MADHAV INSTITUTE OF TECHNOLOGY & SCIENCE, GWALIOR

(A Govt. Aided UGC Autonomous Institute Affiliated to RGPV, Bhopal)

NAAC Accredited with A++ Grade

Computer Science and Engineering

Information Security
150513
Lab File

Submitted to -

Prof. Amit Kumar Manjhvar
Prof. Smita Parte
Prof. Manisha Pathak

Submitted by -

Suyash Agarwal
0901CS211123
CSE 3rd Year

INDEX

S. No.	Experiments	Date	Signature
1	Perform encryption, decryption using the following substitution techniques I. Ceaser cipher II. Hill Cipher		
2	Perform encryption and decryption using following transposition techniques Rail fence - Row & Column Transformation		
3	Implement Playfair Cipher with key entered by user.		
4	Implement polyalphabetic Cipher		
5	Implement AutoKey Cipher		
6	Implement Hill Cipher..		
7	Implement Rail fence technique		
8	Implement Transposition technique		
9	Implement substitution technique		
10	Demonstrate intrusion detection system (ids) using any tool (snort or any other s/w)		

Experiment – 01

Aim – To Perform encryption, decryption using the following substitution techniques I. Caeser cipher II. Hill Cipher.

Implementation

(1) Caeser Cipher

The Caesar cipher is a straightforward substitution cipher that moves the alphabetic characters a predetermined number of places. Here's how to use the Caesar cipher for encryption and decryption.

Encryption:

- The Caesar cipher requires shifting each letter in the plaintext a predetermined number of places (key) down the alphabet in order to encrypt a message. If your key is 3, for instance, "A" becomes "D," "B" becomes "E," and so on.
- Say you wish to use the Caesar cipher with a key value of 3 to encrypt the message "HELLO":
- H -> K
- E -> H
- L -> O
- L -> O
- O -> R

So, "HELLO" would be encrypted as "KHOOR."

Decryption:

- The same number of locations as the encryption key must be moved backward for each letter in the ciphertext in order to decrypt a message that has been encrypted using the Caesar cipher.
- For instance, you would move each letter back three spaces if you had the ciphertext "KHOOR" and knew the key was three.
- K -> H
- H -> E
- O -> L
- O -> L
- R -> O

So, "KHOOR" would be decrypted as "HELLO."

Implementation code for Caeser cipher.

```
def caesar_encrypt(message, key):
    encrypted_message = ""

    for char in message:
```

```
        if char.isalpha():
            base = ord('a') if char.islower() else ord('A')
            encrypted_message += chr((ord(char) - base + key) % 26 + base)
        else:
            encrypted_message += char

    return encrypted_message

def caesar_decrypt(message, key):
    return caesar_encrypt(message, 26 - key)  # Decrypt by shifting in the opposite
direction

plaintext = "HELLO"
key = 3

encrypted = caesar_encrypt(plaintext, key)
print("Encrypted:", encrypted)

decrypted = caesar_decrypt(encrypted, key)
print("Decrypted:", decrypted)
```

(2) Hill Cipher

The Hill cipher is a more complicated encryption method that encrypts and decrypts communications using matrix multiplication. The Hill cipher may be used for both encryption and decryption, as seen below:

Encryption

- Select a key matrix that is invertible (has an inverse matrix) and often 2x2 or 3x3.
- Transform your plaintext into numbers (A = 0, B = 1,..., Z = 25).
- Convert each plaintext chunk into a column vector by dividing it into pieces of the same size as your key matrix.
- Divide each column vector of the plaintext by 26 to multiply the key matrix.
- Rewrite the obtained numbers (0=A, 1=B,..., 25=Z) into letters.

Decryption

- Select the encryption key matrix that was previously used.
- The key matrix's inverse should be calculated.
- Transform the ciphertext's characters into numbers.
- Create a column vector for each chunk of the ciphertext that is equal in size to the size of your key matrix.
- Divide each column vector of the ciphertext by 26 to multiply the opposite key matrix.
- Then change the numbers back to letters.

Implementation code for hill cipher.

```
import numpy as np


def hill_encrypt(message, key_matrix):
    message = message.replace(" ", "").upper()
    key_size = key_matrix.shape[0]
    if len(message) % key_size != 0:
        raise ValueError("Message length must be a multiple of the key size.")
    encrypted_message = ""
    for i in range(0, len(message), key_size):
        message_chunk = message[i:i + key_size]
        message_vector = np.array([ord(char) - ord('A') for char in message_chunk])
        result_vector = np.dot(key_matrix, message_vector) % 26
        encrypted_message += ''.join([chr(val + ord('A')) for val in result_vector])
    return encrypted_message
def hill_decrypt(encrypted_message, key_matrix):
    key_matrix_inv = np.linalg.inv(key_matrix)
    key_matrix_inv = key_matrix_inv.astype(int)
    key_matrix_inv = key_matrix_inv % 26
    return hill_encrypt(encrypted_message, key_matrix_inv)
# Example usage
key = "GYBNQKURP"
message = "HELLOHILL"
key_matrix = np.array([[ord(c) - ord('A') for c in key]])
message_matrix = np.array([[ord(c) - ord('A') for c in message]])
encrypted = hill_encrypt(message, key_matrix)
print("Encrypted:", encrypted)


decrypted = hill_decrypt(encrypted, key_matrix)
print("Decrypted:", decrypted)
```

Experiment – 02

Aim - Perform encryption and decryption using following transposition techniques Rail fence - Row & Column Transformation.

Implementation

(1) Rail Fence Transposition - Row-Wise Encryption:

In the row-wise Rail Fence transposition technique, you write the message diagonally along a set number of "rails," and then read it off row by row.

Implemented code

```
def rail_fence_encrypt_rowwise(message, rails):
    encrypted = ['' for _ in range(rails)]
    row, direction = 0, 1
    for char in message:
        encrypted[row] += char
        if row == 0:
            direction = 1
        elif row == rails - 1:
            direction = -1
        row += direction
    return ''.join(encrypted)
message = "HELLOWORLD"
rails = 3
encrypted = rail_fence_encrypt_rowwise(message, rails)
print("Row-Wise Encrypted:", encrypted)
```

(2) Rail Fence Transposition - Row-Wise Decryption:

Decryption for the row-wise Rail Fence technique is similar to encryption. You create an empty rail matrix and fill it with the ciphertext in a zigzag pattern. Then, you read the rails row by row to recover the original message.

Implemented code

```
def rail_fence_decrypt_rowwise(ciphertext, rails):
    rail_lengths = [0] * rails
    row, direction = 0, 1
    for char in ciphertext:
        rail_lengths[row] += 1
        if row == 0:
            direction = 1
        elif row == rails - 1:
            direction = -1
        row += direction
    rail_positions = [0] * rails
    message = [' '] * len(ciphertext)
    for i, char in enumerate(ciphertext):
        rail = rail_positions.index(min(rail_positions))
        message[i] = char
        rail_positions[rail] += 1
    return ''.join(message)
decrypted = rail_fence_decrypt_rowwise(encrypted, rails)
print("Row-Wise Decrypted:", decrypted)
```

(3) Rail Fence Transposition - Column-Wise Encryption

In the column-wise Rail Fence transposition technique, you write the message into a matrix column by column, and then read it off row by row.

Implemented code

```
def rail_fence_encrypt_columnwise(message, rails):

    message_len = len(message)

    cols = (message_len + rails - 1) // rails

    rail_matrix = [[' ' for _ in range(cols)] for _ in range(rails)]

    col, direction = 0, 1

    for char in message:

        rail_matrix[col][col // rails] = char

        if col == 0:

            direction = 1

        elif col == rails - 1:

            direction = -1
```

```
        col += direction
    encrypted = ''
    for row in rail_matrix:
        encrypted += ''.join(row)
    return encrypted
encrypted_columnwise = rail_fence_encrypt_columnwise(message, rails)
print("Column-Wise Encrypted:", encrypted_columnwise)
```

(4) Rail Fence Transposition - Column-Wise Decryption:

Decryption for the column-wise Rail Fence technique is similar to encryption. You create an empty rail matrix, fill it with the ciphertext column by column, and then read the matrix row by row to recover the original message.

Implemented code

```
def rail_fence_decrypt_columnwise(ciphertext, rails):
    message_len = len(ciphertext)
    cols = (message_len + rails - 1) // rails
    rail_matrix = [[' ' for _ in range(cols)] for _ in range(rails)]
    col, direction = 0, 1
    for i, char in enumerate(ciphertext):
        rail_matrix[col][col // rails] = char
        if col == 0:
            direction = 1
        elif col == rails - 1:
            direction = -1
        col += direction
    decrypted = ''
    for i in range(cols):
        for j in range(rails):
            if rail_matrix[j][i] != ' ':
                decrypted += rail_matrix[j][i]
    return decrypted
decrypted_columnwise = rail_fence_decrypt_columnwise(encrypted_columnwise,
rails)
print("Column-Wise Decrypted:", decrypted_columnwise)
```

Experiment – 03

Aim - Implement Playfair Cipher with key entered by user.

Implementation

The Playfair Cipher is a historical symmetric encryption technique that operates on pairs of letters from the plaintext, substituting them with other letters based on a key matrix. It employs rules such as row/column swaps and filler letter insertion for encryption and decryption, and was used for secure communication in the past, though it's considered relatively insecure by modern cryptographic standards.

(1) Python code

```
def generate_playfair_matrix(key):
    key = key.replace(" ", "").upper()
    matrix = [['' for _ in range(5)] for _ in range(5)]
    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"
    key_index = 0
    # Fill the matrix with the key
    for i in range(5):
        for j in range(5):
            if key_index < len(key):
                matrix[i][j] = key[key_index]
                key_index += 1
            else:
                for letter in alphabet:
                    if letter not in key and letter not in ''.join(matrix[i]) + ''.join([matrix[k][j] for k in range(5)]):
                        matrix[i][j] = letter
                        break
    return matrix
def preprocess_text(text):
    # Remove spaces and convert to uppercase
```

```
    text = text.replace(" ", "").upper()
    # Replace 'J' with 'I'
    text = text.replace("J", "I")
    # Add a placeholder between repeated letters
    i = 0
    while i < len(text) - 1:
        if text[i] == text[i + 1]:
            text = text[:i + 1] + 'X' + text[i + 1:]
        i += 2
    # If the length is odd, add a 'Z' at the end
    if len(text) % 2 != 0:
        text += 'Z'
    return text
def find_positions(matrix, char):
    for i in range(5):
        for j in range(5):
            if matrix[i][j] == char:
                return i, j
def playfair_encrypt(plain_text, key):
    matrix = generate_playfair_matrix(key)
    plain_text = preprocess_text(plain_text)
    cipher_text = ""
    for i in range(0, len(plain_text), 2):
        char1, char2 = plain_text[i], plain_text[i + 1]
        row1, col1 = find_positions(matrix, char1)
        row2, col2 = find_positions(matrix, char2)
        if row1 == row2:
            cipher_text += matrix[row1][(col1 + 1) % 5] + matrix[row2][(col2 + 1) % 5]
        elif col1 == col2:
            cipher_text += matrix[(row1 + 1) % 5][col1] + matrix[(row2 + 1) % 5][col2]
```

```
        else:
            cipher_text += matrix[row1][col2] + matrix[row2][col1]
    return cipher_text
def main():
    key = input("Enter the Playfair key (no spaces, uppercase letters): ")
    plain_text = input("Enter the plaintext to encrypt: ")
    cipher_text = playfair_encrypt(plain_text, key)
    print("Cipher Text:", cipher_text)
if __name__ == "__main__":
    main()
```

Experiment – 04

Aim - Implement polyalphabetic Cipher.

Implementation

Polyalphabetic Cipher is a cryptographic technique that uses multiple substitution alphabets to encrypt plaintext, where each letter in the plaintext can be substituted with different letters or symbols depending on its position or the key, making it more complex and secure compared to simple monoalphabetic ciphers.

(1) Python code

```
def vigenere_encrypt(plain_text, key):
    plain_text = plain_text.upper()
    key = key.upper()
    encrypted_text = ""
    key_length = len(key)
    for i in range(len(plain_text)):
        if plain_text[i].isalpha():
            # Calculate the shift value for this letter
            shift = ord(key[i % key_length]) - ord('A')
            # Encrypt the letter
            encrypted_letter = chr(((ord(plain_text[i]) - ord('A') + shift) % 26) + ord('A'))
            encrypted_text += encrypted_letter
        else:
            # Non-alphabetic characters are not encrypted
            encrypted_text += plain_text[i]
    return encrypted_text
def vigenere_decrypt(cipher_text, key):
    cipher_text = cipher_text.upper()
    key = key.upper()
    decrypted_text = ""
```

```
    key_length = len(key)
    for i in range(len(cipher_text)):
        if cipher_text[i].isalpha():
            # Calculate the shift value for this letter
            shift = ord(key[i % key_length]) - ord('A')
            # Decrypt the letter
            decrypted_letter = chr(((ord(cipher_text[i]) - ord('A') - shift) % 26) + ord('A'))
            decrypted_text += decrypted_letter
        else:
            decrypted_text += cipher_text[i]
    return decrypted_text
def main():
    choice = input("Enter 'E' to encrypt or 'D' to decrypt: ").upper()
    if choice != 'E' and choice != 'D':
        print("Invalid choice. Please enter 'E' to encrypt or 'D' to decrypt.")
        return
    key = input("Enter the keyword (letters only): ")
    if not key.isalpha():
        print("Invalid keyword. Please use letters only.")
        return
    if choice == 'E':
        plain_text = input("Enter the plaintext: ")
        encrypted_text = vigenere_encrypt(plain_text, key)
        print("Encrypted Text:", encrypted_text)
    else:
        cipher_text = input("Enter the cipher text: ")
        decrypted_text = vigenere_decrypt(cipher_text, key)
        print("Decrypted Text:", decrypted_text)
if __name__ == "__main__":
        main()
```

Experiment – 05

Aim - Implement AutoKey Cipher.

Implementation

AutoKey Cipher is a cryptographic method derived from the Vigenère Cipher, in which the key is based on the plaintext itself, extending the key with successive letters from the plaintext to encrypt or decrypt the message.

(1) Python Code

```
def autokey_encrypt(plain_text, key):
    plain_text = plain_text.upper()
    key = key.upper()
    encrypted_text = ""
    key_stream = key + plain_text
    for i in range(len(plain_text)):
        if plain_text[i].isalpha():
            shift = ord(key_stream[i]) - ord('A')
            encrypted_letter = chr(((ord(plain_text[i]) - ord('A') + shift) % 26) + ord('A'))
            encrypted_text += encrypted_letter
        else:
            encrypted_text += plain_text[i]
    return encrypted_text
def autokey_decrypt(cipher_text, key):
    cipher_text = cipher_text.upper()
    key = key.upper()
    decrypted_text = ""
    key_stream = key
    for i in range(len(cipher_text)):
        if cipher_text[i].isalpha():
            # Calculate the shift value for this letter
```

```
            shift = ord(key_stream[i]) - ord('A')
            # Decrypt the letter
            decrypted_letter = chr(((ord(cipher_text[i]) - ord('A') - shift + 26) % 26) + ord('A'))
            decrypted_text += decrypted_letter
            # Update the key stream with the decrypted letter
            key_stream += decrypted_letter
        else:
            # Non-alphabetic characters are not decrypted
            decrypted_text += cipher_text[i]
    return decrypted_text
def main():
    choice = input("Enter 'E' to encrypt or 'D' to decrypt: ").upper()
    if choice != 'E' and choice != 'D':
        print("Invalid choice. Please enter 'E' to encrypt or 'D' to decrypt.")
        return
    key = input("Enter the keyword (letters only): ")
    if not key.isalpha():
        print("Invalid keyword. Please use letters only.")
        return
    if choice == 'E':
        plain_text = input("Enter the plaintext: ")
        encrypted_text = autokey_encrypt(plain_text, key)
        print("Encrypted Text:", encrypted_text)
    else:
        cipher_text = input("Enter the cipher text: ")
        decrypted_text = autokey_decrypt(cipher_text, key)
        print("Decrypted Text:", decrypted_text)
if __name__ == "__main__":
    main()
```

Experiment – 06

Aim - Implement Hill Cipher.

Implementation

Hill Cipher is a polygraphic substitution cipher that employs matrix multiplication to encrypt and decrypt messages, operating on blocks of letters rather than individual letters, enhancing security and complexity compared to monoalphabetic ciphers.

(1) Python code

```
import numpy as np

def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('Modular inverse does not exist')
    else:
        return x % m

def matrix_to_text(matrix, n):
    result = ""
    for i in range(n):
        for j in range(n):
            result += chr(matrix[i][j] % 26 + 65)
    return result

def text_to_matrix(text, n):
    matrix = []
```

```
    for i in range(0, len(text), n):
        row = []
        for j in range(n):
            if i + j < len(text):
                row.append(ord(text[i + j]) - 65)
            else:
                row.append(0)
        matrix.append(row)
    return matrix
def hill_encrypt(plain_text, key):
    n = len(key)
    if len(plain_text) % n != 0:
        raise Exception('Plain text length must be a multiple of the key matrix size')
    key_matrix = text_to_matrix(key, n)
    plain_matrix = text_to_matrix(plain_text, n)
    encrypted_matrix = np.dot(plain_matrix, key_matrix) % 26
    return matrix_to_text(encrypted_matrix, n)
def hill_decrypt(cipher_text, key):
    n = len(key)
    if len(cipher_text) % n != 0:
        raise Exception('Cipher text length must be a multiple of the key matrix size')
    key_matrix = text_to_matrix(key, n)
    key_matrix_inverse = np.array([[modinv(key_matrix[i][j], 26) for j in range(n)] for i in
range(n)])
    cipher_matrix = text_to_matrix(cipher_text, n)
    decrypted_matrix = np.dot(cipher_matrix, key_matrix_inverse) % 26
    return matrix_to_text(decrypted_matrix, n)
def main():
    key = input("Enter the key (letters only, e.g., 'GYBNQKURP'): ").upper()
    plain_text = input("Enter the plaintext: ").upper()
    encrypted_text = hill_encrypt(plain_text, key)
```

```
    decrypted_text = hill_decrypt(encrypted_text, key)
    print("Encrypted Text:", encrypted_text)
    print("Decrypted Text:", decrypted_text)
if __name__ == "__main__":
    main()
```

Experiment – 07

Aim - Implement Rail fence technique

Implementation

Rail Fence Cipher is a transposition cipher that rearranges characters by writing them in a zigzag pattern along a set number of "rails" or lines and then reading off the characters row by row, offering a basic method of obfuscating text.

(1) Python code

```
def rail_fence_encrypt(plain_text, num_rails):
    rail_fence = [[' ' for _ in range(len(plain_text))] for _ in range(num_rails)]
    rail = 0
    direction = 1  # Direction 1 means moving down, -1 means moving up
    for char in plain_text:
        rail_fence[rail][0] = char
        rail += direction
        if rail == num_rails or rail == -1:
            direction *= -1  # Change direction when reaching the top or bottom
            rail += 2 * direction  # Move to the next rail
    encrypted_text = ''.join(''.join(row) for row in rail_fence)
    return encrypted_text
def rail_fence_decrypt(cipher_text, num_rails):
    rail_fence = [[' ' for _ in range(len(cipher_text))] for _ in range(num_rails)]
    pattern = list(range(num_rails)) + list(range(num_rails - 2, 0, -1))
    pattern_length = len(pattern)
    i = 0
    for rail in range(num_rails):
        j = 0
        while rail < num_rails and j < len(cipher_text):
```

```
            rail_fence[rail][j] = cipher_text[i]
            i += 1
            j += 1
            rail += 1
        i -= 1
        rail -= 2
        while rail >= 0 and j < len(cipher_text):
            rail_fence[rail][j] = cipher_text[i]
            i += 1
            j += 1
            rail -= 2
    decrypted_text = ''.join(''.join(row) for row in rail_fence)
    return decrypted_text
def main():
    choice = input("Enter 'E' to encrypt or 'D' to decrypt: ").upper()
    if choice != 'E' and choice != 'D':
        print("Invalid choice. Please enter 'E' to encrypt or 'D' to decrypt.")
        return
    num_rails = int(input("Enter the number of rails: "))
    if choice == 'E':
        plain_text = input("Enter the plaintext: ")
        encrypted_text = rail_fence_encrypt(plain_text, num_rails)
        print("Encrypted Text:", encrypted_text)
    else:
        cipher_text = input("Enter the cipher text: ")
        decrypted_text = rail_fence_decrypt(cipher_text, num_rails)
        print("Decrypted Text:", decrypted_text)
if __name__ == "__main__":
    main()
```

Experiment – 08

Aim - Implement Transposition technique.

Implementation

Transposition techniques are a class of cryptographic algorithms that involve the rearrangement of characters or blocks of plaintext to create ciphertext. One of the common transposition techniques is the Columnar Transposition Cipher.

(1) Python code

```
def encrypt_columnar_transposition(plain_text, key):
    plain_text = plain_text.replace(" ", "").upper()
    key = key.upper()
    num_columns = len(key)
    num_rows = -(-len(plain_text) // num_columns)  # Ceiling division
    grid = [['' for _ in range(num_columns)] for _ in range(num_rows)]
    index = 0
    for col in range(num_columns):
        for row in range(num_rows):
            if index < len(plain_text):
                grid[row][col] = plain_text[index]
                index += 1
    column_positions = {key[i]: i for i in range(num_columns)}
    sorted_key = ''.join(sorted(key))
    encrypted_text = ''
    for col_char in sorted_key:
        col = column_positions[col_char]
        for row in range(num_rows):
            encrypted_text += grid[row][col]
    return encrypted_text
def decrypt_columnar_transposition(cipher_text, key):
```

```
    key = key.upper()
    num_columns = len(key)
    num_rows = -(-len(cipher_text) // num_columns)  # Ceiling division
    grid = [['' for _ in range(num_columns)] for _ in range(num_rows)]
  _chars_in_last_col = len(cipher_text) % num_columns
    column_positions = {key[i]: i for i in range(num_columns)}
    num_full_columns = num_columns - num_chars_in_last_col
    chars_in_full_columns = len(cipher_text) // num_columns
    index = 0
    for col_char in key:
        col = column_positions[col_char]
        if col < num_full_columns:
            num_chars_in_col = chars_in_full_columns
        else:
            num_chars_in_col = chars_in_full_columns + 1
        for row in range(num_rows):
            grid[row][col] = cipher_text[index]
            index += 1
    decrypted_text = ''
    for row in range(num_rows):
        for col in range(num_columns):
            decrypted_text += grid[row][col]
    return decrypted_text
def main():
    choice = input("Enter 'E' to encrypt or 'D' to decrypt: ").upper()
    if choice != 'E' and choice != 'D':
        print("Invalid choice. Please enter 'E' to encrypt or 'D' to decrypt.")
        return
    key = input("Enter the key (letters only): ")
    if not key.isalpha():
```

```
        print("Invalid key. Please use letters only.")
        return
    if choice == 'E':
        plain_text = input("Enter the plaintext: ")
        encrypted_text = encrypt_columnar_transposition(plain_text, key)
        print("Encrypted Text:", encrypted_text)
    else:
        cipher_text = input("Enter the cipher text: ")
        decrypted_text = decrypt_columnar_transposition(cipher_text, key)
        print("Decrypted Text:", decrypted_text)
if __name__ == "__main__":
    main()
```

Experiment – 09

Aim - Implement substitution technique.

Implementation

Substitution Technique: A cryptographic method that involves replacing each character in the plaintext with another character based on a predetermined substitution rule, typically used to obscure the original content of the message.

(1) Python Code

```
def caesar_cipher_encrypt(plain_text, shift):
    encrypted_text = ""
    for char in plain_text:
        if char.isalpha():
            is_upper = char.isupper()
            char_code = ord(char)
            char_code = (char_code - ord('A' if is_upper else 'a') + shift) % 26
            char_code += ord('A' if is_upper else 'a')
            encrypted_text += chr(char_code)
        else:
            encrypted_text += char
    return encrypted_text
def caesar_cipher_decrypt(cipher_text, shift):
    return caesar_cipher_encrypt(cipher_text, -shift)
def main():
    choice = input("Enter 'E' to encrypt or 'D' to decrypt: ").upper()
    if choice != 'E' and choice != 'D':
        print("Invalid choice. Please enter 'E' to encrypt or 'D' to decrypt.")
        return
    shift = int(input("Enter the shift value (integer): "))
    if choice == 'E':
```

```
        plain_text = input("Enter the plaintext: ")
        encrypted_text = caesar_cipher_encrypt(plain_text, shift)
        print("Encrypted Text:", encrypted_text)
    else:
        cipher_text = input("Enter the cipher text: ")
        decrypted_text = caesar_cipher_decrypt(cipher_text, shift)
        print("Decrypted Text:", decrypted_text)
if __name__ == "__main__":
    main()
```

Experiment – 10

Aim - Demonstrate intrusion detection system (ids) using any tool (snort or any other s/w)

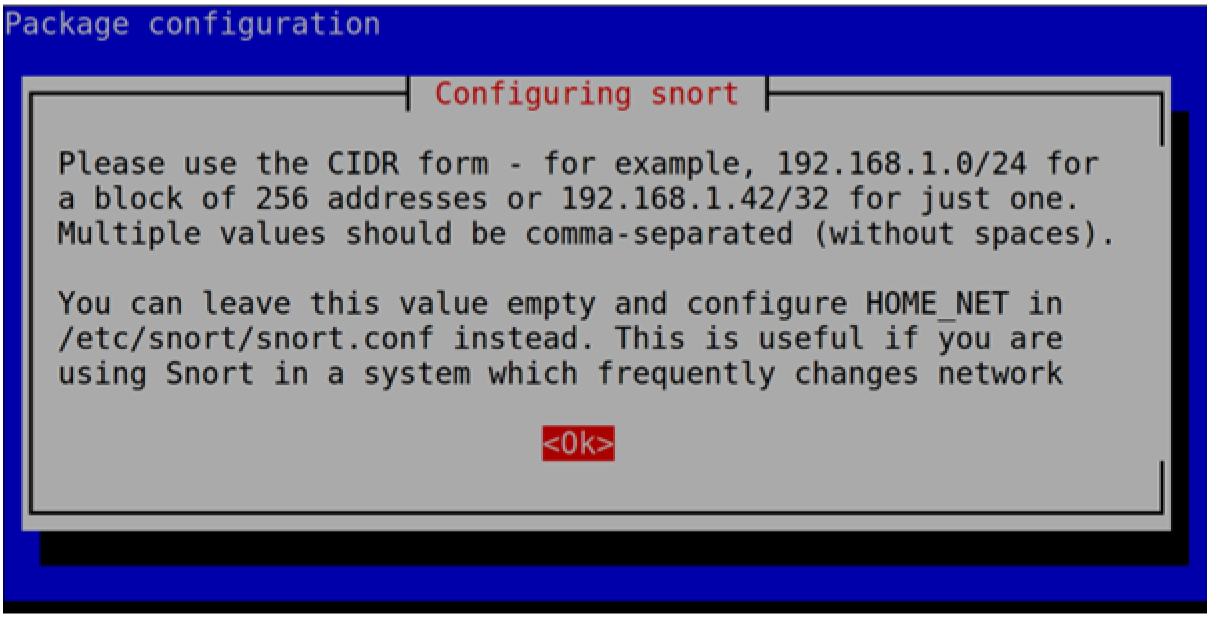
Implementation

Snort is an Intrusion Detection System which analyzes the traffic and packets to detect anomalies, such as malicious traffic, and report them.

You can install Snort using the *apt* packages manager on Debian or Ubuntu as shown in the following screenshot:

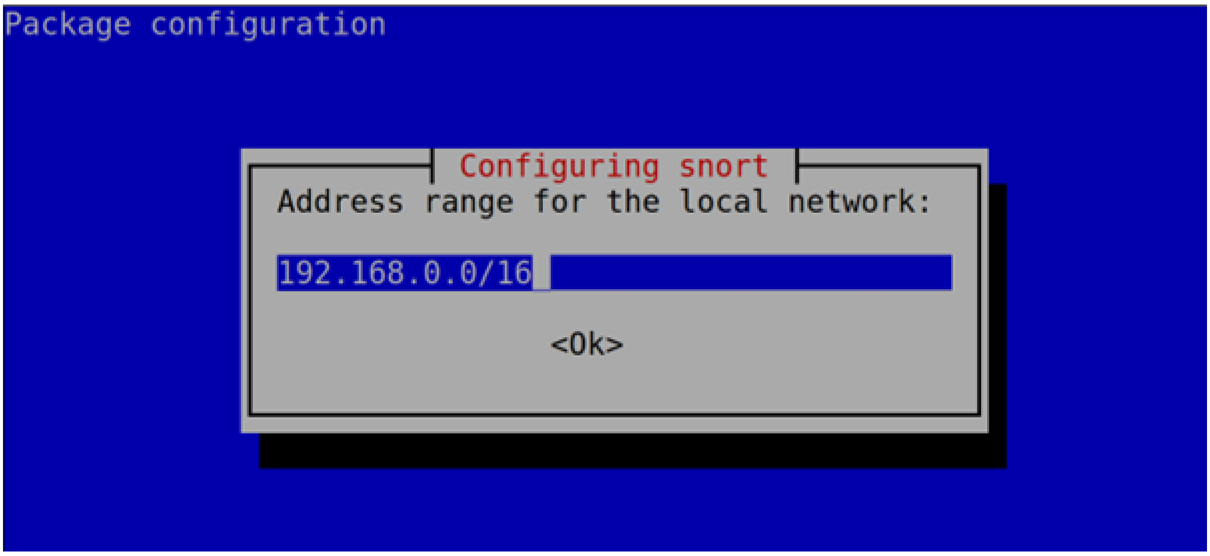
```
sudo apt install snort
```

During the installation process, you will be asked to define your network. Press ***OK*** to continue with the next step.



Now, type your network address in CIDR format. Normally, Snort auto detects it successfully.

Then, press ***OK*** or ***ENTER***. Don’t worry about this step; this configuration can be edited later.



Red Hat based Linux distribution users can download the Snort package from https://www.snort.org/downloads#snort-downloads and then install it by running the following command, where <*Version*> must be replaced with the current version that you downloaded.

```
sudo yum snort-<<em>Version</em>>.rpm
```

Snort contains two main types of rules: community rules developed by the Snort community and official rules. You can always update the community rules by default. But to update the official rules, you need an Oink Code – a code which allows you to download the latest rules.