Islamic University

Department of
Information and Communication Technology

"LAB REPORT"

Course Code: ICT-4110

Course Name:
Cryptography and Network Security Laboratory

Submitted to:

Dr. Tarek Hasan Al Mahmud
Professor
Department of Information and Communication Technology
Islamic University, Bangladesh

Submitted by:

Name: S M Sajjad Hossain Soykot
Roll: 2018040
Reg. No: 1315
Session: 2020-21

INDEX

Experiment No: 01

Implementation of Caesar Cipher

Theory

The Caesar Cipher is one of the simplest and oldest substitution ciphers. In this method, each letter of the plaintext is shifted by a fixed number of positions in the alphabet.

Example (key = 3):

A → D, B → E, C → F

Mathematical Representation

Encryption:

$$C = (P + k) \bmod 26$$

Decryption:

$$P = (C - k) \bmod 26$$

Where:

- P = Plaintext letter
- C = Ciphertext letter
- k = Shift key

Algorithm

Encryption

1. Take plaintext input
2. Take shift key input
3. For each alphabetic character:
    - Convert to numeric position
    - Shift by key
    - Apply modulo 26
4. Form the ciphertext

Decryption

1. Take ciphertext input
2. Use the same shift key
3. Shift backward by the key
4. Recover the plaintext

Source Code

```
def encrypt(text, key):
    result = ""
    key %= 26
    for ch in text:
        if ch.isalpha():
            base = ord('A') if ch.isupper() else ord('a')
            result += chr((ord(ch) - base + key) % 26 + base)
        else:
            result += ch
    return result

def decrypt(text, key):
    result = ""
    key %= 26
    for ch in text:
        if ch.isalpha():
            base = ord('A') if ch.isupper() else ord('a')
            result += chr((ord(ch) - base - key) % 26 + base)
        else:
            result += ch
    return result

plaintext = input("Enter plaintext: ")
key = int(input("Enter key: "))

print("Plaintext:", plaintext)
print("Key:", key)

cipher = encrypt(plaintext, key)
print("Encrypted text:", cipher)

plain = decrypt(cipher, key)
print("Decrypted text:", plain)
```

Sample Output

```
Enter plaintext: HELLO
Enter key: 3

Encrypted text: KHOOR
Decrypted text: HELLO
```

Experiment No: 02

Implementation of Playfair Cipher

Theory

The Playfair Cipher is a digraph substitution cipher. It encrypts pairs of letters instead of single letters. A 5×5 matrix is constructed using a keyword. The letters I and J are treated as the same character.

Rules:

1. If both letters are in the same row, replace each with the letter to its right circularly.
2. If both letters are in the same column, replace each with the letter below it circularly.
3. Otherwise, replace each letter with the letter in the same row but in the column of the other letter.

If a pair contains repeated letters, a filler such as X is inserted between them.

Mathematical Example

`Keyword: MONARCHY`

```
M O N A R
C H Y B D
E F G I K
L P Q S T
U V W X Z
```

Plaintext pair: HE

Positions:

- H → row 2, column 2
- E → row 3, column 1

Rectangle rule applies:

- H → C (same row, column of E)
- E → F (same row, column of H)

So,

$$HE \rightarrow \mathrm{CF}$$

Algorithm

Encryption

1. Generate a 5×5 key matrix using a keyword.
2. Remove duplicate letters from the keyword.
3. Convert plaintext to uppercase and remove spaces.
4. Replace J with I.
5. Divide plaintext into pairs.
6. Insert X between repeated letters in a pair.
7. If one letter remains, add X at the end.
8. Encrypt each pair using Playfair rules.

Decryption

1. Take ciphertext in pairs.
2. Apply reverse Playfair rules:
    - Same row → move left
    - Same column → move up
    - Rectangle → swap columns
3. Recover the plaintext.

Source Code

```
def generate_key_matrix(key):
    key = key.upper().replace("J", "I")
    seen = set()
    matrix_list = []

    for ch in key:
        if ch.isalpha() and ch not in seen:
            seen.add(ch)
            matrix_list.append(ch)

    for ch in "ABCDEFGHIKLMNOPQRSTUVWXYZ":
        if ch not in seen:
            seen.add(ch)
            matrix_list.append(ch)

    return [matrix_list[i:i+5] for i in range(0, 25, 5)]

def find_position(matrix, ch):
    for i in range(5):
        for j in range(5):
            if matrix[i][j] == ch:
                return i, j
```

```
def prepare_text(text):
    text = text.upper().replace("J", "I")
    text = "".join(ch for ch in text if ch.isalpha())

    prepared = ""
    i = 0
    while i < len(text):
        a = text[i]
        b = text[i + 1] if i + 1 < len(text) else 'X'

        if a == b:
            prepared += a + 'X'
            i += 1
        else:
            prepared += a + b
            i += 2

    if len(prepared) % 2 != 0:
        prepared += 'X'

    return prepared

def encrypt_playfair(text, matrix):
    text = prepare_text(text)
    cipher = ""

    for i in range(0, len(text), 2):
        a, b = text[i], text[i+1]
        r1, c1 = find_position(matrix, a)
        r2, c2 = find_position(matrix, b)

        if r1 == r2:
            cipher += matrix[r1][(c1 + 1) % 5]
            cipher += matrix[r2][(c2 + 1) % 5]
        elif c1 == c2:
            cipher += matrix[(r1 + 1) % 5][c1]
            cipher += matrix[(r2 + 1) % 5][c2]
        else:
            cipher += matrix[r1][c2]
            cipher += matrix[r2][c1]

    return cipher

def decrypt_playfair(text, matrix):
    text = "".join(ch for ch in text.upper() if ch.isalpha())
    plain = ""

    for i in range(0, len(text), 2):
        a, b = text[i], text[i+1]
        r1, c1 = find_position(matrix, a)
        r2, c2 = find_position(matrix, b)

        if r1 == r2:
            plain += matrix[r1][(c1 - 1) % 5]
            plain += matrix[r2][(c2 - 1) % 5]
        elif c1 == c2:
            plain += matrix[(r1 - 1) % 5][c1]
            plain += matrix[(r2 - 1) % 5][c2]
        else:
            plain += matrix[r1][c2]
            plain += matrix[r2][c1]

    return plain

keyword = input("Enter keyword: ")
```

```
plaintext = input("Enter plaintext: ")

matrix = generate_key_matrix(keyword)

print("Key Matrix:")
for row in matrix:
    print(" ".join(row))

cipher = encrypt_playfair(plaintext, matrix)
print("Encrypted text:", cipher)

plain = decrypt_playfair(cipher, matrix)
print("Decrypted text:", plain)
```

Sample Output

```
Enter keyword: MONARCHY
Enter plaintext: HELLO

Key Matrix:
M O N A R
C H Y B D
E F G I K
L P Q S T
U V W X Z

Encrypted text: CFSUPM
Decrypted text: HELXLO
```

Experiment No: 03

Implementation of Hill Cipher

Theory

The Hill Cipher is a polygraphic substitution cipher based on linear algebra. It encrypts blocks of letters using matrix multiplication.

Each letter is represented as:

$$A = 0,\ B = 1,\ C = 2,\ \ldots,\ Z = 25$$

Encryption formula:

$$C = K \times P \pmod{26}$$

Decryption formula:

$$P = K^{-1} \times C \pmod{26}$$

Where:
K = Key matrix
P = Plaintext vector
C = Ciphertext vector
K^{-1} = Inverse of key matrix modulo 26

A 2 × 2 key matrix is commonly used for simple implementation.

Mathematical Example

Let the key matrix be:

$$K = \begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix}$$

Plaintext pair: HI

$$H = 7, \quad I = 8$$

So,

$$P = \begin{bmatrix} 7 \\ 8 \end{bmatrix}$$

Encryption:

$$C = K \times P \pmod{26}$$

$$C = \begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \end{bmatrix} \pmod{26}$$

$$C = \begin{bmatrix} 45 \\ 54 \end{bmatrix} \pmod{26} = \begin{bmatrix} 19 \\ 2 \end{bmatrix}$$

Therefore,

$$19 = T, \quad 2 = C$$

So, HI is encrypted as TC.

Algorithm

Encryption

4. Choose a 2×2 invertible key matrix.
5. Convert plaintext letters to numerical values using $A = 0$ to $Z = 25$.
6. If plaintext length is odd, append X.
7. Divide plaintext into pairs.
8. Multiply each pair with the key matrix.
9. Apply modulo 26 to the result.
10. Convert numbers back to letters to obtain ciphertext.

Decryption

1. Compute the inverse of the key matrix modulo 26.
2. Divide ciphertext into pairs.
3. Multiply each pair with the inverse matrix.
4. Apply modulo 26.
5. Convert numbers back to letters to obtain plaintext.

Source Code

```
def mod_inverse(a, m):
    a %= m
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    raise ValueError("No modular inverse exists.")

def matrix_inverse_2x2(matrix):
    a, b = matrix[0]
    c, d = matrix[1]
    det = (a * d - b * c) % 26
    det_inv = mod_inverse(det, 26)

    inv = [
        [( d * det_inv) % 26, (-b * det_inv) % 26],
        [(-c * det_inv) % 26, ( a * det_inv) % 26]
    ]
    return inv

def process_text(text):
    text = "".join(ch for ch in text.upper() if ch.isalpha())
    if len(text) % 2 != 0:
        text += 'X'
    return text

def text_to_numbers(text):
    return [ord(ch) - ord('A') for ch in text]
```

```
def numbers_to_text(nums):
    return "".join(chr(n % 26 + ord('A')) for n in nums)


def encrypt_hill(text, key):
    text = process_text(text)
    nums = text_to_numbers(text)
    result = []

    for i in range(0, len(nums), 2):
        pair = nums[i:i+2]
        c1 = (key[0][0]*pair[0] + key[0][1]*pair[1]) % 26
        c2 = (key[1][0]*pair[0] + key[1][1]*pair[1]) % 26
        result.extend([c1, c2])

    return numbers_to_text(result)

def decrypt_hill(cipher, key):
    inv_key = matrix_inverse_2x2(key)
    nums = text_to_numbers(cipher)
    result = []

    for i in range(0, len(nums), 2):
        pair = nums[i:i+2]
        p1 = (inv_key[0][0]*pair[0] + inv_key[0][1]*pair[1]) % 26
        p2 = (inv_key[1][0]*pair[0] + inv_key[1][1]*pair[1]) % 26
        result.extend([p1, p2])

    return numbers_to_text(result)

key = [[3, 3], [2, 5]]

plaintext = input("Enter plaintext: ")
print("Plaintext:", plaintext)
print("Key Matrix:", key)
cipher = encrypt_hill(plaintext, key)
print("Encrypted text:", cipher)

plain = decrypt_hill(cipher, key)
print("Decrypted text:", plain)
```

Sample Output

```
Plaintext: HELP
Key Matrix: [[3, 3], [2, 5]]
Encrypted text: HIAT
Decrypted text: HELP
```

Experiment No: 04

Implementation of Vigenère Cipher

Theory

The Vigenère Cipher is a polyalphabetic substitution cipher. It uses a keyword to determine the shift for each letter in the plaintext.

Each plaintext letter is shifted according to the corresponding letter in the repeated keyword.

Encryption formula:

$$C_i = (P_i + K_i) \bmod 26$$

Decryption formula:

$$P_i = (C_i - K_i) \bmod 26$$

Where:
P_i = Plaintext letter value
C_i = Ciphertext letter value
K_i = Key letter value

Mathematical Example

Plaintext: ATTACK
Keyword: LEMON

Convert letters to numbers:

Step	Plaintext (P_i)	P_i Value	Key (K_i)	K_i Value	(P_i + K_i)	Mod 26	Cipher Value	Cipher Letter
1	A	0	L	11	0 + 11 = 11	11	11	L
2	T	19	E	4	19 + 4 = 23	23	23	X
3	T	19	M	12	19 + 12 = 31	5	5	F
4	A	0	O	14	0 + 14 = 14	14	14	O
5	C	2	N	13	2 + 13 = 15	15	15	P
6	K	10	L	11	10 + 11 = 21	21	21	V

Encrypted text: LXFOPV

Algorithm

Encryption

1. Take plaintext input.
2. Take keyword input.
3. Repeat keyword until its length matches the plaintext.
4. Convert letters to numerical values using $A = 0$ to $Z = 25$.
5. Apply $C_i = (P_i + K_i) \bmod 26$.
6. Convert result back to letters.

Decryption

1. Take ciphertext input.
2. Repeat keyword to match ciphertext length.
3. Apply $P_i = (C_i - K_i) \bmod 26$.
4. Convert numbers back to letters to recover plaintext.

Source Code

```
def generate_key(text, key):
    key = key.upper()
    text = text.upper()
    result = ""
    j = 0
    for ch in text:
        if ch.isalpha():
            result += key[j % len(key)]
            j += 1
        else:
            result += ch
    return result

def encrypt_vigenere(text, key):
    text = text.upper()
    key = generate_key(text, key)
    cipher = ""

    for t, k in zip(text, key):
        if t.isalpha():
            cipher += chr((ord(t) - ord('A') + ord(k) - ord('A')) % 26 + ord('A'))
        else:
            cipher += t
    return cipher
```

```
def decrypt_vigenere(cipher, key):
    cipher = cipher.upper()
    key = generate_key(cipher, key)
    plain = ""

    for c, k in zip(cipher, key):
        if c.isalpha():
            plain += chr((ord(c) - ord('A') - (ord(k) - ord('A'))) % 26 + ord('A'))
        else:
            plain += c
    return plain

plaintext = input("Enter plaintext: ")
keyword = input("Enter keyword: ")

print("Plaintext:", plaintext)
print("Keyword:", keyword)
cipher = encrypt_vigenere(plaintext, keyword)
print("Encrypted text:", cipher)

plain = decrypt_vigenere(cipher, keyword)
print("Decrypted text:", plain)
```

Sample Output

```
Enter plaintext: ATTACKATDAWN
Enter keyword: LEMON

Encrypted text: LXFOPVEFRNHR
Decrypted text: ATTACKATDAWN
```

Experiment No: 05

Implementation of Rail Fence Cipher

Theory

The Rail Fence Cipher is a transposition cipher. It does not substitute characters but rearranges their positions.

The plaintext is written in a zigzag pattern across a given number of rails depth and then read row by row to form the ciphertext.

For example, for depth $= 3$, the pattern follows a down-up zigzag structure.

Illustration Example

Plaintext: WEAREDISCOVERED
Depth: 3

```
W . . . E . . . C . . . R
. E . R . D . S . O . E .
. . A . . . I . . . V . .
```

Reading row-wise:

Row 1: WECR
Row 2: ERDSOE
Row 3: AIV

Ciphertext: WECRERDSOEAIV

Algorithm

Encryption

1. Take plaintext input.
2. Take rail depth input.
3. Create a zigzag pattern across the rails.
4. Place characters following the zigzag path.
5. Read characters row by row.
6. Combine them to form ciphertext.

Decryption

1. Determine zigzag positions for given depth.
2. Fill ciphertext row by row into those positions.
3. Traverse the zigzag path to reconstruct plaintext.

Source Code

```
def encrypt_rail_fence(text, key):
    if key <= 1:
        return text

    rail = ['' for _ in range(key)]
    row = 0
    direction = 1

    for ch in text:
        rail[row] += ch
        row += direction

        if row == 0 or row == key - 1:
            direction *= -1

    return ''.join(rail)

def decrypt_rail_fence(cipher, key):
    if key <= 1:
        return cipher

    pattern = [['\n' for _ in range(len(cipher))] for _ in range(key)]

    row, direction = 0, 1
    for col in range(len(cipher)):
        pattern[row][col] = '*'
        row += direction
        if row == 0 or row == key - 1:
            direction *= -1

    index = 0
    for i in range(key):
        for j in range(len(cipher)):
            if pattern[i][j] == '*' and index < len(cipher):
                pattern[i][j] = cipher[index]
                index += 1

    result = []
    row, direction = 0, 1
    for col in range(len(cipher)):
        result.append(pattern[row][col])
        row += direction
        if row == 0 or row == key - 1:
            direction *= -1

    return ''.join(result)

plaintext = input("Enter plaintext: ")
depth = int(input("Enter depth: "))
cipher = encrypt_rail_fence(plaintext, depth)
print("Encrypted text:", cipher)
plain = decrypt_rail_fence(cipher, depth)
print("Decrypted text:", plain)
```

Sample Output

```
Enter plaintext: WEAREDISCOVEREDFLEEATONCE
Enter depth: 3

Encrypted text: WECRLTEERDSOEEFEAOCAIVDEN
Decrypted text: WEAREDISCOVEREDFLEEATONCE
```

Experiment No: 06

Implementation of RSA Algorithm

Theory

RSA is an asymmetric encryption algorithm. It uses two keys:

- Public key for encryption
- Private key for decryption

Steps:

1. Choose two prime numbers p and q.
2. Compute $n = p \times q$.
3. Compute $\phi(n) = (p-1)(q-1)$.
4. Choose e such that $\gcd(e, \phi(n)) = 1$.
5. Compute d such that:

$$d \times e \equiv 1 \pmod{\phi(n)}$$

Public key: (e, n)
Private key: (d, n)

Encryption formula:

$$C = P^e \bmod n$$

Decryption formula:

$$P = C^d \bmod n$$

Mathematical Example

Let:

$$p = 61, \quad q = 53$$

Compute:

$$n = p \times q = 61 \times 53 = 3233$$

$$\phi(n) = (61-1)(53-1) = 60 \times 52 = 3120$$

Choose:

$$e = 17$$

Find d such that:

$$d \times 17 \equiv 1 \pmod{3120}$$

$$d = 2753$$

So,

Public Key: (17,3233)
Private Key: (2753,3233)

Algorithm

Key Generation

1. Select prime numbers p and q.
2. Calculate n and $\phi(n)$.
3. Choose e such that $\gcd(e, \phi(n)) = 1$.
4. Compute d as the modular inverse of e modulo $\phi(n)$.

Encryption

1. Convert plaintext characters to numerical values ASCII or mapping.
2. Compute $C = P^e \bmod n$ for each value.

Decryption

1. Compute $P = C^d \bmod n$ for each encrypted value.
2. Convert numerical values back to characters.

Source Code

```
from math import gcd

def mod_inverse(e, phi):
    for d in range(1, phi):
        if (d * e) % phi == 1:
            return d
    raise ValueError("Modular inverse not found.")

def rsa_keygen(p, q, e):
    n = p * q
    phi = (p - 1) * (q - 1)

    if gcd(e, phi) != 1:
        raise ValueError("e must be coprime with phi(n).")

    d = mod_inverse(e, phi)
    return (e, n), (d, n)

def rsa_encrypt(text, public_key):
    e, n = public_key
    return [pow(ord(ch), e, n) for ch in text]

def rsa_decrypt(cipher, private_key):
    d, n = private_key
    return ''.join(chr(pow(c, d, n)) for c in cipher)
```

```
p = 61
q = 53
e = 17

public_key, private_key = rsa_keygen(p, q, e)

print("Public Key:", public_key)
print("Private Key:", private_key)

plaintext = input("Enter plaintext: ")
cipher = rsa_encrypt(plaintext, public_key)
print("Encrypted text:", cipher)

plain = rsa_decrypt(cipher, private_key)
print("Decrypted text:", plain)
```

Sample Output

```
p = 61, q = 53, e = 17
Public Key: (17, 3233)
Private Key: (2753, 3233)

Plaintext: HELLO
Encrypted text: [3000, 28, 2726, 2726, 1307]
Decrypted text: HELLO
```

Experiment No: 07

Implementation of Diffie-Hellman Key Exchange

Theory

Diffie-Hellman Key Exchange is used to establish a shared secret key between two parties over an insecure channel.

Public Values

Prime number: p
Primitive root: g

Private Values

a = private key of user A
b = private key of user B

Public Keys

$$A = g^a \bmod p$$

$$B = g^b \bmod p$$

Shared Secret

For user A:

$$K = B^a \bmod p$$

For user B:

$$K = A^b \bmod p$$

Both users obtain the same shared secret key.

Mathematical Example

Let:

$$p = 23, \quad g = 5$$

Private keys:

$$a = 6, \quad b = 15$$

Public keys:

$$A = 5^6 \bmod 23$$

$$\mathrm{A} = 15625 \bmod 23 = 8$$

$$B = 5^{15} \bmod 23 = 19$$

Shared key User A:

$$K = 19^{6} \bmod 23 = 2$$

Shared key User B:

$$K = 8^{15} \bmod 23 = 2$$

Thus, both obtain the same shared key.

Algorithm

1. Choose a prime number p and primitive root g.
2. User A selects private key a.
3. User B selects private key b.
4. Compute public keys:

$$A = g^{a} \bmod p$$

$$B = g^{b} \bmod p$$

5. Exchange public keys.
6. Compute shared secret:

   User A

$$K = B^{a} \bmod p$$

   User B

$$K = A^{b} \bmod p$$

7. Both obtain the same shared key.

Source Code

```
def diffie_hellman(p, g, a, b):
    A = pow(g, a, p)
    B = pow(g, b, p)

    shared_A = pow(B, a, p)
    shared_B = pow(A, b, p)

    return A, B, shared_A, shared_B

p = int(input("Enter prime number p: "))
g = int(input("Enter primitive root g: "))
a = int(input("Enter private key of A: "))
b = int(input("Enter private key of B: "))

A, B, shared_A, shared_B = diffie_hellman(p, g, a, b)

print("Public key of A:", A)
print("Public key of B:", B)
print("Shared key for A:", shared_A)
print("Shared key for B:", shared_B)
```

Sample Output

```
Enter prime number p: 23
Enter primitive root g: 5
Enter private key of A: 6
Enter private key of B: 15

Public key of A: 8
Public key of B: 19
Shared key for A: 2
Shared key for B: 2
```