

Department of Computer Science and Engineering

Jagannath University, Dhaka-1100

*Course Title:* Cryptography and Information Security Lab

*Course Code: CSEL-4110*

***Submitted By:***

*Name****:*** *Shamim*

*ID****:*** *B190305006*

***Submitted To:***

*Name****:*** *DR. MOHAMMED NASIR UDDIN*

*Professor*

*Department of**CSE Jagannath University*

Table of Contents

[Additive Cipher 3](#_Toc180608898)

[Vigenere Cipher 5](#_Toc180608899)

[Multiplicative Cipher 7](#_Toc180608900)

[Affine Cipher 8](#_Toc180608901)

[Playfair Cipher 9](#_Toc180608902)

[Data Encryption Standard (DES) 13](#_Toc180608903)

[Advanced Encryption Standard (AES) 14](#_Toc180608904)

[*RSA* Encryption 15](#_Toc180608905)

[ElGamal Encryption 16](#_Toc180608906)

[Rabin Cryptosystem 17](#_Toc180608907)

Additive Cipher

# Encryption Function

def encryption(plaintext, key):

text = plaintext.lower()

#Range of lowercase letter is 97 to 122

ciphertext = ""

for char in text:

order = ord(char)

if order >= 97 and order <= 122:

order = order - 97

order = (order + key) % 26

order = order + 97

new\_char = chr(order)

ciphertext = ciphertext + new\_char

else:

ciphertext = ciphertext + char

return ciphertext

#Decryption Function

def decryption(ciphertext, key):

text = ciphertext.upper()

#Range of uppercase letter is 65 to 90

plaintext = ""

for char in text:

order = ord(char)

if order >= 65 and order <= 90:

order = order - 65

order = (order - key) % 26

order = order + 65

new\_char = chr(order)

plaintext = plaintext + new\_char

else:

plaintext = plaintext + char

return plaintext

#Input Section

plaintext = input("Enter the plaintext: ")

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

#Function Calling

ciphertext = encryption(plaintext, key)

decrypted\_text = decryption(ciphertext, key)

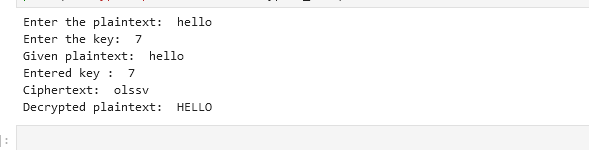
#Output Section

print("Given plaintext: ", plaintext)

print("Entered key : ", key)

print("Ciphertext: ", ciphertext)

print("Decrypted plaintext: ", decrypted\_text)



# Vigenere Cipher

#Key Generation

def key\_generation(key):

key\_len = len(key)

key\_stream = [0]\*key\_len

key = key.lower()

for i in range(key\_len):

order = ord(key[i]) - 97

key\_stream[i] = order

return key\_stream

#Encryption Function

def encryption(plaintext, key\_stream):

text = plaintext.lower()

key\_size = len(key\_stream)

ciphertext = ""

j = 0

for char in text:

order = ord(char)

if order>=97 and order<=122:

#Storing the key for current plaintext character

key = key\_stream[j]

if j==(key\_size-1):

j = 0

else:

j = j+1

#Calculating the ciphertext charater

order = order - 97

order = (order + key) % 26

order = order + 97

new\_char = chr(order)

ciphertext = ciphertext + new\_char

else:

ciphertext = ciphertext + char

return ciphertext

#Decryption Function

def decryption(ciphertext, key\_stream):

text = ciphertext.upper()

key\_size = len(key\_stream)

plaintext = ""

j = 0

for char in text:

order = ord(char)

if order>=65 and order<=90:

#Storing the key for current ciphertext character

key = key\_stream[j]

if j==(key\_size-1):

j = 0

else:

j = j+1

#Calculating the plaintext charater

order = order - 65

order = (order - key) % 26

order = order + 65

new\_char = chr(order)

plaintext = plaintext + new\_char

else:

plaintext = plaintext + char

return plaintext

#Input Section

plaintext = input("Enter the plaintext: ")

key = input("Enter the key: ")

#Function Calling

key\_stream = key\_generation(key)

ciphertext = encryption(plaintext, key\_stream)

decrypted\_text = decryption(ciphertext, key\_stream)

#Output Section

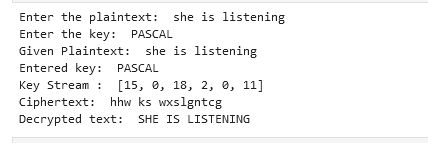
print("Given Plaintext: ", plaintext)

print("Entered key: ", key)

print("Key Stream : ", key\_stream)

print("Ciphertext: ", ciphertext)

print("Decrypted text: ", decrypted\_text)



# Multiplicative Cipher

#Encryption Function

def encryption(plaintext, key):

text = plaintext.lower()

ciphertext = ""

for char in text:

order = ord(char)

#Range of lowercase letter is 97 to 122

if order>= 97 and order<=122:

order = order - 97

order = (order \* key) % 26

order = order + 97

new\_char = chr(order)

ciphertext = ciphertext + new\_char

else:

ciphertext = ciphertext + char

return ciphertext

#Decryption Function

def decryption(ciphertext, key):

text = ciphertext.upper()

#finding multiplicative inverse of the key

key\_inv = pow(key, -1, 26)

plaintext = ""

for char in text:

order = ord(char)

#Range of uppercase letter is 65 to 90

if order>= 65 and order<=90:

order = order - 65

order = (order \* key\_inv) % 26

order = order + 65

new\_char = chr(order)

plaintext = plaintext + new\_char

else:

plaintext = plaintext + char

return plaintext

#Input Section

plaintext = input("Enter the plaintext: ")

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

#Function Calling

ciphertext = encryption(plaintext, key)

decrypted\_text = decryption(ciphertext, key)

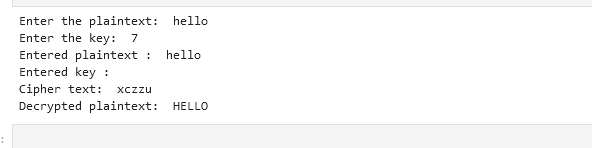
#Output Section

print("Entered plaintext : ", plaintext)

print("Entered key : ")

print("Cipher text: ", ciphertext)

print("Decrypted plaintext: ", decrypted\_text)



# Affine Cipher

#Encryption Function

def encryption(plaintext, key1, key2):

text = plaintext.lower()

ciphertext = ""

for char in text:

order = ord(char)

#Range for lowercase letter is 97 to 122

if order>=97 and order<=122:

order = order - 97

order = ((order \* key1) + key2) % 26

order = order + 97

new\_char = chr(order)

ciphertext = ciphertext + new\_char

else :

ciphertext = ciphertext + char

return ciphertext

#Decrption Function

def decryption(ciphertext, key1, key2):

text = ciphertext.upper()

#finding the inverse of key1 mod 26

key1\_inv = pow(key1, -1, 26)

plaintext = ""

for char in text:

order = ord(char)

#Range for uppercase letter is 65 to 90

if order>=65 and order<=90:

order = order - 65

order = ((order - key2) \* key1\_inv) % 26

order = order + 65

new\_char = chr(order)

plaintext = plaintext + new\_char

else:

plaintext = plaintext + char

return plaintext

#Input section

plaintext = input("Enter the plaintext: ")

key1 = int(input("Enter the first key: "))

key2 = int(input("Enter the second key : "))

#Function calling

ciphertext = encryption(plaintext, key1, key2)

decrypted\_text = decryption(ciphertext, key1, key2)

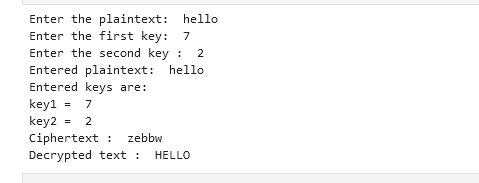
#Output Section

print("Entered plaintext: ", plaintext)

print("Entered keys are: \nkey1 = ", key1, "\nkey2 = ", key2)

print("Ciphertext : ", ciphertext)

print("Decrypted text : ", decrypted\_text)



# Playfair Cipher

import numpy as np

#Declaring the 5\*5 key matrix

key\_matrix = np.array(

[

['L', 'G', 'D', 'B', 'A'],

['Q', 'M', 'H', 'E', 'C'],

['U', 'R', 'N', 'J', 'F'],

['X', 'V', 'S', 'O', 'K'],

['Z', 'Y', 'W', 'T', 'P']

]

)

#Creating transpose matrix of the key matrix

transpose\_key\_matrix = np.transpose(key\_matrix)

#Input Section

plaintext = input("Enter the plaintext: ")

print("Given Plaintext: ", plaintext)

#Removing all the whitespaces from the plaintext

text = plaintext.replace(" ","")

text\_len = len(text)

text = text.upper()

#Replace all "I" in the plaintext to "j"

text = text.replace("I", "J")

#Make pair of two(different) characters from plaintext

plaintextpair = []

i = 0

while i < text\_len:

char1 = text[i]

char2 = ""

#If the letter is the last character of the plaintext add a vogus character "X"

if (i+1) == len(text):

char2 = "X"

#Else add the next character

else:

char2 = text[i+1]

#If the two characters are different insert them in the pair

if char1 != char2:

plaintextpair.append(char1+char2)

i = i+2

#else add "X" as the second character

else:

plaintextpair.append(char1+"X")

i = i + 1

print("Pairs of plaintext :", plaintextpair)

#Encryption Function

ciphertext = ""

ciphertextpair = []

for pair in plaintextpair:

apply\_rule = True

#Rule 1: If the two characters are in the same row replace them with their right charater

if apply\_rule :

for row in range(5):

if pair[0] in key\_matrix[row] and pair[1] in key\_matrix[row]:

for i in range(5):

if key\_matrix[row][i]==pair[0]:

char1 = key\_matrix[row][(i+1)%5]

elif key\_matrix[row][i]==pair[1]:

char2 = key\_matrix[row][(i+1)%5]

apply\_rule = False

ciphertextpair.append(char1+char2)

ciphertext = ciphertext + char1 + char2

#Rule 2: If the two characters are in the same column replace them with their below character

#for this we will use transpose matrix

if apply\_rule :

for column in range(5):

if pair[0] in transpose\_key\_matrix[column] and pair[1] in transpose\_key\_matrix[column]:

for i in range(5):

if transpose\_key\_matrix[column][i]==pair[0]:

char1 = transpose\_key\_matrix[column][(i+1)%5]

elif transpose\_key\_matrix[column][i]==pair[1]:

char2 = transpose\_key\_matrix[column][(i+1)%5]

apply\_rule = False

ciphertextpair.append(char1+char2)

ciphertext = ciphertext + char1 + char2

#Rule 3: If the two letters are not in the same row or column,replace them with letter

# that is in its own row but in the same column as the other letter.

if apply\_rule :

for row in range (5):

for column in range (5):

if key\_matrix[row][column] == pair[0]:

x0 = row

y0 = column

elif key\_matrix[row][column] == pair[1]:

x1 = row

y1 = column

char1 = key\_matrix[x0][y1]

char2 = key\_matrix[x1][y0]

ciphertextpair.append(char1+char2)

ciphertext = ciphertext + char1 + char2

print("Ciphertext: ", ciphertext)

#Decryption Function

decryptedtext = ""

for pair in ciphertextpair:

apply\_rule = True

#Rule 1: If the two characters are in the same row replace them with their left charater

if apply\_rule :

for row in range(5):

if pair[0] in key\_matrix[row] and pair[1] in key\_matrix[row]:

for i in range(5):

if key\_matrix[row][i]==pair[0]:

char1 = key\_matrix[row][(i-1)%5]

elif key\_matrix[row][i]==pair[1]:

char2 = key\_matrix[row][(i-1)%5]

apply\_rule = False

decryptedtext = decryptedtext + char1 + char2

#Rule 2: If the two characters are in the same column replace them with their upper character

#for this we will use transpose matrix

if apply\_rule :

for column in range(5):

if pair[0] in transpose\_key\_matrix[column] and pair[1] in transpose\_key\_matrix[column]:

for i in range(5):

if transpose\_key\_matrix[column][i]==pair[0]:

char1 = transpose\_key\_matrix[column][(i-1)%5]

elif transpose\_key\_matrix[column][i]==pair[1]:

char2 = transpose\_key\_matrix[column][(i-1)%5]

apply\_rule = False

decryptedtext = decryptedtext + char1 + char2

#Rule 3: If the two letters are not in the same row or column,replace them with letter

# that is in its own row but in the same column as the other letter.

if apply\_rule :

for row in range (5):

for column in range (5):

if key\_matrix[row][column] == pair[0]:

x0 = row

y0 = column

elif key\_matrix[row][column] == pair[1]:

x1 = row

y1 = column

char1 = key\_matrix[x0][y1]

char2 = key\_matrix[x1][y0]

decryptedtext = decryptedtext + char1 + char2

print("Decrypted text: ", decryptedtext.lower())









# Data Encryption Standard (DES)

import base64

from Crypto.Cipher import DES

from Crypto.Random import get\_random\_bytes

#Input plaintext

plaintext = input("Enter the plaintext: ");

#Padding the plaintext

while len(plaintext) % 8 != 0:

plaintext = plaintext + " "

#Create a random key

key = get\_random\_bytes(8)

#Create model of the cipher

des = DES.new(key, DES.MODE\_ECB)

#Encryption Part

ciphertext = des.encrypt(plaintext.encode('utf-8'))

print("Ciphertext: ", base64.b64encode(ciphertext))

#Decryptiom Part

decryptedtext = des.decrypt(ciphertext)

print("Decrypted text : ", decryptedtext.decode())

# Advanced Encryption Standard (AES)

import base64

from Crypto.Cipher import AES

from Crypto.Random import get\_random\_bytes

plaintext = b'This is a secret message'

key = get\_random\_bytes(16)

cipher = AES.new(key, AES.MODE\_EAX)

ciphertext, tag = cipher.encrypt\_and\_digest(plaintext)

print("Ciphertext : ", base64.b64encode(ciphertext))

print("Tag : ", tag)

decrypt\_cipher = AES.new(key, AES.MODE\_EAX, nonce=cipher.nonce)

decrypted\_text =decrypt\_cipher.decrypt\_and\_verify(ciphertext, tag)

print("Decrypted text: ", decrypted\_text.decode())

*RSA* Encryption

*#A Python Code for Encryption Using RSA Algorithm*

**from** Crypto.PublicKey **import** RSA

**from** Crypto.Cipher **import** PKCS1\_OAEP

*#Function for generating public and private key*

**def** generate\_key\_pair():

key **=** RSA**.**generate(2048)

public\_key **=** key**.**publickey()**.**export\_key()

private\_key **=** key**.**export\_key()

**return** public\_key, private\_key

*#Encryption Function*

**def** encrypt(message, public\_key):

cipher **=** PKCS1\_OAEP**.**new(RSA**.**import\_key(public\_key))

encrypted\_message **=** cipher**.**encrypt(message)

**return** encrypted\_message

*#Decryption Function*

**def** decrypt(encrypted\_message, private\_key):

cipher **=** PKCS1\_OAEP**.**new(RSA**.**import\_key(private\_key))

decrypted\_message **=** cipher**.**decrypt(encrypted\_message)

**return** decrypted\_message

*# Example usage*

plaintext **=** b"This is a secret message from TAJ"

print("Plaintext:", plaintext)

print("----output----")

*# Generate key pair*

public\_key, private\_key **=** generate\_key\_pair()

*# Encrypt the message*

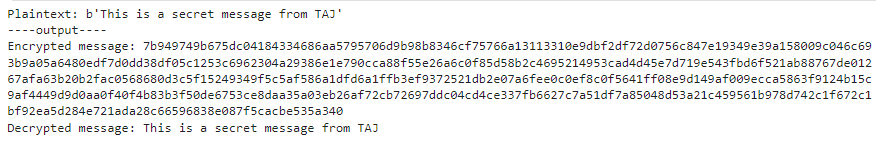
encrypted\_message **=** encrypt(plaintext, public\_key)

print("Encrypted message:", encrypted\_message**.**hex())

*# Decrypt the message*

decrypted\_message **=** decrypt(encrypted\_message, private\_key)

print("Decrypted message:", decrypted\_message**.**decode())



ElGamal Encryption

*# Sympy is a Python library for symbolic mathematics.*

**from** sympy **import** primitive\_root,randprime

**import** random

*# The number for which you want to find the primitive root*

prime **=** randprime(124,10**\*\***3)

root **=** primitive\_root(prime)

d**=**random**.**randint(1,(prime**-**2)) *# It is private key.*

e**=**(pow(root,d)**%prime**) # It is public key.

r**=**random**.**randint(1,10) *# Select a random integer.*

*#Define the plaintext.*

plaintext **=** "This is a secret message"

*# Encryption Algorithm.*

ciphertext**=**[]

**for** char **in** plaintext:

ciphertext1**=**(pow(root,r)**%prime**)

ciphertext2**=**((ord(char)**\***pow(e,r))**%prime**)

ciphertext**.**append((ciphertext1,ciphertext2))

print(ciphertext)

*#Decryption Algorithm*

plaintext**=**""

**for** pair **in** ciphertext:

ciphertext1,ciphertext2**=**pair

value**=**pow(ciphertext1,d)

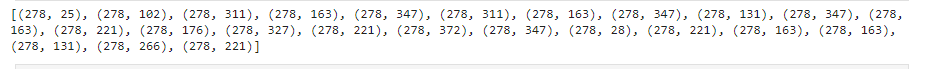
multinv **=** pow(value,**-**1,prime)

decrypt\_char **=** (ciphertext2**\***multinv) **%** prime

plaintext **+=** chr(decrypt\_char)

print(plaintext)







Rabin Cryptosystem

# Helper function: Extended Euclidean Algorithm to find the modular inverse

def extended\_gcd(a, b):

if b == 0:

return a, 1, 0

gcd, x1, y1 = extended\_gcd(b, a % b)

x = y1

y = x1 - (a // b) \* y1

return gcd, x, y

def mod\_inverse(a, m):

gcd, x, \_ = extended\_gcd(a, m)

if gcd != 1:

raise ValueError("Modular inverse does not exist")

else:

return x % m

# Helper function: Modular Exponentiation

def mod\_exp(base, exp, mod):

result = 1

base = base % mod

while exp > 0:

if (exp % 2) == 1: # If exp is odd, multiply base with result

result = (result \* base) % mod

exp = exp >> 1 # Divide the exponent by 2

base = (base \* base) % mod # Square the base

return result

# Step 1: Key Generation

def generate\_keys():

# Prime numbers p and q (small values for simplicity, use larger primes in real-world use)

p = 7

q = 11

n = p \* q

return p, q, n

# Step 2: Encryption (Public key is n, plaintext is m)

def encrypt(m, n):

return mod\_exp(m, 2, n)

# Step 3: Decryption (Private keys are p, q)

def decrypt(c, p, q, n):

# Compute square roots modulo p and q

mp = mod\_exp(c, (p + 1) // 4, p)

mq = mod\_exp(c, (q + 1) // 4, q)

# Use Chinese Remainder Theorem to get four possible plaintexts

inv\_q = mod\_inverse(q, p)

inv\_p = mod\_inverse(p, q)

x1 = (mp \* q \* inv\_q + mq \* p \* inv\_p) % n

x2 = (mp \* q \* inv\_q - mq \* p \* inv\_p) % n

# Return the four possible solutions

return x1, n - x1, x2, n - x2

# Example Usage

if \_\_name\_\_ == "\_\_main\_\_":

# Generate keys

p, q, n = generate\_keys()

print(f"Public key (n): {n}, Private keys (p, q): ({p}, {q})")

# Sample message (must be smaller than n)

m = 5

print(f"Original message: {m}")

# Encrypt the message

c = encrypt(m, n)

print(f"Encrypted message (ciphertext): {c}")

# Decrypt the ciphertext

possible\_messages = decrypt(c, p, q, n)

print(f"Decrypted possible messages: {possible\_messages}")

