**SCHOOL OF TECHNOLOGY**

**PANDIT DEENDAYAL ENERGY UNIVERSITY GANDHINAGAR, GUJARAT, INDIA**

# Computer Science & Engineering LAB Journal

# (2023-24) Information Security Lab(20CP304P)

# 

**Student Name: \_\_\_\_\_\_Bhavya Mehta\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Enrollment No.: \_\_21BCP024\_\_\_\_\_**

**Course with Semester: Information Security Lab | Sem 5**

**Division: \_\_01\_\_**

**Group: \_\_01\_\_\_**

**Faculty: Dr. Hargeet Kaur**

November 2023

21BCP024

Bhavya Mehta

July 2023



**C**

**ERTIFICATE**

This is to certify that

Mr.

\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Enrolment

no

**.**

\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_

of

**5th**

**Sem**

**ester**

degree course in

**Computer**

**Science**

**and**

**E**

**ngineering**

has satisfactorily prepared and presented his

/

her Term

Work in

**Information Security Lab (21CP304P)**

within

four walls of the laboratory of this Institute during

\_\_\_\_\_\_\_\_\_

\_\_

to

\_\_\_\_\_\_\_\_\_

\_\_

.

**Date of Submission: 02/11/23**

**Dr. Hargeet kaur HARGEET KAU**

**Faculty**

**SCHOOL OF TECH**

**N**

**OLOGY**

**PA**

**N**

**DIT**

**DEE**

**N**

**DAYAL E**

**N**

**ERGY U**

**N**

**IVERSITY**

**GA**

**N**

**DHI**

**N**

**AGAR, GUJARAT, I**

**N**

**DIA**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **SCHOOL OF TECHNOLOGY**  **PANDIT DEENDAYAL ENERGY UNIVERSITY GANDHINAGAR, GUJARAT, INDIA**    **INDEX**     |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Sr. No.** | **Name of Experiment** | **Page No.** | **Date** | **Sign** | | **01** | Download and Practise Cryptool. |  |  |  | | **02** | Study and Implement program for Ceaser Cipher with Encryption, Decryption, Brute Force Attack, and Frequency Analysis functions. |  |  |  | | **03** | Study and Implement a program for Transposition (Columnar) cipher ton encrypt and decrypt the message. |  |  |  | | **04** | Study and Implement a program for Railfence cipher to encrypt and decrypt the message. |  |  |  | | **05** | Study and implement a program for Playfair cipher to encrypt and decrypt the message. |  |  |  | | **06** | Study and implement a program for Playfair cipher to encrypt and  decrypt the message |  |  |  | | **07** | Study and implement a program for n-gram Hill cipher. |  |  |  | | **08** |  |  |  |  | | **09** | Study and implement RSA encryption and decryption functions |  |  |  | | **10** | Use RSA for generation and verification of digital signature on file. |  |  |  | |

**Experiment No : 01**

**Aim:**

Download and practise cryptool.

**Introduction:**

CrypTool is an [open-source](https://en.wikipedia.org/wiki/Open-source_software) project that is a free [e-learning](https://en.wikipedia.org/wiki/E-learning) software for illustrating [cryptographic and cryptanalytic concepts](https://en.wikipedia.org/wiki/Cryptography). According to "Hakin9", CrypTool is worldwide the most widespread e-learning software in the field of [cryptology](https://en.wikipedia.org/wiki/Cryptology).

CrypTool implements more than 400 [algorithms](https://en.wikipedia.org/wiki/Algorithm). Users can adjust these with own parameters. To introduce users to the field of [cryptography](https://en.wikipedia.org/wiki/Cryptography), the organization created multiple graphical interface software containing an online documentation, analytic tools and algorithms. They contain most [classical ciphers](https://en.wikipedia.org/wiki/Classical_cipher), as well as modern symmetric and [asymmetric](https://en.wikipedia.org/wiki/Public-key_cryptography) cryptography including RSA, [ECC](https://en.wikipedia.org/wiki/Elliptic_curve_cryptography), [digital signatures](https://en.wikipedia.org/wiki/Digital_signature), hybrid encryption, [homomorphic encryption](https://en.wikipedia.org/wiki/Homomorphic_encryption), and [Diffie–Hellman key exchange](https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange). Methods from the area of [quantum cryptography](https://en.wikipedia.org/wiki/Quantum_cryptography) (like [BB84 key exchange protocol](https://en.wikipedia.org/wiki/BB84)) and the area of [post-quantum cryptography](https://en.wikipedia.org/wiki/Post-quantum_cryptography) (like [McEliece](https://en.wikipedia.org/wiki/McEliece_cryptosystem" \o "McEliece cryptosystem), WOTS, [Merkle-Signature-Scheme](https://en.wikipedia.org/wiki/Merkle_signature_scheme), [XMSS, XMSS\_MT, and SPHINCS](https://en.wikipedia.org/wiki/Hash-based_cryptography)) are implemented. In addition to the algorithms, solvers (analyzers) are included, especially for classical ciphers. Other methods (for instance [Huffman code](https://en.wikipedia.org/wiki/Huffman_coding), [AES](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard), [Keccak](https://en.wikipedia.org/wiki/SHA-3), [MSS](https://en.wikipedia.org/wiki/Merkle_signature_scheme)) are visualized.

**Cryptanalysis:**

Cryptanalysis is the study and process of analyzing and decrypting ciphers, codes, and encrypted text without using the real key. Alternately, we can say it’s the technique of accessing a communication’s plain text content when you don’t have access to the decryption key.

**Applications:**

* Cryptool can be used in various aspects of cryptographic and cryptanalytic concepts.
* Cryptography is used in many applications like banking transactions cards, computer passwords, and e- commerce transactions.

**References:**

1. <https://en.wikipedia.org/wiki/CrypTool>
2. <https://www.c-sharpcorner.com/article/encryption-decryption-using-cryptool/#:~:text=Cryptool%20is%20an%20open%2Dsource%20and%20freeware%20program%20that%20can,and%20decryption%20of%20various%20algorithms>

**Experiment No : 02**

**Aim :**Study and Implement program for Ceaser Cipher with Encryption, Decryption, Brute Force Attack, and Frequency Analysis functions.

**Introduction :** The Caesar cipher is one of the simplest and oldest encryption techniques used to secure information. Named after Julius Caesar, who is said to have used this method to communicate secretly, the Caesar cipher is a type of substitution cipher where each letter in the plaintext is shifted a certain number of positions down or up the alphabet.

Here's how the Caesar cipher works:

**Encryption:**

To encrypt a message using the Caesar cipher, you choose a fixed number called the "key" or "shift." Each letter in the plaintext is replaced by a letter that is a fixed number of positions down the alphabet.

For example, with a shift of 3:

* 'A' becomes 'D'
* 'B' becomes 'E'
* 'C' becomes 'F'
* ...
* 'X' becomes 'A'
* 'Y' becomes 'B'
* 'Z' becomes 'C'

So, the plaintext "HELLO" would be encrypted as "KHOOR" with a shift of 3.

**Decryption:**

To decrypt the message, you reverse the process. You choose the same key but shift the letters in the opposite direction. In the case of a shift of 3, you would shift each letter 3 positions back up the alphabet.

So, the ciphertext "KHOOR" would be decrypted as "HELLO" with a shift of 3.

**Program (Source Code):**

|  |  |
| --- | --- |
| **Code:** | **def encrypt(text, salt):**  **result = ""**  **text = text.lower()**  **for i in range(len(text)):**  **char = text[i]**  **result += chr((ord(char) + salt - 97) % 26 + 97)**  **return result**  **def decrypt(text, salt):**  **result = ""**  **text = text.lower()**  **for i in range(len(text)):**  **char = text[i]**  **result += chr((ord(char) - salt - 97) % 26 + 97)**  **return result**  **def brute\_force(text):**  **for i in range(26):**  **print(i, " ", decrypt(text, i))**  **def frequency\_analysis(text):**  **freq = {}**  **for i in range(len(text)):**  **char = text[i]**  **if char in freq:**  **freq[char] += 1**  **else:**  **freq[char] = 1**  **print(freq)**  **def main():**  **print("Enter the text to be encrypted")**  **text = input()**  **print("Enter the salt value")**  **salt = int(input())**  **print("The encrypted text is")**  **print(encrypt(text, salt))**  **print("The decrypted text is")**  **print(decrypt(encrypt(text, salt), salt))**  **print("The brute force decryption is")**  **brute\_force(encrypt(text, salt))**  **print("The frequency analysis is")**  **frequency\_analysis(encrypt(text, salt))**  **main()** |
| **Output:** |  |
| **Output (Cryptool):** |  |

**Cryptanalysis :**

The Caesar Cipher, although historically significant, has numerous shortcomings that render it highly insecure for modern cryptographic purposes:

* Restricted Key Options: The Caesar Cipher offers only 25 potential keys (one for each letter shift), making it vulnerable to brute force attacks in which all potential shifts are tested to find the correct one.
* Weak Security: Given its small key space, the Caesar Cipher can be easily deciphered through frequency analysis, where common letters in the ciphertext are matched to their probable plaintext equivalents.
* Predictability: The Caesar Cipher's simplicity results in predictability, especially when messages contain commonplace words or phrases. Attackers can exploit patterns and context to determine the key and decrypt the message.
* Inadequate Against Language Analysis: As the Caesar Cipher simply shifts letters by a fixed value, it provides no defense against linguistic analysis, whereby the structure and grammar of the original text can still offer insights to attackers.
* Single-Layer Encryption: The Caesar Cipher provides just a solitary layer of encryption, rendering it susceptible to attacks that exploit known characteristics of the cipher.
* Given these significant drawbacks, the Caesar Cipher is unsuitable for secure communication. Modern cryptographic techniques employ much more robust algorithms, utilizing intricate mathematical functions and larger key spaces to ensure formidable security against various forms of attacks.

**Applications :**

1.Educational Purposes: The Caesar cipher can be used to introduce students to basic concepts of cryptography and encryption. It helps them understand the idea of substituting characters to create ciphertext and the concept of keys and key spaces.

2. Puzzles and Games: The Caesar cipher can be used in puzzles, brain teasers, or escape room scenarios. Participants might need to decode messages to proceed in the game or solve a mystery.

3. Basic Encryption in Historical Reenactments: In historical reenactments or themed events, the Caesar cipher can be used to add an element of authenticity. Participants might use it to exchange messages that need to be decoded to enhance the atmosphere of the event.

4. Children's Activities: The Caesar cipher can be used for simple secret messages among children, allowing them to have fun while learning about encoding and decoding messages.

5. Simple Text Obfuscation: In contexts where only a casual level of security is needed, such as hiding messages from casual observers, the Caesar cipher can be applied. For example, to share a phone number or a basic message on a public forum without immediately revealing the content.

6. Part of More Complex Ciphers: The Caesar cipher can serve as a building block for more complex ciphers. It can be used as a component in a larger encryption scheme to add an additional layer of security.

**References :**

1. <https://www.cryptool.org/en/cto/caesar>
2. <https://www.javatpoint.com/caesar-cipher-technique>

**Experiment No : 03**

**Aim :** Study and Implement a program for Transposition (Columnar) cipher ton encrypt and decrypt the message.

**Introduction :** The Columnar Transposition Cipher is a simple method of encryption that involves rearranging the characters of a plaintext message in a grid or matrix, followed by reading the characters column by column to obtain the ciphertext. The key to this cipher is the arrangement of columns in the grid.

Here's how the Caesar cipher works:

**Encryption Process:**

Key: Choose a key that determines the order of columns in the grid. The key is usually a permutation of numbers that represent the order in which columns should be read.

Plaintext: Write the plaintext message row by row in the grid, filling the rows from left to right. Any empty spaces are usually filled with a placeholder character.

Rearrange Columns: Rearrange the columns of the grid based on the order specified by the key. Columns are read from left to right according to the key's permutation.

Ciphertext: Read the characters column by column from the rearranged grid to obtain the ciphertext.

**Decryption:**

Decryption is the reverse process of encryption:

Key: Use the same key that was used for encryption.

Ciphertext: Write the ciphertext column by column in the grid, filling the columns from top to bottom.

Rearrange Columns: Rearrange the columns of the grid based on the order specified by the key.

Plaintext: Read the characters row by row from the rearranged grid to obtain the original plaintext.

**Program (Source Code):**

|  |  |
| --- | --- |
| **Code:** | alphabets = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']  def encryption(message, key):  message = message.lower()  key = key.lower()  key\_length = len(key)  encrypted\_text = [""] \* key\_length  while (True):  if len(message) % len(key) != 0:  message = message + 'z'  elif len(message) % len(key) == 0:  break  else:  pass  for column in range(key\_length):  pointer = column  while pointer < len(message):  encrypted\_text[column] += message[pointer]  pointer += key\_length  key\_string = [\*key]  key\_index = []  for i in key\_string:  if i in alphabets:  key\_index.append(alphabets.index(i))  print(key\_index)  sorted\_key\_index = sorted(key\_index)  encrypt = []  for i in sorted\_key\_index:  index = key\_index.index(i)  encrypt.append(encrypted\_text[index])  encrypted\_final = ''.join(encrypt)  print(encrypted\_final)  def decryption(encrypted, key):  encrypted = encrypted.lower()  key = key.lower()  plain\_text = []  key\_length = len(key)  encrypted\_message\_length = len(encrypted)  divide = int(encrypted\_message\_length / key\_length)  encrypted\_text\_list = []  for i in range(len(key)):  s = encrypted[divide \* i:divide \* (i + 1)]  encrypted\_text\_list.append(s)  key\_string = [\*key]  key\_index = []  for i in key\_string:  if i in alphabets:  key\_index.append(alphabets.index(i))  sorted\_key\_index = sorted(key\_index)  encrypted\_column\_vise = []  for i in key\_index:  index = sorted\_key\_index.index(i)  encrypted\_column\_vise.append(encrypted\_text\_list[index])  for i in range(divide):  for j in range(len(key)):  plain\_text.append(encrypted\_column\_vise[j][i])  plain = ''.join(plain\_text)  print(plain)  def main():  print("Enter the message to be encrypted")  message = input()  print("Enter the key")  key = input()  encryption(message, key)  print("Enter the message to be decrypted")  encrypted = input()  print("Enter the key")  key = input()  decryption(encrypted, key)  if \_\_name\_\_ == "\_\_main\_\_":  main() |
| **Output:** |  |
| **Output (Cryptool):** |  |

**Cryptanalysis :**

**Brute Force:**

Brute force involves trying all possible permutations of the columns (or rows, depending on the encryption method) to see if any of them result in a valid plaintext. Since there can be many possible permutations, this method becomes impractical for longer messages or larger key lengths.

Here's how the brute force approach works:

* Generate all possible permutations of the columns (key) and decrypt the message using each permutation.
* For each decrypted message, apply some form of frequency analysis or other pattern recognition techniques to determine if the resulting text looks like valid plaintext.
* Choose the decryption that makes the most sense based on these patterns.

**Frequency Analysis:**

Frequency analysis involves analyzing the frequency distribution of characters in the ciphertext and comparing it to the expected frequency distribution of characters in the language (usually English). Transposition ciphers tend to preserve the frequency of individual letters, but they may disrupt the frequency of letter pairs and triplets.

Here's how frequency analysis could be applied to a transposition cipher:

* Compute the frequency distribution of letters in the ciphertext.
* Compare the frequency distribution to the expected frequency distribution of letters in the given language. For English, the most common letters are E, T, A, O, I, N, S, H, R, and D.
* Look for patterns in the bigrams and trigrams (pairs and triplets) of letters. In English, certain pairs like "TH", "ER", "AN", "RE", etc., are very common. If you find similar patterns, it could help you identify potential column permutations.
* Try different column orders based on the patterns you've identified and see if any of them result in meaningful plaintext.

**Applications :**

The Columnar Transposition Cipher, despite its vulnerability to certain types of cryptanalysis, has found applications in various scenarios due to its simplicity and ease of implementation. While it might not be suitable for highly sensitive or critical communications, it has been historically used for the following purposes:

Educational Purposes: The Columnar Transposition Cipher is often used as an introductory example in cryptography courses to teach basic concepts of encryption, permutation, and transposition. It helps students understand the fundamental principles of encryption before moving on to more complex ciphers.

Puzzles and Games: The cipher has been used in puzzle creation, escape room challenges, and interactive games. Participants must decipher messages encrypted using the Columnar Transposition Cipher to progress in the game.

Pen-and-Paper Puzzles: Magazines, puzzle books, and websites occasionally present encrypted messages that can be solved using simple ciphers like Columnar Transposition. These puzzles are meant to engage readers and challenge their problem-solving skills.

**References :**

1. [https://www.cryptool.org](https://www.cryptool.org/en/cto/caesar)
2. <https://www.geeksforgeeks.org/columnar-transposition-cipher/>

**Experiment No : 04**

**Aim :** Study and Implement a program for Railfence cipher to encrypt and decrypt the message.

**Introduction :** The Rail Fence Cipher is a simple transposition cipher that rearranges the characters of a message to encrypt and decrypt it. It works by writing the characters of the message in a zigzag pattern along a set number of "rails" (rows), and then reading off the characters row by row to create the encrypted text.

**Program (Source Code):**

|  |  |
| --- | --- |
| **Code:** | def rail\_fence\_encrypt(text, key):  encrypted = [''] \* key  index = 0  direction = 1  for char in text:  encrypted[index] += char  if index == 0:  direction = 1  elif index == key - 1:  direction = -1  index += direction  return ''.join(encrypted)  def rail\_fence\_decrypt(text, key):  decrypted = [''] \* key  index = 0  direction = 1  for char in text:  decrypted[index] += '\*'  if index == 0:  direction = 1  elif index == key - 1:  direction = -1  index += direction  text\_index = 0  for i in range(key):  for j in range(len(decrypted[i])):  decrypted[i] = decrypted[i][:j] + text[text\_index] + decrypted[i][j + 1:]  text\_index += 1  result = ''  index = 0  direction = 1  for i in range(len(text)):  result += decrypted[index][0]  decrypted[index] = decrypted[index][1:]  if index == 0:  direction = 1  elif index == key - 1:  direction = -1  index += direction  return result  # Example usage  text = "railfencecipherexample"  key = 3  encrypted\_text = rail\_fence\_encrypt(text, key)  print("Encrypted:", encrypted\_text)  decrypted\_text = rail\_fence\_decrypt(encrypted\_text, key)  print("Decrypted:", decrypted\_text) |
| **Output:** |  |
| **Output (Cryptool):** |  |

**Cryptanalysis :**

Cryptanalysis for the Rail Fence Cipher involves attempting to break the encryption and recover the original plaintext without knowing the key. The Rail Fence Cipher is relatively simple, and one common approach to cryptanalysis is brute force, where you try all possible key values. However, since the key is typically small (the number of rails), this process can be quick.

Here are some methods to cryptanalyze the Rail Fence Cipher:

1. Brute Force: Try all possible key values (number of rails) and see if any of them produce meaningful plaintext. This approach is feasible because the key space is limited by the number of rails, which is usually not very large.

2. Frequency Analysis: If the plaintext message is in English (or another language), you can analyze the frequency distribution of letters and common letter combinations. English has distinct patterns of letter frequencies, such as 'E' being the most common letter. If you notice a significant deviation from expected frequencies, you might be able to deduce the key.

3. Pattern Recognition: Depending on the message length and the key, certain patterns might be recognizable in the ciphertext. For example, if the key divides the message length evenly, you might see repeating patterns in the ciphertext. Identifying such patterns could help reveal the key.

4. Known Plaintext Attack: If you have access to the corresponding plaintext-ciphertext pairs, you can attempt to deduce the key based on the observed patterns between them.

5. Trial and Error: You can try various methods for decryption, such as filling the rails one by one, filling diagonally, or other patterns. If you discover a method that produces meaningful plaintext, you might be closer to the correct key.

6. Automated Tools: There are online tools and software available that can automate the process of trying different key values to decrypt the ciphertext.

Remember that the Rail Fence Cipher is not particularly strong, especially when using a small number of rails. It's more of a historical and educational cipher rather than one used for serious encryption. As a result, cryptanalysis techniques are often effective in breaking the cipher.

**Applications :**

The Rail Fence Cipher, while not a strong encryption method, has been used historically and can have some educational and recreational applications. Some of its applications include:

1. Educational Purposes: The Rail Fence Cipher is often used as a teaching tool to introduce basic concepts of cryptography and transposition ciphers to students and beginners. It's a simple example that helps learners understand the principles of encryption and decryption.

2. Puzzle and Challenges: The Rail Fence Cipher can be used in puzzle games, escape rooms, or challenge scenarios where participants need to decode a message to proceed. Its simplicity makes it accessible for participants of various skill levels.

3. Historical Reenactments: The Rail Fence Cipher was historically used during times when secure communication was limited. It can be employed in historical reenactments, events, or exhibits to showcase how messages were encrypted in the past.

4. Recreational Cryptography: The cipher can be used for fun and recreational purposes, such as creating puzzles for friends or family members to solve. It's a lighthearted way to engage with cryptography.

**References :**

1. [https://www.cryptool.org](https://www.cryptool.org/en/cto/caesar)
2. <https://www.geeksforgeeks.org/rail-fence-cipher-encryption-decryption/>

**Experiment No : 05**

**Aim :** Study and implement a program for Playfair cipher to encrypt and decrypt the message.

**Introduction :** The Playfair cipher is a manual symmetric encryption technique that was invented by Charles Wheatstone in 1854.

The Playfair Cipher encryption technique can be used to encrypt or encode a message. It operates exactly like typical encryption. The only difference is that it encrypts a digraph, or a pair of two letters, instead of a single letter. An initial 5×5 matrix key table is created. The plaintext encryption key is made out of the matrix’s alphabetic characters. Be mindful that you shouldn’t repeat the letters. There are 26 alphabets however, there are only 25 spaces in which we can place a letter. Here's a basic overview of how the Playfair cipher works:-

**Key Preparation:**

1. Start with a secret key, which is typically a keyword or phrase. Spaces and duplicate letters are removed, and the alphabet is often extended to include all 26 letters (usually by combining 'I' and 'J' into one letter).

2.Create a 5x5 matrix (usually referred to as the Playfair matrix) using the modified key. Fill

the matrix with the unique letters from the key in a row-wise manner, avoiding duplicates.

3.Fill any remaining cells in the matrix with the remaining letters of the alphabet, excluding those used in the key.

**Encryption:**

1. Break the plaintext into pairs of letters (digraphs).

2. Apply the following rules to each digraph:

• If both letters in the digraph are in the same row of the matrix, replace them with

the letters to their right (wrapping around if necessary).

• If both letters are in the same column, replace them with the letters below

(wrapping around if necessary).

• If the letters are in different rows and columns, form a rectangle with the two

letters and replace them with the letters at the opposite corners of the rectangle.

3. Repeat the process for all digraphs in the plaintext.

4. The result is the ciphertext.

**Decryption:**

1. Use the same key and matrix used for encryption to

prepare for decryption.

2. Break the ciphertext into digraphs.

3. Apply the reverse of the encryption rules to each digraph to obtain the original plaintext. The Playfair cipher was historically used for secure communication, especially during the late 19th and early 20th centuries. However, it's relatively simple compared to modern encryption methods and is considered insecure for serious cryptographic applications today.

**Program (Source Code):**

|  |  |
| --- | --- |
| **Code:** | plain\_text = "helloworld"  keyword = "bhavya"  def generate\_playfair\_matrix(key):  key = key.replace(" ", "").upper()  key = key.replace("J", "I")  key\_set = set(key)  alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"  remaining\_chars = [char for char in alphabet if char not in key\_set]  playfair\_matrix = []  for char in key:  if char not in playfair\_matrix:  playfair\_matrix.append(char)  for char in remaining\_chars:  playfair\_matrix.append(char)  matrix = [['' for i in range(5)] for j in range(5)]  i = 0  for j in range(0, len(playfair\_matrix), 5):  matrix[i] = playfair\_matrix[j:j + 5]  i += 1  return matrix  def prepare\_input(text):  text = text.replace(" ", "").upper()  text = text.replace("J", "I")  prepared\_text = ""  i = 0  while i < len(text):  if i == len(text) - 1:  prepared\_text += text[i] + "X"  i += 1  elif text[i] == text[i + 1]:  prepared\_text += text[i] + "X"  i += 1  else:  prepared\_text += text[i] + text[i + 1]  i += 2  return prepared\_text  def find\_coordinates(matrix, char):  row = 0  col = 0  for i in range(5):  for j in range(5):  if matrix[i][j] == char:  return i, j  def playfair\_encrypt(plaintext, key):  matrix = generate\_playfair\_matrix(key)  text = prepare\_input(plaintext)  ciphertext = ""  for i in range(0, len(text), 2):  char1 = text[i]  char2 = text[i + 1]  row1, col1 = find\_coordinates(matrix, text[i])  row2, col2 = find\_coordinates(matrix, text[i + 1])  if row1 == row2:  ciphertext += matrix[row1][(col1 + 1) % 5] + matrix[row2][(col2 + 1) % 5]  elif col1 == col2:  ciphertext += matrix[(row1 + 1) % 5][col1] + matrix[(row2 + 1) % 5][col2]  else:  ciphertext += matrix[row1][col2] + matrix[row2][col1]  return ciphertext  def playfair\_decrypt(ciphertext, key):  matrix = generate\_playfair\_matrix(key)  text = prepare\_input(ciphertext)  plain\_text = ""  for i in range(0, len(text), 2):  char1 = text[i]  char2 = text[i + 1]  row1, col1 = find\_coordinates(matrix, text[i])  row2, col2 = find\_coordinates(matrix, text[i + 1])  if row1 == row2:  plain\_text += matrix[row1][(col1 - 1) % 5] + matrix[row2][(col2 - 1) % 5]  elif col1 == col2:  plain\_text += matrix[(row1 - 1) % 5][col1] + matrix[(row2 - 1) % 5][col2]  else:  plain\_text += matrix[row1][col2] + matrix[row2][col1]  return plain\_text  print("Cipher Text:", playfair\_encrypt(plain\_text, keyword))  print("Plain Text:", playfair\_decrypt(playfair\_encrypt(plain\_text, keyword), keyword)) |
| **Output:** |  |
| **Output (Cryptool):** |  |

**Cryptanalysis :**

**Here are some of the main cipher analysis of Playfair cipher:-**

**1.Frequency Analysis**:

In the Playfair cipher, digraphs (pairs of letters) are encrypted rather than individual letters.

This makes frequency analysis more challenging than with monoalphabetic ciphers like the

Caesar cipher.

However, common digraphs in the plaintext may still appear with some frequency in the

ciphertext. For example, if "TH" is a common digraph in English text, it may still appear as a

frequently occurring digraph in the ciphertext.

**2.** **Patterns and Repeating Digraphs**:

The Playfair cipher may create repeating patterns in the ciphertext, particularly when the same digraphs are used multiple times in the plaintext. For example, if "HELLO" is encrypted, and "L" appears twice, the corresponding ciphertext

digraphs for "L" might be the same.

**3**. **Known Plaintext Attack**:

If an attacker knows or can guess part of the plaintext and its corresponding ciphertext, they

can gain insights into the key.

For instance, if "HELLO" encrypts to "KEMON," the attacker knows that "H" maps to "K"

and "E" maps to "E," which provides some information about the key matrix.

**4. Brute Force**:

• A brute-force attack on the Playfair cipher involves systematically trying all possible keys.

• The Playfair key space is somewhat reduced compared to monoalphabetic ciphers but still

requires a large number of attempts, making it computationally expensive to break through

brute force.

**5**. **Key Patterns**:

If there are patterns or weaknesses in how the key matrix is generated, it can make the cipher

easier to analyse.

For example, if the key matrix is poorly chosen and lacks sufficient randomness, it could make the cipher vulnerable to attack.

**Applications:**

**1.Puzzles and Games**: Playfair ciphers can be used in puzzles, riddles, or educational games.

Participants can enjoy the challenge of decoding messages encrypted with this classical cipher.

**2.Educational Tools**: Teachers and educators can use Playfair ciphers to demonstrate basic

concepts of cryptography and encourage students to explore encryption techniques.

**3.Historical Reenactments**: In historical reenactments or period-themed events, the Playfair

cipher can be employed to add an authentic touch to communications, as it was historically

used in the 19th and early 20th centuries.

**4.Personal Messages**: For casual or non-sensitive communication between friends or family

members, Playfair ciphers can be used to add a fun and personalized touch to messages.

**5.Escape Rooms**: Escape room challenges often involve deciphering codes and ciphers to

progress in the game. Playfair ciphers can be one of the elements in such puzzles.

**References :**

1. [https://www.cryptool.org](https://www.cryptool.org/en/cto/caesar)
2. <https://www.geeksforgeeks.org/>

**Experiment No : 06**

**Aim :** Study and implement a program for Playfair cipher to encrypt and

decrypt the message.

**Introduction :** The Playfair cipher is a manual symmetric encryption technique that was invented by Charles Wheatstone in 1854.

The Playfair Cipher encryption technique can be used to encrypt or encode a message. Itoperates exactly like typical encryption. The only difference is that it encrypts a digraph,or a pair of two letters, instead of a single letter. An initial 5×5 matrix key table is created. The plaintext encryption key is made out of the matrix’s alphabetic characters. Be mindful that you shouldn’t repeat the letters. There are 26 alphabets however, there are only 25 spaces in which we can place a letter. Here's a basic overview of how the Playfair cipher works:-

**Key Preparation:**

1. Start with a secret key, which is typically a keyword or phrase. Spaces and duplicate letters are removed, and the alphabet is often extended to include all 26 letters (usually by combining 'I' and 'J' into one letter).

2.Create a 5x5 matrix (usually referred to as the Playfair matrix) using the modified key. Fill

the matrix with the unique letters from the key in a row-wise manner, avoiding duplicates.

3.Fill any remaining cells in the matrix with the remaining letters of the alphabet, excluding those used in the key.

**Encryption:**

1. Break the plaintext into pairs of letters (digraphs).

2. Apply the following rules to each digraph:

• If both letters in the digraph are in the same row of the matrix, replace them with

the letters to their right (wrapping around if necessary).

• If both letters are in the same column, replace them with the letters below

(wrapping around if necessary).

• If the letters are in different rows and columns, form a rectangle with the two

letters and replace them with the letters at the opposite corners of the rectangle.

3. Repeat the process for all digraphs in the plaintext.

4. The result is the ciphertext.

**Decryption:**

1. Use the same key and matrix used for encryption to

prepare for decryption.

2. Break the ciphertext into digraphs.

3. Apply the reverse of the encryption rules to each digraph to obtain the original plaintext. The Playfair cipher was historically used for secure communication, especially during the late 19th and early 20th centuries. However, it's relatively simple compared to modern encryption methods and is considered insecure for serious cryptographic applications today.

**Program (Source Code):**

|  |  |
| --- | --- |
| **Code:** | plain\_text = "helloworld"  keyword = "bhavya"  def generate\_playfair\_matrix(key):  key = key.replace(" ", "").upper()  key = key.replace("J", "I")  key\_set = set(key)  alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"  remaining\_chars = [char for char in alphabet if char not in key\_set]  playfair\_matrix = []  for char in key:  if char not in playfair\_matrix:  playfair\_matrix.append(char)  for char in remaining\_chars:  playfair\_matrix.append(char)  matrix = [['' for i in range(5)] for j in range(5)]  i = 0  for j in range(0, len(playfair\_matrix), 5):  matrix[i] = playfair\_matrix[j:j + 5]  i += 1  return matrix  def prepare\_input(text):  text = text.replace(" ", "").upper()  text = text.replace("J", "I")  prepared\_text = ""  i = 0  while i < len(text):  if i == len(text) - 1:  prepared\_text += text[i] + "X"  i += 1  elif text[i] == text[i + 1]:  prepared\_text += text[i] + "X"  i += 1  else:  prepared\_text += text[i] + text[i + 1]  i += 2  return prepared\_text  def find\_coordinates(matrix, char):  row = 0  col = 0  for i in range(5):  for j in range(5):  if matrix[i][j] == char:  return i, j  def playfair\_encrypt(plaintext, key):  matrix = generate\_playfair\_matrix(key)  text = prepare\_input(plaintext)  ciphertext = ""  for i in range(0, len(text), 2):  char1 = text[i]  char2 = text[i + 1]  row1, col1 = find\_coordinates(matrix, text[i])  row2, col2 = find\_coordinates(matrix, text[i + 1])  if row1 == row2:  ciphertext += matrix[row1][(col1 + 1) % 5] + matrix[row2][(col2 + 1) % 5]  elif col1 == col2:  ciphertext += matrix[(row1 + 1) % 5][col1] + matrix[(row2 + 1) % 5][col2]  else:  ciphertext += matrix[row1][col2] + matrix[row2][col1]  return ciphertext  def playfair\_decrypt(ciphertext, key):  matrix = generate\_playfair\_matrix(key)  text = prepare\_input(ciphertext)  plain\_text = ""  for i in range(0, len(text), 2):  char1 = text[i]  char2 = text[i + 1]  row1, col1 = find\_coordinates(matrix, text[i])  row2, col2 = find\_coordinates(matrix, text[i + 1])  if row1 == row2:  plain\_text += matrix[row1][(col1 - 1) % 5] + matrix[row2][(col2 - 1) % 5]  elif col1 == col2:  plain\_text += matrix[(row1 - 1) % 5][col1] + matrix[(row2 - 1) % 5][col2]  else:  plain\_text += matrix[row1][col2] + matrix[row2][col1]  return plain\_text  print("Cipher Text:", playfair\_encrypt(plain\_text, keyword))  print("Plain Text:", playfair\_decrypt(playfair\_encrypt(plain\_text, keyword), keyword)) |
| **Output:** |  |
| **Output (Cryptool):** |  |

**Cryptanalysis :**

**Here are some of the main cipher analysis of Playfair cipher:-**

**1.Frequency Analysis**:

In the Playfair cipher, digraphs (pairs of letters) are encrypted rather than individual letters.

This makes frequency analysis more challenging than with monoalphabetic ciphers like the

Caesar cipher.

However, common digraphs in the plaintext may still appear with some frequency in the

ciphertext. For example, if "TH" is a common digraph in English text, it may still appear as a

frequently occurring digraph in the ciphertext.

**2.** **Patterns and Repeating Digraphs**:

The Playfair cipher may create repeating patterns in the ciphertext, particularly when the same digraphs are used multiple times in the plaintext. For example, if "HELLO" is encrypted, and "L" appears twice, the corresponding ciphertext

digraphs for "L" might be the same.

**3**. **Known Plaintext Attack**:

If an attacker knows or can guess part of the plaintext and its corresponding ciphertext, they

can gain insights into the key.

For instance, if "HELLO" encrypts to "KEMON," the attacker knows that "H" maps to "K"

and "E" maps to "E," which provides some information about the key matrix.

**4. Brute Force**:

• A brute-force attack on the Playfair cipher involves systematically trying all possible keys.

• The Playfair key space is somewhat reduced compared to monoalphabetic ciphers but still

requires a large number of attempts, making it computationally expensive to break through

brute force.

**5**. **Key Patterns**:

If there are patterns or weaknesses in how the key matrix is generated, it can make the cipher

easier to analyse.

For example, if the key matrix is poorly chosen and lacks sufficient randomness, it could make the cipher vulnerable to attack.

**Applications:**

**1.Puzzles and Games**: Playfair ciphers can be used in puzzles, riddles, or educational games.

Participants can enjoy the challenge of decoding messages encrypted with this classical cipher.

**2.Educational Tools**: Teachers and educators can use Playfair ciphers to demonstrate basic

concepts of cryptography and encourage students to explore encryption techniques.

**3.Historical Reenactments**: In historical reenactments or period-themed events, the Playfair

cipher can be employed to add an authentic touch to communications, as it was historically

used in the 19th and early 20th centuries.

**4.Personal Messages**: For casual or non-sensitive communication between friends or family

members, Playfair ciphers can be used to add a fun and personalized touch to messages.

**5.Escape Rooms**: Escape room challenges often involve deciphering codes and ciphers to

progress in the game. Playfair ciphers can be one of the elements in such puzzles.

**References :**

1. [https://www.cryptool.org](https://www.cryptool.org/en/cto/caesar)
2. <https://www.geeksforgeeks.org/>

**Experiment No : 07**

**Aim :** Study and implement a program for n-gram Hill cipher.

**Introduction :** A standard Hill cipher uses a square matrix as its key to perform encryption and decryption. The key matrix's size is determined by the size of the block used, typically 2x2 for bigrams. N-gram Hill ciphers generalize this concept by allowing the use of larger n-grams, such as trigrams or quadrigrams, in the encryption process.

The key idea behind N-gram Hill ciphers is to consider a larger context of characters when encrypting and decrypting messages. This increases the complexity of the cipher and makes it more resistant to certain types of cryptanalysis, as frequency analysis becomes more challenging with larger n-grams.

The encryption and decryption processes in an N-gram Hill cipher involve matrix operations over finite fields, typically modulo the size of the character set (e.g., 26 for the English alphabet). The key matrix determines the exact operations required for both encryption and decryption. The security of the cipher relies on the key remaining secret.

In practice, N-gram Hill ciphers are more computationally intensive and require more extensive key management compared to standard Hill ciphers. However, they offer stronger security, especially against frequency-based attacks.

**Program (Source Code):**

|  |  |
| --- | --- |
| **Code:** | import numpy as np  def matrix\_inverse(matrix, mod):  det = int(np.linalg.det(matrix))  det\_inv = pow(det, -1, mod)  adj\_matrix = np.round(det\_inv \* np.linalg.inv(matrix)).astype(int) % mod  return adj\_matrix  def encrypt(plaintext, key\_matrix):  plaintext = plaintext.replace(" ", "").upper()  key\_size = len(key\_matrix)  plaintext\_len = len(plaintext)  if plaintext\_len % key\_size != 0:  padding = key\_size - (plaintext\_len % key\_size)  plaintext += "X" \* padding  ciphertext = ""  for i in range(0, len(plaintext), key\_size):  block = plaintext[i:i + key\_size]  block\_vector = np.array([ord(char) - ord('A') for char in block])  encrypted\_block = np.dot(key\_matrix, block\_vector) % 26  ciphertext += ''.join([chr(char + ord('A')) for char in encrypted\_block])  return ciphertext  def decrypt(ciphertext, key\_matrix):  key\_matrix\_inv = matrix\_inverse(key\_matrix, 26)  key\_size = len(key\_matrix)  plaintext = ""  for i in range(0, len(ciphertext), key\_size):  block = ciphertext[i:i + key\_size]  block\_vector = np.array([ord(char) - ord('A') for char in block])  decrypted\_block = np.dot(key\_matrix\_inv, block\_vector) % 26  plaintext += ''.join([chr(char + ord('A')) for char in decrypted\_block])  return plaintext  def main():  choice = input("Choose 'E' for Encryption or 'D' for Decryption: ").upper()  if choice not in ('E', 'D'):  print("Invalid choice.")  return  key\_size = int(input("Enter the size of the key matrix: "))  if key\_size <= 1:  print("Key matrix size should be greater than 1.")  return  key\_matrix = []  print("Enter the key matrix elements row by row (separate by space):")  for \_ in range(key\_size):  row = input().split()  if len(row) != key\_size:  print(f"Each row of the key matrix should have {key\_size} elements.")  return  key\_matrix.append([int(x) % 26 for x in row])  key\_matrix = np.array(key\_matrix)  if key\_matrix.shape[0] != key\_matrix.shape[1]:  print("Key matrix should be square.")  return  det = int(np.linalg.det(key\_matrix))  mod = 26  if np.gcd(det, mod) != 1:  print("The key matrix is not invertible. Please choose a different matrix.")  return  if choice == 'E':  plaintext = input("Enter the plaintext: ")  ciphertext = encrypt(plaintext, key\_matrix)  print("Ciphertext:", ciphertext)  else:  ciphertext = input("Enter the ciphertext: ")  decrypted\_text = decrypt(ciphertext, key\_matrix)  print("Decrypted Text:", decrypted\_text)  if \_name\_ == "\_main\_":  main() |
| **Output:** |  |
| **Output (Cryptool):** |  |

**Cryptanalysis :**

**Here are some of the main cipher analysis of Playfair cipher:-**

**Frequency Analysis:** This technique involves analyzing the frequency of n-grams (typically single letters in the English alphabet) in the ciphertext. The frequencies of n-grams in English text follow a specific pattern. For example, 'E' is the most common single letter in English text, followed by 'T', 'A,' and 'O'. If you can identify the most frequent n-grams in the ciphertext, you can make educated guesses about the key. This method becomes more complex when dealing with larger n-grams.

**Known Plaintext Attack:** If you have access to the corresponding plaintext and ciphertext, you can use this information to derive the key. For example, if you know that a certain word exists in both the plaintext and ciphertext, you can determine the key value for that portion of the key matrix.

**Chosen Plaintext Attack:** In a chosen plaintext attack, the attacker has the ability to choose plaintexts and encrypt them with the unknown key. By analyzing the ciphertext generated from these chosen plaintexts, the attacker can make inferences about the key. For example, if the attacker encrypts 'A' and observes the corresponding ciphertext, they can potentially deduce part of the key.

**Brute Force Attack:** If the size of the key matrix is small enough, an attacker can systematically try all possible combinations of keys. This is known as a brute force attack. The larger the key matrix, the more computationally intensive this attack becomes.

**Applications:**

**Secure Communication:** Hill ciphers can be used to encrypt and decrypt messages, providing confidentiality in secure communication between parties. For example, they can be used in secure email communication.

**Secure File Storage:** Hill ciphers can encrypt files and data at rest, ensuring that stored information remains confidential and protected from unauthorized access. This can be applied to file storage systems and databases.

**Cryptography Education:** Hill ciphers are often used in educational settings to teach the principles of classical encryption and matrix-based encryption methods. Students can learn about the fundamentals of encryption and decryption through Hill ciphers.

**References :**

1. [https://www.cryptool.org](https://www.cryptool.org/en/cto/caesar)
2. <https://www.geeksforgeeks.org/>

**Experiment No : 08**

**Aim :**.

**Introduction :**

**Program (Source Code):**

|  |  |
| --- | --- |
| **Code:** |  |
| **Output:** |  |
| **Output (Cryptool):** |  |

**Cryptanalysis :**

**Applications:**

**References :**

**Experiment No : 09**

**Aim :** Study and implement RSA encryption and decryption functions.

**Introduction :** The RSA algorithm is a public-key signature algorithm developed by Ron Rivest, Adi Shamir, and Leonard Adleman. Their paper was first published in 1977, and the algorithm uses logarithmic functions to keep the working complex enough to withstand brute force and streamlined enough to be fast post-deployment. The image below shows it verifies the digital signatures using RSA methodology.

When using RSA for encryption and decryption of general data, it reverses the key set usage.

Unlike signature verification, it uses the receiver’s public key to encrypt the data, and it uses the receiver’s private key in decrypting the data. Thus, there is no need to exchange any keys in this scenario.

There are two broad components when it comes to RSA cryptography, they are:

* Key Generation: Generating the keys to be used for encrypting and decrypting the data to be exchanged.
* Encryption/Decryption Function: The steps that need to be run when scrambling and recovering the data.

**What Are Digital Signatures?**

Digital signatures serve the purpose of authentication and verification of documents and files.

This is crucial to prevent tampering during official papers’ transmission and prevent digital manipulation or forgery.

**Program (Source Code):**

|  |  |
| --- | --- |
| **Code:** | from math import gcd  def RSA(p, q, message):  # Calculate n  n = p \* q   # Calculate totient, t  t = (p - 1) \* (q - 1)   # Select public key, e  for i in range(2, t):  if gcd(i, t) == 1:  e = i  break   # Select private key, d  j = 2 # Start from 2, as 1 is not suitable for d  while True:  if (j \* e) % t == 1:  d = j  break  j += 1   # Performing encryption  ct = (message \*\* e) % n  print(f"Encrypted message is {ct}")   # Performing decryption  mes = (ct \*\* d) % n  print(f"Decrypted message is {mes}")  # Take user input for p, q, and message p = int(input("Enter the value of p: ")) q = int(input("Enter the value of q: ")) message = int(input("Enter the message to be encrypted: "))  # Test the RSA function with user input RSA(p, q, message) |
| **Output:** |  |

**Cryptanalysis :**

**Here are some of the main cipher analysis of Playfair cipher:-**

1. **Small Key Sizes:** If RSA is implemented with very small key sizes (e.g., 2-digit primes), it becomes vulnerable to brute force attacks.

1. **Lack of Modulus Complexity:** Small key sizes make it easier for an attacker to factor the modulus, **n**, and thereby determine the prime factors, **p** and **q**.

1. **Simple Factoring:** In this scenario, the attacker can factor **n** into **p** and **q** using basic trial division without the need for complex factorization algorithms.

**Applications:**

1. **Secure Data Transmission:** RSA is widely used for encrypting data in transit, securing sensitive information during online transactions and communications.
2. **Digital Signatures:** RSA is employed for creating digital signatures, verifying the authenticity and integrity of digital documents, emails, and software.

1. **Secure Web Browsing:** RSA is a fundamental part of SSL/TLS protocols, ensuring secure and encrypted web connections (HTTPS).

1. **Secure Authentication:** RSA-based tokens and smart cards provide secure two-factor authentication for online accounts and systems.

1. **Key Exchange:** RSA facilitates secure key exchange in asymmetric encryption, allowing secure communication over an insecure channel.

**References :**

1. <https://www.javatpoint.com/rsa-encryption-algorithm>

**Experiment No : 10**

**Aim :** Use RSA for generation and verification of digital signature on file.

**Introduction :**

Sender's Authentication Experiment:

Sender's authentication is a critical component of secure communication systems. It ensures that the entity claiming to be the sender of a message is, in fact, the legitimate sender and not an imposter. In the digital age, where information is transmitted electronically and online interactions are commonplace, verifying the authenticity of the sender is essential to prevent unauthorized access, data breaches, and identity theft.

In this experiment, we aim to explore various methods and techniques for sender's authentication. We will investigate how these methods can be employed to confirm the identity of the message sender, which is fundamental in building trust and ensuring the integrity of the communication process. Through this experiment, we hope to gain insights into the strengths and weaknesses of different sender's authentication mechanisms, ultimately contributing to the development of more secure communication protocols and systems.

Message Confidentiality and Sender's Authentication Experiment:

In the realm of secure communication, ensuring not only the confidentiality of messages but also the authentication of the sender is of paramount importance. The need for robust sender's authentication mechanisms is evident in scenarios where sensitive information is exchanged, be it in the realm of financial transactions, healthcare, or any other domain where the privacy and security of information are critical.

This experiment is designed to investigate methods and techniques that simultaneously address message confidentiality and sender's authentication. We aim to explore how encryption, cryptographic protocols, and digital signatures can be integrated to provide a comprehensive solution that safeguards both the privacy of the message content and the confirmation of the sender's identity. By doing so, we aim to develop a deeper understanding of the complexities involved in achieving both these security objectives and to assess the practicality of implementing such a system in real-world communication channels.

Through this experiment, we seek to contribute to the ongoing efforts to enhance the security of digital communication and to enable secure, trustworthy exchanges of information in an increasingly interconnected and data-driven world.

**Program (Source Code):**

|  |  |
| --- | --- |
| **Code:** | import math import random from sympy import isprime  def generate\_keypair(p, q):  n = p \* q  phi = (p - 1) \* (q - 1)  while True:  e = random.randrange(1, phi)  if math.gcd(e, phi) == 1:  break  d = mod\_inverse(e, phi)  return ((n, e), (n, d))  def mod\_inverse(a, m):  m0, x0, x1 = m, 0, 1  while a > 1:  q = a // m  m, a = a % m, m  x0, x1 = x1 - q \* x0, x0  return x1 + m0 if x1 < 0 else x1  def sign\_message(private\_key, message):  n, d = private\_key  signature = pow(message, d, n)  return signature  def verify\_signature(public\_key, message, signature):  n, e = public\_key  decrypted\_signature = pow(signature, e, n)  return message == decrypted\_signature  def is\_prime(num):  if isprime(num):  return True  else:  return False  def generate\_prime(bits):  while True:  num = random.getrandbits(bits)  if is\_prime(num):  return num  p = generate\_prime(10) q = generate\_prime(10) public\_key, private\_key = generate\_keypair(p, q) print(f'Public key: {public\_key} \nPrivate key: {private\_key}') message = 4 print("Original message:", message) signature = sign\_message(private\_key, message) print("Signature:", signature)  if verify\_signature(public\_key, message, signature):  print("Signature is valid.") else:  print("Signature is invalid.")  import math import random from sympy import isprime   def generate\_keypair(p, q):  n = p \* q  phi = (p - 1) \* (q - 1)   while True:  e = random.randrange(1, phi)  if math.gcd(e, phi) == 1:  break  d = mod\_inverse(e, phi)  return ((n, e), (n, d))   def mod\_inverse(a, m):  m0, x0, x1 = m, 0, 1  while a > 1:  q = a // m  m, a = a % m, m  x0, x1 = x1 - q \* x0, x0  return x1 + m0 if x1 < 0 else x1   def sign\_message(private\_key, message):  n, d = private\_key  signature = pow(message, d, n)  return signature   def verify\_signature(public\_key, private\_key\_receiver, encrypted\_value):  n\_receiver, d\_receiver = private\_key\_receiver  n\_sender, e\_sender = public\_key  decrypted\_value = pow(encrypted\_value, d\_receiver, n\_receiver)  decrypted\_value = str(decrypted\_value)  message = int(decrypted\_value[0])  signature = int(decrypted\_value[1:])  print("Message:", message)  decrypted\_signature = pow(signature, e\_sender, n\_sender)  return message == decrypted\_signature   def encrypt(public\_key, plaintext):  n, e = public\_key  c = pow(plaintext, e, n)  return c   def decrypt(private\_key, ciphertext):  n, d = private\_key  m = pow(ciphertext, d, n)  return m   def is\_prime(num):  if isprime(num):  return True  else:  return False   def generate\_prime(bits):  while True:  num = random.getrandbits(bits)  if is\_prime(num):  return num   p = generate\_prime(16) q = generate\_prime(16) public\_key, private\_key = generate\_keypair(p, q) print(f"Public key Sender: {public\_key}\nPrivate key Sender: {private\_key}") message = 4 print("Original message:", message) signature = sign\_message(private\_key, message) print("Signature:", signature)  concatenated\_str = str(message) + str(signature) concatenated\_value = int(concatenated\_str) print("Concatenated value:", concatenated\_value)  p\_receiver = generate\_prime(64) q\_receiver = generate\_prime(64) public\_key\_receiver, private\_key\_receiver = generate\_keypair(p\_receiver, q\_receiver)  print(f"Public key Receiver: {public\_key\_receiver}\nPrivate key Receiver: {private\_key\_receiver}") encrypted\_value = encrypt(public\_key\_receiver, concatenated\_value) print("Encrypted value:", encrypted\_value)  if verify\_signature(public\_key, private\_key\_receiver, encrypted\_value):  print("Signature is valid.") else:  print("Signature is invalid.") |
| **Output:** |  |

**Cryptanalysis :**

1. **Brute Force Attack:** An attacker could attempt to break the RSA encryption by trying all possible private keys. The security of RSA depends on the difficulty of factoring a large composite number n (the modulus). As long as the modulus is sufficiently large, this should be computationally infeasible.
2. **Integer Factorization:** RSA's security is based on the assumption that it is hard to factor the modulus n into its prime factors p and q. Cryptanalysis methods like Pollard's Rho, Quadratic Sieve, and General Number Field Sieve (GNFS) have been developed to try and factor large integers. The larger the prime numbers p and q, the more secure RSA becomes against these attacks.
3. **Chosen Plaintext Attack:** If an attacker can choose plaintexts and observe their corresponding ciphertexts, they may gain information about the private key.
4. **Random Number Generation:** Weak random number generation can lead to vulnerabilities in RSA. If the random numbers used in key generation are predictable, it could make the private key easier to derive.

**Applications:**

**Applications of Sender's Authentication:**

1. **Email and Messaging Services**: Sender's authentication is used to verify the identity of email senders and message originators, reducing the risk of phishing and spam.

1. **Online Banking and Financial Transactions**: It ensures the legitimate identity of users during online banking, preventing unauthorized access to financial accounts.

1. **Access Control Systems**: Sender's authentication is essential in physical access control systems, such as card readers and biometric security systems, to grant access only to authorized individuals.

**Applications of Message Confidentiality and Sender's Authentication:**

1. **Secure Messaging Applications**: These applications ensure that messages are both confidential and authenticated, protecting the privacy of users.

1. **Government and Military Secure Communication**: Used in classified communications where both sender's authentication and message confidentiality are critical.

1. **Healthcare Data Protection**: It safeguards patient data during transmission and ensures that healthcare providers are genuine.

**References :**

<https://www.geeksforgeeks.org/pgp>[-authentication-and-confidentiality/](https://www.geeksforgeeks.org/pgp-authentication-and-confidentiality/)