius Caesar used the shift cipher (additive cipher) technique to communicate with his officers. For this reason, the shift cipher technique is called the Caesar cipher. The Caesar cipher is a kind of replacement (substitution) cipher, where all letters of plain text is replaced by another letter.

Let's take an example to understand the Caesar cipher. Suppose we are shifting with 1, then A will be replaced by B, B will be replaced by C, C will be replaced by D, D will be replaced by C, and this process continues until the entire plain text is finished.

Caesar ciphers is a weak method of cryptography. It can be easily hacked. It means the message encrypted by this method can be easily decrypted.

Plaintext: It is a simple message written by the user.

Ciphertext**:** It is an encrypted message after applying some techniques.

The formula of encryption is: $E_n(x) = (x + n) \bmod 26$.

The formula of decryption is: $D_n(x) = (xi - n) \bmod 26$

If any case (Dn) value becomes negative (-ve), in this case, we will add 26 in the negative value.
Where,

E denotes the encryption
D denotes the decryption
x denotes the letters value
n denotes the key value (shift value)

Table for encryption and decryption for caesar cipher:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Code & Output:

```python
def encrypt(text,s):
    result = ""
    for i in range(len(text)):
        char = text[i]
        if (char.isupper()):
            result += chr((ord(char) + s-65) % 26 + 65)
        else:
            result += chr((ord(char) + s - 97) % 26 + 97)

    return result
text = "IAMZISHNU"
s = 4
print ("Text  : " + text)
print ("Shift : " + str(s))
print ("Cipher: " + encrypt(text,s))
```

```
Text  : IAMZISHNU
Shift : 4
Cipher: MEQDMWLRY


...Program finished with exit code 0
Press ENTER to exit console.
```

Learning :

Here we have learned about caesar cipher where we implemented the encryption process in python. The reverse way is used to decrypt the ciphert exts as here the key remains the same.

EXPERIMENT -02

AIM: It is required to implement vignere cipher.

Theory for Vigenere Cipher:

The vigenere cipher is an algorithm that is used to encrypting and decrypting the text. The Vigenere cypher is a method that combines a number of interconnected Caesar cyphers to encrypt an alphabetic text. It is based on the letters in a keyword. This polyalphabetic substitution cypher serves as an illustration. This algorithm is simple to comprehend and use. The 26-by-26 Vigenère cypher employs a table with the row and column headings A to Z. This table is also known as the Vigenere Table or Vigenere Square. We'll be using the Vigenere Table. The 26 English letters are included in the first row of this table. The letters are cyclically moved to the left one place in each row starting with the second row. For instance, the letter A goes to the end when B is moved to the first place on the second row.

- The table consists of the alphabet written out 26 times in different rows, each alphabet shifted cyclically to the left compared to the previous alphabet, corresponding to the 26 possible Caesar Ciphers.
- At different points in the encryption process, the cipher uses a different alphabet from one of the rows.
- The alphabet used at each point depends on a repeating keyword.

When the vigenere table is not given, the encryption and decryption are done by Vigenar Vigenere algebraic formula in this method (convert the letters (A-Z) into the numbers (0-25)).

Formula of encryption is, Ei = (Pi + Ki) mod

Formula of decryption is, Di = (Ei - Ki) mod 26

If any case (Di) value becomes negative (-ve), in this case, we will add 26 in the negative value

The group of 26 alphabets are like below

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Code & Output:

```python
def generateKey(string, key):
    key = list(key)
    if len(string) == len(key):
        return(key)
    else:
        for i in range(len(string) -
                       len(key)):
            key.append(key[i % len(key)])
    return("" . join(key))

def cipherText(string, key):
    cipher_text = []
    for i in range(len(string)):
        x = (ord(string[i]) +
             ord(key[i])) % 26
        x += ord('A')
        cipher_text.append(chr(x))
    return("" . join(cipher_text))

def originalText(cipher_text, key):
    orig_text = []
    for i in range(len(cipher_text)):
        x = (ord(cipher_text[i]) -
             ord(key[i]) + 26) % 26
        x += ord('A')
        orig_text.append(chr(x))
    return("" . join(orig_text))

if __name__ == "__main__":
    string = "CITYOFINDIA"
    keyword = "ZISHNU"
    key = generateKey(string, keyword)
    cipher_text = cipherText(string,key)
    print("Ciphertext :", cipher_text)
    print("Original/Decrypted Text :",
           originalText(cipher_text, key))
```

input

```
Ciphertext : BQLFBZHVVPN
Original/Decrypted Text : CITYOFINDIA
```

Learning Outcome :

Here we have learned about caesar cipher and vigenere cipher where we implemented the encryption process in python. The reverse way is used to decrypt the ciphertexts as here the key remains the same.

EXPERIMENT -03

AIM: It is required to implement playfair cipher.

Aim:

To implement a program to show encryption and decryption through the playfair Cipher

Theory:

The first useful digraph substitution cypher was the Playfair cypher. The plan was created in 1854 by Charles Wheatstone, but it was given the name Lord Playfair in honor of the person who encouraged the employment of the cypher. Unlike typical cyphers, the playfair cipher encrypts two alphabets (or digraphs) rather than just one.

British forces employed it for tactical objectives during the Second Boer War and World War I, while Australian forces utilized it for similar reasons during World War II. This was due to the fact that Playfair is easy to use and doesn't need any specialized equipment.

Algorithm of Playfair cipher:

1. Pair cannot be made with same letter. Break the letter in single and add a bogus letter to the previous letter.

Plain Text: “hello”
After Split: ‘he’ ‘lx’ ‘lo’
Here ‘x’ is the bogus letter.

2. If the letter is standing alone in the process of pairing, then add an extra bogus letter with the alone letter

Plain Text: “helloe”
AfterSplit: ‘he’ ‘lx’ ‘lo’ ‘ez’
Here ‘z’ is the bogus letter.

- If both the letters are in the same column: Take the letter below each one (going back to the top if at the bottom).
- If both the letters are in the same row: Take the letter to the right of each one (going back to the leftmost if at the rightmost position).
- If neither of the above rules is true: Form a rectangle with the two letters and take the letters on the horizontal opposite corner of the rectangle.

Code & Output:

```
def toLowerCase(text):
    return text.lower()

def removeSpaces(text):
    newText = ""
    for i in text:
        if i == " ":
            continue
        else:
            newText = newText + i
    return newText

def Diagraph(text):
    Diagraph = []
    group = 0
    for i in range(2, len(text), 2):
        Diagraph.append(text[group:i])

        group = i
    Diagraph.append(text[group:])
    return Diagraph
def FillerLetter(text):
    k = len(text)
    if k % 2 == 0:
        for i in range(0, k, 2):
            if text[i] == text[i+1]:
                new_word = text[0:i+1] + str('x') + text[i+1:]
                new_word = FillerLetter(new_word)
                break
            else:
                new_word = text
    else:
        for i in range(0, k-1, 2):
            if text[i] == text[i+1]:
                new_word = text[0:i+1] + str('x') + text[i+1:]
                new_word = FillerLetter(new_word)
                break
            else:
                new_word = text
    return new_word


list1 = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm',
         'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

```python
def generateKeyTable(word, list1):
    key_letters = []
    for i in word:
        if i not in key_letters:
            key_letters.append(i)

    compElements = []
    for i in key_letters:
        if i not in compElements:
            compElements.append(i)
    for i in list1:
        if i not in compElements:
            compElements.append(i)

    matrix = []
    while compElements != []:
        matrix.append(compElements[:5])
        compElements = compElements[5:]

    return matrix


def search(mat, element):
    for i in range(5):
        for j in range(5):
            if(mat[i][j] == element):
                return i, j


def encrypt_RowRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    if e1c == 4:
        char1 = matr[e1r][0]
    else:
        char1 = matr[e1r][e1c+1]

    char2 = ''
    if e2c == 4:
        char2 = matr[e2r][0]
    else:
        char2 = matr[e2r][e2c+1]

    return char1, char2
```

```python
def encrypt_ColumnRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    if e1r == 4:
        char1 = matr[0][e1c]
    else:
        char1 = matr[e1r+1][e1c]

    char2 = ''
    if e2r == 4:
        char2 = matr[0][e2c]
    else:
        char2 = matr[e2r+1][e2c]

    return char1, char2


def encrypt_RectangleRule(matr, e1r, e1c, e2r, e2c):
    char1 = ''
    char1 = matr[e1r][e2c]

    char2 = ''
    char2 = matr[e2r][e1c]

    return char1, char2


def encryptByPlayfairCipher(Matrix, plainList):
    CipherText = []
    for i in range(0, len(plainList)):
        c1 = 0
        c2 = 0
        ele1_x, ele1_y = search(Matrix, plainList[i][0])
        ele2_x, ele2_y = search(Matrix, plainList[i][1])

        if ele1_x == ele2_x:
            c1, c2 = encrypt_RowRule(Matrix, ele1_x, ele1_y, ele2_x, ele2_y)
        elif ele1_y == ele2_y:
            c1, c2 = encrypt_ColumnRule(Matrix, ele1_x, ele1_y, ele2_x, ele2_y
        else:
            c1, c2 = encrypt_RectangleRule(
                Matrix, ele1_x, ele1_y, ele2_x, ele2_y)
```

```python

        cipher = c1 + c2
        CipherText.append(cipher)
    return CipherText


text_Plain = 'Technological'
text_Plain = removeSpaces(toLowerCase(text_Plain))
PlainTextList = Diagraph(FillerLetter(text_Plain))
if len(PlainTextList[-1]) != 2:
    PlainTextList[-1] = PlainTextList[-1]+'z'

key = "Medalist"
print("Key text:", key)
key = toLowerCase(key)
Matrix = generateKeyTable(key, list1)

print("Plain Text:", text_Plain)
CipherList = encryptByPlayfairCipher(Matrix, PlainTextList)

CipherText = ""
for i in CipherList:
    CipherText += i
print("CipherText:", CipherText)

```

input

```
Key text: Medalist
Plain Text: technological
CipherText: sdtnfumufsblcl
```

Learning Outcomes:

Here, we learned about the encryption process of playfair cipher The reverse process is used to decrypt the cipher text into the plain text.

EXPERIMENT -04

AIM: It is required to implement hill cipher.

THEORY:

A polygraphic substitution cipher built on linear algebra is the Hill cipher. A number modulo 26 represents each letter. It is common to employ the straightforward formula A = 0, B = 1,..., Z = 25, however this is not a necessary component of the encryption. Each block of n letters, which is thought of as an n-component vector, is multiplied by an invertible n n matrix against modulus 26 to encrypt a message. Each block is multiplied by the inverse of the encryption matrix to decrypt the message.

The set of invertible n n matrices should contain the cypher key, which is the matrix used for encryption (modulo 26).

Here, polygraphic substitution cipher defines that Hill Cipher can work seamlessly with digraphs (two-letter blocks), trigraphs (three-letter blocks), or any multiple-sized blocks for building a uniform cipher.

Hill Cipher is based on a particular mathematical topic of linear Algebra and the sophisticated use of matrices in general, as well as rules for modulo arithmetic. As a prerequisite, it would be better for learners and professionals to have a sound understanding of both linear Algebra and Matrices. Thus, most of the problems and solutions for Hill Ciphers are mathematical, making it easy to withhold or hide letters precisely.

We will study Hill Cipher encryption and decryption procedures in solving 2×2 and 3×3 matrices. Although it can be used for higher matrices (4×4, 5×5, or 6×6), it also requires a higher and advanced level of mathematics, adding more complexity. Here, we have used simple examples that provide in-depth knowledge on this topic.

Code & Output :

```python
keyMatrix = [[0] * 3 for i in range(3)]

messageVector = [[0] for i in range(3)]

cipherMatrix = [[0] for i in range(3)]

def getKeyMatrix(key):
    k = 0
    for i in range(3):
        for j in range(3):
            keyMatrix[i][j] = ord(key[k]) % 65
            k += 1

def encrypt(messageVector):
    for i in range(3):
        for j in range(1):
            cipherMatrix[i][j] = 0
            for x in range(3):
                cipherMatrix[i][j] += (keyMatrix[i][x] *
                                       messageVector[x][j])
            cipherMatrix[i][j] = cipherMatrix[i][j] % 26

def HillCipher(message, key):

    getKeyMatrix(key)

    for i in range(3):
        messageVector[i][0] = ord(message[i]) % 65

    encrypt(messageVector)

    CipherText = []
    for i in range(3):
        CipherText.append(chr(cipherMatrix[i][0] + 65))

```

```python

    print("Ciphertext: ", "".join(CipherText))

def main():

    message = "nitiwit"
    key = "GYBNQKURP"

    HillCipher(message, key)

if __name__ == "__main__":
    main()

```

input

```
Ciphertext:  HTF
```

LEARNING OUTCOMES:

Here, we have learned about hill cipher, how to encrypt using hill cipher and how efficient it is. The reverse way will introduce the decryption part

EXPERIMENT -05

Aim:

It is required to implement the Diffie-Hellman Exchange algorithm.

THEORY:

Using the elliptic curve to produce points and the parameters to obtain the secret key, the Diffie-Hellman method is being utilized to create a shared secret that may be used for secret conversations while transferring data over a public network.

For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables, one prime P and G (a primitive root of P) and two private values a and b.

P and G are both publicly available numbers. Users (say Alice and Bob) pick private values a and b and they generate a key and exchange it publicly. The opposite person receives the key and that generates a secret key, after which they have the same secret key to encrypt.

Diffie-Hellman algorithm:

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 using the elliptic curve to generate points and get the secret key using the parameters.

For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables, one prime P and G (a primitive root of P) and two private values a and b.

P and G are both publicly available numbers. Users (say Alice and Bob) pick private values a and b and they generate a key and exchange it publicly. The opposite person receives the key and that generates a secret key, after which they have the same secret key to encrypt.

Code & Output:

```python
from random import randint

if __name__ == '__main__':

    P = 20

    G = 8


    print('The Value of P is :%d'%(P))
    print('The Value of G is :%d'%(G))

    a = 4
    print('The Private Key a for Alice is :%d'%(a))

    x = int(pow(G,a,P))

    b = 3
    print('The Private Key b for Bob is :%d'%(b))

    y = int(pow(G,b,P))

    ka = int(pow(y,a,P))

    kb = int(pow(x,b,P))

    print('Secret key for the Alice is : %d'%(ka))
    print('Secret Key for the Bob is : %d'%(kb))
```

input

```
The Private Key a for Alice is :4
The Private Key b for Bob is :3
Secret key for the Alice is : 16
Secret Key for the Bob is : 16
```

LEARNING OUTCOMES:

Here, we have learned about Diffie-Hellman exchange algorithm, how to encrypt using it and how efficient it is. The reverse way will introduce the decryption part.

EXPERIMENT -06

AIM: It is required to implement S-DES sub-key generation.

Theory: The general design of the streamlined DES. An 8-bit block of plaintext (for example, 10111101) and a 10-bit key are the inputs for the S-DES encryption technique, which yields an 8-bit block of ciphertext as the output. The 10-bit key that was used to encrypt the initial 8-bit block of plaintext is entered into the S-DES decryption method together with an 8-bit block of ciphertext.

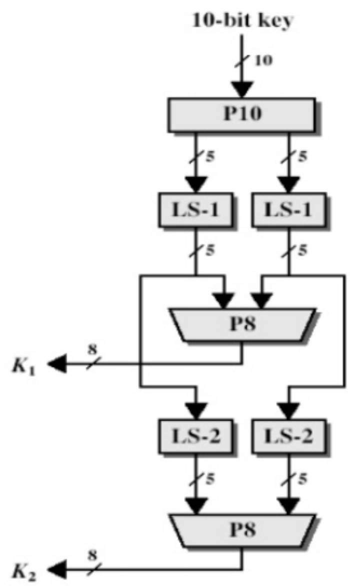


Figure: key generation for S-DES

S-DES depends on the use of a 10-bit key shared between sender and receiver. From this key, two 8-bit subkeys are produced for use in particular stages of the encryption and decryption algorithm. First, permute the key in the following fashion. Let the 10-bit key be designated as (k1, K2, k3, k4, k5, k6, k7, k8, k9, k10). Then the permutation P10 is defined as:

P10 (k1, K2, k3, k4, k5, k6, k7, k8, k9, k10) = (k3, k5, K2, k7, k4, k10 10, k1, k9, k8, k6) P10 can be concise.

Code & Output :

```python
import numpy as np

def table_shift(array, table_array):
    array_shifted = np.zeros(table_array.shape[0], dtype='int')
    for index, value in enumerate(table_array): array_shifted[index] = array[value - 1]
    return array_shifted

def array_split(array):
    left_split = array[:int(len(array) / 2)]
    right_split = array[int(len(array) / 2):]
    return left_split, right_split

def shifting_LtoR(array):
    temp = array[0]
    for index in range(1, len(array)): array[index - 1] = array[index]
    array[len(array) - 1] = temp
    return array

table_p_10 = np.array([3, 5, 2, 7, 4, 10, 1, 9, 8, 6])
table_p_08 = np.array([6, 3, 7, 4, 8, 5, 10, 9])

key = list('0001011011')

def split_and_merge(key):
    left_split, right_split = array_split(key)
    return np.concatenate((shifting_LtoR(left_split), shifting_LtoR(right_split)))

def key_generation_1(key, table):
    k = table_shift(key, table)
    key_merge = split_and_merge(k)
    return table_shift(key_merge, table)

def key_generation_2(key, table): return split_and_merge(key)

print("Name: Zishnendu Sarker ")
print("Lab : 05 ")
key_1 = key_generation_1(key, table_p_10)
print("".join([str(elem) for elem in key_1]))  #1000111010

key_2 = key_generation_2(key_1, table_p_08)
print("".join([str(elem) for elem in key_2]))  #0001110101
```

input

```
Name: Zishnendu Sarker
Lab : 05
1001110100
0011101001
```

Learning Outcome:

The process of encrypting a plaintext message into an encrypted message with the use of S-DES has been divided into multi-steps which may help you to understand it as easily as possible. Points should be remembered: It is a block cipher, It has 8-bit block size of plain text or cipher text, it uses 10-bit key size for encryption, it is a symmetric cipher, it has Two Rounds.

EXPERIMENT -07

AIM: It is required to implement generate SHA-1 hash.

Theory: A hash function called SHA-1 (Secure Hash Algorithm 1) creates a 160-bit hash code (message digest) corresponding to an input. The American National Security Agency began the SHA-1 in 1993. The security programs and protocols TLS, SSL, PGP, SSH, IPsec, and S/MIME all use it extensively.

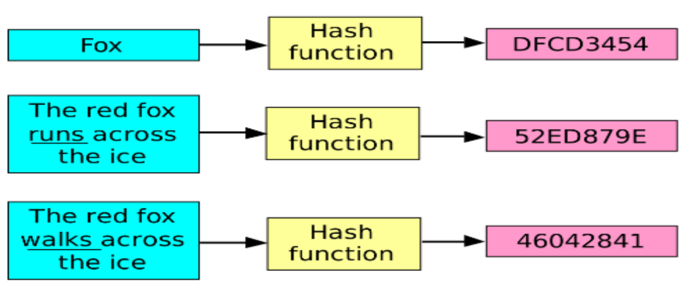
Compared to MD5, the SHA-1 hashing method is more secure. It is key to securing confidential data, such as passwords and credit card details.

In cryptography, SHA-1 (Secure Hash Algorithm 1) is a cryptographically broken but still widely used hash function that takes an input and produces a 160-bit (20-byte) hash value known as a message digest – typically rendered as 40 hexadecimal digits.

SHA-1 produces a 160-bit hash value or message digests from the inputted data (data that requires encryption), which resembles the hash value of the MD5 algorithm. It uses 80 rounds of cryptographic operations to encrypt and secure a data object. Some of the protocols that use SHA-1 include:

- Transport Layer Security (TLS)
- Secure Sockets Layer (SSL)
- Pretty Good Privacy (PGP)
- Secure Shell (SSH)
- Secure/Multipurpose Internet Mail Extensions (S/MIME)
- Internet Protocol Security (IPSec)

SHA-1 is commonly used in cryptographic applications and environments where the need for data integrity is high. It is also used to index hash functions and identify data corruption and checksum errors.



Code & Output:

```python
import hashlib

# initializing string
str = "sristi"


result = hashlib.sha256(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA256 is : ")
print(result.hexdigest())

print ("\r")

# initializing string
str = "sristi"


result = hashlib.sha384(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA384 is : ")
print(result.hexdigest())

print ("\r")

# initializing string
str = "sristi"


result = hashlib.sha224(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA224 is : ")
print(result.hexdigest())
```

```
print ("\r")

# initializing string
str = "sristi"


result = hashlib.sha512(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA512 is : ")
print(result.hexdigest())

print ("\r")

# initializing string
str = "sristi"


result = hashlib.sha1(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA1 is : ")
print(result.hexdigest())
```

input

```
The hexadecimal equivalent of SHA384 is :
e7ce5a962d28de95592c9a5867939a53fa38ad65c76b996bb7874074de11fa32e845a3bd10f9a034819179de0a
366e9d

The hexadecimal equivalent of SHA224 is :
f49442ca0043a9b7c3d7a7ed6ebd240349c89cbe77047537d0a810c7

The hexadecimal equivalent of SHA512 is :
6a9fd6f6bf6ebac32e5e76f0591e20ce213dbcb11464b70dcdb228d78068650b2e37b3918c449dacb62247fbba
942c05328c1bc600a38a885aeade160377db1a

The hexadecimal equivalent of SHA1 is :
6ba3d3754f5de7d5c6ec71e20e6164daf7f056ed
```

Learning Outcome:

The four round constants k are 230 times the square roots of 2, 3, 5, and 10. However, they were incorrectly rounded to the nearest integer instead of being rounded to the nearest odd integer, with equilibrated proportions of zero and one bit. As well, choosing the square root of 10 (which is not a prime) made it a common factor for the two other chosen square roots of primes 2 and 5, with possibly usable arithmetic properties across successive rounds, reducing the strength of the algorithm against finding collisions on some bits. The first four starting values for h0 through h3 are the same with the MD5 algorithm, and the fifth (for h4) is similar. However, they were not properly verified for being resistant against inversion of the few first rounds to infer possible collisions on some bits, usable by multiblock differential attacks.

properties across successive rounds, reducing the strength of the algorithm against finding collisions on some bits.

The first four starting values for h0 through h3 are the same with the MD5 algorithm, and the fifth (for h4) is similar. However, they were not properly verified for being resistant against inversion of the few first rounds to infer possible collisions on some bits, usable by multiblock differential attacks.

EXPERIMENT -08

AIM: It is required to implement a signature scheme - Digital Signature Standard.

Theory:

When sending an electronic message, the authenticity of the message is checked using digital signatures. A public key system is used by an algorithm for digital signatures. The intended receiver confirms the message with the intended transmitter's public key after the intended transmitter signs it with his or her private key. Message authentication, message integrity, and non-repudiation services can all be offered by a digital signature.

Digital Signature Scheme: In RSA, d is private; e and n are public.

- Alice creates her digital signature using S=M^d mod n where M is the message
- Alice sends Message M and Signature S to Bob
- Bob computes M1=S^e mod n
- If M1=M then Bob accepts the data sent by Alice.

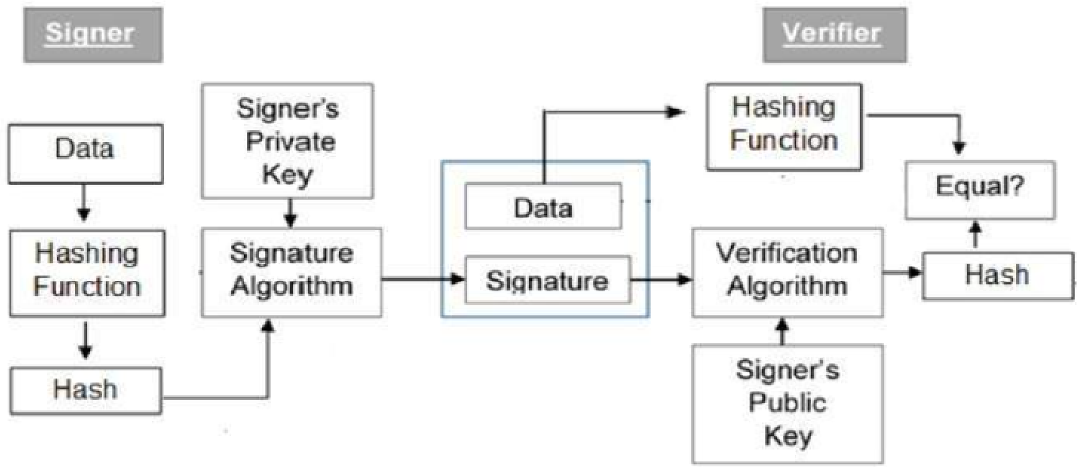
A hash code is generated from the message and given as an input to the signature function on the sender side. The other inputs to a signature function include a unique random number k for the signature, the private key of sender PR(a), and global public key i.e., PU(g).

The output of the signature function consists of two components: s & r, which is concatenated with the input message and then sent to the receiver.

Signature = {s, r}.

At the receiver side, the hash code for the message sent is generated by the receiver by applying a hash function. The verification function is used for verifying the message and signature sent by the sender. The verification function takes hash code generated, signature components s and r, the public key of the sender (PU(a)), and global public key.

The signature function is compared with the output of the verification function and if both the values match, the signature is valid because A valid signature can only be generated by the sender using its private key.



Code & Output:

```
# Function to find gcd
# of two numbers
def euclid(m, n):

    if n == 0:
        return m
    else:
        r = m % n
        return euclid(n, r)


# Program to find
# Multiplicative inverse
def exteuclid(a, b):

    r1 = a
    r2 = b
    s1 = int(1)
    s2 = int(0)
    t1 = int(0)
    t2 = int(1)

    while r2 > 0:

        q = r1//r2
        r = r1-q * r2
        r1 = r2
        r2 = r
        s = s1-q * s2
        s1 = s2
        s2 = s
        t = t1-q * t2
        t1 = t2
        t2 = t

    if t1 < 0:
        t1 = t1 % a

    return (r1, t1)

# Enter two large prime
# numbers p and q
p = 823
q = 953
n = p * q
Pn = (p-1)*(q-1)
# Generate encryption key
# in range 1<e<Pn
key = []
for i in range(2, Pn):

    gcd = euclid(Pn, i)
```

```python

    if gcd == 1:
        key.append(i)

# Select an encryption key
# from the above list
e = int(313)
# Obtain inverse of
# encryption key in Z_Pn
r, d = exteuclid(Pn, e)
if r == 1:
    d = int(d)
    print("decryption key is: ", d)

else:
    print("Multiplicative inverse for\
    the given encryption key does not \
    exist. Choose a different encryption key ")

# Enter the message to be sent
M = 19070
# Signature is created
S = (M**d) % n


M1 = (S**e) % n



if M == M1:
    print("As M = M1, Accept the\
    message sent by Zishnu")
else:
    print("As M not equal to M1,\
    Do not accept the message\
    sent by Zishnu ")
```

input

```
decryption key is:  160009
As M = M1, Accept the    message sent by Zishnu
```

Learning Outcome:

From the experiment, we have learned that asymmetric cryptographic methods like digital signatures are employed to confirm the legitimacy of digital messages and documents. It makes use of the idea of public/private key pairs, where the two keys are mathematically connected and offer security benefits above and above those of handwritten signatures. A digital signature is encrypted using a person's private key, and only that person's public key may be used to decrypt the signature.

EXPERIMENT -09

AIM: Study and use the Wireshark for various network protocols.

THEORY:

The most famous and commonly used network protocol analyzer in the world is called Wireshark. It is the de facto (and frequently de jure) standard across many commercial and non-profit firms, governmental organisations, and educational institutions because it enables you to observe what's occurring on your network at a microscopic level. The continued development of Gerald Combs' 1998 project, Wireshark, is made possible by the voluntary contributions of networking specialists from all around the world.

Wireshark has a rich feature set which includes the following:

Thorough examination of hundreds of procedures, with new ones being added on a regular basis. offline analysis and real-time capture. standard packet browser with three panes. Multi-platform: runs on several operating systems, including Windows, Linux, macOS, Solaris, FreeBSD, and NetBSD. A GUI or the TTY-mode TShark programme can be used to browse network data that has been captured.The industry's strongest display filters. Detailed VoIP analysis several various capture file types can be read/writable. Gzip-compressed capture files may instantly be uncompressed. Many protocols, including IPsec, ISAKMP, Kerberos, SNMPv3, SSL/TLS, WEP, and WPA/WPA2, are supported in terms of decryption. The packet list may be analysed quickly and easily by using colouring rules. Export options for output include XML, PostScript®, CSV, and plain text.

Implementation and Output:

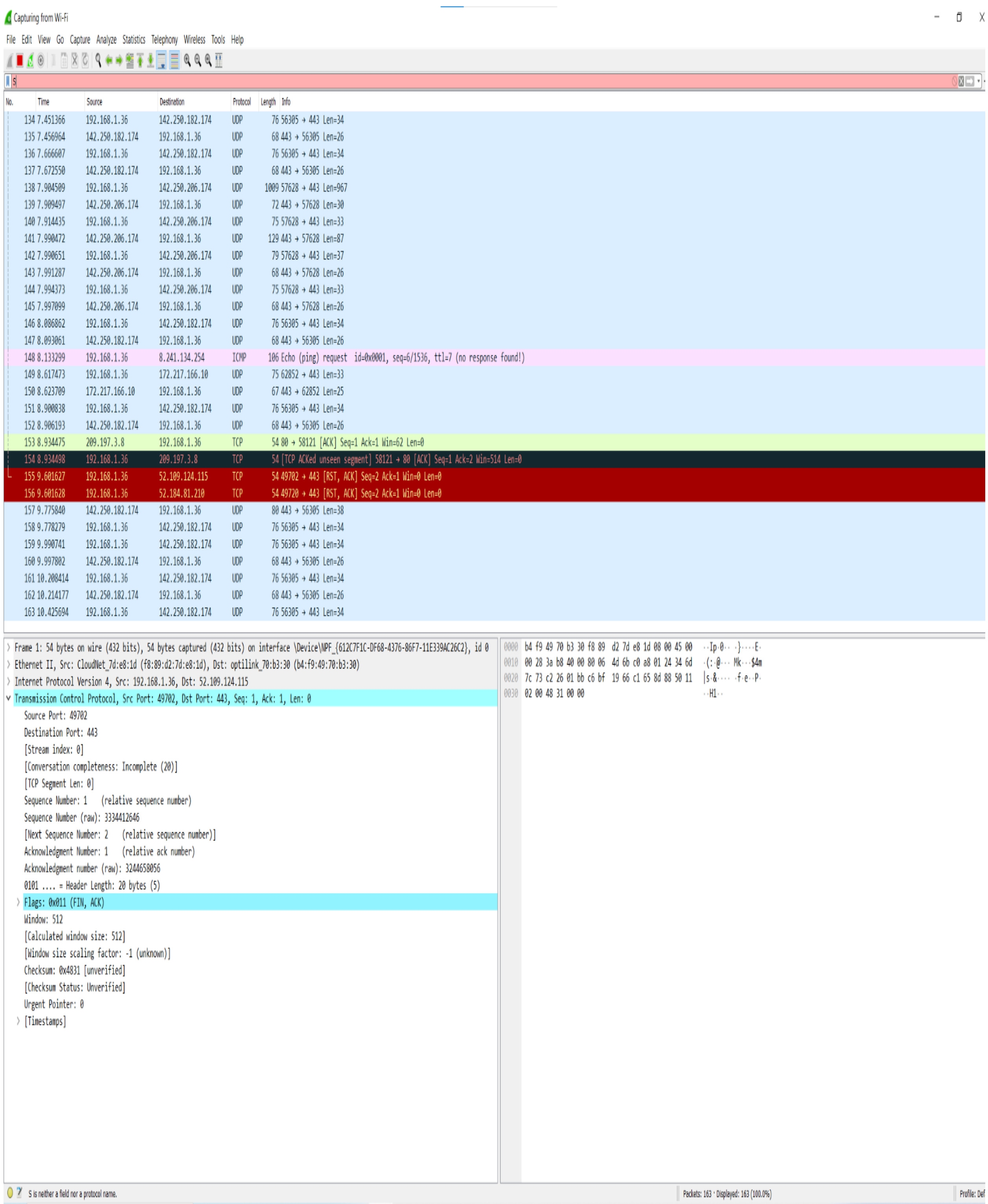


Fig: for TCP protocol

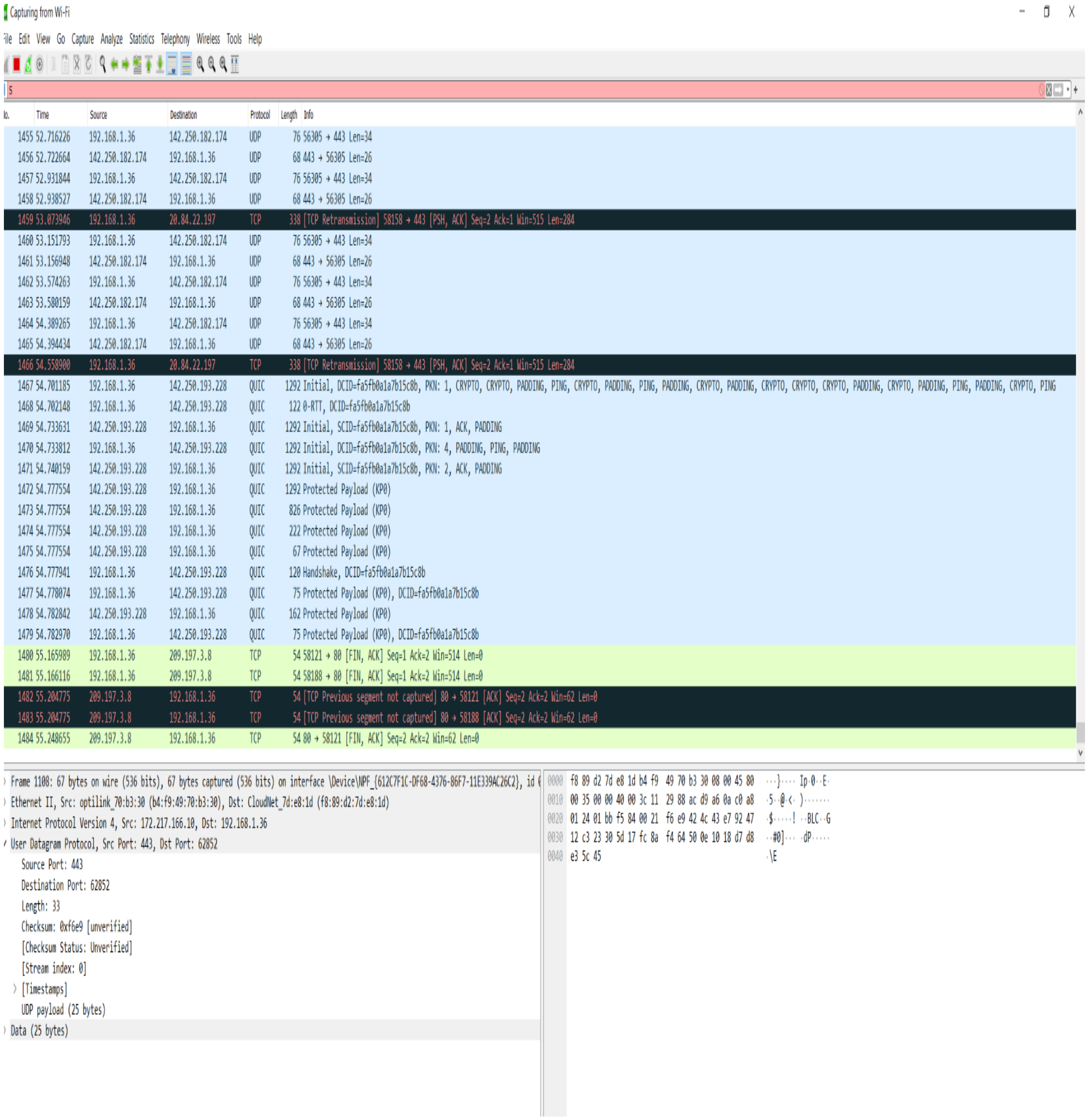


Fig: for UDP protocol

Learning Outcome:

From here , we learn how to sync with network layer, and how to identify the network protocol.

Network protocol and their usage and identification