****

FACULTY OF ENGINEERING AND TECHNOLOGY

BACHELOR OF TECHNOLOGY

**INFORMATION AND**

**NETWORK SECURITY**

**(203105311)**

7th SEMESTER

7A13

**NAME**: **VISHAL SURESH UMAVANE**

**ENROLLMENT NUMBER**: **210306105201**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SR No.** | **Practical List** | **Start Date** | **End Date** | **Sign** | **Marks** |
| 1 | Implement Caesar cipher encryption-decryption |  |  |  |  |
| 2 | Implement Monoalphabetic cipher encryption-decryption |  |  |  |  |
| 3 | Implement Playfair cipher encryption-decryption |  |  |  |  |
| 4 | Implement Polyalphabetic cipher encryption-decryption |  |  |  |  |
| 5 | Implement Hill cipher encryption-decryption |  |  |  |  |
| 6 | Implement Simple Transposition encryption-decryption |  |  |  |  |
| 7 | Implement One time pad encryption-decryption |  |  |  |  |
| 8 | Implement Diffi-Hellmen Key exchange Method |  |  |  |  |
| 9 | Implement RSA encryption-decryption algorithm |  |  |  |  |
| 10 | Demonstrate working of Digital Signature using Cryptool |  |  |  |  |

**Index Table**

**Practical 1**

**Aim :** Implement Caesar cipher encryption-decryption

**Theory** : The Caesar cipher is a basic form of encryption where each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. It's a type of substitution cipher, named after Julius Caesar who reportedly used it to protect his military messages.

**Encryption Process**

1. Choose a shift value: This is the number of positions each letter will be shifted
2. Shift each letter: For each letter in the plaintext, replace it with the letter that is the shift
3. value positions down the alphabet. Wrap around from 'Z' to 'A' if necessary.

**Decryption Process**

1. Determine the shift value: You need to know the shift value used for encryption.
2. Shift each letter back: For each letter in the ciphertext, shift it back by the same number of positions used for encryption.

**Implementation :**

#include <iostream>

#include <string>

// Function to encrypt the text using Caesar Cipher

std::string encryptCaesarCipher(std::string text, int shift) {

std::string result = "";

// Traverse text

for (int i = 0; i < text.length(); i++) {

char ch = text[i];

// Encrypt uppercase letters

if (isupper(ch))

result += char(int(ch + shift - 65) % 26 + 65);

// Encrypt lowercase letters

else if (islower(ch))

result += char(int(ch + shift - 97) % 26 + 97);

// Encrypt digits

else if (isdigit(ch))

result += char(int(ch + shift - 48) % 10 + 48);

// Leave other characters unchanged

else

result += ch;

}

return result;

}

// Function to decrypt the text using Caesar Cipher

std::string decryptCaesarCipher(std::string text, int shift) {

std::string result = "";

// Traverse text

for (int i = 0; i < text.length(); i++) {

char ch = text[i];

// Decrypt uppercase letters

if (isupper(ch))

result += char(int(ch - shift - 65 + 26) % 26 + 65);

// Decrypt lowercase letters

else if (islower(ch))

result += char(int(ch - shift - 97 + 26) % 26 + 97);

// Decrypt digits

else if (isdigit(ch))

result += char(int(ch - shift - 48 + 10) % 10 + 48);

// Leave other characters unchanged

else

result += ch;

}

return result;

}

int main() {

std::string text;

int shift;

char choice;

std::cout << "Enter the text: ";

std::getline(std::cin, text);

std::cout << "Enter the shift value: ";

std::cin >> shift;

std::cout << "Do you want to (e)ncrypt or (d)ecrypt? ";

std::cin >> choice;

if (choice == 'e' || choice == 'E') {

std::cout << "Encrypted text: " << encryptCaesarCipher(text, shift) << std::endl;

} else if (choice == 'd' || choice == 'D') {

std::cout << "Decrypted text: " << decryptCaesarCipher(text, shift) << std::endl;

} else {

std::cout << "Invalid choice" << std::endl;

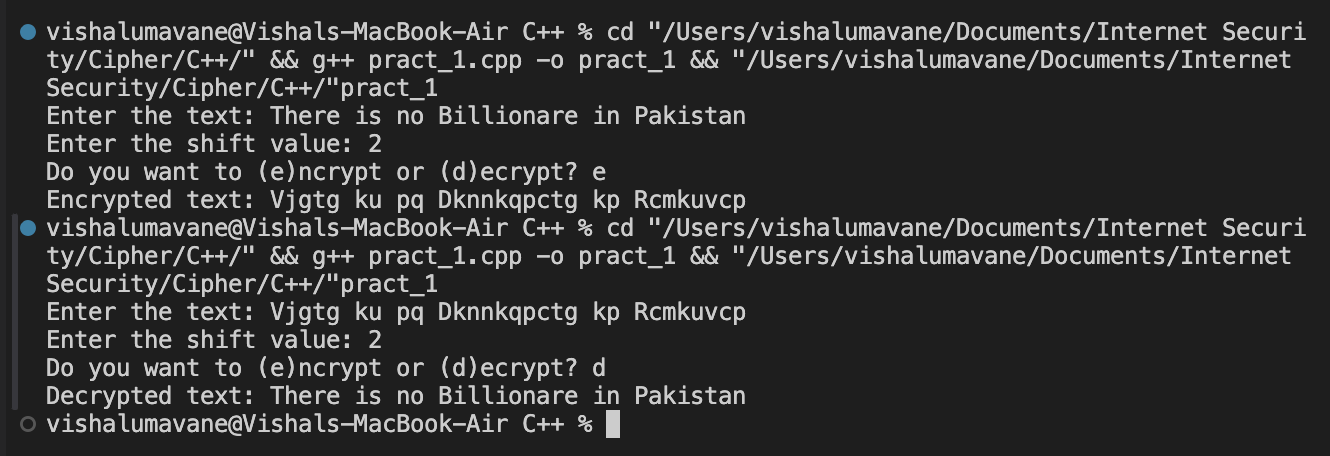
}

return 0;

}

**Outputs :**

Encryption & Decryption



**Practical 2**

**Aim**: Implement Monoalphabetic cipher encryption-decryption

**Implementation :**

#include <iostream>

#include <unordered\_map>

#include <string>

// Function to generate the encryption and decryption maps based on the key

void generateMaps(std::string key, std::unordered\_map<char, char>& encryptMap, std::unordered\_map<char, char>& decryptMap) {

std::string alphabet = "abcdefghijklmnopqrstuvwxyz";

for (int i = 0; i < alphabet.length(); i++) {

encryptMap[alphabet[i]] = key[i];

decryptMap[key[i]] = alphabet[i];

}

}

// Function to encrypt the text using Monoalphabetic Cipher

std::string encryptMonoalphabeticCipher(std::string text, std::unordered\_map<char, char>& encryptMap) {

std::string result = "";

for (char ch : text) {

if (isalpha(ch)) {

char lower = tolower(ch);

result += isupper(ch) ? toupper(encryptMap[lower]) : encryptMap[lower];

} else {

result += ch;

}

}

return result;

}

// Function to decrypt the text using Monoalphabetic Cipher

std::string decryptMonoalphabeticCipher(std::string text, std::unordered\_map<char, char>& decryptMap) {

std::string result = "";

for (char ch : text) {

if (isalpha(ch)) {

char lower = tolower(ch);

result += isupper(ch) ? toupper(decryptMap[lower]) : decryptMap[lower];

} else {

result += ch;

}

}

return result;

}

int main() {

std::string text, key;

char choice;

std::unordered\_map<char, char> encryptMap, decryptMap;

// Prompt for key

std::cout << "Enter the 26-letter key for the cipher (e.g., QWERTYUIOPASDFGHJKLZXCVBNM): ";

std::cin >> key;

// Generate maps

generateMaps(key, encryptMap, decryptMap);

// Clear the input buffer

std::cin.ignore();

// Prompt for text and choice

std::cout << "Enter the text: ";

std::getline(std::cin, text);

std::cout << "Do you want to (e)ncrypt or (d)ecrypt? ";

std::cin >> choice;

if (choice == 'e' || choice == 'E') {

std::cout << "Encrypted text: " << encryptMonoalphabeticCipher(text, encryptMap) << std::endl;

} else if (choice == 'd' || choice == 'D') {

std::cout << "Decrypted text: " << decryptMonoalphabeticCipher(text, decryptMap) << std::endl;

} else {

std::cout << "Invalid choice" << std::endl;

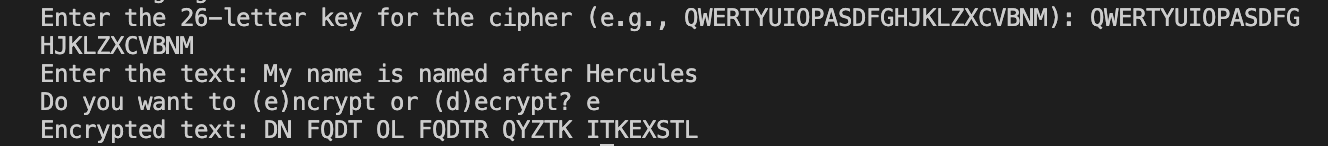
}

return 0;

}

**Outputs :**

Encryption



**Practical 3**

**Aim :** Implement Playfair cipher encryption-decryption

**Implementation :**

#include <iostream>

#include <vector>

#include <algorithm>

#include <cctype>

#include <string>

#include <unordered\_set>

// Function to generate the Playfair matrix based on the key

void generateMatrix(std::string key, char matrix[5][5]) {

std::string keyString = "";

std::unordered\_set<char> usedChars;

// Add key characters to keyString, removing duplicates and ignoring 'J'

for (char ch : key) {

ch = toupper(ch);

if (ch == 'J') ch = 'I';

if (usedChars.find(ch) == usedChars.end() && isalpha(ch)) {

keyString += ch;

usedChars.insert(ch);

}

}

// Add remaining letters to keyString

for (char ch = 'A'; ch <= 'Z'; ch++) {

if (ch == 'J') continue;

if (usedChars.find(ch) == usedChars.end()) {

keyString += ch;

usedChars.insert(ch);

}

}

// Fill the matrix

int k = 0;

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 5; j++) {

matrix[i][j] = keyString[k++];

}

}

}

// Function to find the position of a character in the matrix

void findPosition(char matrix[5][5], char ch, int &row, int &col) {

if (ch == 'J') ch = 'I'; // Treat 'J' as 'I'

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 5; j++) {

if (matrix[i][j] == ch) {

row = i;

col = j;

return;

}

}

}

}

// Function to process text by removing non-alphabetic characters and handling duplicate letters in digraphs

std::string processText(std::string text) {

std::string result = "";

for (char ch : text) {

if (isalpha(ch)) {

ch = toupper(ch);

result += (ch == 'J') ? 'I' : ch;

}

}

// Handle duplicate letters in digraphs

for (size\_t i = 0; i < result.length(); i += 2) {

if (i + 1 < result.length() && result[i] == result[i + 1]) {

result.insert(i + 1, "X");

}

}

// If the processed text has an odd number of characters, add 'X' at the end

if (result.length() % 2 != 0) {

result += 'X';

}

return result;

}

// Function to encrypt the text using Playfair Cipher

std::string encryptPlayfairCipher(std::string text, char matrix[5][5]) {

std::string result = "";

text = processText(text);

for (size\_t i = 0; i < text.length(); i += 2) {

char first = text[i];

char second = text[i + 1];

int row1, col1, row2, col2;

findPosition(matrix, first, row1, col1);

findPosition(matrix, second, row2, col2);

if (row1 == row2) {

result += matrix[row1][(col1 + 1) % 5];

result += matrix[row2][(col2 + 1) % 5];

} else if (col1 == col2) {

result += matrix[(row1 + 1) % 5][col1];

result += matrix[(row2 + 1) % 5][col2];

} else {

result += matrix[row1][col2];

result += matrix[row2][col1];

}

}

return result;

}

// Function to decrypt the text using Playfair Cipher

std::string decryptPlayfairCipher(std::string text, char matrix[5][5]) {

std::string result = "";

text = processText(text);

for (size\_t i = 0; i < text.length(); i += 2) {

char first = text[i];

char second = text[i + 1];

int row1, col1, row2, col2;

findPosition(matrix, first, row1, col1);

findPosition(matrix, second, row2, col2);

if (row1 == row2) {

result += matrix[row1][(col1 + 4) % 5];

result += matrix[row2][(col2 + 4) % 5];

} else if (col1 == col2) {

result += matrix[(row1 + 4) % 5][col1];

result += matrix[(row2 + 4) % 5][col2];

} else {

result += matrix[row1][col2];

result += matrix[row2][col1];

}

}

return result;

}

int main() {

std::string text, key;

char choice;

char matrix[5][5];

// Prompt for key

std::cout << "Enter the key for the cipher: ";

std::getline(std::cin, key);

// Generate matrix

generateMatrix(key, matrix);

// Prompt for text and choice

std::cout << "Enter the text: ";

std::getline(std::cin, text);

std::cout << "Do you want to (e)ncrypt or (d)ecrypt? ";

std::cin >> choice;

if (choice == 'e' || choice == 'E') {

std::cout << "Encrypted text: " << encryptPlayfairCipher(text, matrix) << std::endl;

} else if (choice == 'd' || choice == 'D') {

std::cout << "Decrypted text: " << decryptPlayfairCipher(text, matrix) << std::endl;

} else {

std::cout << "Invalid choice" << std::endl;

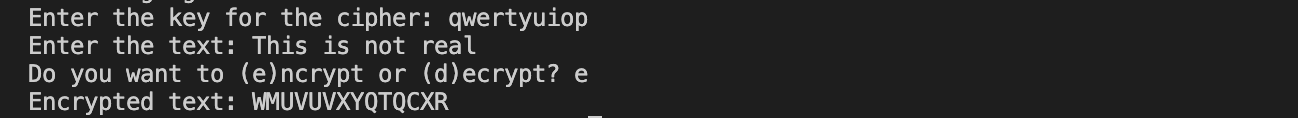
}

return 0;

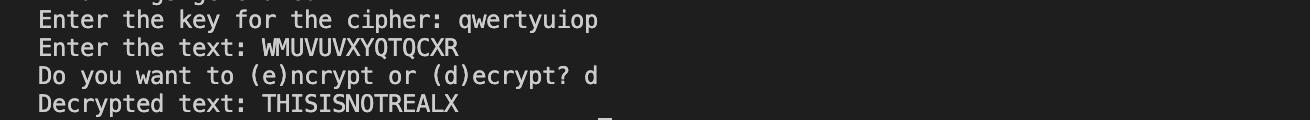
}

**Outputs :**

Encryption



Decryption



**Practical 4**

**Aim :** Implement Polyalphabetic cipher encryption-decryption

**Implementation :**

#include <iostream>

#include <string>

// Function to extend the key to match the length of the text

std::string extendKey(const std::string &text, const std::string &key) {

std::string extendedKey = key;

int textLength = text.length();

int keyLength = key.length();

for (int i = 0; i < textLength - keyLength; i++) {

extendedKey += key[i % keyLength];

}

return extendedKey;

}

// Function to encrypt the text using Vigenère Cipher

std::string encryptVigenereCipher(const std::string &text, const std::string &key) {

std::string encryptedText = "";

std::string extendedKey = extendKey(text, key);

for (size\_t i = 0; i < text.length(); i++) {

char ch = text[i];

if (isalpha(ch)) {

char base = isupper(ch) ? 'A' : 'a';

char keyCh = toupper(extendedKey[i]) - 'A';

encryptedText += (ch - base + keyCh) % 26 + base;

} else {

encryptedText += ch;

}

}

return encryptedText;

}

// Function to decrypt the text using Vigenère Cipher

std::string decryptVigenereCipher(const std::string &text, const std::string &key) {

std::string decryptedText = "";

std::string extendedKey = extendKey(text, key);

for (size\_t i = 0; i < text.length(); i++) {

char ch = text[i];

if (isalpha(ch)) {

char base = isupper(ch) ? 'A' : 'a';

char keyCh = toupper(extendedKey[i]) - 'A';

decryptedText += (ch - base - keyCh + 26) % 26 + base;

} else {

decryptedText += ch;

}

}

return decryptedText;

}

int main() {

std::string text, key;

char choice;

// Prompt for key

std::cout << "Enter the key for the cipher: ";

std::getline(std::cin, key);

// Prompt for text and choice

std::cout << "Enter the text: ";

std::getline(std::cin, text);

std::cout << "Do you want to (e)ncrypt or (d)ecrypt? ";

std::cin >> choice;

if (choice == 'e' || choice == 'E') {

std::cout << "Encrypted text: " << encryptVigenereCipher(text, key) << std::endl;

} else if (choice == 'd' || choice == 'D') {

std::cout << "Decrypted text: " << decryptVigenereCipher(text, key) << std::endl;

} else {

std::cout << "Invalid choice" << std::endl;

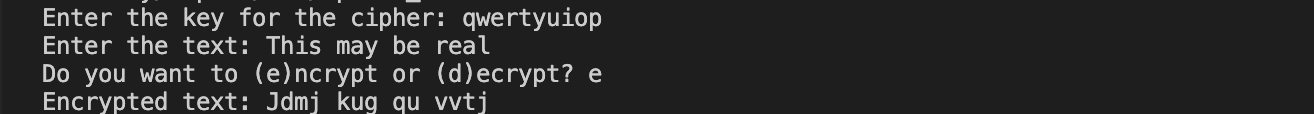
}

return 0;

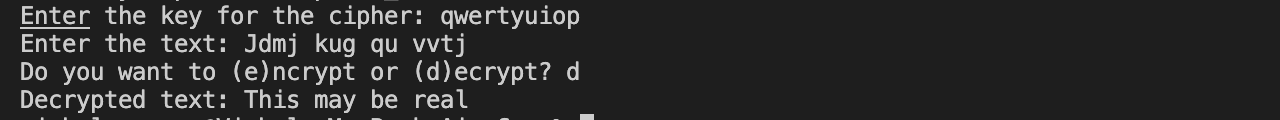
}

**Outputs :**

Encryption



Decryption



**Practical 5**

**Aim** : Implement hill cipher encryption and decryption

**Implementation :**

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

// Function to generate the key matrix from the key string

void getKeyMatrix(string key, int keyMatrix[][3]) {

int k = 0;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

keyMatrix[i][j] = (key[k]) % 65;

k++;

}

}

}

// Function to get the cofactor matrix

void getCofactor(int matrix[3][3], int temp[3][3], int p, int q, int n) {

int i = 0, j = 0;

for (int row = 0; row < n; row++) {

for (int col = 0; col < n; col++) {

if (row != p && col != q) {

temp[i][j++] = matrix[row][col];

if (j == n - 1) {

j = 0;

i++;

}

}

}

}

}

// Function to calculate the determinant of the matrix

int determinant(int matrix[3][3], int n) {

int det = 0;

if (n == 1) return matrix[0][0];

int temp[3][3];

int sign = 1;

for (int i = 0; i < n; i++) {

getCofactor(matrix, temp, 0, i, n);

det += sign \* matrix[0][i] \* determinant(temp, n - 1);

sign = -sign;

}

return det;

}

// Function to find adjoint of a matrix

void adjoint(int matrix[3][3], int adj[3][3]) {

if (3 == 1) {

adj[0][0] = 1;

return;

}

int sign = 1, temp[3][3];

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

getCofactor(matrix, temp, i, j, 3);

sign = ((i + j) % 2 == 0) ? 1 : -1;

adj[j][i] = (sign) \* (determinant(temp, 3 - 1));

}

}

}

// Function to find the modular inverse of a number

int modInverse(int a, int m) {

a = a % m;

for (int x = 1; x < m; x++) {

if ((a \* x) % m == 1)

return x;

}

return -1;

}

// Function to find the inverse of the key matrix

bool inverseKeyMatrix(int keyMatrix[3][3], int inverse[3][3]) {

int det = determinant(keyMatrix, 3);

int invDet = modInverse(det, 26);

if (invDet == -1) {

cout << "Inverse doesn't exist";

return false;

}

int adj[3][3];

adjoint(keyMatrix, adj);

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

inverse[i][j] = (adj[i][j] \* invDet) % 26;

if (inverse[i][j] < 0) inverse[i][j] += 26;

}

}

return true;

}

// Function to encrypt the message

void encrypt(int cipherMatrix[][1], int keyMatrix[][3], int messageVector[][1]) {

for (int i = 0; i < 3; i++) {

cipherMatrix[i][0] = 0;

for (int j = 0; j < 3; j++) {

cipherMatrix[i][0] += keyMatrix[i][j] \* messageVector[j][0];

}

cipherMatrix[i][0] = cipherMatrix[i][0] % 26;

}

}

// Function to decrypt the message

void decrypt(int plainMatrix[][1], int inverseKeyMatrix[][3], int cipherVector[][1]) {

for (int i = 0; i < 3; i++) {

plainMatrix[i][0] = 0;

for (int j = 0; j < 3; j++) {

plainMatrix[i][0] += inverseKeyMatrix[i][j] \* cipherVector[j][0];

}

plainMatrix[i][0] = plainMatrix[i][0] % 26;

}

}

// Function to implement Hill Cipher encryption

void HillCipherEncrypt(string message, string key) {

int keyMatrix[3][3];

getKeyMatrix(key, keyMatrix);

// Pad the message to make its length a multiple of 3

while (message.length() % 3 != 0) {

message += 'X'; // Padding with 'X'

}

string CipherText;

for (size\_t i = 0; i < message.length(); i += 3) {

int messageVector[3][1];

for (int j = 0; j < 3; j++) {

messageVector[j][0] = (message[i + j]) % 65;

}

int cipherMatrix[3][1];

encrypt(cipherMatrix, keyMatrix, messageVector);

for (int j = 0; j < 3; j++) {

CipherText += cipherMatrix[j][0] + 65;

}

}

// Print the ciphertext

cout << "Ciphertext: " << CipherText << endl;

}

// Function to implement Hill Cipher decryption

void HillCipherDecrypt(string ciphertext, string key) {

int keyMatrix[3][3];

getKeyMatrix(key, keyMatrix);

int inverseMatrix[3][3];

if (!inverseKeyMatrix(keyMatrix, inverseMatrix)) {

cout << "Key matrix is not invertible. Decryption aborted." << endl;

return;

}

string PlainText;

for (size\_t i = 0; i < ciphertext.length(); i += 3) {

int cipherVector[3][1];

for (int j = 0; j < 3; j++) {

cipherVector[j][0] = (ciphertext[i + j]) % 65;

}

int plainMatrix[3][1];

decrypt(plainMatrix, inverseMatrix, cipherVector);

for (int j = 0; j < 3; j++) {

PlainText += plainMatrix[j][0] + 65;

}

}

// Print the plaintext

cout << "Plaintext: " << PlainText << endl;

}

int main() {

string message;

string key = "GYBNQKURP";

cout << "Enter the message: ";

getline(cin, message);

HillCipherEncrypt(message, key);

string ciphertext;

cout << "Enter the ciphertext: ";

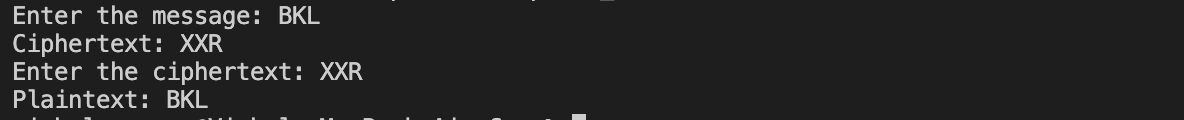
getline(cin, ciphertext);

HillCipherDecrypt(ciphertext, key);

return 0;

}

**Outputs :**



**Practical 6**

**Aim** : Implement Simple Transposition encryption-decryption

**Implementation :**

#include <iostream>

#include <string>

#include <cstring>

using namespace std;

// Encryption function

string Encryption(int no\_rows, int len\_key, int len\_msg, string msg, int col\_val[])

{

int x = 0;

char enc\_mat[no\_rows + 1][len\_key];

// creating the matrix

for (int i = 0; i < no\_rows + 1; i++) {

for (int j = 0; j < len\_key; j++) {

// initializes the positions with '\_' after the end of message

if (x >= len\_msg) {

enc\_mat[i][j] = '\_';

}

else {

enc\_mat[i][j] = msg[x];

}

x++;

}

}

int t = 1;

string cipher = "";

// finding the cipher text according to the value of col\_val matrix

while (t <= len\_key) {

for (int i = 0; i < len\_key; i++) {

int k = col\_val[i];

if (k == t) {

for (int j = 0; j < no\_rows + 1; j++) {

cipher += enc\_mat[j][i];

}

t++;

}

}

}

return cipher;

}

// Decryption function

string Decryption(int no\_rows, int len\_key, string cipher, int col\_val[])

{

char dec\_mat[no\_rows + 1][len\_key];

int x = 0, t = 1;

// rearrange the matrix according to the col\_val

while (t <= len\_key) {

for (int i = 0; i < len\_key; i++) {

int k = col\_val[i];

if (k == t) {

for (int j = 0; j < no\_rows + 1; j++) {

dec\_mat[j][i] = cipher[x];

x++;

}

t++;

}

}

}

string message = "";

for (int i = 0; i < no\_rows + 1; i++) {

for (int j = 0; j < len\_key; j++) {

// replacing the '\_' with space

if (dec\_mat[i][j] == '\_') {

dec\_mat[i][j] = ' ';

}

message += dec\_mat[i][j];

}

}

return message;

}

int main()

{

// Taking input for the message and key from the user

string msg, key;

cout << "Enter the message to be encrypted: ";

getline(cin, msg);

cout << "Enter the key: ";

getline(cin, key);

int len\_key = key.length();

int len\_msg = msg.length();

int val = 1, count = 0, ind;

int col\_val[len\_key];

// initializing col\_val matrix with 0

memset(col\_val, 0, sizeof(col\_val));

// numbering the key alphabets according to its ASCII value

while (count < len\_key) {

int min = 999;

for (int i = 0; i < len\_key; i++) {

if ((min > int(key[i])) && (col\_val[i] == 0)) {

min = int(key[i]);

ind = i;

}

}

col\_val[ind] = val;

count++;

val++;

}

int no\_rows = len\_msg / len\_key;

// encrypted text

string cipher\_text = Encryption(no\_rows, len\_key, len\_msg, msg, col\_val);

cout << "Encrypted Message: " << cipher\_text << endl;

// decrypted text

string original\_msg = Decryption(no\_rows, len\_key, cipher\_text, col\_val);

cout << "Decrypted Message: " << original\_msg << endl;

return 0;

}

**Output :**

****

**Practical 7**

**Aim** : Implement One time pad encryption-decryption

**Implementation :**

#include <iostream>

#include <string>

#include <algorithm> // For transform function

using namespace std;

// Method 1: Returning encrypted text

string stringEncryption(string text, string key)

{

// Initializing cipherText

string cipherText = "";

// Initialize cipher array of key length

// which stores the sum of corresponding no.'s

// of plainText and key.

int cipher[key.length()];

for (int i = 0; i < key.length(); i++) {

cipher[i] = text.at(i) - 'A' + key.at(i) - 'A';

}

// If the sum is greater than 25

// subtract 26 from it

// and store that resulting value

for (int i = 0; i < key.length(); i++) {

if (cipher[i] > 25) {

cipher[i] = cipher[i] - 26;

}

}

// Converting the no.'s into integers

// Convert these integers to corresponding

// characters and add them up to cipherText

for (int i = 0; i < key.length(); i++) {

int x = cipher[i] + 'A';

cipherText += (char)x;

}

// Returning the cipherText

return cipherText;

}

// Method 2: Returning plain text

static string stringDecryption(string s, string key)

{

// Initializing plain text

string plainText = "";

// Initializing integer array of key length

// which stores difference

// of corresponding no.'s of

// each character of cipherText and key

int plain[key.length()];

// Running for loop for each character

// subtracting and storing in the array

for (int i = 0; i < key.length(); i++) {

plain[i] = s.at(i) - 'A' - (key.at(i) - 'A');

}

// If the difference is less than 0

// add 26 and store it in the array.

for (int i = 0; i < key.length(); i++) {

if (plain[i] < 0) {

plain[i] = plain[i] + 26;

}

}

// Converting int to corresponding char

// add them up to plainText

for (int i = 0; i < key.length(); i++) {

int x = plain[i] + 'A';

plainText += (char)x;

}

// Returning plainText

return plainText;

}

// Method 3: Main driver method

int main()

{

// Declaring plain text and key

string plainText, key;

// Taking input for plain text and key from the user

cout << "Enter the message to be encrypted: ";

getline(cin, plainText);

cout << "Enter the key: ";

getline(cin, key);

// Converting plain text and key to uppercase

transform(plainText.begin(), plainText.end(), plainText.begin(), ::toupper);

transform(key.begin(), key.end(), key.begin(), ::toupper);

// Encrypting the plain text

string encryptedText = stringEncryption(plainText, key);

// Printing cipher Text

cout << "Cipher Text - " << encryptedText << endl;

// Decrypting the encrypted text

string decryptedText = stringDecryption(encryptedText, key);

// Printing the decrypted message

cout << "Decrypted Message - " << decryptedText << endl;

return 0;

}

**Output :**



# Practical 8

## **AIM:** Implement Diffie Hellman key exchange method.

### Theory:

The Diffie-Hellman key exchange is a cryptographic protocol that allows two parties to securely share a secret key over an insecure channel. It works by each party selecting a private key, then generating and exchanging public keys based on a shared prime number and base. Both parties can then independently compute the same shared secret using their own private key and the other party's public key. The security of the protocol relies on the difficulty of solving the discrete logarithm problem, making it secure even if the exchanged public keys are intercepted.

**Key Concepts:**

### Public Parameters:

* + **Prime number p**: A large prime number.
  + **Base g**: A primitive root modulo p, typically a small integer.

Both p and g are public and can be known by everyone.

### Private Keys:

* + **Alice’s private key a**: A secret number randomly chosen by Alice.
  + **Bob’s private key b**: A secret number randomly chosen by Bob.

These private keys are kept secret and are not shared with anyone.

### Public Keys:

* + **Alice’s public key A**: Calculated as A=ga mod p.
  + **Bob’s public key B**: Calculated as B=gb mod p.

These public keys are shared between Alice and Bob.

**Key Exchange Process:**

### Global Parameters:

* + Both Alice and Bob agree on a common prime number p and a base g, where g is a primitive root modulo p.

### Select Private Keys:

* + Alice selects a private key XA such that XA<p.
  + Bob selects a private key XB such that XB<p.

### Calculate Public Keys:

* + Alice computes her public key YA using the formula YA= gX A mod p.
  + Bob computes his public key YB using the formula YB=. gX Bmod p.

### Exchange Public Keys:

* + Alice and Bob share their public keys YA and YB with each other.

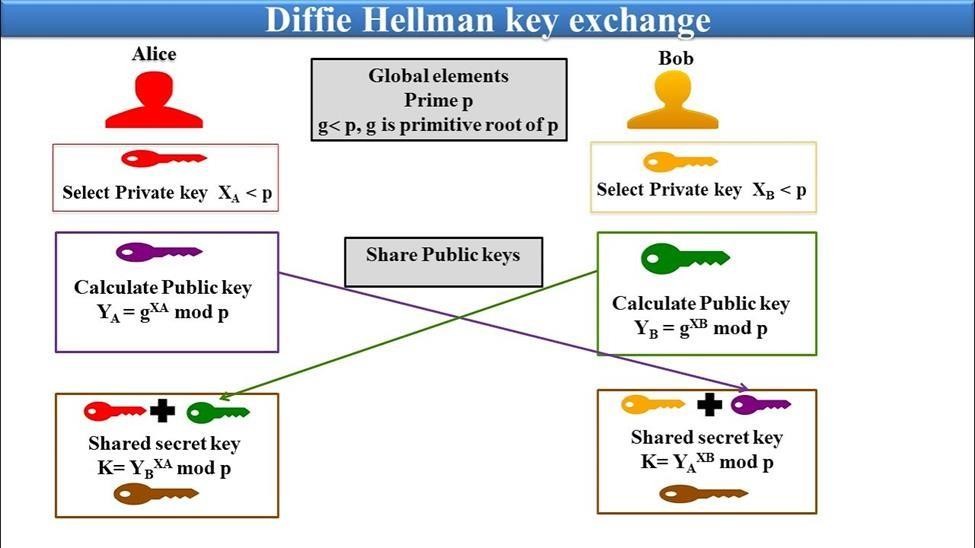
### Calculate the Shared Secret Key:

* + Alice computes the shared secret key K using Bob's public key YB and her private key XA with the formula K = YBXAmod p.
  + Bob computes the shared secret key K using Alice's public key YA and her

private key XB with the formula K = Y X mod p.

A B

Both Alice and Bob will arrive at the same shared secret key K, which can be used for secure communication. The exchange process relies on the difficulty of calculating the discrete logarithm, ensuring that the shared secret remains secure even if an eavesdropper knows the public keys.



### Code:

def modular\_pow(base, exponent, mod): result = 1

base = base % mod while exponent > 0:

if (exponent % 2) == 1:

result = (result \* base) % mod exponent = exponent >> 1

base = (base \* base) % mod return result

def diffie\_hellman(p, g, a, b): A = modular\_pow(g, a, p) B = modular\_pow(g, b, p)

shared\_secret\_alice = modular\_pow(B, a, p) shared\_secret\_bob = modular\_pow(A, b, p)

return A, B, shared\_secret\_alice, shared\_secret\_bob if name == " main ":

p = int(input("Enter a prime number (p): "))

g = int(input("Enter a primitive root modulo p (g): ")) a = int(input("Enter Alice's private key (a): "))

b = int(input("Enter Bob's private key (b): "))

A, B, shared\_secret\_alice, shared\_secret\_bob = diffie\_hellman(p, g, a, b) print("\nDiffie-Hellman Key Exchange:")

print("Alice's public key (A):", A) print("Bob's public key (B):", B)

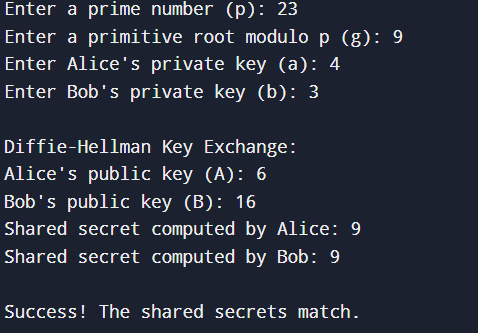
print("Shared secret computed by Alice:", shared\_secret\_alice) print("Shared secret computed by Bob:", shared\_secret\_bob)

if shared\_secret\_alice == shared\_secret\_bob: print("\nSuccess! The shared secrets match.")

else:

print("\nError! The shared secrets do not match.")

### Output:



**Result:**

The implementation of Diffie Hellman key exchange algorithm encryption and decryption has been successfully completed.

# Practical - 9

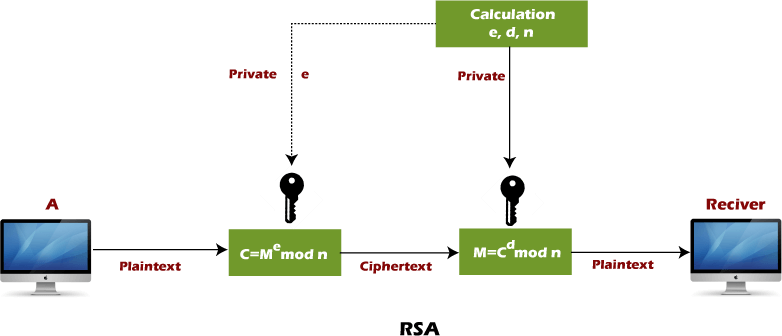
## **AIM:** Implement RSA encryption-decryption algorithm.

### Theory:

The RSA algorithm (Rivest-Shamir-Adleman) is one of the first public-key cryptosystems and is widely used for secure data transmission. It is an asymmetric encryption algorithm, meaning it uses two different keys: a public key for encryption and a private key for decryption.

### Steps in RSA Algorithm:

1. **Key Generation**:
   * Choose two large prime numbers, p and q.
   * Compute n = p×q. This is used as part of both the public and private keys.
   * Compute Euler's totient function ϕ(n)=(p−1)×(q−1).
   * Choose an integer e such that 1<e<ϕ(n) and e is coprime with ϕ(n). This e becomes the **public key exponent**.
   * Compute d, the modular multiplicative inverse of e, such that d×e≡1(mod ϕ (n)). This d becomes the **private key exponent**.
   * **Public Key**: (n,e)
   * **Private Key**: (n,d)
2. **Encryption**: To encrypt a message m (where m is a number such that 0≤m<n):
   * Ciphertext ccc is calculated as: c = me mod n
3. **Decryption:** To decrypt the ciphertext c:
   * The original message m is recovered using: m = cd mod n



### Code:

import random

# Function to compute the greatest common divisor def gcd(a, b):

while b != 0:

a, b = b, a % b return a

# Function to find modular inverse using Extended Euclidean Algorithm def modinv(a, m):

m0, x0, x1 = m, 0, 1

if m == 1:

return 0 while a > 1:

q = a // m

m, a = a % m, m

x0, x1 = x1 - q \* x0, x0 if x1 < 0:

x1 += m0 return x1

# Function to generate RSA keys def generate\_keys():

# Function to check if a number is prime def is\_prime(num):

if num <= 1:

return False

for i in range(2, int(num \*\* 0.5) + 1): if num % i == 0:

return False

return True

# Generate two large prime numbers, p and q p = q = 0

while not is\_prime(p):

p = random.randint(50, 100) # Smaller primes for simplicity increase range for real security

while not is\_prime(q) or q == p:

q = random.randint(50, 100)

n = p \* q

phi = (p - 1) \* (q - 1)

# Choose e such that 1 < e < phi and gcd(e, phi) = 1 e = random.randrange(2, phi)

while gcd(e, phi) != 1:

e = random.randrange(2, phi)

# Calculate d (modular inverse of e) d = modinv(e, phi)

# Return public and private keys return (n, e), (n, d)

# Function to encrypt a message def encrypt(public\_key, message):

n, e = public\_key

message\_int = [ord(char) for char in message] # Convert each character to its ASCII value ciphertext = [(m \*\* e) % n for m in message\_int]

return ciphertext

# Function to decrypt a message

def decrypt(private\_key, ciphertext): n, d = private\_key

decrypted\_message = [(c \*\* d) % n for c in ciphertext]

message = ''.join([chr(m) for m in decrypted\_message]) # Convert ASCII values back to characters

return message

# Main program def main():

# Key generation

public\_key, private\_key = generate\_keys() print(f"Public Key: {public\_key}") print(f"Private Key: {private\_key}")

# User input for message

message = input("Enter a message to encrypt: ")

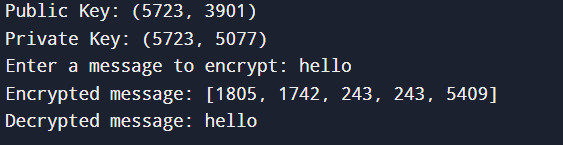
# Encrypt the message

encrypted\_message = encrypt(public\_key, message) print(f"Encrypted message: {encrypted\_message}") # Decrypt the message

decrypted\_message = decrypt(private\_key, encrypted\_message) print(f"Decrypted message: {decrypted\_message}")

if name == " main ": main()

### Output:



**Result:**

The implementation of RSA algorithm encryption and decryption has been successfully completed.

# Practical - 10

## **AIM:** Demonstrate working of Digital Signature using Cryptool.

### Theory:

**Digital Signature:**

A digital signature in cryptography is a mechanism that ensures the authenticity, integrity, and non- repudiation of digital messages or documents. It's similar to a handwritten signature or a stamped seal but offers far more security. Digital signatures use asymmetric cryptography, which involves a pair of keys: a private key and a public key.

### Key Components of Digital Signatures:

1. **Private Key:** Used by the signer to create the signature. Only the owner of the private key can sign the document.
2. **Public Key:** Used by others to verify the signature. Anyone with the public key can check whether the document was signed by the corresponding private key.

### Process:

1. **Hashing:** First, the document or message is hashed using a cryptographic hash function (e.g., SHA-256). This creates a fixed-size output (hash) representing the document.
2. **Signing:** The hash of the document is then encrypted with the signer's private key. This encrypted hash is the digital signature.

### Verification:

* + The receiver decrypts the signature using the sender's public key, obtaining the original hash.
  + The receiver also hashes the received document using the same hash function.
  + If the decrypted hash matches the hash of the received document, the signature is valid, confirming that the document hasn't been altered and was signed by the legitimate private key owner.

**CrypTool:**

**CrypTool** is an open-source project that provides a suite of tools and educational resources for learning and experimenting with cryptography. It is widely used by students, educators, and cryptography enthusiasts to understand cryptographic concepts and algorithms through a hands- on approach.

### Key Features of CrypTool:

1. **Cryptographic Algorithms:** CrypTool supports a wide range of cryptographic algorithms, including:
   * **Symmetric encryption:** AES, DES, Blowfish, etc.
   * **Asymmetric encryption:** RSA, ElGamal, ECC, etc.
   * **Hash functions:** MD5, SHA family, etc.

### Digital signatures and certificates

* + **Key exchange protocols:** Diffie-Hellman, etc.

1. **Visual Representation:** It provides visual demonstrations of how encryption, decryption, and other cryptographic operations work. This is particularly helpful for understanding complex algorithms and their inner workings.
2. **Hands-on Exercises:** Users can perform cryptographic tasks such as encryption, decryption, key generation, digital signing, and verification. CrypTool enables you to experiment with different algorithms, key sizes, and modes of operation.
3. **CrypTool Variants:** There are several versions of CrypTool, each designed for different needs:
   * **CrypTool 1 (CT1):** A desktop application focused on classical and modern cryptography.
   * **CrypTool 2 (CT2):** Provides a modern, flexible environment for experimenting with cryptographic algorithms. It includes drag-and-drop functionality for visual programming and workflows.
   * **JCrypTool (JCT):** A Java-based version for cross-platform usage, with a focus on modularity.
   * **CrypTool Online (CTO):** A web-based version that allows users to experiment with cryptography through their web browser without needing to install software.
4. **Analysis Tools:** CrypTool offers tools for analyzing cryptographic strengths, such as:
   * **Frequency analysis:** For breaking classical ciphers (Caesar, Vigenère, etc.).

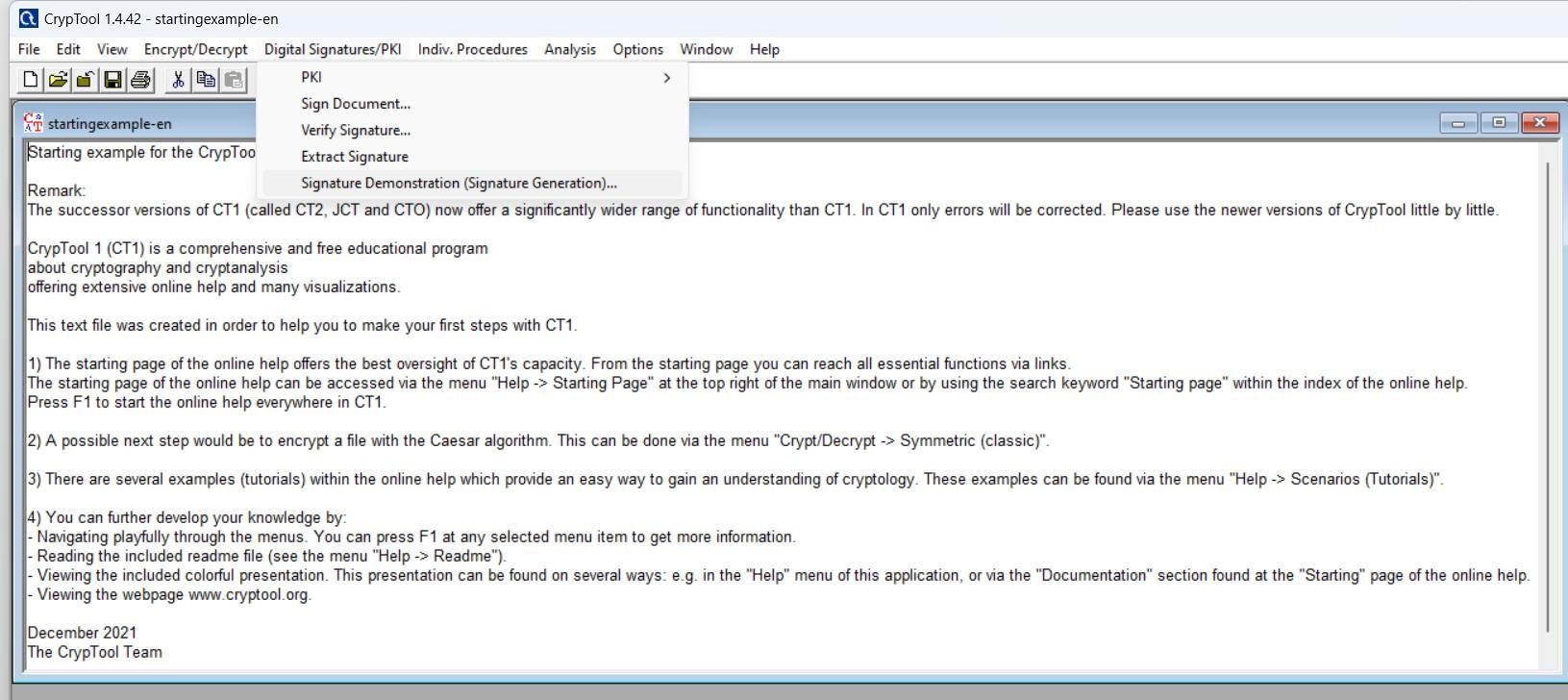
### Side-channel attack simulations

* + **Statistical tests** on the randomness of cipher outputs.

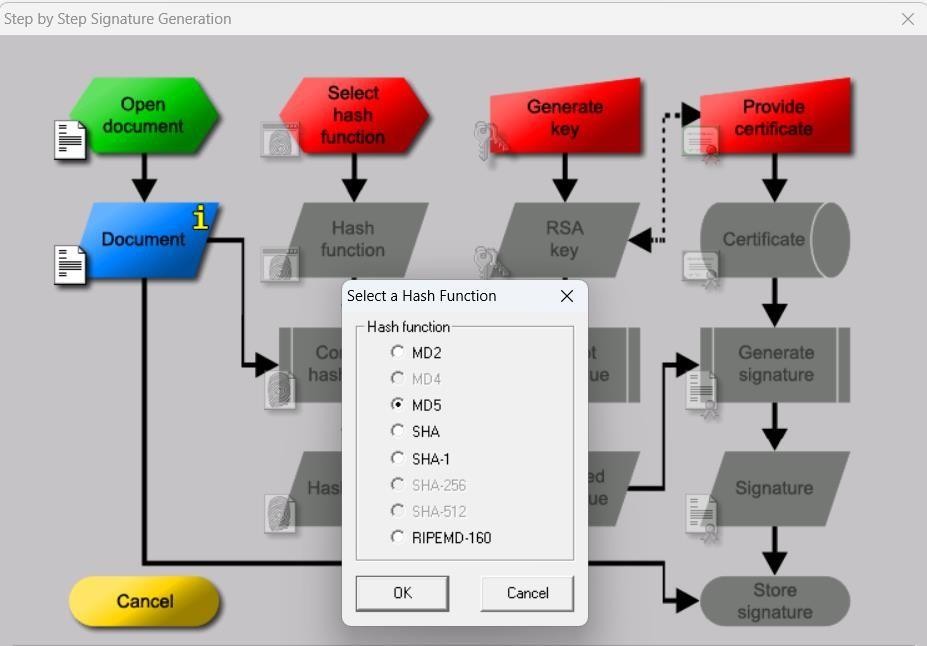
1. **Educational Tutorials:** CrypTool provides step-by-step tutorials and explanations of cryptographic concepts, making it an excellent resource for both beginners and advanced learners.

Following are the steps involved in generating a digital signature using cryptool:

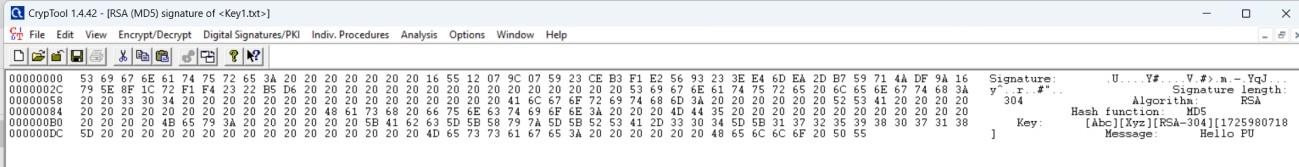
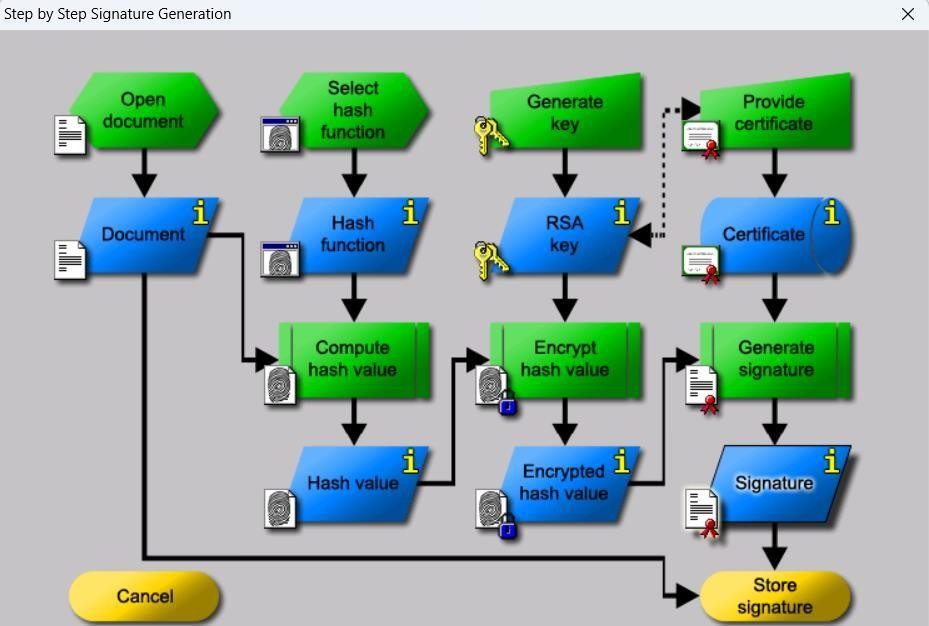
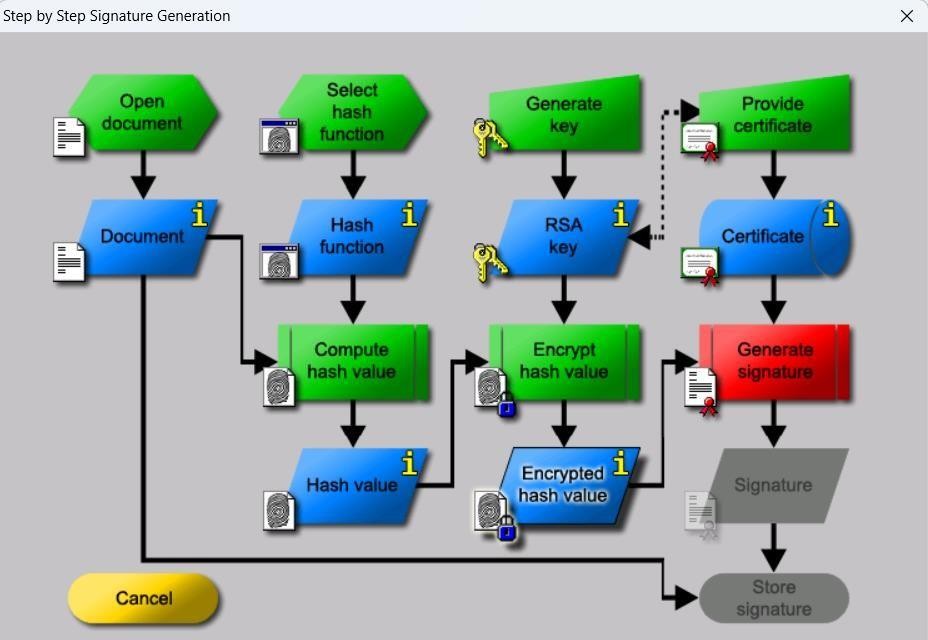
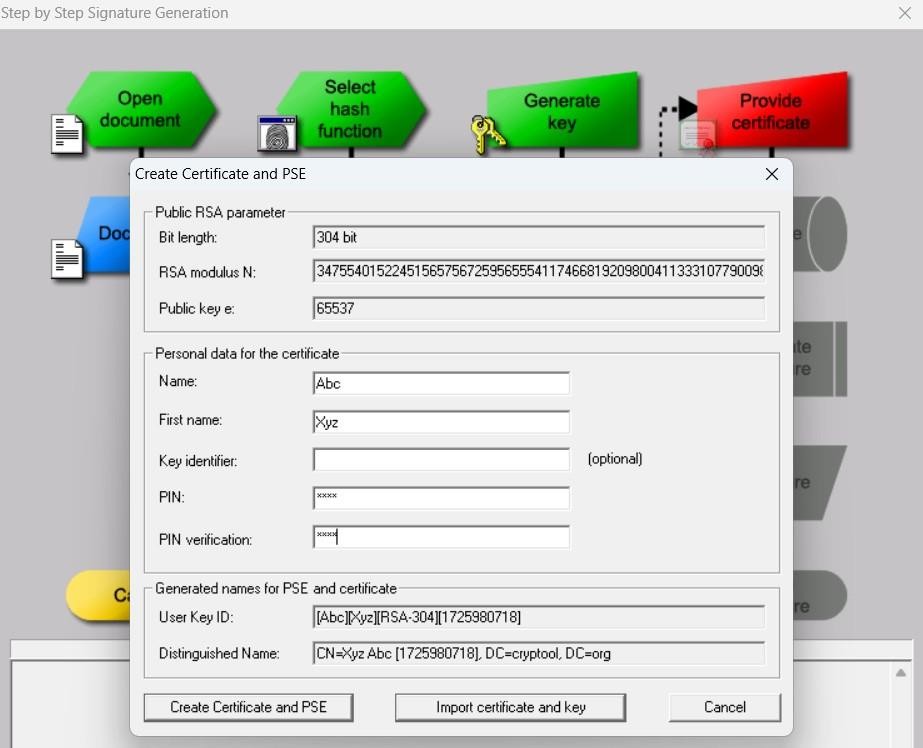
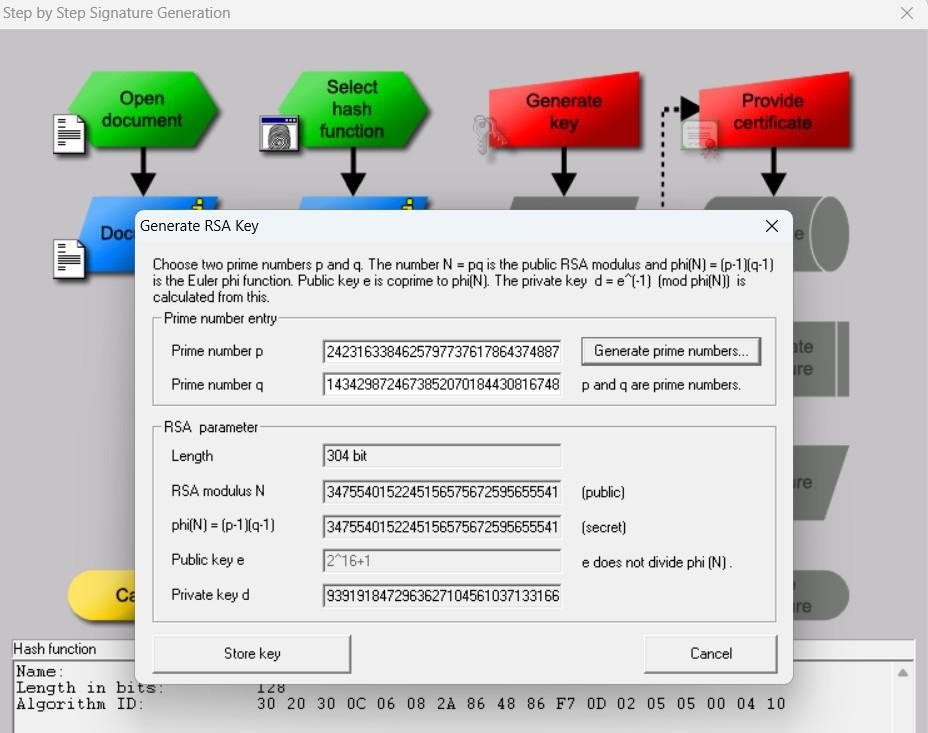
1. Launch CrypTool: Open CrypTool, a software that offers various cryptographic functions.
2. Load/Enter the Message: Input the message or file you want to sign. You can either type the text manually or import a file.



1. Select Hash Function: Choose a hash function (e.g., SHA-256) to generate a fixed-size hash (message digest) from the original message.



1. Choose Asymmetric Key Algorithm: Select a public-key cryptography algorithm like RSA or DSA for signing. You will need a private key for signing and a public key for verification.
2. Generate Key Pair: If you don’t already have a key pair, generate a new one (private and public key). The private key will be used to create the digital signature.



1. Sign the Message: Using the private key, the message digest (from step 3) is encrypted, producing the digital signature.
2. Save or Export: Save or export the digital signature along with the original message if needed.
3. Verify the Signature (optional): Use the public key to verify the digital signature to ensure the message hasn’t been tampered with.

These steps allow the sender to sign the message, ensuring authenticity and integrity.