Cryptography and Information Security Lab

CSEL-4110

Assignment on Various Cryptographic Techniques and Algorithms

Submitted By

MD. WALIUL ISLAM RAYHAN
ID: B190305034

Submitted To

Dr. Mohammed Nasir Uddin
Professor
Dept. of CSE
Jagannath University, Dhaka - 1100

Additive Cipher (Caesar Cipher)

An Additive Cipher is a type of substitution cipher where each letter in the plaintext is shifted by a certain number of positions down the alphabet. It's also known as a Caesar Cipher when the shift value is fixed (typically by 1 or 3), but in an additive cipher, the shift can be any value in the range 0 to 25 (mod 26).

The encryption rule for an additive cipher is:

$$\mathrm{C}=(P+K) \bmod 26$$

Where:

- C is the ciphertext letter
- P is the plaintext letter
- K is the key (shift value)
- mod26 ensures the values wrap around if they exceed 25 (the number of letters in the English alphabet)

Similarly, for decryption:

$$\mathrm{P}=(\mathrm{C}-\mathrm{K}) \bmod 26$$

This method is one of the simplest classical encryption techniques but is weak against frequency analysis since it follows predictable patterns.

Python Code for Additive Cipher

Below is a Python implementation where the user can input plaintext, the shift (key), and the program will both encrypt and decrypt using the 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)
```

Example

Input:

Plaintext: HELLO and Shift (Key): 3

Process: Each letter is shifted by 3 positions down the alphabet.

- H becomes K
- E becomes H
- L becomes O

- L becomes O
- becomes R

Ciphertext: KHOOR

Output:

Encrypted Ciphertext: KHOOR

Decrypted Text: HELLO

This simple cipher demonstrates how shifting letters can produce encryption, but it’s also easy to break using various cryptanalysis methods due to the limited number of possible shifts.

Multiplicative Cipher

The Multiplicative Cipher is a type of substitution cipher in which each letter in the plaintext is multiplied by a fixed key. This cipher works by mapping each letter to a number (A = 0, B = 1, ..., Z = 25) and then multiplying this number by the key modulo 26.

The encryption rule for a multiplicative cipher is:

$$C = (P \times K) \bmod 26$$

Where:

- C is the ciphertext letter
- P is the plaintext letter
- K is the key (multiplicative value)

For decryption, the inverse of the key modulo 26 is required. The decryption formula is:

$$P = (C \times K^{-1}) \bmod 26$$

Where:

K^{-1} is the multiplicative inverse of the key under modulo 26. Not all keys are valid; the key must be coprime with 26 (i.e., their greatest common divisor is 1) for the cipher to work.

Python Code for Multiplicative Cipher

Below is a Python implementation that allows the user to input plaintext and a key, and it performs both encryption and decryption.

```
# Function to find multiplicative inverse of key under mod 26
def multiplicative_inverse(key):
    for i in range(26):
        if (key * i) % 26 == 1:
            return i
return None

# Function to encrypt using multiplicative cipher
def encrypt_multiplicative_cipher(plaintext, key):
    if gcd(key, 26) != 1:
        raise ValueError("Key is not valid. It must be coprime with 26.")

    ciphertext = ''
    for char in plaintext:
        if char.isalpha():
            shift_base = 65 if char.isupper() else 97
```

```
            encrypted_char = chr(((ord(char) - shift_base) * key) % 26 + shift_base)
            ciphertext += encrypted_char
        else:
            ciphertext += char # non-alphabet characters remain unchanged
    return ciphertext

# Function to decrypt using multiplicative cipher
def decrypt_multiplicative_cipher(ciphertext, key):
    inverse_key = multiplicative_inverse(key)
    if inverse_key is None:
        raise ValueError("Multiplicative inverse not found. Invalid key.")

    plaintext = ''
    for char in ciphertext:
        if char.isalpha():
            shift_base = 65 if char.isupper() else 97
            decrypted_char = chr(((ord(char) - shift_base) * inverse_key) % 26 + shift_base)
            plaintext += decrypted_char
        else:
            plaintext += char # non-alphabet characters remain unchanged
    return plaintext

# Function to calculate the greatest common divisor (GCD)
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Main program to accept user input
if __name__ == "__main__":
    # Input from the user
    plaintext = input("Enter the plaintext: ")
    key = int(input("Enter the multiplicative key: "))

    # Encrypt the plaintext
    try:
        ciphertext = encrypt_multiplicative_cipher(plaintext, key)
        print(f"Encrypted Ciphertext: {ciphertext}")

        # Decrypt the ciphertext
        decrypted_text = decrypt_multiplicative_cipher(ciphertext, key)
        print(f"Decrypted Text (should match the original plaintext): {decrypted_text}")
    except ValueError as e:
        print(f"Error: {e}")
```

Example

Input:

Plaintext: HELLO

Multiplicative Key: 5

Process:

1. Convert letters to numbers: H = 7, E = 4, L = 11, O = 14
2. Multiply each letter by 5 and apply modulo 26:
   - H: (7×5) mod 26 = 9 → J
   - E: (4×5) mod 26 = 20 → U
   - L: (11×5) mod 26 = 5 → F
   - L: (11×5) mod 26 = 5 → F
   - O: (14×5) mod 26 = 18 → S

Ciphertext: JUFFS

Decryption Process:

1. First, we find the multiplicative inverse of 5 modulo 26, which is 21 because (5×21) mod 26=1.
2. Now, multiply the ciphertext by 21 and apply modulo 26:
   - J: (9×21) mod 26 = 7 → H
   - U: (20×21) mod 26 = 4 → E
   - F: (5×21) mod 26 = 11 → L
   - F: (5×21) mod 26 = 11 → L
   - S: (18×21) mod 26 = 14 → O

Decrypted Text: HELLO

Key Consideration:

The key must be coprime with 26, meaning the greatest common divisor (GCD) of the key and 26 must be 1. If the key isn't valid, the algorithm will throw an error.

This encryption technique is more complex than an additive cipher but still vulnerable to attacks such as frequency analysis.

Affine Cipher

The Affine Cipher is a type of substitution cipher that combines both multiplicative and additive ciphers. It operates by first multiplying the plaintext character by a key (mod 26) and then adding another key. The affine cipher uses the following formula for encryption:

$$C = (P \times K_1 + K_2) \bmod 26$$

Where:

- C is the ciphertext letter
- P is the plaintext letter (converted to a number from 0 to 25, where A = 0, B = 1, ..., Z = 25)
- K_1 is the multiplicative key, which must be coprime with 26
- K_2 is the additive key

For decryption, the process is reversed using the following formula:

$$P = (C - K_2) \times K_1^{-1} \bmod 26$$

Where:

- K_1^{-1} is the multiplicative inverse of K_1 under modulo 26.

Python Code for Affine Cipher

Here is a Python implementation of the Affine Cipher, which allows the user to input plaintext, a multiplicative key K_1, and an additive key K_2. The program handles both encryption and decryption.

```
# Function to find multiplicative inverse of K1 under mod 26
def multiplicative_inverse(K1):
    for i in range(26):
        if (K1 * i) % 26 == 1:
            return i
    return None

# Function to calculate the greatest common divisor (GCD)
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Function to encrypt using affine cipher
def encrypt_affine_cipher(plaintext, K1, K2):
    if gcd(K1, 26) != 1:
```

```
        raise ValueError("Multiplicative key K1 must be coprime with 26.")

    ciphertext = ''
    for char in plaintext:
        if char.isalpha():
            shift_base = 65 if char.isupper() else 97
            encrypted_char = chr(((ord(char) - shift_base) * K1 + K2) % 26 + shift_base)
            ciphertext += encrypted_char
        else:
            ciphertext += char # non-alphabet characters remain unchanged
    return ciphertext

# Function to decrypt using affine cipher
def decrypt_affine_cipher(ciphertext, K1, K2):
    inverse_K1 = multiplicative_inverse(K1)
    if inverse_K1 is None:
        raise ValueError("Multiplicative inverse not found. Invalid K1.")

    plaintext = ''
    for char in ciphertext:
        if char.isalpha():
            shift_base = 65 if char.isupper() else 97
            decrypted_char = chr(((ord(char) - shift_base - K2) * inverse_K1) % 26 + shift_base)
            plaintext += decrypted_char
        else:
            plaintext += char # non-alphabet characters remain unchanged
    return plaintext

# Main program to accept user input
if __name__ == "__main__":
    # Input from the user
    plaintext = input("Enter the plaintext: ")
    K1 = int(input("Enter the multiplicative key (K1): "))
    K2 = int(input("Enter the additive key (K2): "))

    # Encrypt the plaintext
    try:
        ciphertext = encrypt_affine_cipher(plaintext, K1, K2)
        print(f"Encrypted Ciphertext: {ciphertext}")

        # Decrypt the ciphertext
        decrypted_text = decrypt_affine_cipher(ciphertext, K1, K2)
        print(f"Decrypted Text (should match the original plaintext): {decrypted_text}")
    except ValueError as e:
        print(f"Error: {e}")
```

Example

Input:

- Plaintext: HELLO
- Multiplicative Key K_1: 5
- Additive Key K_2: 8

Process:

1. Convert letters to numbers: H = 7, E = 4, L = 11, O = 14
2. Apply the encryption formula $C = (P \times K_1 + K_2) \bmod 26$:
   - H: (7×5+8) mod 26 = 17 → R
   - E: (4×5+8) mod 26 = 2 → C
   - L: (11×5+8) mod 26 = 11 → L
   - L: (11×5+8) mod 26 = 11 → L
   - O: (14×5+8) mod 26 = 0 → A

Ciphertext: RCLLA

Decryption Process:

1. Find the multiplicative inverse of K_1 =5 under mod 26, which is 21 (because 5×21mod26=1).
2. Apply the decryption formula $P = (C - K_2) \times K_1^{-1} \bmod 26$:
   - R: ((17−8) ×21) mod 26=7 → H
   - C: ((2−8) ×21) mod26=4 → E
   - L: ((11−8) ×21) mod26=11 → L
   - L: ((11−8) ×21) mod26=11 → L
   - A: ((0−8) ×21) mod26=14 → O

Decrypted Text: HELLO

Key Considerations:

- K_1, and K_2 must be coprime with 26 to ensure a valid multiplicative inverse exists. Common valid choices for K_1 include 1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, and 25.
- K_2 can be any integer between 0 and 25.

The Affine Cipher is stronger than simple substitution ciphers, but it is still susceptible to frequency analysis, especially when the keys are not carefully chosen.

Auto Key Cipher

The Auto Key Cipher is a polyalphabetic substitution cipher, which is an enhancement of the Vigenère cipher. In this cipher, the keyword is followed by the plaintext itself as the key, which increases the cryptographic security compared to traditional ciphers.

In the Auto Key Cipher:

- The first part of the key is the provided keyword.
- The rest of the key is formed by appending the plaintext to the keyword.

For encryption:

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

Where:

- C_i is the *i*-th character of the ciphertext,
- P_i is the *i*-th character of the plaintext,
- K_i is the corresponding character from the keyword (or plaintext, after the keyword),
- The result is taken modulo 26 (since there are 26 letters in the alphabet).

For decryption:

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

The key for decryption will be derived from both the keyword and part of the already decrypted text.

Python Code for Auto Key Cipher

Here's a Python implementation that allows the user to input plaintext and a keyword, and the program performs both encryption and decryption using the Auto Key cipher.

```
# Function to encrypt using Auto Key cipher
def encrypt_auto_key(plaintext, keyword):
    plaintext = plaintext.upper().replace(" ", "")
    keyword = keyword.upper()

    # Generate the key by appending the plaintext after the keyword
    key = keyword + plaintext[:len(plaintext) - len(keyword)]

    ciphertext = ''
    for i in range(len(plaintext)):
        # Encryption formula: (P + K) mod 26
        P = ord(plaintext[i]) - 65
```

```
        K = ord(key[i]) - 65
        C = (P + K) % 26
        ciphertext += chr(C + 65)
    return ciphertext

# Function to decrypt using Auto Key cipher
def decrypt_auto_key(ciphertext, keyword):
    ciphertext = ciphertext.upper().replace(" ", "")
    keyword = keyword.upper()

    # The key starts with the keyword and is extended by the decrypted text
    key = keyword
    plaintext = ''

    for i in range(len(ciphertext)):
        # Decryption formula: (C - K) mod 26
        C = ord(ciphertext[i]) - 65
        K = ord(key[i]) - 65
        P = (C - K + 26) % 26
        decrypted_char = chr(P + 65)
        plaintext += decrypted_char
        # Append decrypted character to key to extend it
        key += decrypted_char
    return plaintext

# Main program to accept user input
if __name__ == "__main__":
    # Input from the user
    plaintext = input("Enter the plaintext: ")
    keyword = input("Enter the keyword: ")

    # Encrypt the plaintext
    ciphertext = encrypt_auto_key(plaintext, keyword)
    print(f"Encrypted Ciphertext: {ciphertext}")

    # Decrypt the ciphertext
    decrypted_text = decrypt_auto_key(ciphertext, keyword)
    print(f"Decrypted Text (should match the original plaintext): {decrypted_text}")
```

Example

Input:

- Plaintext: HELLO
- Keyword: KEY

Process:

1. The key is formed by appending the plaintext to the keyword:
   - Key: KEYHE
2. Encrypt the plaintext using the key:
   - H = 7, K = 10: (7+10) mod 26 = 17 → R
   - E = 4, E = 4: (4+4) mod 26 = 8 → I
   - L = 11, Y = 24: (11+24) mod 26 = 9 → J
   - L = 11, H = 7: (11+7) mod 26 = 18 → S
   - = 14, E = 4: (14+4) mod 26 = 18 → S

Ciphertext: RIJSS

Decryption Process:

1. Start with the keyword KEY.
2. Decrypt the ciphertext step by step, and append each decrypted character to the key:

   R: (17−10) mod 26 = 7 → H (key becomes KEYH)

   I: (8−4) mod 26 = 4 → E (key becomes KEYHE)

   J: (9−24) mod 26 = 11 → L (key becomes KEYHEL)

   S: (18−7) mod 26 = 11 → L (key becomes KEYHELL)

   S: (18−4) mod 26 = 14 → O

Decrypted Text: HELLO

Key Considerations:

- The key grows dynamically by including the plaintext during encryption or decrypted text during decryption.
- This cipher is still susceptible to some forms of cryptanalysis (like known-plaintext attacks), but it's stronger than simple substitution ciphers.

Playfair Cipher

The Playfair Cipher is a digraph substitution cipher, meaning it encrypts pairs of letters (digraphs) instead of single letters, which provides better security than simple monoalphabetic ciphers like Caesar or Vigenère. It was invented by Charles Wheatstone in 1854 but was named after Lord Playfair, who promoted its use.

Features:

1. **Key Matrix:** The Playfair cipher uses a 5x5 matrix of letters constructed using a keyword. The keyword is written first, and then the remaining letters of the alphabet (excluding one, typically "J") fill up the matrix. For example, if the keyword is "PLAYFAIR", the matrix might look like:

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

   - **In this matrix:**

   "I" and "J" are considered the same letter (so only "I" is used).

2. **Encryption Rules:** The plaintext is divided into pairs of letters (digraphs) and then encrypted according to their positions in the matrix.
   - **Same row:** If the letters are in the same row, replace each with the letter to its right (wrap around to the start if necessary).
   - **Same column:** If the letters are in the same column, replace each with the letter below it (wrap around to the top if necessary).
   - **Rectangle:** If the letters form a rectangle, replace them with the letters on the same row but in the opposite corners of the rectangle.
   - **Identical pair:** If there is a pair of identical letters, insert a filler letter (usually "X") between them and treat them as two different pairs.

3. **Decryption:** The process is reversed, using the same key matrix and reversing the encryption rules.

Python Code for Playfair Cipher

Here is a Python implementation of the Playfair cipher, which allows the user to input plaintext and a keyword. The program handles both encryption and decryption.

```
import numpy as np

# Declaring the 5x5 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 bogus 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 character
    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
    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 letters
    # in the same row as the other letter but in their respective columns.
    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 character
    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
    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 letters
    # in the same row as the other letter but in their respective columns.
    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())
```

Example

Input:

- Plaintext: HELLO
- Keyword: PLAYFAIR

Process:

1. Create a 5x5 matrix from the keyword PLAYFAIR:

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

2. Split the plaintext into digraphs:
    - HELLO, becomes HE LX LO
3. Encrypt each digraph:
    - HE: Rectangle rule → KC
    - LX: Rectangle rule → YW
    - LO: Rectangle rule → OP

    So, the ciphertext is KCYWOP.

Output:

- Encrypted Ciphertext: KCYWOP
- Decrypted Text: HELXLO (with the filler X added during encryption)

Decryption Process:

Using the same matrix and reversing the rules, KCYWOP is decrypted back to HELXLO.

Vigenère Cipher

The Vigenère Cipher is a polyalphabetic substitution cipher that uses a keyword to perform multiple shifts on the plaintext. It is one of the most famous classical ciphers and is much stronger than a simple Caesar cipher due to its multiple shifting techniques.

Encryption: Each letter of the plaintext is shifted by a few positions based on the corresponding letter of the repeating keyword. Formula for encryption:

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

Where:

- P_i is the i-th letter of the plaintext (converted to a number from 0 to 25, where A=0, B=1, ..., Z=25)
- K_i is the i-th letter of the repeating keyword (also converted to a number)
- C_i is the resulting ciphertext letter.

Decryption: The reverse process of encryption, using:

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

Where:

- C_i is the ciphertext letter, and
- K_i is the corresponding letter of the keyword.

Python Code for Vigenère Cipher

Here's the Python implementation that accepts user input for both plaintext and the keyword and handles both encryption and decryption.

```
# Function to convert letter to number (A=0, B=1, ..., Z=25)
def letter_to_number(letter):
    return ord(letter.upper()) - ord('A')

# Function to convert number to letter (0=A, 1=B, ..., 25=Z)
def number_to_letter(number):
    return chr((number % 26) + ord('A'))

# Function to repeat the keyword to match the length of the plaintext
def repeat_keyword(plaintext, keyword):
    keyword = keyword.upper()
```

```
    keyword_repeated = ''
    keyword_length = len(keyword)
    for i in range(len(plaintext)):
        if plaintext[i].isalpha():
            keyword_repeated += keyword[i % keyword_length]
        else:
            keyword_repeated += plaintext[i]  # Keep non-alphabet characters unchanged
    return keyword_repeated

# Function to encrypt the plaintext using Vigenère cipher
def encrypt_vigenere(plaintext, keyword):
    plaintext = plaintext.upper()
    keyword_repeated = repeat_keyword(plaintext, keyword)

    ciphertext = ''
    for i in range(len(plaintext)):
        if plaintext[i].isalpha():  # Encrypt only alphabetic characters
            P = letter_to_number(plaintext[i])
            K = letter_to_number(keyword_repeated[i])
            C = (P + K) % 26  # Vigenère encryption formula
            ciphertext += number_to_letter(C)
        else:
            ciphertext += plaintext[i]  # Keep non-alphabet characters unchanged

    return ciphertext

# Function to decrypt the ciphertext using Vigenère cipher
def decrypt_vigenere(ciphertext, keyword):
    ciphertext = ciphertext.upper()
    keyword_repeated = repeat_keyword(ciphertext, keyword)

    plaintext = ''
    for i in range(len(ciphertext)):
        if ciphertext[i].isalpha():  # Decrypt only alphabetic characters
            C = letter_to_number(ciphertext[i])
            K = letter_to_number(keyword_repeated[i])
            P = (C - K + 26) % 26  # Vigenère decryption formula
            plaintext += number_to_letter(P)
        else:
            plaintext += ciphertext[i]  # Keep non-alphabet characters unchanged

    return plaintext

# Main program to accept user input
if __name__ == "__main__":
    # Input from the user
    plaintext = input("Enter the plaintext: ")
```

```
keyword = input("Enter the keyword: ")
# Encrypt the plaintext
ciphertext = encrypt_vigenere(plaintext, keyword)
print(f"Encrypted Ciphertext: {ciphertext}")

# Decrypt the ciphertext
decrypted_text = decrypt_vigenere(ciphertext, keyword)
print(f"Decrypted Text (should match the original plaintext): {decrypted_text}")
```

Example

Input:

- Plaintext: HELLO WORLD
- Keyword: KEY

Process:

1. Repeat the keyword to match the length of the plaintext (ignoring spaces and punctuation):
   - Plaintext: HELLO WORLD
   - Repeated keyword: KEYKE KEYKE
2. Encryption using the Vigenère formula:
   - H = 7, K = 10: (7+10) mod 26=17 → R
   - E = 4, E = 4: (4+4) mod 26=8 → I
   - L = 11, Y = 24: (11+24) mod 26=9 → J
   - L = 11, K = 10: (11+10) mod 26=21 → V
   - = 14, E = 4: (14+4) mod 26=18 → S

   Space remains unchanged

   - W = 22, K = 10: (22+10) mod 26=6 → G
   - O= 14, E = 4: (14+4) mod 26=18 → S
   - R = 17, Y = 24: (17+24) mod 26=15 → P
   - L = 11, K = 10: (11+10) mod 26=21 → V
   - D = 3, E = 4: (3+4) mod 26=7 → H

Ciphertext: RIJVS GSPVH

Output:

- Encrypted Ciphertext: RIJVS GSPVH
- Decrypted Text: HELLO WORLD

Key Points:

- **Strength:** The Vigenère cipher is much stronger than simple substitution ciphers like Caesar because of its polyalphabetic nature. A letter can be encrypted into different ciphertext letters depending on its position.

- **Keyword:** The security of the Vigenère cipher relies heavily on the length and complexity of the keyword. A short or simple keyword can make it vulnerable to attacks such as frequency analysis.

Feistel Cipher

A Feistel Cipher is a symmetric structure used to build block ciphers. It's named after the German-born cryptographer Horst Feistel and is the basis of many block ciphers, including DES (Data Encryption Standard). The main advantage of the Feistel structure is that encryption and decryption operations are almost identical, which simplifies implementation.

Feistel Cipher Structure:

1. **Input:** The plaintext is divided into two halves, typically denoted as L (left half) and R (right half).
2. **Rounds:** The cipher undergoes multiple rounds of processing, and in each round:
   - The left half becomes the right half for the next round.
   - The right half is XORed with a function applied to the left half and a round-specific key.
   - This function is called the round function (F).
3. **Decryption:** Decryption in the Feistel cipher uses the same structure as encryption, but the round keys are applied in reverse order.

Basic Structure:

- $L_{i+1}=R_i$
- $R_{i+1}= L_i \oplus F(R_i, K_i)$

Where:

- L_i and R_i are the left and right halves of the input at round i,
- F is the round function,
- K_i is the round key for round i,
- $\oplus$ is the XOR operation.

Python Code for Feistel Cipher

Here is a basic Python implementation of a Feistel Cipher. In this example:

- The plaintext is divided into two halves.
- A simple XOR function serves as the round function.
- We use a fixed number of rounds for encryption and decryption.

```
# Example round function: simple XOR with the key
def round_function(data, key):
    return data ^ key

# Function to encrypt using the Feistel cipher
def feistel_encrypt(plaintext, keys, num_rounds):
    # Split the plaintext into two halves
    L = plaintext >> 8  # Left half (upper 8 bits)
    R = plaintext & 0xFF  # Right half (lower 8 bits)

    # Apply the Feistel rounds
    for i in range(num_rounds):
        new_L = R
        new_R = L ^ round_function(R, keys[i])
        L = new_L
        R = new_R

    # Combine the two halves and return the ciphertext
    ciphertext = (L << 8) | R
    return ciphertext

# Function to decrypt using the Feistel cipher
def feistel_decrypt(ciphertext, keys, num_rounds):
    # Split the ciphertext into two halves
    L = ciphertext >> 8  # Left half (upper 8 bits)
    R = ciphertext & 0xFF  # Right half (lower 8 bits)

    # Apply the Feistel rounds in reverse order
    for i in range(num_rounds - 1, -1, -1):
        new_R = L
        new_L = R ^ round_function(L, keys[i])
        L = new_L
        R = new_R

    # Combine the two halves and return the decrypted plaintext
    plaintext = (L << 8) | R
    return plaintext

# Main program to accept user input
if __name__ == "__main__":
    # Example usage with 16-bit block and 8-bit keys
    num_rounds = 4
    keys = [0x3, 0x7, 0xF, 0xA]  # Example round keys (4 rounds)

    # Input plaintext (16-bit)
    plaintext = int(input("Enter a 16-bit plaintext (as a number): "))
```

```
# Encrypt the plaintext
ciphertext = feistel_encrypt(plaintext, keys, num_rounds)
print(f"Ciphertext (in hexadecimal): {ciphertext:04x}")

# Decrypt the ciphertext
decrypted_text = feistel_decrypt(ciphertext, keys, num_rounds)
print(f"Decrypted Text (in decimal, should match original): {decrypted_text}")
```

Example

Input:

- Plaintext: 1234 (decimal)

Process:

1. The plaintext 1234 (decimal) is split into two 8-bit halves:

   Left half: 4D (hexadecimal) or 77 (decimal)

   Right half: D2 (hexadecimal) or 210 (decimal)

2. **Feistel Rounds:**

- Round 1:

  Left half = R = 210

  Right half = L ^ F (R, K1) = 77 ^ F (210, 0x03)

- Continue for 4 rounds with different keys.

3. **Ciphertext:** After 4 rounds, the final left and right halves are combined to form the ciphertext.

Output:

- Ciphertext: dad0
- Decrypted Text: 1234 (original plaintext after decryption)

Applications:

The Feistel structure is used in many well-known block ciphers such as:

- DES (Data Encryption Standard): A widely used but now considered insecure Feistel-based block cipher.
- Blowfish: Another popular cipher that uses the Feistel structure.

The Feistel Cipher is a highly flexible and powerful design used in modern block ciphers. Its key feature is that the same process is used for both encryption and decryption, just by reversing the order of the round keys.

DES (Data Encryption Standard)

DES (Data Encryption Standard) is a symmetric-key block cipher that was widely used for encryption. It encrypts data in fixed-size blocks of 64 bits (8 bytes) and uses a 56-bit key (the key is 64 bits long, but 8 bits are used for parity checking). DES operates through 16 rounds of Feistel structure, making it a complex and secure algorithm (though it is now considered insecure due to advances in computing power).

Key Concepts of DES:

1. **Block Size:** DES processes data in 64-bit blocks.
2. **Key Size:** DES uses a 64-bit key, but only 56 bits are used for encryption (the remaining 8 bits are parity bits).
3. **Rounds:** DES operates in 16 rounds of processing using a Feistel structure.
4. **S-boxes:** DES uses substitution boxes (S-boxes) for non-linear substitutions of data.
5. **Initial and Final Permutation:** DES includes an initial permutation (IP) before the 16 rounds and a final permutation (FP) after the rounds.

Python Code for Simplified DES (64-bit)

Here's a simplified Python implementation of DES that performs encryption and decryption of 64-bit blocks using a 56-bit key.

```
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())
```

This code implements encryption and decryption using DES (Data Encryption Standard) in ECB (Electronic Codebook) mode. The DES algorithm is a symmetric-key block cipher that encrypts data in 64-bit (8-byte) blocks and uses a fixed key of 8 bytes (64 bits). In this code, the pycryptodome library is used for cryptographic functions, which provides DES implementations among others.

Simplicity: This code provides a simplified structure of DES. In a real DES implementation, each round would involve a more complex round function with expansion, substitution (via S-boxes), and permutation.

Key Schedule: In real DES, the 56-bit key is expanded into 16 subkeys, one for each round. This simplified version uses the same key for all rounds, but in real DES, the subkeys are generated using bit rotations and permutations.

Security: DES is no longer considered secure due to its small key size (56 bits). It has been replaced by Triple DES and AES for modern cryptographic applications.

Example of the Process

- Input: HelloWorld

Step-by-Step Execution:

1. **Padding:** The original text HelloWorld is 10 characters long. To make it a multiple of 8, 6 spaces will be added at the end: HelloWorld.

2. **Key Generation:** A random 8-byte key is generated (e.g., b'\x92\x01\xef\xa1k\xf4\x91\x8e').

3. **Encryption**

- The plaintext HelloWorld is encrypted using the key in ECB mode.
- The resulting ciphertext is printed in Base64 format (e.g., Ciphertext: b'vQWPIUM85EtS1/KXt2tXTQ==').

4. **Decryption:**
    - The ciphertext is decrypted using the same key.
    - The decrypted text (which includes padding) is printed (e.g., Decrypted text: HelloWorld).

Example Output:

- Enter the plaintext: HelloWorld
- Ciphertext: b'vQWPIUM85EtS1/KXt2tXTQ=='
- Decrypted text: HelloWorld

Security Considerations

- **ECB Mode:** While this code demonstrates how to use DES, ECB mode is not secure for most real-world applications because identical plaintext blocks result in identical ciphertext blocks, allowing an attacker to infer patterns.

- **DES Weakness:** DES is now considered insecure due to its short key length (56-bit keys are vulnerable to brute-force attacks). Modern algorithms like AES (Advanced Encryption Standard) are recommended for secure encryption.

AES (Advanced Encryption Standard)

AES (Advanced Encryption Standard) is a symmetric block cipher used widely across the globe to protect sensitive data. It was adopted as a replacement for DES due to its improved security and efficiency. AES works on fixed 128-bit (16-byte) blocks of data and supports three different key sizes: 128-bit, 192-bit, and 256-bit.

Features of AES:

1. **Block Size:** AES operates on blocks of 128 bits (16 bytes).
2. **Key Sizes:** AES supports three key sizes:
   - 128-bit key (16 bytes)
   - 192-bit key (24 bytes)
   - 256-bit key (32 bytes)
3. **Rounds:** The number of encryption rounds depends on the key size:
   - 10 rounds for 128-bit keys
   - 12 rounds for 192-bit keys
   - 14 rounds for 256-bit keys
4. **Modes of Operation:** AES can be used with different modes, such as ECB (Electronic Codebook), CBC (Cipher Block Chaining), CTR (Counter Mode), etc. CBC is more secure than ECB because it introduces randomness using an initialization vector (IV).

Python Code for AES (in CBC mode)

Here is a Python implementation of AES encryption and decryption using the pycryptodome library. This example uses CBC mode, which requires both a key and an initialization vector (IV) to ensure that identical plaintext blocks do not result in identical ciphertext blocks.

```
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())
```

Example of the Process

Input:

- Plaintext: This is a secret message

Steps:

1. **Key Generation:** A random 128-bit key is generated (e.g., b'\xe2\x12\x95\x92\xfd\xa3\x7f\xd3\x12\x93\x12\xe2\x8f\xb5\xa5\x9e').
2. **Cipher Creation:** A cipher object in EAX mode is created with the key and a random nonce.
3. **Encryption:** The plaintext is encrypted using AES, producing the ciphertext. An authentication tag is generated. AES encrypts the plaintext using a random key and nonce in EAX mode, producing both the ciphertext and an authentication tag.
4. **Decryption:** The ciphertext is decrypted using the same key and nonce. The tag is used to verify the integrity of the data. AES decrypts the ciphertext and verifies the integrity using the same key, nonce, and the tag.

Output:

- Ciphertext: b'q+kRE8Xq/zG6fMKkRZRPEZfBPuyHrSRhIm0Y5SxNGPU='
- Tag: b'\xbf\xb9\xb3\x8b\xe8\xbf~<p\xf1O\xe5\xc3&\xd6>'
- Decrypted text: This is a secret message

Security: AES is considered very secure and is widely used in many encryption standards. EAX mode ensures both confidentiality and data integrity by encrypting and authenticating the message.

RSA (Rivest-Shamir-Adleman) Algorithm

RSA is one of the most widely used asymmetric encryption algorithms. Unlike symmetric encryption algorithms like AES and DES, RSA uses a pair of keys:

- **Public Key:** Used for encryption. This key is shared with everyone.
- **Private Key:** Used for decryption. This key is kept secret by the owner.

RSA relies on the difficulty of factoring large prime numbers to provide security. It works by generating two large prime numbers, and the public and private keys are derived from these numbers.

Key Concepts:

1. **Asymmetric Encryption:** RSA uses two different keys (public and private), unlike symmetric encryption, where the same key is used for both encryption and decryption.
2. **Key Pair:** The public key is used for encryption, and the private key is used for decryption.
3. **Security:** The strength of RSA is based on the difficulty of factoring large composite numbers, making it secure for encrypting sensitive data.

Python Code for RSA Encryption and Decryption

Below is a Python implementation of RSA encryption and decryption using the pycryptodome library. It demonstrates how to generate RSA keys, encrypt a message using the public key, and decrypt it is using the private key.

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Random import get_random_bytes
import base64

# Generate RSA key pair (private and public key)
key = RSA.generate(2048)

# Extract public and private keys
private_key = key.export_key()
public_key = key.publickey().export_key()

print("Private Key: ", private_key.decode('utf-8'))
print("Public Key: ", public_key.decode('utf-8'))
```

```
# Save the keys into files (optional)
with open("private.pem", "wb") as private_file:
    private_file.write(private_key)

with open("public.pem", "wb") as public_file:
    public_file.write(public_key)

# Load the public and private keys (from string for this example)
public_key_obj = RSA.import_key(public_key)
private_key_obj = RSA.import_key(private_key)

# Create RSA cipher object with the public key for encryption
cipher_rsa = PKCS1_OAEP.new(public_key_obj)

# Input plaintext
plaintext = input("Enter the plaintext: ")

# Encrypt the plaintext
ciphertext = cipher_rsa.encrypt(plaintext.encode('utf-8'))
print("Ciphertext (Base64 Encoded):", base64.b64encode(ciphertext).decode('utf-8'))

# Decryption Part
# Create RSA cipher object with the private key for decryption
cipher_rsa_decrypt = PKCS1_OAEP.new(private_key_obj)

# Decrypt the ciphertext
decryptedtext = cipher_rsa_decrypt.decrypt(ciphertext)
print("Decrypted text:", decryptedtext.decode('utf-8'))
```

Explanation:

1. **Key Pair Generation (RSA Key Generation):** The RSA.generate(2048) function generates a 2048-bit RSA key pair. RSA keys can be of varying sizes, but 2048 bits is commonly used because it offers a good balance between security and performance.
2. **Extract Public and Private Keys:**
   - Private Key: The private_key is derived from the generated key pair, and it is stored for decryption.
   - Public Key: The public_key is extracted from the generated key pair and is used for encryption.
3. **Saving the Keys to Files (Optional):** The keys are saved into .pem files, which can be shared and reused later. This step is optional and demonstrates how to store RSA keys.
4. **Creating RSA Cipher Object for Encryption (PKCS1_OAEP):** This is a padding scheme used in RSA encryption to ensure secure encryption. It provides a layer of security against attacks like chosen-plaintext attacks.

5. **Encryption:** The plaintext is first converted into bytes using UTF-8 encoding and then encrypted using the RSA public key. The ciphertext is printed after Base64 encoding for readability.
6. **Decryption:** To decrypt the ciphertext, a new RSA cipher object is created using the private key. The ciphertext is then decrypted, and the original plaintext is recovered.

Example

Input:

- Enter the plaintext: Hello RSA Encryption!

Output:

- **Private Key:**

-----BEGIN RSA PRIVATE KEY-----

MIIEpQIBAAKCAQEAyKf7ASNywDJ3lYVj...

-----END RSA PRIVATE KEY-----

- **Public Key:**

-----BEGIN PUBLIC KEY-----

MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8A...

-----END PUBLIC KEY-----

Ciphertext (Base64 Encoded): fwAbdW7upMVaD2qfbhC...

Decrypted text: Hello RSA Encryption!

Security Considerations

- **Public vs Private Keys:** The public key is freely distributed, while the private key must be kept secret. Only the holder of the private key can decrypt the message.
- **Asymmetric Encryption:** RSA is slower than symmetric encryption (like AES) because of the complex mathematical operations, so it's typically used for encrypting small amounts of data or for exchanging symmetric keys in hybrid encryption systems.
- **Key Size:** Larger RSA keys (e.g., 3072 or 4096 bits) offer more security but are slower in performance. 2048-bit keys are considered secure for most purposes.

RSA is widely used in secure communication protocols like SSL/TLS for establishing encrypted connections over the internet. It is also used for digital signatures, where the private key signs the message and the public key verifies it.

Rabin Cryptosystem

The Rabin Cryptosystem is an asymmetric encryption algorithm, like RSA, but with provable security based on the difficulty of factoring large integers. The key strength of the Rabin cryptosystem is that it is as secure as the problem of integer factorization. However, one of the challenges with Rabin is that decryption yields multiple possible plaintexts, and additional techniques (like redundancy or padding) are needed to determine the correct plaintext.

Key Features:

1. **Asymmetric Encryption:** It uses a pair of keys—public key for encryption and private key for decryption.
2. **Mathematical Foundation:** The security relies on the hardness of factoring a product of two large prime numbers.
3. **Multiple Decryption Outputs:** Decryption results in four possible plaintexts, and extra methods (like redundancy checks) are used to identify the correct one.

Components of the Rabin Cryptosystem:

- **Key Generation:** Two large prime numbers p and q are generated such that $p \equiv 3 \bmod 4$ and $q \equiv 3 \bmod 4$. The public key is $N = p \times q$, and the private key consists of p, q, and N.
- **Encryption:** The plaintext message m is encrypted using the formula:

  $$C = m^2 \bmod N$$

  where N is the public key, and C is the ciphertext.

1. **Decryption:** The ciphertext is decrypted by computing the square roots modulo p and q. The Chinese Remainder Theorem (CRT) is used to combine the results, yielding four possible plaintexts:

   $$m_1, -m_1, m_2, -m_2$$

   These four values are the square roots of the ciphertext modulo N.

Python Implementation of the Rabin Cryptosystem

Below is a Python implementation of the Rabin Cryptosystem. The code generates a public-private key pair, encrypts a message using the public key, and decrypts the message using the private key, yielding four possible plaintexts.

```
import random
import sympy

# Function to generate keys
def generate_keys(bits):
    # Generate two large primes p and q such that p, q ≡ 3 (mod 4)
    p = sympy.randprime(2**(bits-1), 2**bits)
    while p % 4 != 3:
        p = sympy.randprime(2**(bits-1), 2**bits)

    q = sympy.randprime(2**(bits-1), 2**bits)
    while q % 4 != 3:
        q = sympy.randprime(2**(bits-1), 2**bits)

    N = p * q  # Public key
    return (N, p, q)

# Encryption function
def encrypt(public_key, message):
    N = public_key
    ciphertext = pow(message, 2, N)
    return ciphertext

# Extended Euclidean Algorithm
def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    gcd, x1, y1 = extended_gcd(b % a, a)
    x = y1 - (b // a) * x1
    y = x1
    return gcd, x, y

# Modular square root
def mod_sqrt(a, p):
    return pow(a, (p + 1) // 4, p)

# Decryption function
def decrypt(private_key, ciphertext):
    p, q = private_key[1], private_key[2]
    N = p * q

    # Compute square roots mod p and mod q
    r_p = mod_sqrt(ciphertext, p)
    r_q = mod_sqrt(ciphertext, q)

    # Use the extended Euclidean algorithm to find y_p and y_q
    _, y_p, y_q = extended_gcd(p, q)
```

```
    # Combine solutions using the Chinese remainder theorem
    m1 = (r_p * q * y_q + r_q * p * y_p) % N
    m2 = (r_p * q * y_q - r_q * p * y_p) % N

    # The four possible roots are m1, m2, -m1, -m2
    solutions = [(m1 % N), (-m1 % N), (m2 % N), (-m2 % N)]

    return solutions

# Example usage
if __name__ == "__main__":
    bits = 16  # Use small bit size for this example, increase for better security
    public_key, p, q = generate_keys(bits)
    private_key = (public_key, p, q)

    message = 42  # Sample message (must be smaller than N)
    print(f"Original message: {message}")

    ciphertext = encrypt(public_key, message)
    print(f"Encrypted message: {ciphertext}")

    possible_messages = decrypt(private_key, ciphertext)
    print(f"Decrypted possible messages: {possible_messages}")
```

Step-by-step Execution:

1. **Key Generation:**
   - The function generate_keys(bits) generates two large primes p and q, both of which satisfy $p \equiv 3 \bmod 4$ and $q \equiv 3 \bmod 4$. These conditions are important to ensure efficient computation of square roots during decryption.
   - The public key is $N = p \times q$, and the private key consists of (p,q).
2. **Encryption:**
   - The function encrypt(public_key, message) encrypts the plaintext message m using the formula:

     $$C = m^2 \bmod N$$

     Here, N is the public key, and m is the plaintext message. The result is the ciphertext C.
3. **Decryption:**
   - The decryption function decrypt(private_key, ciphertext) works by finding the square roots of the ciphertext modulo p and modulo q.

- Chinese Remainder Theorem (CRT) is used to combine the solutions obtained from mod p and mod q. The extended Euclidean algorithm helps calculate the necessary coefficients for combining the square roots.
- Four possible plaintexts are produced: m_1, $-m_1$, m_2, $-m_2$. These are the four possible square roots of the ciphertext.

4. **Extended Euclidean Algorithm:**
   - The extended_gcd(a, b) function computes the greatest common divisor (GCD) of a and b and finds the coefficients needed for the Chinese Remainder Theorem. This is essential for combining the results from the mod p and mod q computations.

Example Execution

Input:

- message = 42

Output:

- Original message: 42
- Encrypted message: 1764
- Decrypted possible messages: [42, 3191, 61, 3172]

This Python implementation of the Rabin Cryptosystem demonstrates the encryption and decryption processes, showing how to generate keys, encrypt a message, and decrypt the resulting ciphertext into four possible solutions. This implementation highlights the key aspects of the Rabin cryptosystem, including the use of the Chinese Remainder Theorem and modular arithmetic.

ElGamal Cryptosystem

The ElGamal Cryptosystem is an asymmetric key encryption algorithm based on the Diffie-Hellman key exchange. It was developed by Taher Elgamal in 1985. ElGamal provides both encryption and digital signature schemes and is widely used due to its simplicity and security.

Key Features:

1. **Asymmetric Encryption:** ElGamal uses a pair of keys—public key for encryption and private key for decryption.
2. **Mathematical Foundation:** The security of ElGamal relies on the difficulty of computing discrete logarithms in a large prime modulus.
3. **Key Sizes:** ElGamal encryption uses large prime numbers for secure communication, typically 1024-bit or 2048-bit primes.
4. **Randomness in Encryption:** Each encryption operation uses a randomly chosen integer, making the ciphertext different for the same plaintext each time.

Components of the ElGamal Cryptosystem:

- **Public Key:** The public key consists of a large prime p, a generator g, and $h=g^x \bmod p$, where x is the private key.
- **Private Key:** The private key is x, which is a randomly chosen integer from [1, p−2].
- **Encryption:** The encryption uses the public key and a random integer k. The ciphertext consists of two parts (c_1, c_2), where:

$$c_1 = g^k \bmod p$$

$$c_2 = m \cdot h^K \bmod p$$

  Here, m is the plaintext.

- **Decryption:** To decrypt the ciphertext (c1, c2), the recipient uses their private key x to recover the original plaintext m:

$$m = c_2 \cdot (c_1^x)^{-1} \bmod p$$

Python Code for ElGamal

Below is the Python implementation of the ElGamal Cryptosystem. The code demonstrates how to generate keys, encrypt a message, and decrypt the ciphertext.

```
import random
import sympy
from sympy import mod_inverse

# Function to generate ElGamal keys
def generate_keys(bits):
    # Generate a large prime p
    p = sympy.randprime(2**(bits-1), 2**bits)
    g = random.randint(2, p-2)  # Generator g, such that 1 < g < p-1

    # Private key x (randomly chosen from [1, p-2])
    x = random.randint(1, p-2)

    # Public key h = g^x mod p
    h = pow(g, x, p)

    # Public key (p, g, h) and private key x
    return (p, g, h), x

# Function to encrypt a message using the ElGamal cryptosystem
def encrypt(public_key, message):
    p, g, h = public_key

    # Random value k (chosen for each encryption, from [1, p-2])
    k = random.randint(1, p-2)

    # Calculate the two parts of the ciphertext
    c1 = pow(g, k, p)  # c1 = g^k mod p
    c2 = (message * pow(h, k, p)) % p  # c2 = m * h^k mod p

    # Ciphertext is (c1, c2)
    return c1, c2

# Function to decrypt the ciphertext using the ElGamal cryptosystem
def decrypt(private_key, public_key, ciphertext):
    p, g, h = public_key
    c1, c2 = ciphertext
    x = private_key

    # Compute the shared secret: s = c1^x mod p
    s = pow(c1, x, p)

    # Compute the inverse of s modulo p
    s_inv = mod_inverse(s, p)

    # Recover the original message: m = c2 * s_inv mod p
    message = (c2 * s_inv) % p
```

```
    return message

# Example usage
if __name__ == "__main__":
    bits = 16  # Use small bit size for this example, increase for better security
    public_key, private_key = generate_keys(bits)

    print(f"Public Key (p, g, h): {public_key}")
    print(f"Private Key (x): {private_key}")

    # Sample message (must be smaller than p)
    message = 42
    print(f"Original message: {message}")

    # Encrypt the message
    ciphertext = encrypt(public_key, message)
    print(f"Ciphertext: {ciphertext}")

    # Decrypt the ciphertext
    decrypted_message = decrypt(private_key, public_key, ciphertext)
    print(f"Decrypted message: {decrypted_message}")
```

Example Execution

Input:

- message = 42

Output:

- Public Key (p, g, h): (50827, 2161, 15610)
- Private Key (x): 22891
- Original message: 42
- Ciphertext: (28977, 19263)
- Decrypted message: 42

Explanation of Steps:

1. **Key Generation:**
   - A large prime number p, a generator g, and the public key h are generated.
   - The private key x is kept secret, and the public key is distributed.
2. **Encryption:**
   - The message m=42 is encrypted using a randomly chosen integer k.
   - The ciphertext (c_1, c_2) is produced, with $c_1 = g^k \bmod p$ and $c_2 = m \cdot h^k \bmod p$.

3. **Decryption:**
    - The ciphertext is decrypted using the private key x, and the original message is recovered.

Security Considerations:

1. **Discrete Logarithm Problem:** The security of ElGamal relies on the difficulty of solving the discrete logarithm problem, i.e., finding x given $g^x \bmod p$.
2. **Randomness in Encryption:** Each encryption uses a fresh random value k, ensuring that even if the same message is encrypted multiple times, the resulting ciphertexts will be different.

This Python implementation of the ElGamal cryptosystem demonstrates the key generation, encryption, and decryption processes. The ElGamal cryptosystem provides a secure method of encryption by leveraging the difficulty of the discrete logarithm problem, making it a popular choice for securing communication.

DSS (Digital Signature Standard)

The Digital Signature Standard (DSS) is a suite of algorithms used for generating and verifying digital signatures. It was proposed by the National Institute of Standards and Technology (NIST) in 1991 and is widely used for secure digital communications. DSS defines the Digital Signature Algorithm (DSA) as the standard for digital signatures. DSA is based on the mathematical problem of discrete logarithms and is commonly used in a variety of security applications, including digital certificates and secure communication protocols.

Key Features of DSS:

1. **Digital Signature Algorithm (DSA):** DSS defines DSA as the standard for digital signatures.
2. **Asymmetric:** DSA uses a pair of keys—a private key for signing and a public key for verifying signatures.
3. **Security:** The security of DSA relies on the difficulty of solving discrete logarithm problems in a finite field.
4. **Hash Function:** DSA uses a hash function (such as SHA-256) to create a digest of the message, which is signed rather than signing the entire message.

Steps in DSS (Digital Signature Algorithm):

1. **Key Generation:**
   - Generate a public and private key pair.
   - The private key is used to sign a message, and the public key is used to verify the signature.
2. **Signature Generation:**
   - The signer generates a hash of the message and signs this hash with their private key using DSA.
3. **Signature Verification:**
   - The verifier uses the public key to verify the signature by checking that it matches the hash of the message.

Python Code Implementation of DSS (DSA):

Below is a Python implementation of the Digital Signature Algorithm (DSA). The code demonstrates key generation, message signing, and signature verification using the pycryptodome library.

```
from Crypto.Signature import DSS
from Crypto.Hash import SHA256
from Crypto.PublicKey import DSA
from Crypto.Random import get_random_bytes

# Function to generate DSA keys
def generate_keys():
    # Generate a DSA private key
    private_key = DSA.generate(2048)

    # Extract the public key
    public_key = private_key.publickey()

    return private_key, public_key

# Function to sign a message using DSA
def sign_message(private_key, message):
    # Hash the message using SHA-256
    hash_obj = SHA256.new(message.encode('utf-8'))

    # Create the signature object using the private key
    signer = DSS.new(private_key, 'fips-186-3')

    # Sign the message
    signature = signer.sign(hash_obj)

    return signature

# Function to verify a signature using DSA
def verify_signature(public_key, message, signature):
    # Hash the message using SHA-256
    hash_obj = SHA256.new(message.encode('utf-8'))

    # Create the verifier object using the public key
    verifier = DSS.new(public_key, 'fips-186-3')

    try:
        # Verify the signature
        verifier.verify(hash_obj, signature)
        print("The signature is valid.")
    except ValueError:
        print("The signature is not valid.")

# Example usage
if __name__ == "__main__":
    # Generate DSA keys
    private_key, public_key = generate_keys()
```

```
print("Private Key:")
print(private_key.export_key().decode('utf-8'))

print("Public Key:")
print(public_key.export_key().decode('utf-8'))

# Sample message
message = "Hello, this is a message to sign with DSA."
print(f"\nOriginal message: {message}")

# Sign the message
signature = sign_message(private_key, message)
print(f"\nSignature (in bytes): {signature}")

# Verify the signature
verify_signature(public_key, message, signature)
```

Example Output:

Private Key:

-----BEGIN PRIVATE KEY-----

MIIBSgIBADCCASwGByqGSM44BAEwggEfAoGBAKCQjGuN...

-----END PRIVATE KEY-----

Public Key:

-----BEGIN PUBLIC KEY-----

MIIBtzCCASsGByqGSM44BAEwggEeAoGBAKCQjGuN...

-----END PUBLIC KEY-----

Original message: Hello, this is a message to sign with DSA.

Signature (in bytes): b'\xb4\xf0\x9d\x8c...'

The signature is valid.

Execution Step:

- **Key Generation:** A 2048-bit DSA key pair (private and public keys) is generated. The private key is used for signing, and the public key is distributed for verification.
- **Message Signing:** The message is hashed using SHA-256, and the resulting hash is signed using the private key, producing the digital signature.
- **Signature Verification:** The verifier uses the public key to check the validity of the signature by comparing the hash of the original message with the signature. If they match, the signature is valid.

Security Considerations:

- **Private Key Security:** The private key must be always kept secret, as anyone with access to it can sign messages on behalf of the legitimate user.
- **Public Key Distribution:** The public key should be distributed securely to ensure that verifiers are using the correct key.
- **Cryptographic Hash Functions:** The choice of a strong hash function (such as SHA-256) is critical for the security of the digital signature. Weak hash functions can lead to vulnerabilities.

This Python implementation demonstrates the key processes in the Digital Signature Standard (DSS) using the Digital Signature Algorithm (DSA). The code shows how to generate DSA keys, sign a message, and verify the authenticity of a digital signature. DSA provides a secure way of ensuring that messages have not been tampered with and that the sender's identity is authentic.

Diffie-Hellman Key Exchange

The Diffie-Hellman Key Exchange (DHKE) is a cryptographic protocol that allows two parties to securely agree on a shared secret over a public, insecure channel. This shared secret can then be used to encrypt subsequent communications between the two parties. The security of Diffie-Hellman relies on the difficulty of computing discrete logarithms.

Key Features:

1. **Asymmetric Key Exchange:** Two parties, without previously sharing any secret, can agree on a shared secret key.
2. **Public Parameters:** The protocol uses two public values—a large prime number p and a generator g (also known as the base).
3. **Security:** The security of Diffie-Hellman is based on the difficulty of solving the discrete logarithm problem, i.e., finding a from $g^a \bmod p$.

Steps in the Diffie-Hellman Key Exchange:

1. **Public Parameters:** Both parties agree on a large prime number p and a base g.
2. **Private Keys:** Each party generates a private key, which is a large random number.
3. **Public Keys:** Each party computes a public key using the formula:

   $$A = g^a \bmod p \text{ and } B = g^b \bmod p$$

   where a and b are the private keys of the two parties.

4. **Shared Secret:** Each party computes the shared secret using the other party's public key and their own private key:

   $$\text{Shared Secret} = B^a \bmod p = A^b \bmod p$$

   Both parties end up with the same shared secret, which can be used as a key for symmetric encryption.

Python Implementation of Diffie-Hellman Key Exchange

Here's a Python implementation of the Diffie-Hellman Key Exchange. It demonstrates how two parties can securely agree on a shared secret key.

```
import random

# Function to generate a large prime p and generator g
def generate_parameters(bits=256):
```

```
    # Generate a large prime number p (a small prime for example purposes)
    p = 23  # In practice, p should be a large prime number
    g = 5   # In practice, g should be a primitive root modulo p
    return p, g

# Function to generate a private key (a random integer in the range [1, p-2])
def generate_private_key(p):
    return random.randint(2, p-2)

# Function to generate a public key based on the private key
def generate_public_key(g, private_key, p):
    return pow(g, private_key, p)

# Function to compute the shared secret
def compute_shared_secret(public_key_other, private_key_self, p):
    return pow(public_key_other, private_key_self, p)

# Example usage of the Diffie-Hellman Key Exchange
if __name__ == "__main__":
    # Step 1: Generate public parameters (p and g)
    p, g = generate_parameters()
    print(f"Public Parameters: p = {p}, g = {g}")

    # Step 2: Each party generates a private key
    private_key_A = generate_private_key(p)
    private_key_B = generate_private_key(p)

    print(f"Private Key of Party A: {private_key_A}")
    print(f"Private Key of Party B: {private_key_B}")

    # Step 3: Each party generates a public key
    public_key_A = generate_public_key(g, private_key_A, p)
    public_key_B = generate_public_key(g, private_key_B, p)

    print(f"Public Key of Party A: {public_key_A}")
    print(f"Public Key of Party B: {public_key_B}")

    # Step 4: Each party computes the shared secret
    shared_secret_A = compute_shared_secret(public_key_B, private_key_A, p)
    shared_secret_B = compute_shared_secret(public_key_A, private_key_B, p)

    print(f"Shared Secret computed by Party A: {shared_secret_A}")
    print(f"Shared Secret computed by Party B: {shared_secret_B}")

    # Check if both parties have the same shared secret
    if shared_secret_A == shared_secret_B:
        print("The shared secret is the same for both parties.")
```

```
else:
    print("Error: The shared secrets do not match.")
```

Example Output:

- Public Parameters: p = 23, g = 5
- Private Key of Party A: 6
- Private Key of Party B: 15
- Public Key of Party A: 8
- Public Key of Party B: 19
- Shared Secret computed by Party A: 2
- Shared Secret computed by Party B: 2
- The shared secret is the same for both parties.

Execution Step:

1. **Parameter Generation:** Both parties agree on a large prime number p and a generator g. These are public and do not need to be kept secret.
2. **Private Key Generation:** Each party generates a private key randomly. This private key is kept secret.
3. **Public Key Generation:** Each party generates a public key using their private key, the generator g, and the prime number p.
4. **Shared Secret Calculation:** Using the other party's public key and their own private key, each party computes the shared secret. This shared secret will be the same for both parties and can be used for symmetric encryption of further communications.

Security Considerations:

- **Discrete Logarithm Problem:** The security of Diffie-Hellman relies on the difficulty of solving the discrete logarithm problem, which is considered computationally hard for large primes.
- **Man-in-the-Middle Attack:** Diffie-Hellman alone does not provide authentication, so it is vulnerable to a man-in-the-middle attack if an attacker intercepts and alters the public keys. To prevent this, Diffie-Hellman is often combined with authentication methods such as digital signatures or certificates.

This Python implementation of the Diffie-Hellman Key Exchange demonstrates how two parties can agree on a shared secret over an insecure channel. The shared secret can then be used as a key for symmetric encryption in secure communication protocols such as TLS (Transport Layer Security). This protocol forms the basis of secure key exchange in many modern cryptographic systems.