Final Year B.Tech. (CSE) – VII [2024-25]

6CS451: Cryptography and Network Security Lab (C&NS Lab)

Date: 01/08/2024

Assignment 1

PRN: 21510042 **Name:** Omkar Rajesh Auti

---

1. Perform encryption, decryption using the following substitution techniques:

a. Ceaser cipher

Ans:

The Caesar Cipher is a simple encryption technique where each letter in a message is shifted by a fixed number of positions in the alphabet. For example, with a shift of 3, "A" becomes "D," "B" becomes "E," and so on. It's one of the oldest known ciphers and is easy to implement but also easy to break.

Python Code:

```python
def caesar_encrypt(text, shift):
    """
    Encrypt the plain text using Caesar cipher.

    Parameters:
    text (str): The input text to be encrypted.
    shift (int): The number of positions to shift each character.

    Returns:
    str: The encrypted text.
    """
    encrypted_text = ""
    for char in text:
        if char.isalpha():
            shift_amount = shift % 26
            if char.islower():
                new_char = chr((ord(char) - ord('a') + shift_amount) % 26 + ord('a'))
            else:
                new_char = chr((ord(char) - ord('A') + shift_amount) % 26 + ord('A'))
```

```python
            encrypted_text += new_char
        else:
            encrypted_text += char
    return encrypted_text


def caesar_decrypt(text, shift):
    """
    Decrypt the encrypted text using Caesar cipher.

    Parameters:
    text (str): The input text to be decrypted.
    shift (int): The number of positions to shift each character back.

    Returns:
    str: The decrypted text.
    """
    return caesar_encrypt(text, -shift)


def main():
    """
    The main function to run the menu-driven program.
    """
    while True:
        print("\nCaesar Cipher Program")
        print("1. Encrypt")
        print("2. Decrypt")
        print("3. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            plain_text = input("\nEnter the plain text: ")
            shift = int(input("Enter the shift value: "))
            encrypted_text = caesar_encrypt(plain_text, shift)
            print(f"\nEncrypted Text: {encrypted_text}")
        elif choice == '2':
            encrypted_text = input("Enter the encrypted text: ")
            shift = int(input("Enter the shift value: "))
```

```python
            decrypted_text = caesar_decrypt(encrypted_text, shift)
            print(f"Decrypted Text: {decrypted_text}")
        elif choice == '3':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB> python -u "c:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 1\caesar_cipher.py"

Caesar Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 1

Enter the plain text: Omkar
Enter the shift value: 3

Encrypted Text: Rpndu

Caesar Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 2
Enter the encrypted text: Rpndu
Enter the shift value: 3
Decrypted Text: Omkar

Caesar Cipher Program
1. Encrypt
2. Decrypt
3. Exit
```

Advantages:

- **Simplicity**: Easy to understand and implement.
- **Efficiency**: Fast encryption and decryption.

Disadvantages:

- **Weak Security**: Vulnerable to frequency analysis and brute-force attacks (only 25 possible shifts).
- **Predictability**: Does not change much between different texts.

b. Playfair cipher

Ans:

The Playfair Cipher is a digraph substitution cipher that encrypts pairs of letters. It uses a 5x5 matrix of letters created from a keyword. To encrypt, locate each letter pair in the matrix and swap or substitute based on their positions. It's more secure than simple substitution ciphers because it encodes pairs of letters rather than individual letters.

Python code:

```
def generate_playfair_matrix(key):
    """
    Generate a 5x5 matrix for the Playfair cipher based on the provided key.

    Parameters:
    key (str): The key to generate the matrix.

    Returns:
    list: A 5x5 matrix for the Playfair cipher.
    """
    key = key.upper().replace("J", "I")
    matrix = []
    used = set()

    for char in key:
        if char not in used and char.isalpha():
            used.add(char)
            matrix.append(char)

    for char in "ABCDEFGHIKLMNOPQRSTUVWXYZ":
        if char not in used:
            used.add(char)
            matrix.append(char)

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


def find_position(matrix, char):
    """
    Find the row and column of a character in the Playfair matrix.
```

```
    Parameters:
    matrix (list): The 5x5 matrix for the Playfair cipher.
    char (str): The character to find in the matrix.

    Returns:
    tuple: The row and column of the character in the matrix.
    """
    for row in range(5):
        for col in range(5):
            if matrix[row][col] == char:
                return row, col
    return None


def playfair_encrypt(text, key):
    """
    Encrypt the plain text using the Playfair cipher.

    Parameters:
    text (str): The input text to be encrypted.
    key (str): The key for the Playfair cipher.

    Returns:
    str: The encrypted text.
    """
    text = text.upper().replace("J", "I").replace(" ", "")
    if len(text) % 2 != 0:
        text += "X"

    matrix = generate_playfair_matrix(key)
    encrypted_text = ""

    for i in range(0, len(text), 2):
        char1, char2 = text[i], text[i + 1]

        if char1 == char2:
            char2 = 'X'
```

```
        row1, col1 = find_position(matrix, char1)
        row2, col2 = find_position(matrix, char2)

        if row1 == row2:
            encrypted_text += matrix[row1][(col1 + 1) % 5]
            encrypted_text += matrix[row2][(col2 + 1) % 5]
        elif col1 == col2:
            encrypted_text += matrix[(row1 + 1) % 5][col1]
            encrypted_text += matrix[(row2 + 1) % 5][col2]
        else:
            encrypted_text += matrix[row1][col2]
            encrypted_text += matrix[row2][col1]

    return encrypted_text


def playfair_decrypt(text, key):
    """
    Decrypt the encrypted text using the Playfair cipher.

    Parameters:
    text (str): The input text to be decrypted.
    key (str): The key for the Playfair cipher.

    Returns:
    str: The decrypted text.
    """
    text = text.upper().replace("J", "I").replace(" ", "")
    matrix = generate_playfair_matrix(key)
    decrypted_text = ""

    for i in range(0, len(text), 2):
        char1, char2 = text[i], text[i + 1]

        row1, col1 = find_position(matrix, char1)
        row2, col2 = find_position(matrix, char2)

        if row1 == row2:
            decrypted_text += matrix[row1][(col1 - 1) % 5]
```

```
            decrypted_text += matrix[row2][(col2 - 1) % 5]
        elif col1 == col2:
            decrypted_text += matrix[(row1 - 1) % 5][col1]
            decrypted_text += matrix[(row2 - 1) % 5][col2]
        else:
            decrypted_text += matrix[row1][col2]
            decrypted_text += matrix[row2][col1]

    return decrypted_text


def main():
    """
    The main function to run the menu-driven program.
    """
    while True:
        print("\nPlayfair Cipher Program")
        print("1. Encrypt")
        print("2. Decrypt")
        print("3. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            plain_text = input("\nEnter the plain text: ")
            key = input("Enter the key: ")
            encrypted_text = playfair_encrypt(plain_text, key)
            print(f"\nEncrypted Text: {encrypted_text}")
        elif choice == '2':
            encrypted_text = input("\nEnter the encrypted text: ")
            key = input("Enter the key: ")
            decrypted_text = playfair_decrypt(encrypted_text, key)
            print(f"\nDecrypted Text: {decrypted_text}")
        elif choice == '3':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please try again.")
```

```python
if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB> python -u "c:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 1\playfair_cipher.py"

Playfair Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 1

Enter the plain text: MEET ME AT NOON
Enter the key: MONARCHY

Encrypted Text: CLKLCLRSANNA

Playfair Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 2

Enter the encrypted text: CLKLCLRSANNA
Enter the key: MONARCHY

Decrypted Text: MEETMEATNOON

Playfair Cipher Program
1. Encrypt
2. Decrypt
3. Exit
```

Advantages:

- **Improved Security**: More secure than Caesar Cipher as it encrypts digraphs (pairs of letters).
- **Simplicity**: Slightly more complex but still relatively easy to implement.

Disadvantages:

- **Key Management**: Requires a good keyword and matrix setup.
- **Vulnerability**: Can still be broken with modern techniques like frequency analysis of digraphs.

c. Hill Cipher

Ans:

The Hill Cipher is a polygraphic substitution cipher that uses linear algebra. It encrypts blocks of text (usually 2x2 or 3x3 matrices) by multiplying them with a key matrix. The key matrix must be invertible for decryption. This method allows for more complex encryption compared to simple substitution ciphers.

Python code:

```python
import numpy as np
```

```python
def mod_inverse(matrix, modulus):
    """
    Calculate the modular inverse of a matrix under a given modulus.

    Parameters:
    matrix (numpy.ndarray): The matrix to invert.
    modulus (int): The modulus value.

    Returns:
    numpy.ndarray: The modular inverse of the matrix.
    """
    det = int(np.round(np.linalg.det(matrix)))
    det_inv = pow(det, -1, modulus)
    matrix_modulus_inv = (
        det_inv * np.round(det * np.linalg.inv(matrix)).astype(int) % modulus
    )
    return matrix_modulus_inv


def hill_encrypt(text, key):
    """
    Encrypt the plain text using the Hill cipher.

    Parameters:
    text (str): The input text to be encrypted.
    key (list of int): The key for the Hill cipher as a flat list.

    Returns:
    str: The encrypted text.
    """
    size = int(len(key) ** 0.5)
    key_matrix = np.array(key).reshape(size, size)
    modulus = 26
    text_vector = np.array([ord(char) - ord('A') for char in text])
    text_vector = text_vector.reshape(-1, size).T
    encrypted_vector = (np.dot(key_matrix, text_vector) % modulus).T
    encrypted_text = ''.join(chr(num + ord('A')) for num in
encrypted_vector.flatten())
    return encrypted_text
```

```
def hill_decrypt(text, key):
    """
    Decrypt the encrypted text using the Hill cipher.

    Parameters:
    text (str): The input text to be decrypted.
    key (list of int): The key for the Hill cipher as a flat list.

    Returns:
    str: The decrypted text.
    """
    size = int(len(key) ** 0.5)
    key_matrix = np.array(key).reshape(size, size)
    modulus = 26
    key_matrix_inv = mod_inverse(key_matrix, modulus)
    text_vector = np.array([ord(char) - ord('A') for char in text])
    text_vector = text_vector.reshape(-1, size).T
    decrypted_vector = (np.dot(key_matrix_inv, text_vector) % modulus).T
    decrypted_text = ''.join(chr(int(num) + ord('A')) for num in
decrypted_vector.flatten())
    return decrypted_text


def main():
    """
    The main function to run the menu-driven program.
    """
    while True:
        print("\nHill Cipher Program")
        print("1. Encrypt")
        print("2. Decrypt")
        print("3. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            plain_text = input("\nEnter the plain text (length multiple of key matrix
size): ").upper().replace(" ", "")
```

```
            key = input("Enter the key matrix (comma-separated integers, e.g., '2,4,5,9' for 2x2 matrix): ")
            key_matrix = list(map(int, key.split(',')))
            size = int(len(key_matrix) ** 0.5)
            if len(plain_text) % size != 0:
                print("Error: The length of the plain text must be a multiple of the key matrix size.")
                continue
            encrypted_text = hill_encrypt(plain_text, key_matrix)
            print(f"\nEncrypted Text: {encrypted_text}")
        elif choice == '2':
            encrypted_text = input("\nEnter the encrypted text: ").upper().replace(" ", "")
            key = input("Enter the key matrix (comma-separated integers, e.g., '2,4,5,9' for 2x2 matrix): ")
            key_matrix = list(map(int, key.split(',')))
            size = int(len(key_matrix) ** 0.5)
            if len(encrypted_text) % size != 0:
                print("Error: The length of the encrypted text must be a multiple of the key matrix size.")
                continue
            decrypted_text = hill_decrypt(encrypted_text, key_matrix)
            print(f"\nDecrypted Text: {decrypted_text}")
        elif choice == '3':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB> python -u "c:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 1\hill_cipher.py"

Hill Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 1

Enter the plain text (length multiple of key matrix size): WILL MEET TOMORROW
Enter the key matrix (comma-separated integers, e.g., '2,4,5,9' for 2x2 matrix): 6,1,3,2

Encrypted Text: KEZDYSRYYHIMPHCI

Hill Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 2

Enter the encrypted text: KEZDYSRYYHIMPHCI
Enter the key matrix (comma-separated integers, e.g., '2,4,5,9' for 2x2 matrix): 6,1,3,2

Decrypted Text: WILLMEETTOMORROW

Hill Cipher Program
1. Encrypt
2. Decrypt
3. Exit
```

Advantages:

- **Polyalphabetic**: Uses linear algebra for encryption, making it stronger than monoalphabetic ciphers.
- **Higher Complexity**: More resistant to frequency analysis due to the use of matrices.

Disadvantages:

- **Complexity**: Requires matrix inversion and modular arithmetic, which can be cumbersome.
- **Key Size**: Key matrix must be invertible, and the length of the plaintext must be a multiple of the matrix size.

d. Vigenere cipher

Ans:

The Vigenère Cipher is a method of encrypting text using a keyword. It works by shifting each letter in the plaintext by an amount determined by the corresponding letter in the keyword. The key repeats itself if it's shorter than the plaintext.

How It Works:

1. **Keyword**: Choose a keyword (e.g., "KEY").
2. **Encryption**:
    - Write the keyword repeatedly above the plaintext.
    - Shift each letter in the plaintext by the position of the corresponding letter in the keyword (A=0, B=1, ..., Z=25).

3. **Decryption**:
    - Use the same keyword to reverse the shifts and recover the plaintext.

Python Code:

```
def vigenere_encrypt(plain_text, key):
    """
    Encrypt the plain text using the Vigenere cipher.

    Parameters:
    plain_text (str): The input text to be encrypted.
    key (str): The key for the Vigenere cipher.

    Returns:
    str: The encrypted text.
    """
    plain_text = plain_text.upper().replace(" ", "")
    key = key.upper().replace(" ", "")
    key_length = len(key)
    encrypted_text = ""

    for i, char in enumerate(plain_text):
        if char.isalpha():
            shift = ord(key[i % key_length]) - ord('A')
            encrypted_char = chr((ord(char) - ord('A') + shift) % 26 + ord('A'))
            encrypted_text += encrypted_char
        else:
            encrypted_text += char

    return encrypted_text


def vigenere_decrypt(cipher_text, key):
    """
    Decrypt the cipher text using the Vigenere cipher.

    Parameters:
    cipher_text (str): The input text to be decrypted.
```

```
    key (str): The key for the Vigenere cipher.

    Returns:
    str: The decrypted text.
    """
    cipher_text = cipher_text.upper().replace(" ", "")
    key = key.upper().replace(" ", "")
    key_length = len(key)
    decrypted_text = ""

    for i, char in enumerate(cipher_text):
        if char.isalpha():
            shift = ord(key[i % key_length]) - ord('A')
            decrypted_char = chr((ord(char) - ord('A') - shift + 26) % 26 + ord('A'))
            decrypted_text += decrypted_char
        else:
            decrypted_text += char

    return decrypted_text


def main():
    """
    The main function to run the menu-driven program.
    """
    while True:
        print("\nVigenere Cipher Program")
        print("1. Encrypt")
        print("2. Decrypt")
        print("3. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            plain_text = input("\nEnter the plain text: ")
            key = input("Enter the key: ")
            encrypted_text = vigenere_encrypt(plain_text, key)
            print(f"\nEncrypted Text: {encrypted_text}")
        elif choice == '2':
            encrypted_text = input("\nEnter the encrypted text: ")
```

```
            key = input("Enter the key: ")
            decrypted_text = vigenere_decrypt(encrypted_text, key)
            print(f"\nDecrypted Text: {decrypted_text}")
        elif choice == '3':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB> python -u "c:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 1\vigenere_cipher.py"

Vigenere Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 1

Enter the plain text: MEET ME AT MORNING
Enter the key: CODE

Encrypted Text: OSHXOSDXOCURKBJ

Vigenere Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 2

Enter the encrypted text: OSHXOSDXOCURKBJ
Enter the key: CODE

Decrypted Text: MEETMEATMORNING

Vigenere Cipher Program
1. Encrypt
2. Decrypt
3. Exit
```

Advantages:

- **Polyalphabetic**: Uses a keyword to shift letters, making it more secure than Caesar Cipher.
- **Improved Security**: Harder to crack with frequency analysis if the keyword is long and complex.

Disadvantages:

- **Keyword Management**: Security depends on the keyword length and complexity.
- **Vulnerabilities**: Can be broken with techniques like the Kasiski examination or frequency analysis if the keyword is short.

Final Year B.Tech. (CSE) – VII [2024-25]

6CS451: Cryptography and Network Security Lab (C&NS Lab)

Date: 05/08/2024

Assignment 2

PRN: 21510042 **Name:** Omkar Rajesh Auti

---

1. Perform encryption and decryption using following transposition techniques

a. Rail fence

Ans:

The Rail Fence Cipher is a type of transposition cipher where the plain text is written in a zigzag pattern across multiple "rails" (rows) and then read row by row to create the cipher text. Decryption involves reconstructing the zigzag pattern to retrieve the original message.

Python code:

```
def rail_fence_encrypt(plain_text, key):
    """
    Encrypt the plain text using the Rail Fence cipher.

    Parameters:
    plain_text (str): The input text to be encrypted.
    key (int): The number of rails (rows) for the Rail Fence cipher.

    Returns:
    str: The encrypted text.
    """
    # Create a list of strings to represent each rail
    rail = ['' for _ in range(key)]
    row, direction = 0, 1

    # Distribute the characters across the rails in a zigzag pattern
    for char in plain_text:
        rail[row] += char
        row += direction
```

```
        # Reverse direction when we reach the top or bottom rail
        if row == 0 or row == key - 1:
            direction *= -1

    # Concatenate all the rails to get the encrypted text
    return ''.join(rail)


def rail_fence_decrypt(cipher_text, key):
    """
    Decrypt the cipher text using the Rail Fence cipher.

    Parameters:
    cipher_text (str): The input text to be decrypted.
    key (int): The number of rails (rows) for the Rail Fence cipher.

    Returns:
    str: The decrypted text.
    """
    # Determine the length of each rail in the zigzag pattern
    pattern = [0] * len(cipher_text)
    row, direction = 0, 1

    for i in range(len(cipher_text)):
        pattern[i] = row
        row += direction

        # Reverse direction when we reach the top or bottom rail
        if row == 0 or row == key - 1:
            direction *= -1

    # Reconstruct the rails from the cipher text
    rail_lengths = [pattern.count(i) for i in range(key)]
    rail_chars = ['' for _ in range(key)]
    pos = 0

    for i in range(key):
        rail_chars[i] = cipher_text[pos:pos + rail_lengths[i]]
        pos += rail_lengths[i]
```

```python
    # Reconstruct the original message by following the zigzag pattern
    result = []
    row_pointers = [0] * key
    for i in range(len(cipher_text)):
        result.append(rail_chars[pattern[i]][row_pointers[pattern[i]]])
        row_pointers[pattern[i]] += 1

    return ''.join(result)


def main():
    """
    The main function to run the menu-driven program.
    """
    while True:
        print("\nRail Fence Cipher Program")
        print("1. Encrypt")
        print("2. Decrypt")
        print("3. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            plain_text = input("\nEnter the plain text: ").replace(" ", "")
            key = int(input("Enter the number of rails: "))
            encrypted_text = rail_fence_encrypt(plain_text, key)
            print(f"\nEncrypted Text: {encrypted_text}")
        elif choice == '2':
            cipher_text = input("\nEnter the encrypted text: ").replace(" ", "")
            key = int(input("Enter the number of rails: "))
            decrypted_text = rail_fence_decrypt(cipher_text, key)
            print(f"\nDecrypted Text: {decrypted_text}")
        elif choice == '3':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please try again.")
```

```python
if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB> python -u "c:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 2\rail_fence.py"

Rail Fence Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 1

Enter the plain text: HELLO FROM OTHER SIDE
Enter the number of rails: 3

Encrypted Text: HOMEDELFOOHRIELRTS

Rail Fence Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 2

Enter the encrypted text: HOMEDELFOOHRIELRTS
Enter the number of rails: 3

Decrypted Text: HELLOFROMOTHERSIDE

Rail Fence Cipher Program
1. Encrypt
2. Decrypt
3. Exit
```

Advantages:

- **Simplicity**: Easy to understand and implement.
- **Low Computation**: Requires minimal computational resources for encryption and decryption.

Disadvantages:

- **Weak Security**: Very easy to break with simple analysis or known-plaintext attacks.
- **Pattern Recognition**: The regular zigzag pattern makes it susceptible to pattern recognition, which can be exploited to decode the message.

b. row and Column Transformation

Ans:

Row and column transformation is a type of transposition cipher where the message is written in a grid (matrix) and the order of rows and columns is changed according to a key.

Row Transposition: Encrypts text by writing it into rows of a grid, then permuting the columns according to a specific key.

Column Transposition: Encrypts text by writing it into columns of a grid, then permuting the rows according to a specific key.

How It Works:

1. **Write** the plaintext into a grid according to the number of rows or columns.
2. **Permute** the rows or columns based on the key.
3. **Read** off the text in the new order to get the ciphertext.

Python code:

```
import math

def create_matrix(text, key_len):
    """
    Create a matrix from the text with the specified number of columns (key length).
    """
    rows = math.ceil(len(text) / key_len)
    matrix = [['' for _ in range(key_len)] for _ in range(rows)]
    k = 0

    for i in range(rows):
        for j in range(key_len):
            if k < len(text):
                matrix[i][j] = text[k]
                k += 1
            else:
                matrix[i][j] = 'X'  # Padding with 'X' if the matrix is not full

    return matrix


def row_column_encrypt(plain_text, row_key, col_key):
    """
    Encrypt the plain text using row and column transformation.

    Parameters:
    plain_text (str): The input text to be encrypted.
    row_key (list): The key to rearrange rows.
    col_key (list): The key to rearrange columns.

    Returns:
```

```
    str: The encrypted text.
    """
    plain_text = plain_text.replace(" ", "")
    key_len = len(col_key)

    # Create the matrix from the plain text
    matrix = create_matrix(plain_text, key_len)

    # Apply the row key
    row_matrix = [matrix[i] for i in row_key]

    # Apply the column key
    encrypted_text = ""
    for row in row_matrix:
        encrypted_row = [row[j] for j in col_key]
        encrypted_text += ''.join(encrypted_row)

    return encrypted_text


def row_column_decrypt(cipher_text, row_key, col_key):
    """
    Decrypt the cipher text using row and column transformation.

    Parameters:
    cipher_text (str): The input text to be decrypted.
    row_key (list): The key to rearrange rows.
    col_key (list): The key to rearrange columns.

    Returns:
    str: The decrypted text.
    """
    key_len = len(col_key)
    rows = len(cipher_text) // key_len

    # Create the matrix to store the rearranged cipher text
    matrix = [['' for _ in range(key_len)] for _ in range(rows)]
    k = 0
```

```python
    # Arrange the cipher text in the matrix based on the column key
    for i in range(len(row_key)):
        for j in col_key:
            matrix[row_key[i]][j] = cipher_text[k]
            k += 1

    # Read the decrypted text row by row
    decrypted_text = ""
    for i in range(rows):
        decrypted_text += ''.join(matrix[i])

    return decrypted_text


def main():
    """
    The main function to run the menu-driven program.
    """
    while True:
        print("\nRow and Column Transformation Cipher Program")
        print("1. Encrypt")
        print("2. Decrypt")
        print("3. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            plain_text = input("\nEnter the plain text: ")
            row_key = list(map(int, input("Enter the row key as a sequence of numbers (e.g., 2 0 1): ").split()))
            col_key = list(map(int, input("Enter the column key as a sequence of numbers (e.g., 1 0 2): ").split()))
            encrypted_text = row_column_encrypt(plain_text, row_key, col_key)
            print(f"\nEncrypted Text: {encrypted_text}")
        elif choice == '2':
            cipher_text = input("\nEnter the encrypted text: ")
            row_key = list(map(int, input("Enter the row key as a sequence of numbers (e.g., 2 0 1): ").split()))
            col_key = list(map(int, input("Enter the column key as a sequence of numbers (e.g., 1 0 2): ").split()))
```

```
            decrypted_text = row_column_decrypt(cipher_text, row_key, col_key)
            print(f"\nDecrypted Text: {decrypted_text}")
        elif choice == '3':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB> python -u "c:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 2\row_column_transformat
ion.py"

Row and Column Transformation Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 1

Enter the plain text: HELLO WORLD
Enter the row key as a sequence of numbers (e.g., 2 0 1): 2 0 1
Enter the column key as a sequence of numbers (e.g., 1 0 2): 1 0 2

Encrypted Text: ROLEHLOLW

Row and Column Transformation Cipher Program
1. Encrypt
2. Decrypt
3. Exit
Enter your choice: 2

Enter the encrypted text: ROLEHLOLW
Enter the row key as a sequence of numbers (e.g., 2 0 1): 2 0 1
Enter the column key as a sequence of numbers (e.g., 1 0 2): 1 0 2

Decrypted Text: HELLOWORL

Row and Column Transformation Cipher Program
1. Encrypt
2. Decrypt
3. Exit
```

Advantages:

- **Increased Security**: More complex than simple transpositions.
- **Flexibility**: Key-based rearrangement can add security.

Disadvantages:

- **Complexity**: Can be more complex to implement and manage compared to simple ciphers.
- **Pattern Recognition**: Still susceptible to pattern analysis if not combined with other encryption methods.

Final Year B.Tech. (CSE) – VII [2024-25]

6CS451: Cryptography and Network Security Lab (C&NS Lab)

Date: 12/08/2024

Assignment 3

PRN: 21510042 **Name:** Omkar Rajesh Auti

---

1. Implementation of Euclidean and Extended Euclidean Algorithm

Ans:

The Euclidean and Extended Euclidean algorithms are essential for finding the greatest common divisor (GCD) of two integers. The Extended Euclidean algorithm also finds the coefficients of Bézout's identity, which are useful in solving linear Diophantine equations and in modular arithmetic.

Euclidean Algorithm

The Euclidean algorithm finds the GCD of two numbers by repeatedly applying the following rule: gcd(a, b) = gcd(b, a % b) until b becomes zero. The GCD is then the non-zero remainder.

Extended Euclidean Algorithm

The Extended Euclidean algorithm not only computes the GCD of two integers a and b, but also finds integers x and y such that ax + by = gcd(a, b).

Python Code:

```
def euclidean_algorithm(a, b):
    """
    Compute the GCD of a and b using the Euclidean algorithm.

    Parameters:
    a (int): First integer.
    b (int): Second integer.

    Returns:
    int: The GCD of a and b.
    """
```

```
    while b != 0:
        a, b = b, a % b
    return a


def extended_euclidean_algorithm(a, b):
    """
    Compute the GCD of a and b, as well as the coefficients x and y
    such that ax + by = gcd(a, b) using the Extended Euclidean algorithm.

    Parameters:
    a (int): First integer.
    b (int): Second integer.

    Returns:
    tuple: (gcd, x, y) where gcd is the GCD of a and b, and x, y are
    the coefficients of Bézout's identity.
    """
    if b == 0:
        return a, 1, 0
    else:
        gcd, x1, y1 = extended_euclidean_algorithm(b, a % b)
        x = y1
        y = x1 - (a // b) * y1
        return gcd, x, y


def main():
    """
    The main function to run the program.
    """
    while True:
        print("\nEuclidean and Extended Euclidean Algorithm")
        print("1. Compute GCD using Euclidean Algorithm")
        print("2. Compute GCD and coefficients using Extended Euclidean
Algorithm")
        print("3. Exit")
        choice = input("Enter your choice: ")
```

```python
        if choice == '1':
            a = int(input("\nEnter the first integer (a): "))
            b = int(input("Enter the second integer (b): "))
            gcd = euclidean_algorithm(a, b)
            print(f"\nGCD of {a} and {b} is: {gcd}")
        elif choice == '2':
            a = int(input("\nEnter the first integer (a): "))
            b = int(input("Enter the second integer (b): "))
            gcd, x, y = extended_euclidean_algorithm(a, b)
            print(f"\nGCD of {a} and {b} is: {gcd}")
            print(f"Coefficients x and y are: x = {x}, y = {y}")
            print(f"\nBézout's identity: {a}*({x}) + {b}*({y}) = {gcd}")
        elif choice == '3':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB> python -u "c:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 3\euclidean.py"

Euclidean and Extended Euclidean Algorithm
1. Compute GCD using Euclidean Algorithm
2. Compute GCD and coefficients using Extended Euclidean Algorithm
3. Exit
Enter your choice: 1

Enter the first integer (a): 15
Enter the second integer (b): 5

GCD of 15 and 5 is: 5

Euclidean and Extended Euclidean Algorithm
1. Compute GCD using Euclidean Algorithm
2. Compute GCD and coefficients using Extended Euclidean Algorithm
3. Exit
Enter your choice: 2

Enter the first integer (a): 48
Enter the second integer (b): 18

GCD of 48 and 18 is: 6
Coefficients x and y are: x = -1, y = 3

Bézout's identity: 48*(-1) + 18*(3) = 6

Euclidean and Extended Euclidean Algorithm
1. Compute GCD using Euclidean Algorithm
2. Compute GCD and coefficients using Extended Euclidean Algorithm
3. Exit
```

This implementation of the Euclidean and Extended Euclidean algorithms is fundamental in cryptography, number theory, and algorithms related to modular arithmetic.

Final Year B.Tech. (CSE) – VII [2024-25]

6CS451: Cryptography and Network Security Lab (C&NS Lab)

Date: 19/08/2024

Assignment 4

PRN: 21510042 **Name:** Omkar Rajesh Auti

---

1. Implementation of Chinese Remainder Theorem (CRT)

Ans:

The Chinese Remainder Theorem (CRT) is a powerful tool in number theory that provides a solution to a system of simultaneous congruences with pairwise coprime moduli. Given a system of congruences, the CRT allows us to find a unique solution modulo the product of the moduli.

Problem Description

Given n congruences: x≡a1 (mod m1), x≡a2 (mod m2) ⋮x ≡ an (mod mn)

Where the moduli m1, m2, …, mn are pairwise coprime, the CRT provides a unique solution modulo M=m1×m2×⋯× mn.

For each congruence x ≡ ai (mod mi), it calculates the partial solution using the formula: x ≡ ai × Mi × inverse(Mi, mi) (mod M) where Mi=M/mi

The final solution is obtained by summing all partial solutions modulo M.

Python code:

```python
def extended_euclidean_algorithm(a, b):
    """
    Compute the GCD of a and b, as well as the coefficients x and y
    such that ax + by = gcd(a, b) using the Extended Euclidean algorithm.

    Parameters:
    a (int): First integer.
    b (int): Second integer.

    Returns:
    tuple: (gcd, x, y) where gcd is the GCD of a and b, and x, y are
```

```
    the coefficients of Bézout's identity.
    """
    if b == 0:
        return a, 1, 0
    else:
        gcd, x1, y1 = extended_euclidean_algorithm(b, a % b)
        x = y1
        y = x1 - (a // b) * y1
        return gcd, x, y


def chinese_remainder_theorem(a, m):
    """
    Solve the system of congruences using the Chinese Remainder Theorem.

    Parameters:
    a (list): List of remainders.
    m (list): List of moduli (must be pairwise coprime).

    Returns:
    int: The smallest non-negative solution to the system of congruences.
    """
    assert len(a) == len(m), "The number of remainders and moduli must be the 
same"

    # Calculate the product of all moduli
    M = 1
    for mi in m:
        M *= mi

    # Initialize the solution
    x = 0

    # Apply the CRT
    for ai, mi in zip(a, m):
        Mi = M // mi  # M_i = M / m_i
        gcd, inverse, _ = extended_euclidean_algorithm(Mi, mi)
        if gcd != 1:
            raise ValueError("Moduli are not pairwise coprime")
```

```python
        x += ai * inverse * Mi

    return x % M


def main():
    """
    The main function to run the program.
    """
    while True:
        print("\nChinese Remainder Theorem (CRT)")
        print("1. Solve System of Congruences")
        print("2. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            n = int(input("\nEnter the number of congruences: "))
            a = []
            m = []
            for i in range(n):
                ai = int(input(f"\nEnter remainder a[{i+1}]: "))
                mi = int(input(f"Enter modulus m[{i+1}]: "))
                a.append(ai)
                m.append(mi)

            solution = chinese_remainder_theorem(a, m)
            print(f"\nThe solution to the system of congruences is: {solution}")
        elif choice == '2':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB> python -u "c:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 4\chinese_remainder_theorem.py"

Chinese Remainder Theorem (CRT)
1. Solve System of Congruences
2. Exit
Enter your choice: 1

Enter the number of congruences: 3

Enter remainder a[1]: 2
Enter modulus m[1]: 3

Enter remainder a[2]: 3
Enter modulus m[2]: 5

Enter remainder a[3]: 2
Enter modulus m[3]: 7

The solution to the system of congruences is: 23

Chinese Remainder Theorem (CRT)
1. Solve System of Congruences
2. Exit
```

Final Year B.Tech. (CSE) – VII [2024-25]

6CS451: Cryptography and Network Security Lab (C&NS Lab)

Date: 26/08/2024

Assignment 5

PRN: 21510042 **Name:** Omkar Rajesh Auti

---

1. Apply DES algorithm for practical applications

Ans:
The Data Encryption Standard (DES) is a symmetric-key algorithm for the encryption of digital data. Although DES is now considered insecure for many applications due to its small key size, it is still an important algorithm for understanding the basics of cryptography.

Practical Application of DES Algorithm

To apply the DES algorithm in a practical application, we can use the **pycryptodome** library in Python, which provides an implementation of DES. Below is an example that demonstrates how to use DES to encrypt and decrypt a message.

Python Code:

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes


def des_encrypt(plain_text, key):
    """
    Encrypt the plain text using DES algorithm.

    Parameters:
    plain_text (str): The text to be encrypted.
    key (bytes): The encryption key (must be 8 bytes long).

    Returns:
    bytes: The encrypted cipher text.
    """
    cipher = DES.new(key, DES.MODE_ECB)
```

```python
    padded_text = pad(plain_text.encode(), DES.block_size)
    encrypted_text = cipher.encrypt(padded_text)
    return encrypted_text


def des_decrypt(cipher_text, key):
    """
    Decrypt the cipher text using DES algorithm.

    Parameters:
    cipher_text (bytes): The encrypted text to be decrypted.
    key (bytes): The decryption key (must be 8 bytes long).

    Returns:
    str: The decrypted plain text.
    """
    cipher = DES.new(key, DES.MODE_ECB)
    decrypted_text = unpad(cipher.decrypt(cipher_text), DES.block_size)
    return decrypted_text.decode()


def main():
    """
    The main function to run the program.
    """
    print("\nDES Encryption and Decryption")

    # Generate a random 8-byte key for DES
    key = get_random_bytes(8)
    print(f"\nGenerated Key (in hexadecimal): {key.hex()}")

    # Input plaintext
    plain_text = input("Enter the plain text to encrypt: ")

    # Encrypt the plaintext
    encrypted_text = des_encrypt(plain_text, key)
    print(f"\nEncrypted Text (in hexadecimal): {encrypted_text.hex()}")

    # Decrypt the ciphertext
```

```python
    decrypted_text = des_decrypt(encrypted_text, key)
    print(f"\nDecrypted Text: {decrypted_text}")


if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB> python -u "c:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 5\des.py"

DES Encryption and Decryption

Generated Key (in hexadecimal): b4c3cd99606f04d5
Enter the plain text to encrypt: We will meet tomorrow at 5pm

Encrypted Text (in hexadecimal): ee41627fed00e3a68e7027242982742008fd4f408e0e232c5402ddb7c332ce53

Decrypted Text: We will meet tomorrow at 5pm
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB>
```

Practical Applications:

- **File Encryption:** DES can be used to encrypt sensitive files before storing them in insecure locations.
- **Secure Communication:** DES ensures that messages sent over a network are unreadable to unauthorized parties.
- **Password Storage:** Encrypting passwords before storing them in databases (though modern standards recommend stronger algorithms like AES).

While DES itself is outdated and not recommended for secure applications, understanding how it works is crucial for grasping more advanced encryption algorithms like AES.

Final Year B.Tech. (CSE) – VII [2024-25]

6CS451: Cryptography and Network Security Lab (C&NS Lab)

Date: 26/08/2024

Assignment 6

PRN: 21510042 **Name:** Omkar Rajesh Auti

---

1. Apply AES algorithm for practical applications

Ans:

The Advanced Encryption Standard (AES) is a widely used symmetric encryption algorithm that is both fast and secure. It is the standard encryption algorithm used by governments, financial institutions, and many other organizations. Unlike DES, which is now considered insecure, AES is robust and provides a high level of security.

Practical Application of AES Algorithm

We can use the **pycryptodome** library in Python to implement AES encryption and decryption. The AES algorithm can work with key sizes of 128, 192, or 256 bits, and it operates on 128-bit blocks. In this example, we'll use AES with a 256-bit key in Cipher Block Chaining (CBC) mode.

Python Code:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes


def aes_encrypt(plain_text, key):
    """
    Encrypt the plain text using AES algorithm.

    Parameters:
```

```
    plain_text (str): The text to be encrypted.
    key (bytes): The encryption key (must be 16, 24, or 32
bytes long).

    Returns:
    bytes: The initialization vector (IV) and the encrypted
cipher text.
    """
    cipher = AES.new(key, AES.MODE_CBC)
    iv = cipher.iv  # Initialization vector
    padded_text = pad(plain_text.encode(), AES.block_size)
    encrypted_text = cipher.encrypt(padded_text)
    return iv, encrypted_text


def aes_decrypt(iv, cipher_text, key):
    """
    Decrypt the cipher text using AES algorithm.

    Parameters:
    iv (bytes): The initialization vector used during
encryption.
    cipher_text (bytes): The encrypted text to be decrypted.
    key (bytes): The decryption key (must be 16, 24, or 32
bytes long).

    Returns:
    str: The decrypted plain text.
    """
    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted_text = unpad(cipher.decrypt(cipher_text),
AES.block_size)
    return decrypted_text.decode()


def main():
    """
    The main function to run the program.
    """
```

```python
    print("\nAES Encryption and Decryption")

    # Generate a random 32-byte key for AES (256-bit)
    key = get_random_bytes(32)
    print(f"\nGenerated Key (in hexadecimal): {key.hex()}")

    # Input plaintext
    plain_text = input("\nEnter the plain text to encrypt: ")

    # Encrypt the plaintext
    iv, encrypted_text = aes_encrypt(plain_text, key)
    print(f"\nInitialization Vector (IV) (in hexadecimal): {iv.hex()}")
    print(f"\nEncrypted Text (in hexadecimal): {encrypted_text.hex()}")

    # Decrypt the ciphertext
    decrypted_text = aes_decrypt(iv, encrypted_text, key)
    print(f"\nDecrypted Text: {decrypted_text}")


if __name__ == "__main__":
    main()
```

<u>Output:</u>

```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB> python -u "c:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 6\aes.py"

AES Encryption and Decryption

Generated Key (in hexadecimal): 9386a2e13110e424f7db8286de5303d18067a35f3e34352d97d93a062c7eb7d1

Enter the plain text to encrypt: Keep it secret!

Initialization Vector (IV) (in hexadecimal): d1802431c456d8fbe65149f3f6cf8f26

Encrypted Text (in hexadecimal): 9a46a061646697d48d93625d2a604423

Decrypted Text: Keep it secret!
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB>
```

<u>Practical Applications of AES:</u>

- **File Encryption:** Encrypting sensitive files before storing them on disk.

- **Secure Communication**: Ensuring that data sent over the network remains confidential.
- **Data Protection in Applications**: Encrypting user data, such as passwords, to protect them from unauthorized access.

AES is widely adopted due to its strength and efficiency, and it remains the standard for securing digital data across various industries.

Final Year B.Tech. (CSE) – VII [2024-25]

6CS451: Cryptography and Network Security Lab (C&NS Lab)

Date: 26/08/2024

Assignment 7

PRN: 21510042 **Name:** Omkar Rajesh Auti

---

1. Implementation of RSA Algorithm

Ans:

The RSA algorithm is one of the first public-key cryptosystems and is widely used for secure data transmission. It is an asymmetric cryptographic algorithm, meaning it uses a pair of keys: a public key for encryption and a private key for decryption. It relies on the mathematical properties of prime numbers.

How RSA Works:

1. **Key Generation:**
   - Choose two large prime numbers p and q.
   - Compute n = p * q.
   - Compute the totient ϕ(n) = (p-1) * (q-1).
   - Choose an encryption key e such that 1 < e < ϕ(n) and gcd(e, ϕ(n)) = 1. The integer e is the public key exponent.
   - Calculate the decryption key d such that d * e ≡ 1 (mod ϕ(n)). The integer d is the private key exponent.
2. **Encryption:**
   - The public key is (n, e).
   - Given a plaintext message M, the ciphertext C is computed as:

     C = M^ e mod n.
3. **Decryption:**

- The private key is (n, d).
- Given a ciphertext C, the plaintext M is recovered as:

  M = C^d mod n

To implement the RSA algorithm **using large prime numbers with 2048 bits** and converting plaintext into numbers, we'll use the **Crypto library in Python**, which provides the **necessary tools to handle such large prime numbers and perform RSA encryption and decryption.**

The large primes and the strong key sizes make RSA secure against most attacks when implemented correctly.

Python Code:

```
import random
from sympy import isprime, mod_inverse

def generate_prime_candidate(length):
    """Generate an odd integer randomly."""
    p = random.getrandbits(length)
    # Ensure p is odd
    p |= (1 << length - 1) | 1
    return p


def generate_prime_number(length):
    """Generate a prime number."""
    p = 4
    while not isprime(p):
        p = generate_prime_candidate(length)
    return p


def generate_keypair(keysize):
    """Generate RSA public and private keys."""
    # Generate two large primes p and q
    p = generate_prime_number(keysize)
    q = generate_prime_number(keysize)

    print("\np: ", p)
```

```python
    print("\nq: ", q)

    # Compute n = p * q
    n = p * q

    # Compute Euler's Totient ϕ(n) = (p-1)*(q-1)
    phi = (p - 1) * (q - 1)

    # Choose an integer e such that 1 < e < phi(n) and gcd(e,
phi(n)) = 1
    e = random.randrange(2, phi)
    g = gcd(e, phi)
    while g != 1:
        e = random.randrange(2, phi)
        g = gcd(e, phi)

    # Compute d, the modular inverse of e
    d = mod_inverse(e, phi)

    # Public key (e, n) and Private key (d, n)
    return ((e, n), (d, n))


def gcd(a, b):
    """Compute the greatest common divisor using Euclid's
algorithm."""
    while b != 0:
        a, b = b, a % b
    return a


def encrypt(public_key, plaintext):
    """Encrypt plaintext using the public key."""
    e, n = public_key
    cipher = [pow(ord(char), e, n) for char in plaintext]
    return cipher


def decrypt(private_key, ciphertext):
    """Decrypt ciphertext using the private key."""
    d, n = private_key
    plain = [chr(pow(char, d, n)) for char in ciphertext]
    return ''.join(plain)


def main():
    """Run RSA algorithm."""
```

```python
    print("RSA Encryption/Decryption")

    keysize = 2048  # Keysize in bits

    # Generate public and private keys
    public_key, private_key = generate_keypair(keysize)

    print(f"\nPublic key: {public_key}")
    print(f"Private key: {private_key}")

    # Input plaintext
    plaintext = input("\nEnter a message to encrypt: ")

    # Encrypt the message
    encrypted_msg = encrypt(public_key, plaintext)
    print(f"\nEncrypted message: {encrypted_msg}")

    # Decrypt the message
    decrypted_msg = decrypt(private_key, encrypted_msg)
    print(f"\nDecrypted message: {decrypted_msg}")


if __name__ == "__main__":
    main()
```

Output:

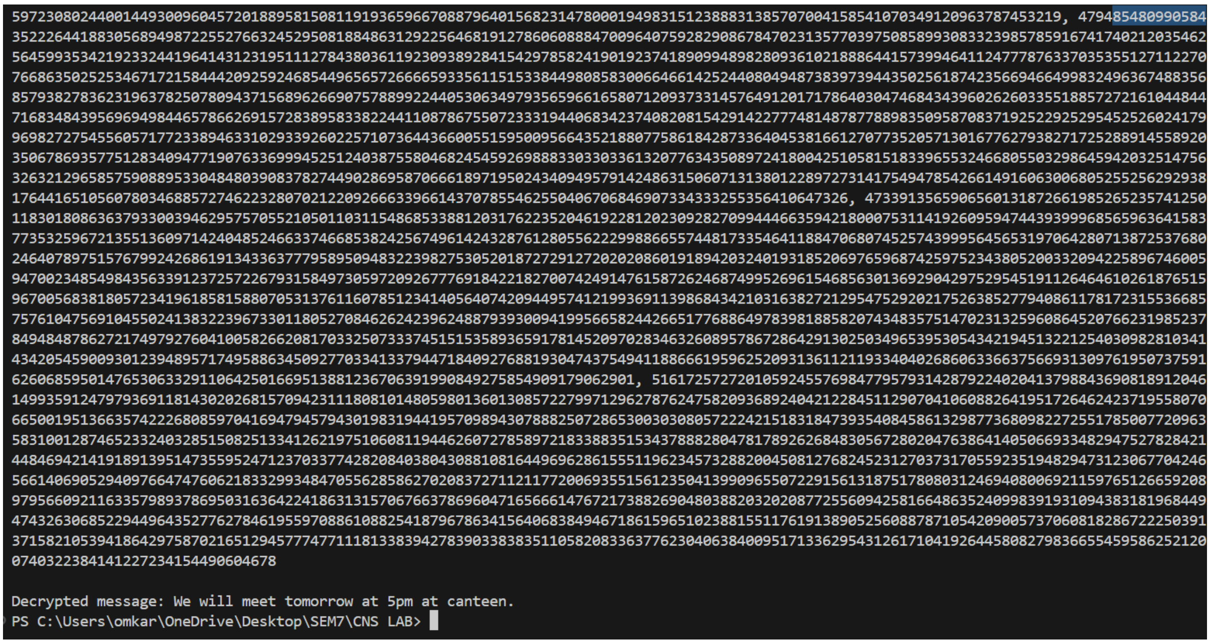
```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB> python -u "c:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 7\rsa.py"
RSA Encryption/Decryption

p:  2511117729564267621843960264874393596030286648632073242384378954074126226790456611323407936333102827475441115880483676119282670056385534
49508293899189675190664113106348815733971244898305817085881930571565768148790725127521598415315536650159923843252962720429603696909393967931
82740223790583833901426152225272387743402175529808209442493897313608011228766596393336833013086745666809426567016410725032420285475577283156
94601239398759675855387290670728750627391069088924306226324681785967213657975793239317394228145389811175650737914239445446317048917749199509
8555352050405543646061695665342056892041980027340950514383609

q:  2531735213582544559249699053928295832506387498914482824806768075268334509376209761735031930885328995758288240314710136138914819185423303
84634070244372527658045461123658897560481656702614562308432905932466428645344614364342115672302552308374693180075014083181114847624868848921
17873749942818699422921063115770487311361218671393543725928027649310151387610837112584180263217569126811649413004814968725886077300689736292
42716426998844626563727288166717348646332511705343985269214512606297242160211217840682031097047699578298911667121104107641120041926681017382
7145082830900643563423346777537593059409446912528234008068013
```

```
Public key: (6012287810814221757315615600137784881742505760087921859159472945410026388869704090401372891100839835257885947705872899151079378
71248242757057290589844486350464444892025428885176900548460871757392316765180732309977029222069579264363402205869658120068514605073710240647
31358680235567587691524913861460415060768461818342272622165919462473913698789185627590939954157260761341737696569537408678117475295144162546
36869357619219087125550147643774983641380261431687004528438737437295805390807273054594717053776640531578609411949822160723264394061180789097
83188162884056343854837522513714486246799111668142252306183778021706753836091615449940892379773724915968905087207682833809603813385928588034
88777978035084919601841834296064388985818323650228584058758580565990525210147216741441638968848609297680521537486309768818246293443213655783
93673645185566644075015173309112612011131801949916184810265388951850118071880613436413672742996701106179200642031285620851291856901422963113
24601489454327013855802155533643781693544018675959486704506758984323696706864025453504391128210451072049601831637178016029937658913906379858
251280454928265079795602303404393864915231091244128220813751723362131799444119719455752133290852828942388306095385292998560717, 635748518138
93054557712434592551457395067609585703954544273806003200072523381694320398727471203299885691392216041949491028683028565280125631565389567534
75225297307942851055369089205013604960108034273276571342034490904451075538253860914533972839465658208707926746074318581961101098437212425723
60686802693067960593117965147386782337714050224819828044898789135709213111167196315723220752209947345512933376471619106465087325603000551604
36106305223285863222573776568016163872221142633737050773249166246351651839992747639755738217848366379228989525068364567659434195448332544970
55387122798868724765691123678216421330624616980629340075635476910484184278451903779907835940853340121247215920940451046601093746458252489193
65265543170031392450952125372853896885329302279389468046219926415694706838905143493764338496710771188298655980167047036779744140005677866525
22673746965140243354133950178156290275087101749741765499552190962675518244980054494192905544582654355987308652452963834180271619430440772823
47884856474883580798204436876752206804187788605494756333564244175839618079929477301699987267764154296994594270383032512146153549447806500401
32149200255400455830492544122459305408415381873076213009146841667996350525822348807463498909644398917)
```

```
Private key: (709535279892813591340587480704620345587503567552261057045491508294867284308870913984181648226008556321125367490066959034254729
024023031671802793544468054049023655894963211915249618569604959472936690481888720686077048143733645225374465748750496492924023852357654315O5
04462862250444262628604675160654143842505875364599258576056708797642724824179602383071631253805868512624464931296564247946106161308289261615
48377711854777065999861255267294411385512402593258481038455084453654072266698990178815451300699289878428987823554264242473505138976711926397
44250615406660557165592942282632685784403462915954786868386289763972353179564164931546043230753791916410270067794330620051520048713776639895
576582227030183111804697129389440365153356095104808349468675312596201496169240357180705013720785684381849898796546507083462011865055457335 74
829538146288487880887694856313535057085310394650237925939432474716519975468731774326900181757296651689790269517746758882913166997760345562 13
116110449501551206696520190040374131636290269758796633937210021693469584889642488546694688107493172460138763768538450227424745563790799459 92
444862903732388758906615206328266542184148059936536947023539974409874020362989055462477920583443546803538118988213729053925541, 635748518138
930545577124345925514573950676095857039545442738060032000725233816943203987274712032998856913922160419494910286830285652801256315653895675 34
752252973079428510553690892050136049601080342732765713420344909044510755382538609145339728394656582087079267460743185819611010984372124257 23
606868026930679605931179651473867823377140502248198280448987891357092131111671963157232207522099473455129333764716191064650873256030005516 04
361063052232858632225737765680161638722211426337370507732491662463516518399927476397557382178483663792289895250683645676594341954483325449 70
553871227988687247656911236782164213306246169806293400756354769104841842784519037799078359408533401212472159209404510466010937464582524891 93
652655431700313924509521253728538968853293022793894680462199264156947068389051434937643384967107711882986559801670470367797441400056778665 25
226737469651402433541339501781562902750871017497417654995521909626755182449800544941929055445826543559873086524529638341802716194304407728 23
478848564748835807982044368767522068041877886054947563335642441758396180799294773016999872677641542969945942703830325121461535494478065004 01
321492002554004558304925441224593054084153818730762130091468416679963505258223488074634989096443989 17)
```

```
Enter a message to encrypt: We will meet tomorrow at 5pm at canteen.
```

```
Encrypted message: [2630432755935098166213940985561694353413640629727034823743211950503747259661865534182331319709636841466880332894379198 38
520303645125560055798361688880969468156868669390949258599713265080384919231318964995020148636890893494290647434291664106115712544396411606 58
040324068959366235787556628470434723248189466036370235462471044995465131498624282403408931506724013000538282059835452956292975710026149024 20
972382264060948131179281464714985533579382001804640516777858950103731428914847642453565104949433423385606669974399658373872613787578457483 07
184943581909718154383703731790525826138946566321026776100742805526792528006689973445421700114560907185134939117836264135093207541358431030 77
804863712571634755865481241948513780517135782125938537358128323416773481064115784302958305273648694961849842166943417814686411371859362427 88
411300771307694699330841351078100328009862518870398282868485918181322027152413108089630610065037851231961475686279332155105149892722519303 60
319343428872325006078624064409974822647447069381381674531740661355169875603785560912389831272988175824870492176910608614317752631539799330 98
879336463614085077811906345625048545739890195834716001990242620854637260450028239374066639011761919492881639491597249726590279479340 4, 51617
257272010592455769847795793142879224020413798843690818912046149935912479793691181430202681570942311180810148059801360130857227997129627876 24
758209368924042122845112907041060882641951726462423719558070665001951366357422268085970416947945794301983194419570989430788825072865300303 08
057222421518318473935408458613298773680982272551785007720963583100128746523324032851508251334126219751060811944626072785897218338835153437 88
828047817892626848305672802047638641405066933482947527828421448469421419189139514735595247123703377428208403804308810816449696286155511962 34
573288200450812768245231270373170559235194829473123067704246566140690529409766474760621833299348470556285862702083727112117720069355156123 50
413990965507229156131875178080312469408006921159765126659208979566092116335798937869503163642241863131570676637869604716566614767217388269 04
803882032020877255609425816648635240998391931094383181968449474326306852294496435277627846195597088610882541879678634156406838494671861596 51
023881551176191389052560887871054209005737060818286722250391371582105394186429758702165129457774771118133839427839033838351105820833637762 30
406384009517133629543126171041926445808279836655459586252120074032238414122723415449060467825823351880628698, 479485480990584352226441883056
894987225527663245295081884863129225646819127860608884700964075928290867847023135770397508589930833239857859167417402120354625645993534219 23
324419641431231951112784380361192309389284154297858241901923741890994898280936102188864415739946411247778763370353551271122707668635025253 46
717215844420925924685449656572666659335611515338449808583006646614252440804948738397394435025618742356694664998324963674883568579382783623 19
637825078094371568962669075788992244053063497935659661658071209373314576491201717864030474684343960262603355188572721610448447168348439569 69
498446578662691572838958338224411087867550723331944068342374082081542914227774814878778898350958708371925229252954525260241799698272754556 05
717723389463310293392602257107364436600551595009566435218807758618428733640453816612707735205713016776279382717252889145589203506786935775 12
834094771907633699945251240387558046824545926988833033033613207763435089724180042510581518339655324668055032986459420325147563263212965857 59
088953304848039083782744902869587066618971950243409495791424863150607131380122897273141754947854266149160630068052552562929381764416510560 78
034688572746223280702122092666339661437078554625504067068469073343332553564106473 26, 541692791770571534229187989738568685947632840219694159 7
803333777409166054008273848322414233454184341397808697005936956148401861305983748395714511421553124173876086901866692029707243041421468789 71
153144254853708940612909009340558422684968614618430924528441434315264053448176059479558280229932776802099184828453103812885194126221344503 22
508486973280943038915682708297040672831618535798953674849416705650564959563716389435644602470681904778277485094857364009049728191094566815 64
```


Decrypted message: We will meet tomorrow at 5pm at canteen.
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB>

Practical Applications of RSA

- **Secure Communication**: Encrypting emails and messages.
- **Digital Signatures:** Verifying the authenticity of a message or document.
- **Key Exchange**: Securely exchanging keys for symmetric encryption algorithms.

RSA is widely used in various security protocols, including SSL/TLS for secure internet communications.

RSA ensures security through the difficulty of factoring large numbers. It is commonly used for securing sensitive data, digital signatures, and in SSL/TLS protocols.

Final Year B.Tech. (CSE) – VII [2024-25]

6CS451: Cryptography and Network Security Lab (C&NS Lab)

Date: 26/08/2024

Assignment 8

PRN: 21510042 **Name:** Omkar Rajesh Auti

---

1. **Implement the Diffie-Hellman Key Exchange algorithm for a given problem**

Ans:

The Diffie-Hellman Key Exchange is a cryptographic algorithm that allows two parties to securely share a secret key over a public channel. This shared key can then be used for encrypted communication. The algorithm allows two parties to generate a shared secret key that can be used for subsequent encryption and decryption, even if the exchange itself is observed by an eavesdropper.

<u>How Diffie-Hellman Works:</u>

1. **Public Parameters:**
    - Both parties agree on a large prime number p and a base g (a primitive root modulo p).
2. **Key Exchange Process:**
    - **Party A** selects a private key 'a' and computes A = g^ a mod p, then sends A to Party B.
    - **Party B** selects a private key 'b' and computes B = g^ b mod p, then sends B to Party A.
3. **Shared Secret:**
    - **Party A** computes the shared secret as S = B^ a mod p.
    - **Party B** computes the shared secret as S = A^ b mod p.

Since both calculations result in the same value, S becomes the shared secret key, even though an eavesdropper only knows p, g, A, and B.

The Diffie-Hellman algorithm securely establishes a shared secret key without transmitting it directly, making it fundamental for secure communications in protocols like SSL/TLS.

To implement the Diffie-Hellman Key Exchange algorithm for client-server communication across two different machines, we will create two Python programs: one for the client and one for the server. The server will generate its public key and share it with the client, and vice versa. Both will then calculate the shared secret key independently.

Python Code:

Server-side program:

```
import socket
import random


def generate_private_key(p):
    """Generate a private key."""
    return random.randint(2, p-2)


def calculate_public_key(g, private_key, p):
    """Calculate the public key."""
    return pow(g, private_key, p)


def calculate_shared_secret(public_key, private_key, p):
    """Calculate the shared secret."""
    return pow(public_key, private_key, p)
```

```
def start_server(host='localhost', port=5000):
    # p = 23
    # g = 5
    p = 104729  # Shared prime number --> 104729 (which is
the 10000th prime number)
    g = 2   # Shared base --> Primitive Root g: 2

    # Generate server's private and public keys
    private_key = generate_private_key(p)
    public_key = calculate_public_key(g, private_key, p)

    # Create server socket
    server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
    server_socket.bind((host, port))
    server_socket.listen(1)
    print(f"Server started. Listening on {host}:{port}")

    conn, addr = server_socket.accept()
    print(f"\nConnected by {addr}")

    # Send the server's public key to the client
    conn.sendall(str(public_key).encode())

    # Receive the client's public key
    client_public_key = int(conn.recv(1024).decode())
    print(f"\nReceived Client's Public Key:
{client_public_key}")

    # Calculate the shared secret
    shared_secret =
calculate_shared_secret(client_public_key, private_key, p)
    print(f"\nShared Secret (Server): {shared_secret}")

    conn.close()
    server_socket.close()


if __name__ == "__main__":
```

```
    start_server()
```

Client-side program:

```
import socket
import random


def generate_private_key(p):
    """Generate a private key."""
    return random.randint(2, p-2)


def calculate_public_key(g, private_key, p):
    """Calculate the public key."""
    return pow(g, private_key, p)


def calculate_shared_secret(public_key, private_key, p):
    """Calculate the shared secret."""
    return pow(public_key, private_key, p)


def start_client(server_host='localhost', server_port=5000):
    # p = 23
    # g = 5
    p = 104729  # Shared prime number --> 104729 (which is
the 10000th prime number)
    g = 2   # Shared base --> Primitive Root g: 2

    # Generate client's private and public keys
    private_key = generate_private_key(p)
    public_key = calculate_public_key(g, private_key, p)

    # Create client socket
    client_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
    client_socket.connect((server_host, server_port))
```

```python
    # Receive the server's public key
    server_public_key =
int(client_socket.recv(1024).decode())
    print(f"\nReceived Server's Public Key:
{server_public_key}")

    # Send the client's public key to the server
    client_socket.sendall(str(public_key).encode())

    # Calculate the shared secret
    shared_secret =
calculate_shared_secret(server_public_key, private_key, p)
    print(f"\nShared Secret (Client): {shared_secret}")

    client_socket.close()


if __name__ == "__main__":
    start_client()
```

<u>Output:</u>

<u>Server output-</u>

```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 8> python server_diffie_hellman.py
Server started. Listening on localhost:5000

Connected by ('127.0.0.1', 52927)

Received Client's Public Key: 45195

Shared Secret (Server): 91540
```

<u>Client output-</u>

```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 8> python client_diffie_hellman.py

Received Server's Public Key: 45505

Shared Secret (Client): 91540
```

<u>Practical Applications of Diffie-Hellman:</u>

- **Secure Communication**: Establishing a shared secret for symmetric encryption over an insecure channel.
- **VPNs**: Secure key exchange for Virtual Private Networks.
- **TLS/SSL**: Part of the key exchange process in securing internet communications.

The Diffie-Hellman algorithm forms the basis of many modern cryptographic protocols and is crucial for secure communication in distributed systems.

Final Year B.Tech. (CSE) – VII [2024-25]

6CS451: Cryptography and Network Security Lab (C&NS Lab)

Date: 01/10/2024

Assignment 9

PRN: 21510042 **Name:** Omkar Rajesh Auti

---

9. Calculate the message digest of a text using the SHA-1 algorithm

Ans:

SHA-1 Algorithm:

SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function that produces a 160-bit hash value (20 bytes), often referred to as a **message digest**. It takes an input message of any size and outputs a fixed-size hash, which is commonly represented as a 40-character hexadecimal number. It was developed by the National Security Agency (NSA) and published by NIST in 1993.

Message Digest of a Text:

A **message digest** is a fixed-size numerical representation of the contents of a message. For SHA-1, this digest is 160 bits long, and any change in the input message, even a single bit, will result in a drastically different digest (this is known as the **avalanche effect**). The message digest ensures data integrity by allowing anyone to verify that the message has not been altered.

To calculate the message digest of a text using the SHA-1 algorithm in Python, you can use the hashlib library, which provides easy access to various hash algorithms, including SHA-1.

1. **hashlib library**:
   - The hashlib library provides various cryptographic hashing algorithms including SHA-1, SHA-256, MD5, etc.

2. **SHA-1 Hash Object**:
   - hashlib.sha1() creates a new SHA-1 hash object.
3. **Updating the Hash**:
   - The update() method takes the input text (which is first encoded into bytes) and updates the hash object with that data.
4. **Getting the Digest**:
   - The hexdigest() method returns the hash value as a hexadecimal string.

<u>Python Code for SHA-1 Message Digest Calculation using hashlib library:</u>

```
import hashlib


# Function to hash a message using SHA-1
def sha1_encrypt(message):
    sha1_hash = hashlib.sha1()
    sha1_hash.update(message.encode('utf-8'))  # Convert the message to bytes
    return sha1_hash.hexdigest()


# Function to verify the hash (like a decryption process)
def verify_hash(original_message, provided_hash):
    original_hash = sha1_encrypt(original_message)
    return original_hash == provided_hash


# Menu-driven system
def menu():
```

```
while True:
    print("\n===== SHA-1 Hashing System =====")
    print("1. Encrypt a message using SHA-1")
    print("2. Verify a message against a given hash")
    print("3. Exit")

    choice = input("Enter your choice (1/2/3): ")

    if choice == '1':
        message = input("Enter the message to hash: ")
        hashed_message = sha1_encrypt(message)
        print(f"\nSHA-1 Hash: {hashed_message}")

    elif choice == '2':
        original_message = input("Enter the original message: ")
        provided_hash = input("Enter the hash to verify against: ")

        if verify_hash(original_message, provided_hash):
            print("\nVerification successful! The message matches the provided hash.")
        else:
            print("\nVerification failed! The message does not match the provided hash.")

    elif choice == '3':
        print("Exiting the program...")
```

```
            break

        else:
            print("Invalid choice. Please choose a valid option.")

if __name__ == "__main__":
    menu()
```

Output:

```
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB> python -u "c:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 9\sha1_using_hashlib.py"
Enter the text to calculate SHA-1 hash: Omkar Auti
SHA-1 Digest: b08b45e760628a5ce25d0e25a3d2c6489cc72ca0
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB> python -u "c:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB\Assignment 9\sha1_without_using_hashlib.py"
Enter the text to calculate SHA-1 hash: Omkar Auti
SHA-1 Digest: b08b45e760628a5ce25d0e25a3d2c6489cc72ca0
PS C:\Users\omkar\OneDrive\Desktop\SEM7\CNS LAB>
```

Advantages of SHA-1:

- **Speed and Efficiency**: SHA-1 was designed to be computationally efficient and can process large amounts of data quickly.
- **Widespread Use**: It has been widely adopted and used for various cryptographic applications, including digital signatures, certificates, and integrity checks.
- **Fixed-Length Output**: Regardless of the input size, the output is always 160 bits, making it convenient to use in various security protocols.

Disadvantages of SHA-1:

- **Weakness to Collisions**: SHA-1 is vulnerable to **collision attacks**, where two different inputs produce the same hash output. This reduces its effectiveness in ensuring data integrity and security.
- **Security Deprecation**: Due to these vulnerabilities, SHA-1 is no longer considered secure for cryptographic purposes. Most modern systems and protocols, including browsers and SSL certificates, have moved to stronger hash functions like SHA-256 or SHA-3.

Importance of SHA-1:

- **Legacy Systems**: Despite its vulnerabilities, SHA-1 was used for many years in security applications such as digital signatures and certificates.
- **Data Integrity**: SHA-1 can still be used to check the integrity of data, ensuring that files have not been altered during transmission.

Security Risks and Vulnerabilities of SHA-1:

- **Collision Attacks**: The primary vulnerability is the possibility of collision attacks. This means that an attacker could potentially create two different messages with the same hash, compromising the authenticity of the data.
- **Birthday Attack**: A specific type of attack known as a **birthday attack** makes it easier to find collisions in SHA-1 due to its 160-bit length, reducing the security level.
- **Deprecation in Modern Systems**: Due to these weaknesses, SHA-1 has been deprecated in most cryptographic protocols like TLS (Transport Layer Security) and digital certificates, where stronger algorithms like SHA-256 are preferred.

While SHA-1 played a significant role in the development of cryptographic standards, its vulnerabilities, especially to collision attacks, have made it unsuitable for modern security applications. Understanding SHA-1's purpose and limitations is important, especially when dealing with legacy systems or understanding the evolution of cryptographic hash functions.

Practical Applications of SHA-1:

1. **Digital Signatures**: SHA-1 was commonly used in creating digital signatures to ensure the authenticity and integrity of documents. It would generate a hash of the message, which is then signed by a private key.
2. **File Integrity Verification**: SHA-1 was used to generate checksums for files to verify that files were not altered during transfer or storage. The recipient could compare the hash of the received file with the original hash to ensure integrity.
3. **Version Control Systems**: In systems like Git, SHA-1 hashes were used to identify commits, ensuring the integrity and tracking of changes in code repositories.
4. **SSL Certificates**: Until 2017, SHA-1 was used in SSL/TLS certificates for secure web communications. The hash was part of the process to ensure a website's identity and secure data transmission.
5. **Password Hashing**: SHA-1 was once used for hashing passwords in databases, providing a layer of security by storing a hashed version of the password instead of the plaintext.

Despite these applications, most systems have transitioned to more secure alternatives due to SHA-1's vulnerabilities.

SHA 512:

SHA-512 Algorithm:

SHA-512 (Secure Hash Algorithm 512) is a cryptographic hash function that generates a 512-bit hash value (64 bytes), typically represented as a 128-character hexadecimal string. SHA-512 is part of the SHA-2 family, which provides greater security than SHA-1 due to its larger output and resistance to collision attacks.

Message Digest of a Text: In SHA-512, the message digest is a 512-bit hash value representing the contents of the message. Any change in the input message, even a single bit, will produce a significantly different hash due to the avalanche effect. This property ensures data integrity and authenticity, making it useful in various security applications.

Overview of SHA-512 Algorithm:

1. **Padding the Message**: The message is padded to ensure its length is a multiple of 1024 bits.
2. **Initialize Hash Values**: There are eight constants (H0 to H7) initialized to specific 64-bit values based on the fractional parts of the square roots of the first eight prime numbers.
3. **Processing the Message in Blocks**: The message is processed in chunks of 1024 bits, updating the hash after each chunk.
4. **Final Output**: After processing all blocks, the hash digest is formed by concatenating the values of H0 through H7.

Python Code for SHA-512 Message Digest Calculation using hashlib Library:

```
import hashlib


# Function to hash a message using SHA-512
def sha512_encrypt(message):
    sha512_hash = hashlib.sha512()
    sha512_hash.update(message.encode('utf-8'))  # Convert the message to bytes
    return sha512_hash.hexdigest()


# Function to verify the hash (like a decryption process)
```

```
def verify_hash(original_message, provided_hash):
    original_hash = sha512_encrypt(original_message)
    return original_hash == provided_hash


# Menu-driven system
def menu():
    while True:
        print("\n===== SHA-512 Hashing System =====")
        print("1. Encrypt a message using SHA-512")
        print("2. Verify a message against a given hash")
        print("3. Exit")
        
        choice = input("Enter your choice (1/2/3): ")

        if choice == '1':
            message = input("Enter the message to hash: ")
            hashed_message = sha512_encrypt(message)
            print(f"\nSHA-512 Hash: {hashed_message}")

        elif choice == '2':
            original_message = input("Enter the original message: ")
            provided_hash = input("Enter the hash to verify against: ")
            
            if verify_hash(original_message, provided_hash):
```

```
            print("\nVerification successful! The message matches the provided hash.")
        else:
            print("\nVerification failed! The message does not match the provided hash.")

    elif choice == '3':
        print("Exiting the program...")
        break

    else:
        print("Invalid choice. Please choose a valid option.")

if __name__ == "__main__":
    menu()
```

Advantages of SHA-512:

- **High Security**: Due to its larger output size, SHA-512 provides strong resistance against collision and preimage attacks.
- **Wide Adoption**: SHA-512 is widely used in security protocols such as SSL/TLS and digital certificates.
- **Fixed-Length Output**: The output is always 512 bits, regardless of the input size, which is useful for secure storage and transmission.

Disadvantages of SHA-512:

- **Computationally Intensive**: SHA-512 requires more processing power and time compared to shorter algorithms like SHA-256 or SHA-1.

- **Not Ideal for Lightweight Applications**: Due to its computational requirements, SHA-512 may not be suitable for lightweight devices or applications with limited processing resources.

Practical Applications of SHA-512:

1. **Digital Certificates**: SHA-512 is commonly used in digital certificates, providing a higher level of security for verifying the authenticity of websites and applications.
2. **Blockchain and Cryptocurrencies**: SHA-512 is often used in the cryptographic aspects of blockchain technology to secure transactions and protect against tampering.
3. **Data Integrity Verification**: SHA-512 is used to verify file integrity, ensuring that files or data have not been tampered with during transmission.
4. **Password Hashing**: Although other algorithms are also used, SHA-512 is sometimes used for hashing passwords in secure applications.

Security Risks and Vulnerabilities of SHA-512:

- **Quantum Computing**: As with other SHA-2 algorithms, SHA-512 may eventually become vulnerable to quantum computing attacks, though this is not an immediate threat.
- **Length Extension Attacks**: SHA-512 is susceptible to length extension attacks, making it unsuitable for certain applications without additional protective measures.

Importance of SHA-512: SHA-512 plays a vital role in modern cryptographic applications, offering high security and reliability for data integrity and authentication. It is considered one of the most secure hash functions in the SHA family, particularly suitable for applications requiring robust protection against tampering.

Final Year B.Tech. (CSE) – VII [2024-25]

6CS451: Cryptography and Network Security Lab (C&NS Lab)

Date: 22/10/2024

Assignment 9

PRN: 21510042 **Name:** Omkar Rajesh Auti

Digital Signature System Documentation

Overview

This Python program implements a digital signature system using RSA encryption and SHA-256 hashing. It provides the ability to generate RSA key pairs (private and public), sign a message by hashing it and creating a signature, verify the signature against the hash of the message, and save the RSA keys to files. The program is menu-driven and operates through a command-line interface.

Key Concepts:

- **RSA Encryption**: An asymmetric encryption method that uses a public-private key pair for secure operations.
- **SHA-256 Hashing**: A cryptographic hash function producing a 256-bit hash value to ensure message integrity.
- **Digital Signature**: A method of verifying the authenticity of a message using cryptographic signatures.

Dependencies

The program uses the following libraries:

- cryptography.hazmat.backends.default_backend
- cryptography.hazmat.primitives.asymmetric.rsa
- cryptography.hazmat.primitives.asymmetric.padding
- cryptography.hazmat.primitives.hashes
- cryptography.hazmat.primitives.serialization
- cryptography.exceptions.InvalidSignature
- hashlib

To install the required libraries, use:

bash

Copy code

pip install cryptography

Functions

1. generate_keys()

Generates a new RSA private and public key pair with a key size of 2048 bits.

- **Returns**:
  - private_key: RSA private key object.
  - public_key: RSA public key object.

2. sign_message(private_key, message)

Hashes the message using SHA-256 and signs the hash using the private key.

- **Parameters**:
  - private_key: RSA private key used to sign the message.
  - message: The message to be signed.
- **Returns**:
  - signature: The digital signature.
  - message_hash: The SHA-256 hash of the message in hexadecimal format.

Note: The message hash is printed to the console.

3. verify_signature(public_key, message_hash, signature)

Verifies the signature by checking it against the provided message hash using the public key.

- **Parameters**:
  - public_key: RSA public key used to verify the signature.
  - message_hash: The SHA-256 hash of the message (hexadecimal format).
  - signature: The signature to verify.
- **Returns**:
  - True: If the signature is valid.
  - False: If the signature is invalid.

4. save_keys_to_file(private_key, public_key)

Saves the RSA private and public keys to files in PEM format (private_key.pem and public_key.pem).

- **Parameters**:
  - private_key: The RSA private key.
  - public_key: The RSA public key.

The private key is saved in an unencrypted format for simplicity.

Menu Interface

The program provides a menu with the following options:

1. Generate RSA Keys

Generates a new RSA key pair (private and public).

2. Sign a Message

Prompts the user to input a message, hashes it, and signs the hash using the private key. The message hash is displayed for user reference.

3. Verify Signature

Prompts the user to input the hash of the message and verifies the signature using the public key.

4. Save Keys to Files

Saves the generated private and public keys to files (private_key.pem and public_key.pem).

5. Exit

Exits the program.

Usage

1. **Generating Keys**: After starting the program, select the option to generate RSA keys.
2. **Signing a Message**: Sign a message by selecting the appropriate menu option and entering the message.
3. **Verifying a Signature**: Verify the signature by inputting the hash of the message.
4. **Saving Keys**: Save the generated RSA keys to files for later use.

Security Considerations

- Ensure that private keys are kept secure and not shared.
- Private keys can be stored encrypted for additional security, though this implementation stores them unencrypted for simplicity.

Code:

```
from cryptography.hazmat.backends import default_backend

from cryptography.hazmat.primitives.asymmetric import rsa,
padding

from cryptography.hazmat.primitives import hashes, serialization

from cryptography.exceptions import InvalidSignature

import hashlib
```

```
# Function to generate RSA private and public keys
def generate_keys():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )
    public_key = private_key.public_key()

    return private_key, public_key

# Function to sign a message and print its hash
def sign_message(private_key, message):
    # Hash the message using SHA-256
    message_hash =
hashlib.sha256(message.encode()).hexdigest()

    # Sign the hashed message using RSA private key
    signature = private_key.sign(
        bytes.fromhex(message_hash),  # Convert the hex hash to
bytes
        padding.PSS(
```

```
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )

    print(f"Hash of the message: {message_hash}")  # Print the
message hash
    return signature, message_hash

# Function to verify the signature using the provided hash
def verify_signature(public_key, message_hash, signature):
    try:
        # Verify the signature using RSA public key
        public_key.verify(
            signature,
            bytes.fromhex(message_hash),  # Convert the hex hash
back to bytes
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
```

```
        )
        return True
    except InvalidSignature:
        return False


# Function to save keys to files
def save_keys_to_file(private_key, public_key):
    # Save private key
    with open("private_key.pem", "wb") as private_file:
        private_file.write(
            private_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.PKCS8,
                encryption_algorithm=serialization.NoEncryption()
            )
        )
    # Save public key
    with open("public_key.pem", "wb") as public_file:
        public_file.write(
            public_key.public_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PublicFormat.SubjectPublicKeyInfo
```

```
        )
    )
    print("Keys saved to files: private_key.pem and
public_key.pem")

# Menu for the digital signature system
def menu():
    private_key, public_key = None, None
    signature = None
    message_hash = None

    while True:
        print("\n===== Digital Signature System =====")
        print("1. Generate RSA Keys")
        print("2. Sign a Message")
        print("3. Verify Signature")
        print("4. Save Keys to Files")
        print("5. Exit")

        choice = input("Enter your choice (1/2/3/4/5): ")

        if choice == '1':
```

```
        # Generate RSA private and public keys
        private_key, public_key = generate_keys()
        print("\nRSA Keys Generated!")

    elif choice == '2':
        # Sign a message
        if private_key is None:
            print("You need to generate RSA keys first.")
        else:
            message = input("Enter the message to sign: ")
            signature, message_hash = sign_message(private_key,
message)
            print("\nMessage signed successfully!")

    elif choice == '3':
        # Verify the signature
        if public_key is None or message_hash is None or
signature is None:
            print("You need to sign a message first.")
        else:
            input_hash = input("Enter the hash of the message to
verify: ")
```

```
            verification_result = verify_signature(public_key,
input_hash, signature)
            if verification_result:
                print("\nSignature verified successfully! The message
is authentic.")
            else:
                print("\nSignature verification failed! The message is
not authentic.")

        elif choice == '4':
            # Save RSA keys to files
            if private_key is None or public_key is None:
                print("You need to generate RSA keys first.")
            else:
                save_keys_to_file(private_key, public_key)

        elif choice == '5':
            print("Exiting the program...")
            break

        else:
            print("Invalid choice. Please try again.")
```

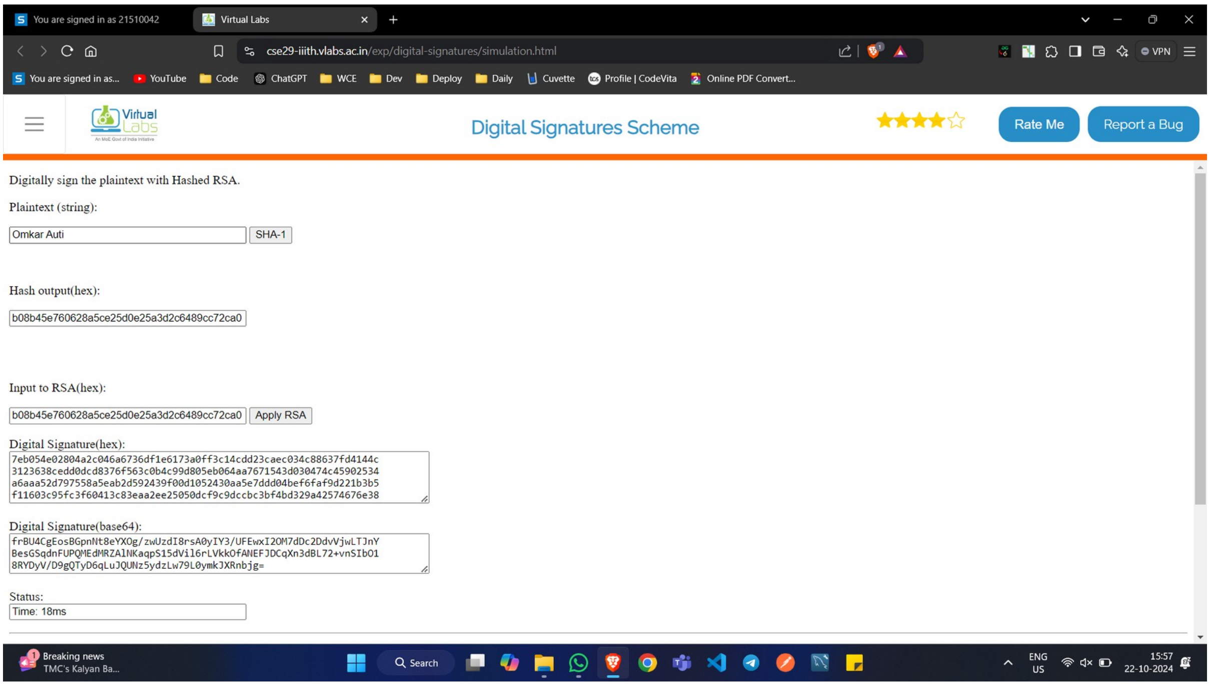
```
if __name__ == "__main__":
    menu()
```

Output:

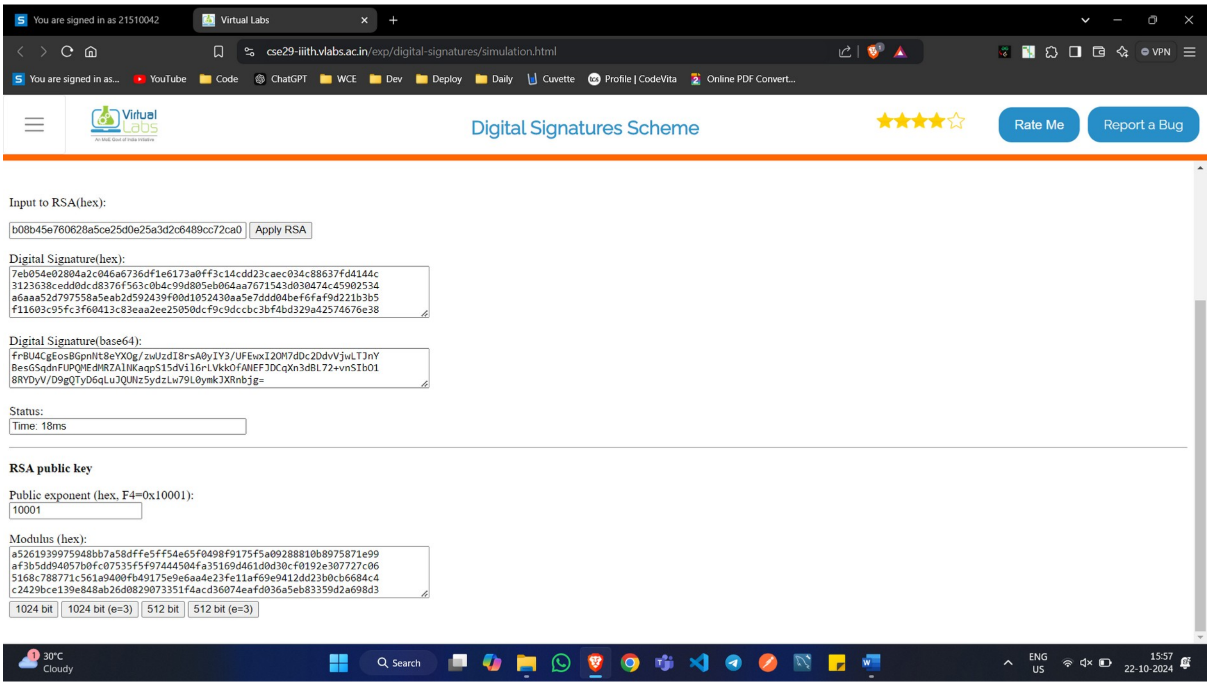
```
===== Digital Signature System =====
1. Generate RSA Keys
2. Sign a Message
3. Verify Signature
4. Save Keys to Files
5. Exit
Enter your choice (1/2/3/4/5): 1

RSA Keys Generated!

===== Digital Signature System =====
1. Generate RSA Keys
2. Sign a Message
3. Verify Signature
4. Save Keys to Files
5. Exit
Enter your choice (1/2/3/4/5): 2
Enter the message to sign: Omkar Auti
Hash of the message: d5eadc6ba2bc54d3df9a539bbf8ab494750a54a5b9af176b6bc3c69018665df5

Message signed successfully!
```

```
Message signed successfully!

===== Digital Signature System =====
1. Generate RSA Keys
2. Sign a Message
3. Verify Signature
4. Save Keys to Files
5. Exit
Enter your choice (1/2/3/4/5): 3
Enter the hash of the message to verify: d5eadc6ba2bc54d3df9a539bbf8ab494750a54a5b9af176b6bc3c69018665df5

Signature verified successfully! The message is authentic.

===== Digital Signature System =====
1. Generate RSA Keys
2. Sign a Message
3. Verify Signature
4. Save Keys to Files
5. Exit
Enter your choice (1/2/3/4/5):
```

VLAB



Virtual Labs
cse29-iiith.vlabs.ac.in/exp/digital-signatures/simulation.html
Digital Signatures Scheme
Rate Me
Report a Bug
Digitally sign the plaintext with Hashed RSA.
Plaintext (string):
Omkar Auti
SHA-1
Hash output(hex):
b08b45e760628a5ce25d0e25a3d2c6489cc72ca0
Input to RSA(hex):
b08b45e760628a5ce25d0e25a3d2c6489cc72ca0
Apply RSA
Digital Signature(hex):
7eb054e02804a2c046a6736df1e6173a0ff3c14cdd23caec034c88637fd4144c
3123638cedd0dcd8376f563c0b4c99d805eb064aa7671543d030474c45902534
a6aaa52d797558a5eab2d592439f00d1052430aa5e7ddd04bef6faf9d221b3b5
f11603c95fc3f60413c83eaa2ee25050dcf9c9dccbc3bf4bd329a42574676e38
Digital Signature(base64):
frBU4CgEosBGpnNt8eYXOg/zwUzdI8rsA0yIY3/UFEwxI2OM7dDc2DdvVjwLTJnY
BesGSqdnFUPQMEdMRZAlNKaqpS15dVil6rLVkkOfANEFJDCqXn3dBL72+vnSIbO1
8RYDyV/D9gQTyD6qLuJQUNz5ydzLw79L0ymkJXRnbjg=
Status:
Time: 18ms
15:57
22-10-2024



Virtual Labs
cse29-iiith.vlabs.ac.in/exp/digital-signatures/simulation.html
Digital Signatures Scheme
Rate Me
Report a Bug
Input to RSA(hex):
b08b45e760628a5ce25d0e25a3d2c6489cc72ca0
Apply RSA
Digital Signature(hex):
7eb054e02804a2c046a6736df1e6173a0ff3c14cdd23caec034c88637fd4144c
3123638cedd0dcd8376f563c0b4c99d805eb064aa7671543d030474c45902534
a6aaa52d797558a5eab2d592439f00d1052430aa5e7ddd04bef6faf9d221b3b5
f11603c95fc3f60413c83eaa2ee25050dcf9c9dccbc3bf4bd329a42574676e38
Digital Signature(base64):
frBU4CgEosBGpnNt8eYXOg/zwUzdI8rsA0yIY3/UFEwxI2OM7dDc2DdvVjwLTJnY
BesGSqdnFUPQMEdMRZAlNKaqpS15dVil6rLVkkOfANEFJDCqXn3dBL72+vnSIbO1
8RYDyV/D9gQTyD6qLuJQUNz5ydzLw79L0ymkJXRnbjg=
Status:
Time: 18ms
RSA public key
Public exponent (hex, F4=0x10001):
10001
Modulus (hex):
a5261939975948bb7a58dffe5ff54e65f0498f9175f5a09288810b8975871e99
af3b5dd94057b0fc07535f5f97444504fa35169d461d0d30cf0192e307727c06
5168c788771c561a9400fb49175e9e6aa4e23fe11af69e9412dd23b0cb6684c4
c2429bce139e848ab26d0829073351f4acd36074eafd036a5eb83359d2a698d3
1024 bit
1024 bit (e=3)
512 bit
512 bit (e=3)
15:57
22-10-2024

Final Year B.Tech. (CSE) – VII [2024-25]

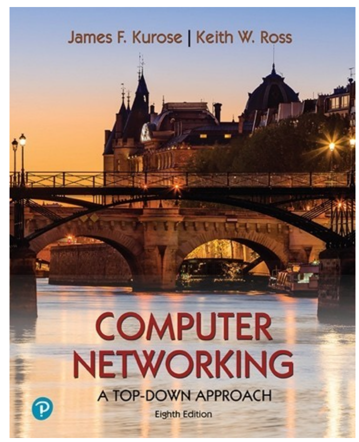
6CS451: Cryptography and Network Security Lab (C&NS Lab)

Date: 21/10/2024

Assignment10

PRN: 21510042 **Name:** Omkar Rajesh Auti

Wireshark Lab: SSL v8.0

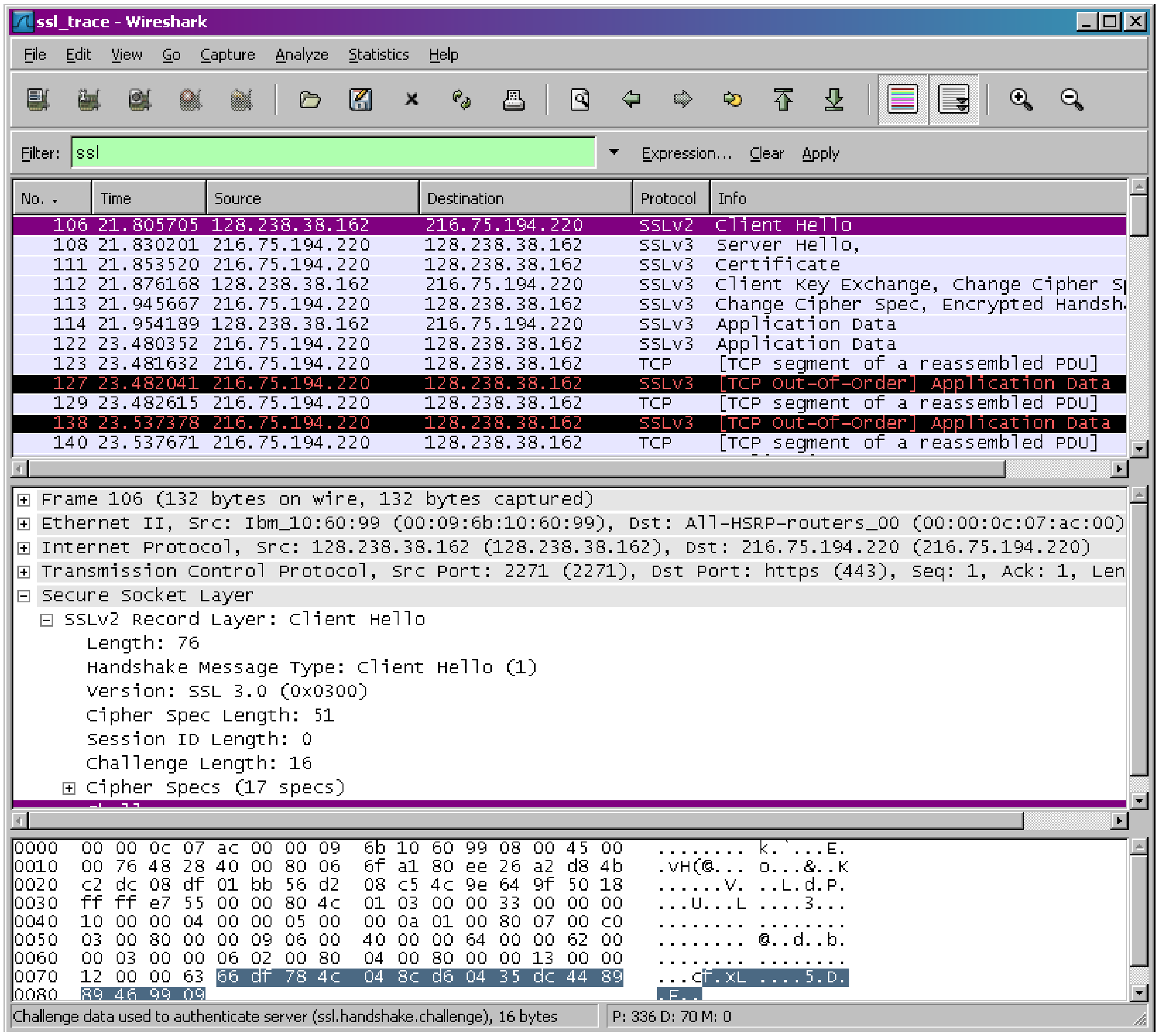


Supplement to *Computer Networking: A Top-Down Approach, 8th ed.,* J.F. Kurose and K.W. Ross

"Tell me and I forget. Show me and I remember. Involve me and I understand." Chinese proverb

---

In this lab, we will investigate the Secure Sockets Layer (SSL) protocol, focusing on the SSL records sent over a TCP connection. We will do so by analyzing a trace of the SSL records sent between your host and an e-commerce server. We will investigate the various SSL record types as well as the fields in the SSL messages.



1. Capturing packets in an SSL session

The first step is to capture the packets in an SSL session. To do this, you should go to your favorite e-commerce site and begin the process of purchasing an item (but terminating before making the actual purpose!). After capturing the packets with Wireshark, you should set the filter so that it displays only the Ethernet frames that contain SSL records sent from and received by your host. (An SSL record is the same thing as an SSL message.) You should obtain something like screenshot on the previous page.

If you have difficulty creating a trace, you should download the zip file http://gaia.cs.umass.edu/wireshark-labs/wireshark-traces.zip and extract the *ssl-ethereal- trace-1* packet trace.

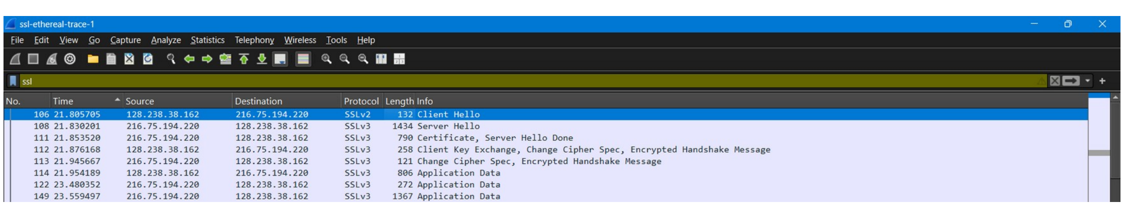
2. A look at the captured trace

Your Wireshark GUI should be displaying only the Ethernet frames that have SSL records. It is important to keep in mind that an Ethernet frame may contain one or more SSL records. (This is very different from HTTP, for which each frame contains either one complete HTTP message or a portion of a HTTP message.) Also, an SSL record may not

completely fit into an Ethernet frame, in which case multiple frames will be needed to carry the record.

Whenever possible, when answering a question below, you should hand in a printout of the packet(s) within the trace that you used to answer the question asked. Annotate the printout[2] to explain your answer. To print a packet, use *File->Print*, choose *Selected packet only*, choose *Packet summary line,* and select the minimum amount of packet detail that you need to answer the question

1. For each of the first 8 Ethernet frames, specify the source of the frame (client or server), determine the number of SSL records that are included in the frame, and list the SSL record types that are included in the frame. Draw a timing diagram between client and server, with one arrow for each SSL record.



first 8 Ethernet frames to determine:

1. Source (client or server)
2. Number of SSL records in the frame
3. SSL record types

Frame 1: (Frame 106)

- Source: Client (128.238.38.162)
- Number of SSL Records: 1
- SSL Record Type: Client Hello (SSLv2)

Frame 2: (Frame 108)

- Source: Server (216.75.194.220)
- Number of SSL Records: 1
- SSL Record Type: Server Hello (SSLv3)

Frame 3: (Frame 111)

- Source: Server (216.75.194.220)
- Number of SSL Records: 2
- SSL Record Types:
    1. Certificate (SSLv3)
    2. Server Hello Done (SSLv3)

Frame 4: (Frame 112)

- Source: Client (128.238.38.162)
- Number of SSL Records: 3
- SSL Record Types:
    1. Client Key Exchange (SSLv3)
    2. Change Cipher Spec (SSLv3)
    3. Encrypted Handshake Message (SSLv3)

Frame 5: (Frame 113)

- Source: Server (216.75.194.220)
- Number of SSL Records: 2
- SSL Record Types:
    1. Change Cipher Spec (SSLv3)
    2. Encrypted Handshake Message (SSLv3)

Frame 6: (Frame 114)

- Source: Client (128.238.38.162)
- Number of SSL Records: 1
- SSL Record Type: Application Data (SSLv3)

Frame 7: (Frame 122)

- Source: Server (216.75.194.220)
- Number of SSL Records: 1
- SSL Record Type: Application Data (SSLv3)

Frame 8: (Frame 149)

- Source: Server (216.75.194.220)
- Number of SSL Records: 1
- SSL Record Type: Application Data (SSLv3)

2. Each of the SSL records begins with the same three fields (with possibly different values). One of these fields is "content type" and has length of one byte. List all three fields and their lengths.

Each SSL record starts with the following three fields:

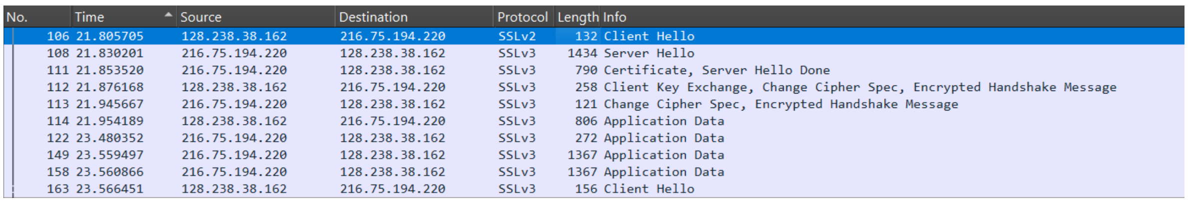
- **Content Type:** 1 byte
- **Version:** 2 bytes
- **Length:** 2 bytes

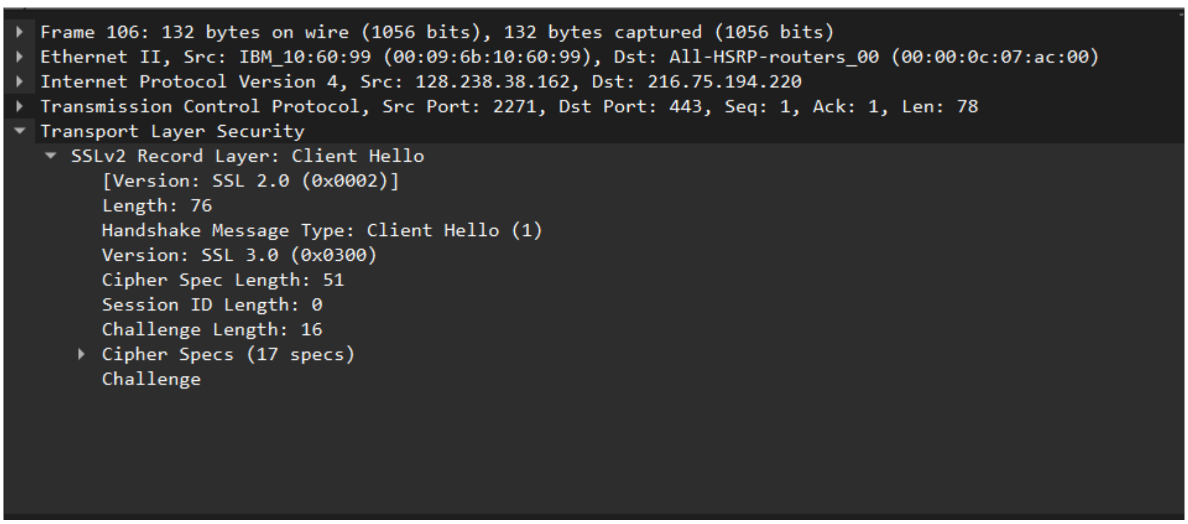
How to Find These Fields:

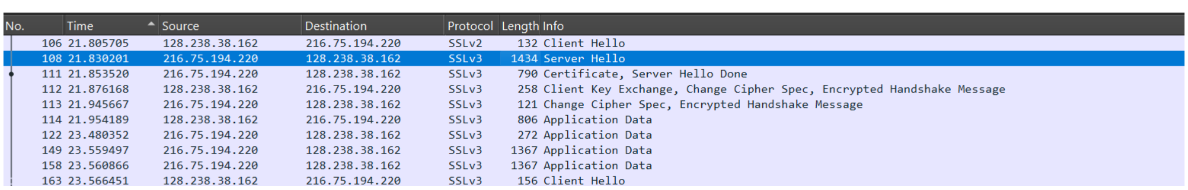
If you are using packet capture software like **Wireshark**, you can find these fields in the packet capture by:

1. **Open Wireshark** and load the captured SSL/TLS packet data (the one you listed).
2. **Select an SSL/TLS packet** from the list and expand the **"Secure Sockets Layer"** or **"Transport Layer Security"** section in the detailed packet view.

3. You will see the **Record Layer** header information, where these fields will be listed:
    - **Content Type:** Displays the type of SSL/TLS record (Handshake, Application Data, etc.)
    - **Version:** The protocol version (e.g., TLS 1.2)
    - **Length:** The size of the encrypted data.







```
Frame 108: 1434 bytes on wire (11472 bits), 1434 bytes captured (11472 bits)
Ethernet II, Src: Cisco_83:e4:54 (00:b0:8e:83:e4:54), Dst: IBM_10:60:99 (00:09:6b:10:60:99)
Internet Protocol Version 4, Src: 216.75.194.220, Dst: 128.238.38.162
Transmission Control Protocol, Src Port: 443, Dst Port: 2271, Seq: 1, Ack: 79, Len: 1380
Transport Layer Security
  SSLv3 Record Layer: Handshake Protocol: Server Hello
    Content Type: Handshake (22)
    Version: SSL 3.0 (0x0300)
    Length: 74
    Handshake Protocol: Server Hello
  TLS segment data (1301 bytes)
```

ClientHello Record:

3. Expand the ClientHello record. (If your trace contains multiple ClientHello records, expand the frame that contains the first one.) What is the value of the content type?

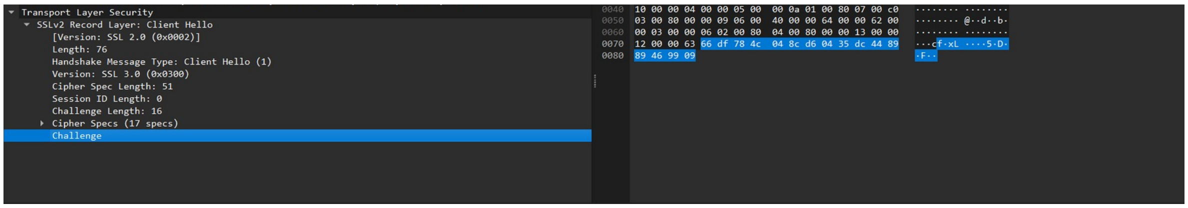
The **ClientHello** record in **Frame 106** is an SSLv2 message with a handshake message type of **Client Hello (1)**.

No.	Time	Source	Destination	Protocol	Length	Info
106	21.805705	128.238.38.162	216.75.194.220	SSLv2	132	Client Hello
108	21.830201	216.75.194.220	128.238.38.162	SSLv3	1434	Server Hello
111	21.853520	216.75.194.220	128.238.38.162	SSLv3	790	Certificate, Server Hello Done
112	21.876168	128.238.38.162	216.75.194.220	SSLv3	258	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
113	21.945667	216.75.194.220	128.238.38.162	SSLv3	121	Change Cipher Spec, Encrypted Handshake Message
114	21.954189	128.238.38.162	216.75.194.220	SSLv3	806	Application Data
122	23.480352	216.75.194.220	128.238.38.162	SSLv3	272	Application Data
149	23.559497	216.75.194.220	128.238.38.162	SSLv3	1367	Application Data
158	23.560866	216.75.194.220	128.238.38.162	SSLv3	1367	Application Data
163	23.566451	128.238.38.162	216.75.194.220	SSLv3	156	Client Hello

```
Frame 106: 132 bytes on wire (1056 bits), 132 bytes captured (1056 bits)
Ethernet II, Src: IBM_10:60:99 (00:09:6b:10:60:99), Dst: All-HSRP-routers_00 (00:00:0c:07:ac:00)
Internet Protocol Version 4, Src: 128.238.38.162, Dst: 216.75.194.220
Transmission Control Protocol, Src Port: 2271, Dst Port: 443, Seq: 1, Ack: 1, Len: 78
Transport Layer Security
  SSLv2 Record Layer: Client Hello
    [Version: SSL 2.0 (0x0002)]
    Length: 76
    Handshake Message Type: Client Hello (1)
    Version: SSL 3.0 (0x0300)
    Cipher Spec Length: 51
    Session ID Length: 0
    Challenge Length: 16
    Cipher Specs (17 specs)
    Challenge
```

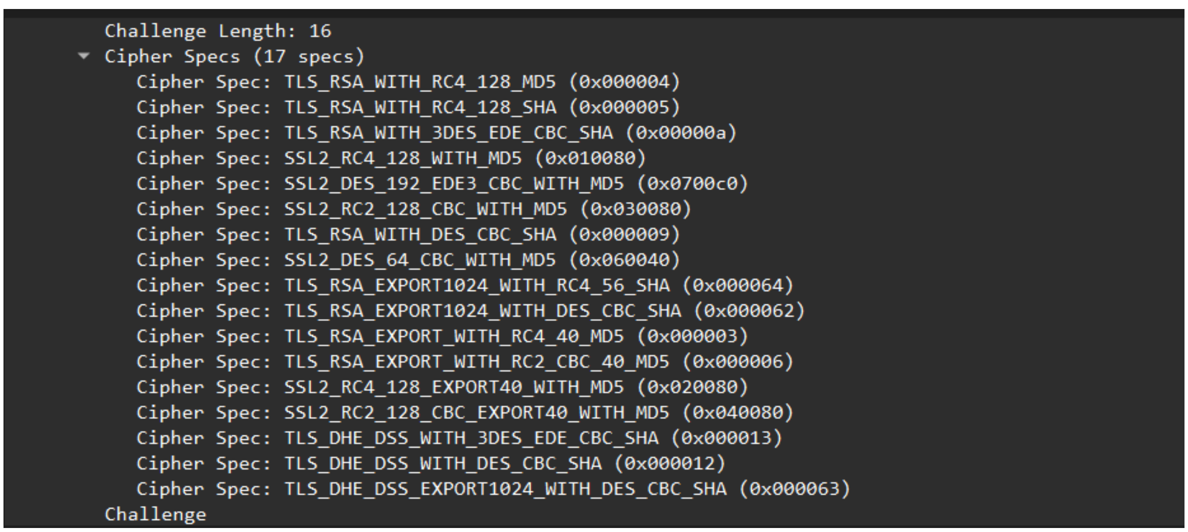
4. Does the ClientHello record contain a nonce (also known as a “challenge”)? If so, what is the value of the challenge in hexadecimal notation?

Answer: YES

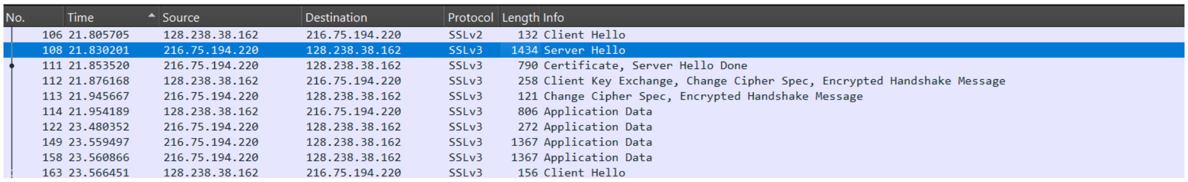


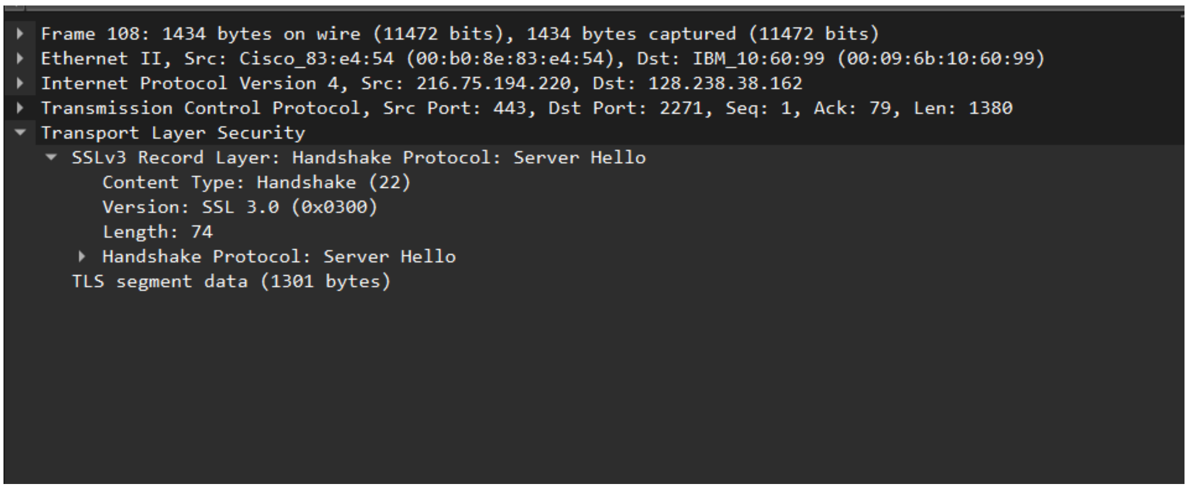
5. Does the ClientHello record advertise the cyber suites it supports? If so, in the first listed suite, what are the public-key algorithm, the symmetric-key algorithm, and the hash algorithm?

Answer: YES



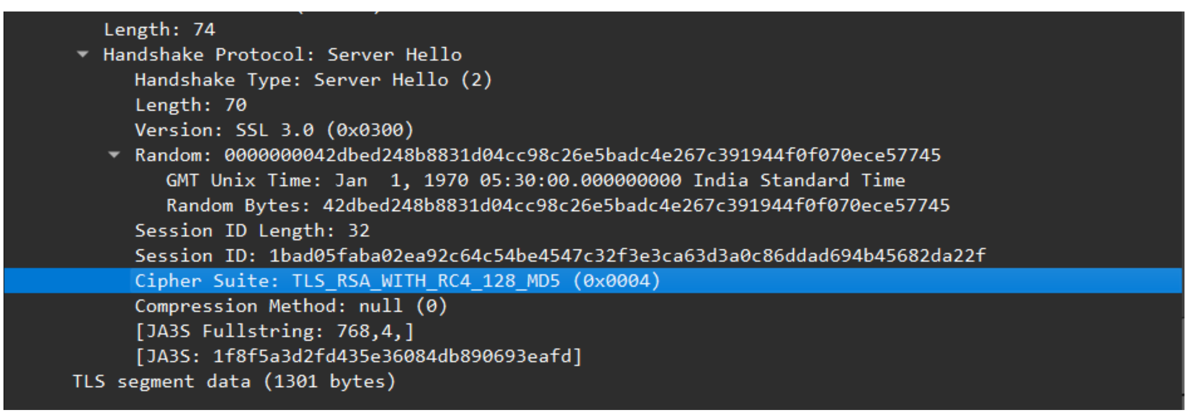
ServerHello Record:





6. Locate the ServerHello SSL record. Does this record specify a chosen cipher suite? What are the algorithms in the chosen cipher suite?

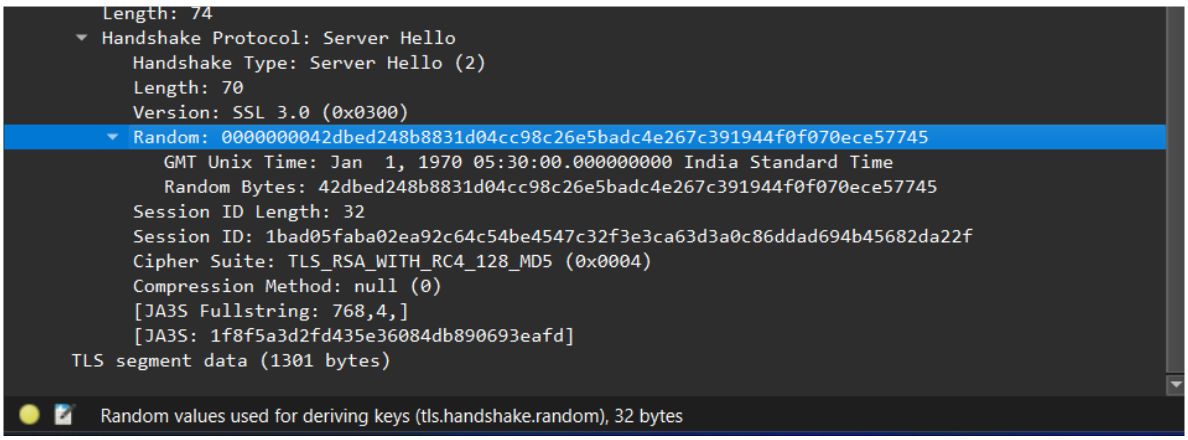
Answer: YES



7. Does this record include a nonce? If so, how long is it? What is the purpose of the client and server nonces in SSL?

Locate the Nonce:

- The **ServerHello** response may not explicitly list a nonce like the **ClientHello** does, but it usually includes a **Session ID** and potentially a **Server Random** value (which acts similarly to a nonce).
- Look for fields labeled **Session ID Length**, **Session ID**, and **Random**.



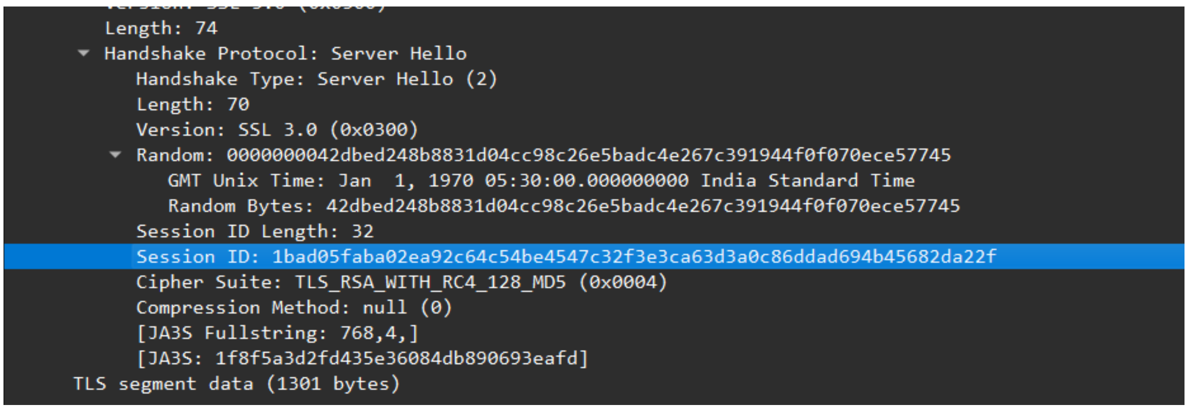
Purpose of Nonce in the ServerHello Record

1. **Session Uniqueness**:
   - Similar to the **ClientHello**, the **Server Random** value helps ensure that the session is unique. It differentiates this session from previous ones.
2. **Key Derivation**:
   - The **Server Random** value is combined with the **Client Random** value (from the **ClientHello**) during the key derivation process to create session keys for encrypting the data exchanged in the session. This ensures that the keys are unique for each session.

3. **Preventing Replay Attacks**:
   - Just as with the client, the server's nonce (or **Server Random**) helps protect against replay attacks, ensuring that each session is independent and cannot be reused maliciously.

8. Does this record include a session ID? What is the purpose of the session ID?

Answer YES

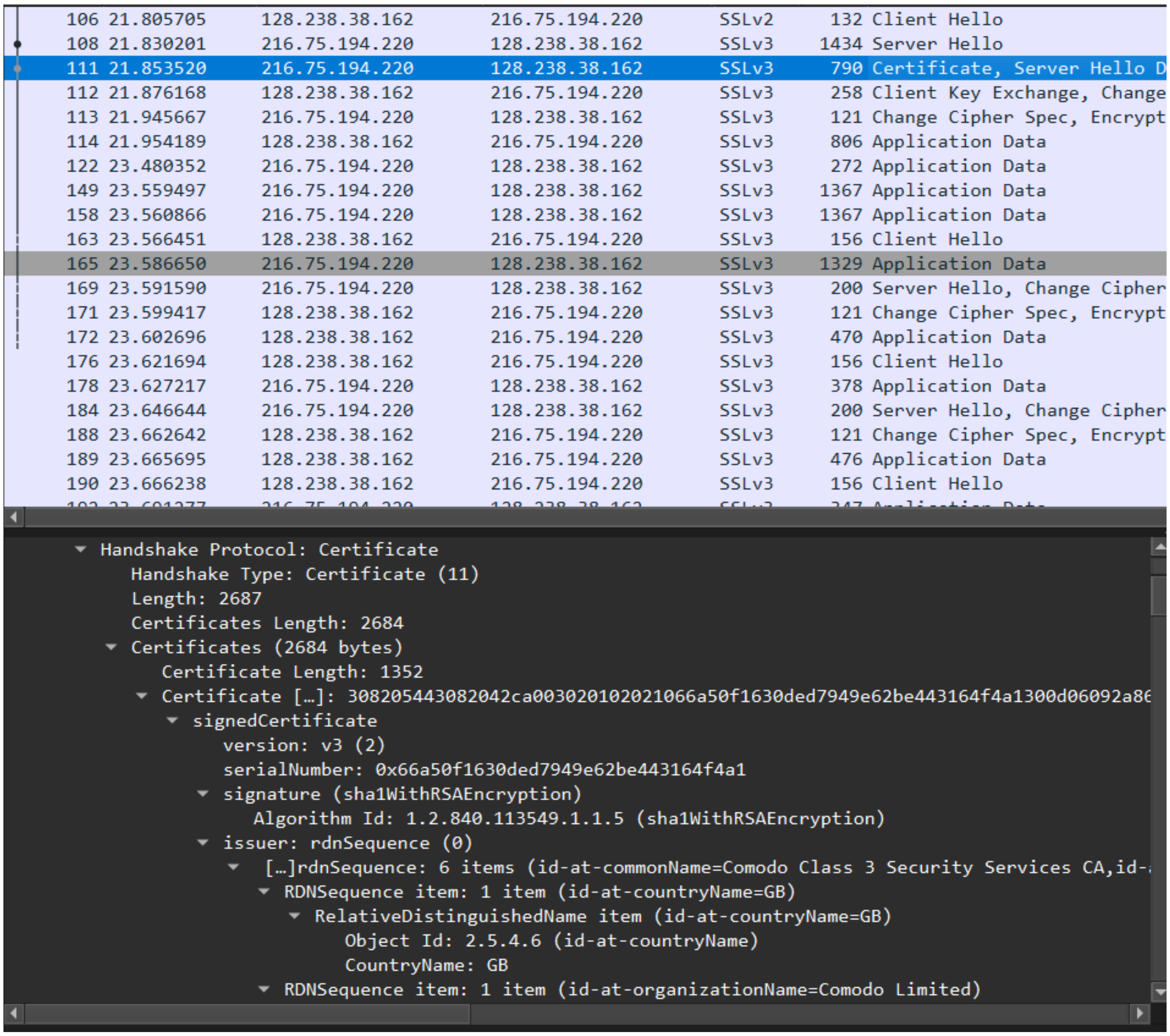


9. Does this record contain a certificate, or is the certificate included in a separate record. Does the certificate fit into a single Ethernet frame?

Answer: YES

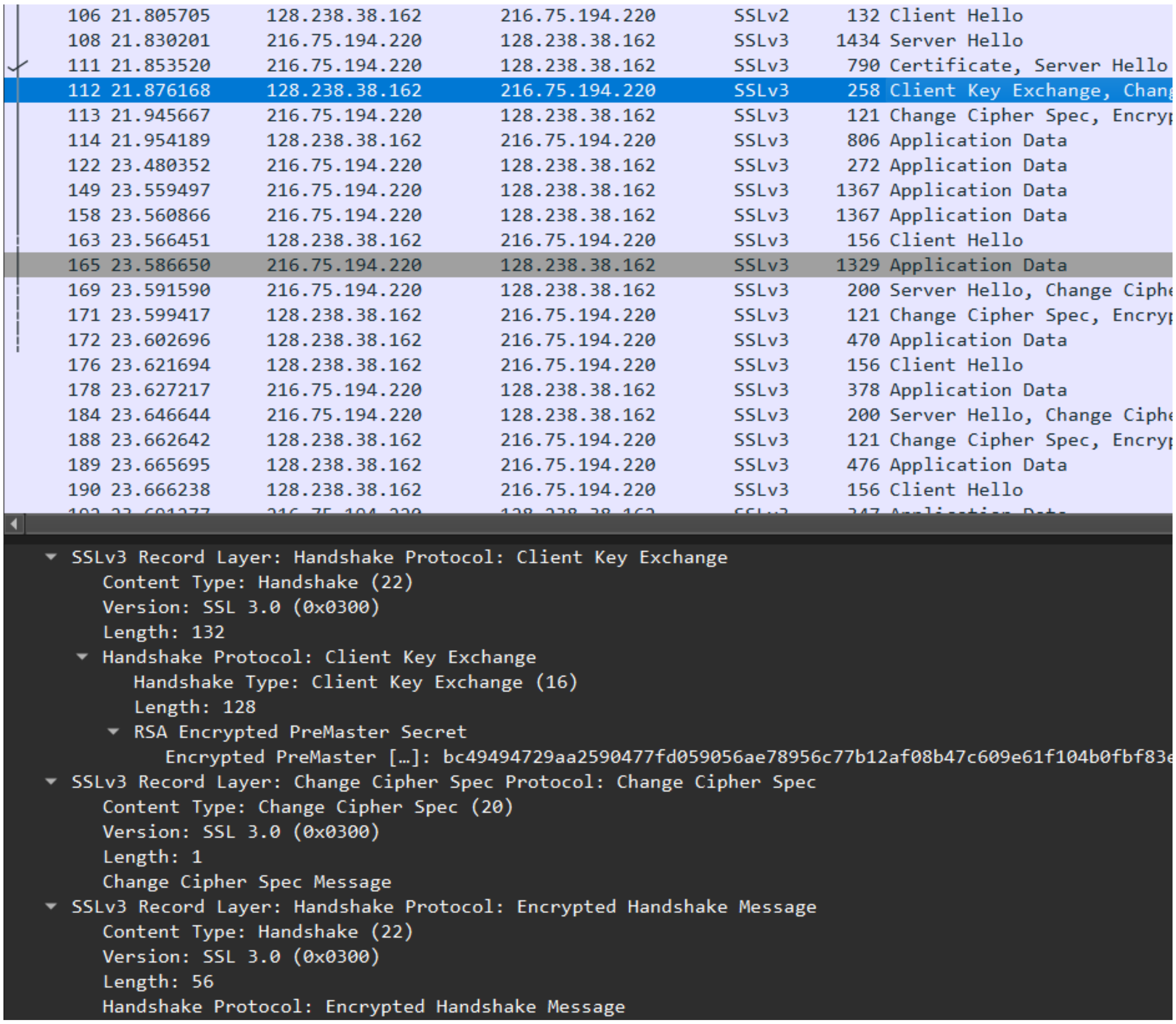
After the **ServerHello** frame, there should be another frame labeled something like **Certificate**. This frame contains the actual server certificate sent by the server.

If the certificate size is less than or equal to 1500 bytes, it will fit into a single Ethernet frame. If it exceeds this size, it will be fragmented across multiple frames.



Client Key Exchange Record:

10. Locate the client key exchange record. Does this record contain a pre-master secret? What is this secret used for? Is the secret encrypted? If so, how? How long is the encrypted secret?



Presence of Pre-Master Secret: The **Client Key Exchange** record does contain the pre-master secret, which is crucial for establishing session keys.

Purpose of the Pre-Master Secret: The pre-master secret is used to derive symmetric session keys that will encrypt the data exchanged between the client and server after the handshake is complete.

Encryption: The pre-master secret is typically encrypted with the server's public key, ensuring that only the server can decrypt it using its private key.

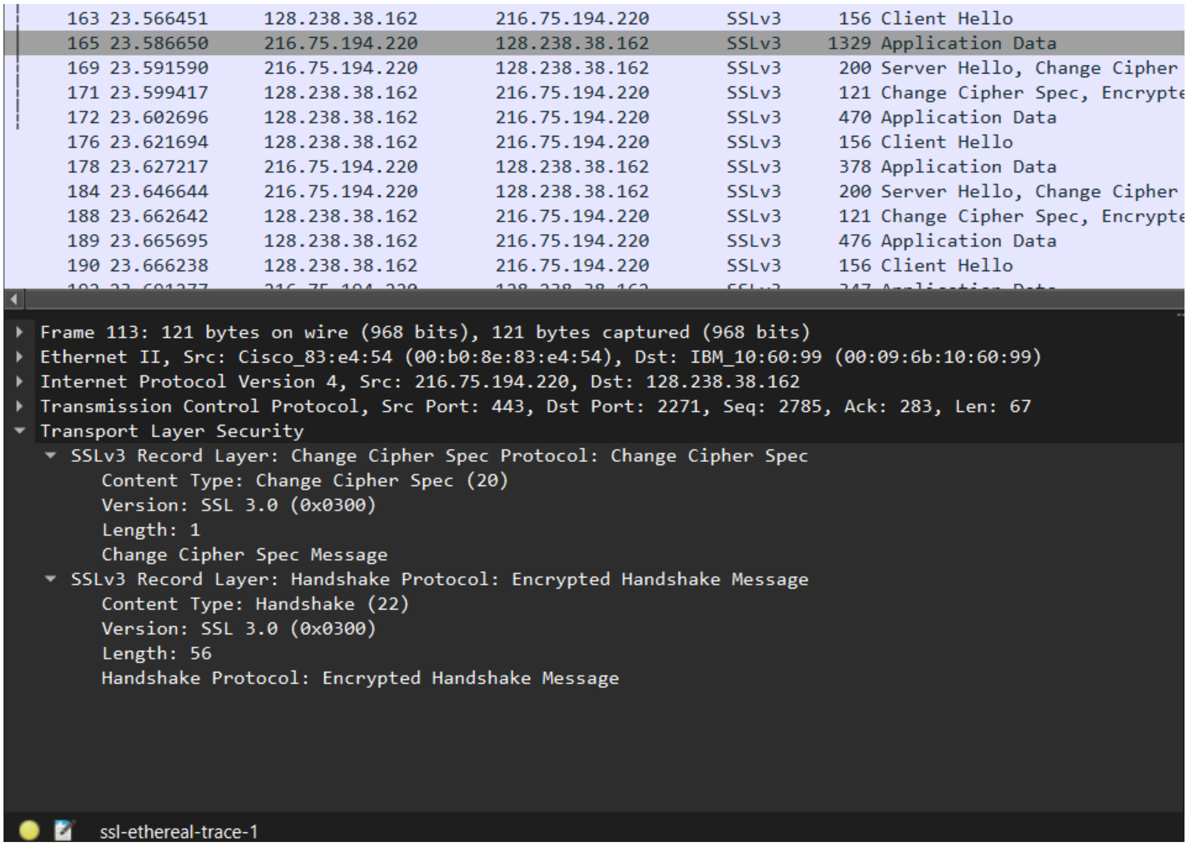
Length of the Encrypted Secret: The length of the encrypted pre-master secret is usually around 128 bytes but can vary based on the cipher suite and specific implementation.

Change Cipher Spec Record (sent by client) and Encrypted Handshake Record:

11. What is the purpose of the Change Cipher Spec record? How many bytes is the record in your trace?

Purpose: The Change Cipher Spec record indicates that the sender is ready to switch to encrypted communication using the new cipher suite and keys.

Size: The record is generally **2 bytes** in total (1 byte for the content type and 1 byte for the Change Cipher Spec message itself).



12. In the encrypted handshake record, what is being encrypted? How?

What is Encrypted: Handshake messages exchanged during the SSL/TLS handshake process.

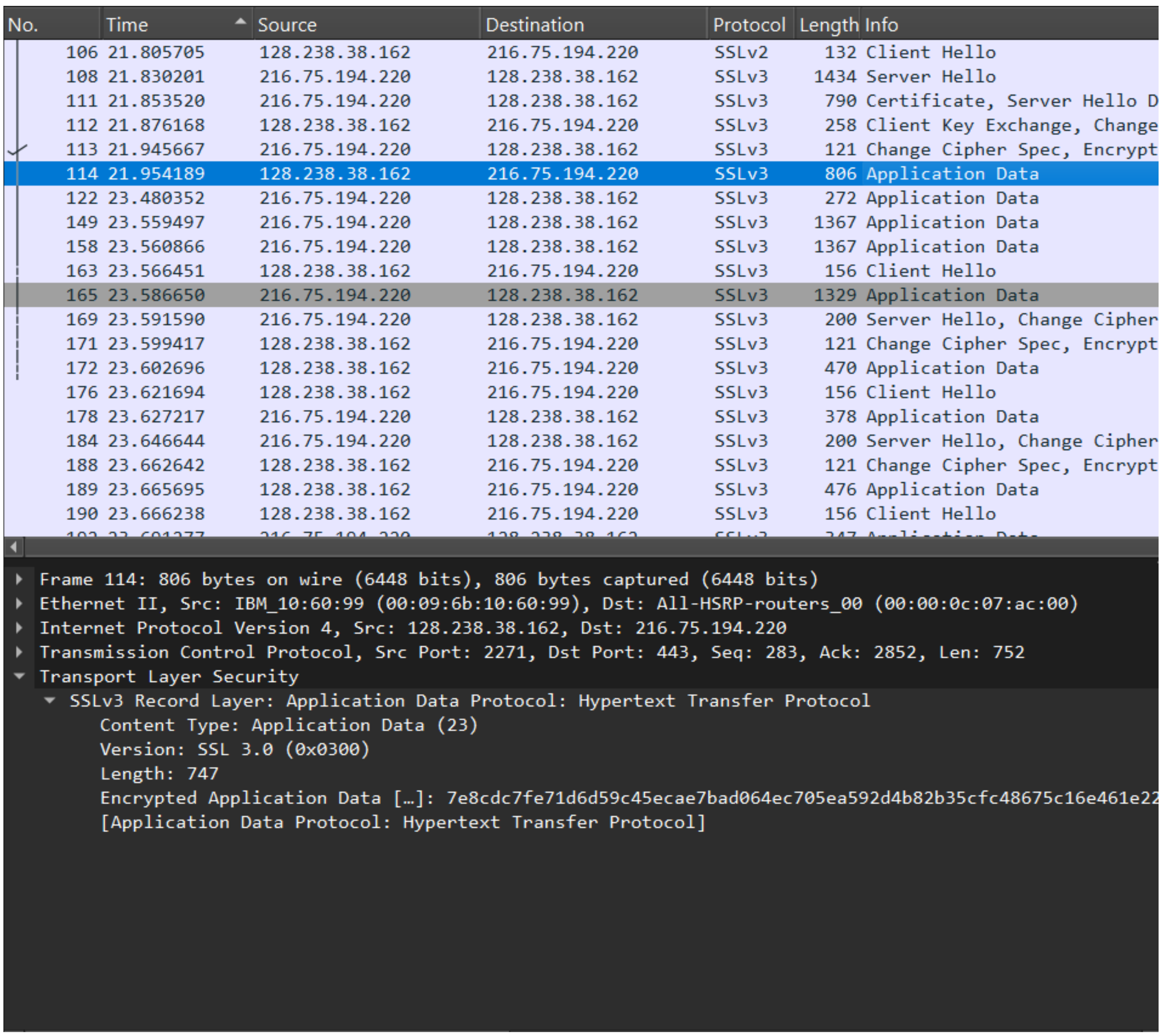
How it is Encrypted: Using symmetric-key algorithms determined by the negotiated cipher suite, leveraging session keys derived from the pre-master secret. The messages are encrypted and often accompanied by a MAC for integrity and authenticity.

13. Does the server also send a change cipher record and an encrypted handshake record to the client? How are those records different from those sent by the client?

Both the client and server send Change Cipher Spec and encrypted handshake records.

The **Change Cipher Spec records** signify readiness for encrypted communication.

The **encrypted handshake records** finalize the handshake and vary in content based on whether they originate from the client or the server, with each party indicating completion of their Application Data



14. How is the application data being encrypted? Do the records containing application data include a MAC? Does Wireshark distinguish between the encrypted application data and the MAC?

Encryption of Application Data:

- In SSL/TLS, application data is encrypted using the symmetric encryption algorithms agreed upon during the handshake process. After the Change Cipher Spec record has been exchanged, both the client and server use the session keys derived from the pre-master secret to encrypt and decrypt application data.
- The specific symmetric encryption algorithm (such as AES, DES, etc.) is part of the cipher suite chosen during the handshake.

Inclusion of MAC (Message Authentication Code):

- Yes, the records containing application data include a MAC. The MAC is calculated over the plaintext data (the application data) along with additional information like sequence numbers and the session keys.
- The MAC serves as a form of integrity check, ensuring that the data has not been tampered with during transmission.

Wireshark Distinction:

- In Wireshark, encrypted application data and the MAC are typically bundled together in the same record. However, Wireshark does display a breakdown of the decrypted application data, allowing users to view the plaintext contents after decryption.
- If the application data is decrypted (for instance, if the session keys are available to Wireshark), the MAC may not be separately shown in the decrypted data, as it is used internally to verify integrity but does not need to be displayed in the application layer.

15. Comment on and explain anything else that you found interesting in the trace.

Use of Different SSL Versions:

The trace indicates a transition from SSLv2 to SSLv3. It's interesting to note the evolution of the SSL protocol versions, as SSLv2 is considered outdated and insecure. Modern applications primarily use TLS, which is the successor to SSL. The presence of SSLv2 could indicate compatibility settings or legacy systems.

Cipher Suite Negotiation:

The ClientHello message lists multiple cipher suites supported by the client. The server chooses one from this list for the session, which can reveal insights into the security posture and configurations of both the client and server. Observing this negotiation process can be critical for understanding potential vulnerabilities.

Challenge and Nonce Usage:

The ClientHello message includes a nonce (challenge), which is a random value used to prevent replay attacks. This is an interesting feature of SSL/TLS that enhances security by ensuring that each session is unique. The presence of nonces shows the protocols' design to handle specific security threats effectively.

Certificate Exchange:

The certificate exchange step during the ServerHello message and subsequent records is crucial for establishing trust. This trace shows the server providing its certificate, which may be signed by a trusted Certificate Authority (CA). The ability to verify this certificate is essential for the client to ensure that it is communicating with the legitimate server.

Packet Sizes and Performance:

Analyzing the sizes of the packets in the trace could provide insights into network performance. Larger packets may indicate bulk data transfers, while smaller packets might signify many small transactions. Identifying patterns in packet sizes could help in optimizing application performance and network resource utilization.

Timing of Records:

Observing the timing between records can provide insights into latency and performance issues. For example, if there are significant delays between the ClientHello and ServerHello messages, it could indicate network congestion or processing delays.

Application Data Records:

The presence of application data records after the handshake signals that secure communication has commenced. Analyzing the types of application data exchanged can provide insights into the nature of the application traffic, whether it's HTTP requests, file transfers, etc.

Network Security Considerations:

The trace can help identify potential security concerns, such as unencrypted traffic, or weak cipher suites. It is important to ensure that strong encryption practices are followed, as vulnerabilities in these areas could lead to exposure of sensitive data.

These points provide a deeper understanding of the SSL handshake process and the resulting secure communication, illustrating both the complexity and importance of cryptographic protocols in modern network security.