

# MIT Art Design and Technology University

**MIT School of Computing, Pune**

**Department of Computer Science and Engineering**

**Lab Manual**

**Practical - Programming Laboratory –V**

**( CC+ IS )**

**Class - L.Y. (SEM-I), <Core-II>**

**A.Y. 2023 - 2024**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **Sr.**  **No.** | **Title** | **Page No.** |
|  | |  |
| 1 | Write a C program that contains a string (char pointer) with a value  \Hello World’. The program should AND or and XOR 2 each character in this string with 127 and display the result. | 03 |
| 2 | Write a Java program to perform encryption and decryption using Caesar Cipher algorithm | 07 |
| 3 | Write a Program to Implement RSA Algorithm. | 14 |
| 4 | Write a Program to Simulate Diffie-Hellman Key Exchange | 21 |
| 5 | Implementation of Columnar Transposition Technique | 27 |
| 6 | Calculate the message digest of a text using the SHA-256 algorithm. | 35 |
| 7 | Demonstrate intrusion detection system (ids) using any tool. | 39 |

# Experiment No:1

**Title:** Write a C program that contains a string (char pointer) with a value \Hello World’. The program should AND or and XOR 2 each character in this string with 127 and display the result.

After completion of this experiment students will be able to: Learn AND, OR and XOR operations on string

**Aim:** Program for AND OR and XOR each character in string operation

### Theory:

**XOR Operation**

There are two inputs and one output in binary XOR (exclusive OR) operation. It is similar to ADD operation which takes two inputs and produces one result i.e. one output.

The inputs and result to a binary XOR operation can only be 0 or 1.The binary XOR operation will always produce a 1 output if either of its inputs is 1 and will produce a 0 output if both of its inputs are 0 or 1.

XOR Truth Table

|  |  |  |
| --- | --- | --- |
| **Input** | | **Output** |
| **X** | **Y** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

### Algorithm/Pseudocode:

1. Start
2. Declare array for storing string
3. Take input from user as string
4. Read the string
5. Find the length of string using built in function
6. Perform AND operation display result character by character
7. Perform XOR operation display result character by character
8. Perform OR operation display result character by character
9. Stop

### Program:

#include <iostream>

#include <cstring> // Include for C++ string functions

int main() {

char str[24], a[24], b[24], c[24]; // Adjusted array size

int x;

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

std::cin >> str;

x = std::strlen(str);

std::cout << "\n AND operation: ";

for (int i = 0; i < x; i++) { // Corrected loop condition

a[i] = str[i] & 127;

std::cout << a[i];

}

std::cout << "\n XOR operation: ";

for (int i = 0; i < x; i++) { // Corrected loop condition

b[i] = str[i] ^ 127;

std::cout << b[i];

}

std::cout << "\n OR operation: ";

for (int i = 0; i < x; i++) { // Corrected loop condition

c[i] = str[i] | 127;

std::cout << c[i];

}

return 0;

}

## OUTPUT:

Enter the string: HelloWorld

AND operation: HelloWorld XOR operation: 7

OR operation:

Certainly! Let's go through the provided C++ code step by step:

cpp

Copy code

#include <iostream>

#include <cstring>

These are the standard include statements for input/output operations (<iostream>) and string manipulation (<cstring>).

cpp

Copy code

int main() {

char str[24], a[24], b[24], c[24]; // Adjusted array size

int x;

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

std::cin >> str;

x = std::strlen(str);

The program starts with the main function.

It declares arrays str, a, b, and c, each of size 24. These arrays will store the input string and the results of bitwise AND, XOR, and OR operations.

x will store the length of the input string.

The user is prompted to enter a string, which is then stored in the str array. The length of the string is calculated using std::strlen (from <cstring>) and stored in the variable x.

cpp

Copy code

std::cout << "\n AND operation: ";

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

a[i] = str[i] & 127;

std::cout << a[i];

}

This section performs a bitwise AND operation between each character in the input string (str[i]) and the value 127.

The results of the AND operation are stored in the array a.

The loop iterates through each character of the string, and the result of each operation is printed.

cpp

Copy code

std::cout << "\n XOR operation: ";

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

b[i] = str[i] ^ 127;

std::cout << b[i];

}

Similar to the AND operation, this section performs a bitwise XOR operation between each character in the input string (str[i]) and the value 127.

The results of the XOR operation are stored in the array b.

The loop iterates through each character of the string, and the result of each operation is printed.

cpp

Copy code

std::cout << "\n OR operation: ";

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

c[i] = str[i] | 127;

std::cout << c[i];

}

This section performs a bitwise OR operation between each character in the input string (str[i]) and the value 127.

The results of the OR operation are stored in the array c.

The loop iterates through each character of the string, and the result of each operation is printed.

cpp

Copy code

return 0;

}

The program finishes with the return 0; statement, indicating successful execution of the program.

# Experiment No:2

**Title:** Write a Java program to perform encryption and decryption using Caesar Cipher algorithm

After completion of this experiment students will be able to: Learn caesar cipher substitution technique

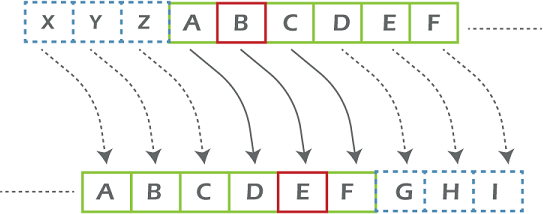
**Aim:** Program for Caesar cipher.

### Theory:

It is one of the simplest and most used encryption techniques. In this technique, each letter of the given text is replaced by a letter of some fixed number of positions down the alphabet.

For example, with a shift of 1, X would be replaced by Y, Y would become Z, and so on. Julius Caesar was the first one who used it for communicating with his officials. Based on his name, this technique was named as Caesar Cipher technique.

An integer value is required to cipher a given text. The integer value is known as shift, which indicates the number of positions each letter of the text has been moved down.



We can mathematically represent the encryption of a letter by a shift n in the following way:

Encryption phase with shift n = En (x) = (x+n)mod 26 Decryption phase with shift n = Dn (x) = (x-n)mod 26 Examples

**Text** : ABCDEFGHIJKLMNOPQRSTUVWXYZ

**Shift** : 23

**Cipher** : XYZABCDEFGHIJKLMNOPQRSTUVW

**Text** : ATTACKATONCE

**Shift** : 4

**Cipher** : EXXEGOEXSRGI

### Algorithm/Pseudocode:

1. Take an input string from the user to encrypt it using the Caesar Cipher technique.
2. Take an input integer from the user for shifting characters. The input integer should be between 0-25.
3. Traverse input string one character at a time.
4. Depending on the encryption and decryption, we transform each character as per the rule.
5. Returns the newly generated string.

### Program:

import java.util.Scanner;

public class CaesarCipherExample {

public static final String ALPHABET = "abcdefghijklmnopqrstuvwxyz";

public static String encryptData(String inputStr, int shiftKey) {

inputStr = inputStr.toLowerCase();

String encryptStr = "";

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

int pos = ALPHABET.indexOf(inputStr.charAt(i));

int encryptPos = (shiftKey + pos) % 26;

if (encryptPos < 0) {

encryptPos = ALPHABET.length() + encryptPos;

}

char encryptChar = ALPHABET.charAt(encryptPos);

encryptStr += encryptChar;

}

return encryptStr;

}

public static String decryptData(String inputStr, int shiftKey) {

inputStr = inputStr.toLowerCase();

String decryptStr = "";

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

int pos = ALPHABET.indexOf(inputStr.charAt(i));

int decryptPos = (pos - shiftKey) % 26;

if (decryptPos < 0) {

decryptPos = ALPHABET.length() + decryptPos;

}

char decryptChar = ALPHABET.charAt(decryptPos);

decryptStr += decryptChar;

}

return decryptStr;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter a string for encryption using Caesar Cipher: ");

String inputStr = sc.nextLine();

System.out.println("Enter the value by which each character in the plaintext message gets shifted: ");

int shiftKey = Integer.valueOf(sc.nextLine());

System.out.println("Encrypted Data ===> " + encryptData(inputStr, shiftKey));

System.out.println("Decrypted Data ===> " + decryptData(encryptData(inputStr, shiftKey), shiftKey));

sc.close();

}

}

Import Statements:

java

Copy code

import java.util.Scanner;

The code begins with importing the Scanner class, which is used later for taking input from the user.

Class Declaration:

java

Copy code

public class CaesarCipherExample {

The code defines a class named CaesarCipherExample.

Constant Declaration:

java

Copy code

public static final String ALPHABET = "abcdefghijklmnopqrstuvwxyz";

The ALPHABET constant is a string containing the lowercase English alphabet.

Encrypt Method:

java

Copy code

public static String encryptData(String inputStr, int shiftKey) {

// Method implementation

}

This method takes an input string and a shift key, then encrypts the input string using the Caesar cipher.

Decrypt Method:

java

Copy code

public static String decryptData(String inputStr, int shiftKey) {

// Method implementation

}

This method takes an encrypted string and a shift key, then decrypts the string using the Caesar cipher.

Main Method:

java

Copy code

public static void main(String[] args) {

// Method implementation

}

The main method is the entry point of the program. It takes user input, calls the encryption and decryption methods, and displays the results.

User Input:

java

Copy code

Scanner sc = new Scanner(System.in);

Creates a Scanner object to read input from the user.

Encryption and Decryption:

java

Copy code

System.out.println("Enter a string for encryption using Caesar Cipher: ");

String inputStr = sc.nextLine();

System.out.println("Enter the value by which each character in the plaintext message gets shifted: ");

int shiftKey = Integer.valueOf(sc.nextLine());

System.out.println("Encrypted Data ===> " + encryptData(inputStr, shiftKey));

System.out.println("Decrypted Data ===> " + decryptData(encryptData(inputStr, shiftKey), shiftKey));

Takes a string input from the user and a shift key.

Calls the encryptData method to encrypt the input string.

Calls the decryptData method to decrypt the encrypted string.

Displays the encrypted and decrypted data.

Closing Scanner:

java

Copy code

sc.close();

Closes the Scanner object to release resources.

The code demonstrates a basic implementation of the Caesar cipher for both encryption and decryption. The Caesar cipher shifts each letter in the plaintext by a fixed number of positions down the alphabet.

## OUTPUT:

Enter a string for encryption using Caesar Cipher:

MITADT

Enter the value by which each character in the plaintext message gets shifted: 3

Encrypted Data ===> plwdgw Decrypted Data ===> mitadt

### Conclusion:

**References:**

1. William Stallings, “Cryptography and Network Security”, Sixth Edition Pearson Education.
2. Atul Kahate, “Cryptography and Network Security”, Tata McGraw-Hill.

# Experiment No: 3

**Title:** Write a Program to Implement RSA Algorithm.

**Aim:** To develop a program for encryption and decryption of the message using RSA algorithm in C Language.

### Theory:

An original message is known as the plaintext, while the coded message is called the ciphertext. The process of converting from plaintext to ciphertext is known as enciphering or encryption; restoring the plaintext from the ciphertext is deciphering or decryption.

* + **Plaintext:** This is the original intelligible message or data that is fed into the algorithm asinput.
  + **Encryption algorithm:** The encryption algorithm performs various substitutions and transformations on the plaintext.
  + **Secret key:** The secret key is also input to the encryption algorithm. The key is a value independent of the plaintext and of the algorithm. The algorithm will produce a different output depending on the specific key being used at the time. The exact substitutions and transformations performed by the algorithm depend on the key.
  + **Ciphertext:** This is the scrambled message produced as output. It depends on the plaintext and the secret key. For a given message, two different keys will produce two different ciphertexts. The ciphertext is an apparently random stream of data and, as it stands, is unintelligible.
  + **Decryption algorithm:** This is essentially the encryption algorithm run in reverse. It takes the ciphertext and the secret key and produces the original plaintext.

### RSA Algorithm:

RSA algorithms have been proposed for public-key cryptography. RSA was developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman at MIT and first published in 1978. Public-key [cryptography,](http://searchsoftwarequality.techtarget.com/definition/cryptography) also known as [asymmetric](http://searchsecurity.techtarget.com/definition/asymmetric-cryptography) [cryptography](http://searchsecurity.techtarget.com/definition/asymmetric-cryptography), uses two different but mathematically linked [keys,](http://searchsecurity.techtarget.com/definition/key) one public and one private. The [public key](http://searchsecurity.techtarget.com/definition/public-key) can be shared with everyone, whereas the [private key](http://searchsecurity.techtarget.com/definition/private-key) must be kept secret. In RSA cryptography, both the public and the private keys

can encrypt a message; the opposite key from the one used to encrypt a message is used to decrypt it. This attribute is one reason why RSA has become the most widely used asymmetric.

RSA is more secure because of factoring large integers that are the product of two large [prime numbers.](http://whatis.techtarget.com/definition/prime-number) Multiplying these two numbers is easy, but determining the original prime numbers from the total factoring is considered infeasible due to the time it would take even using today’s super computers.

The RSA scheme is a block cipher in which the plaintext and ciphertext are integers between 0 and *n* - 1 for some *n*. A typical size for *n* is 1024 bits. That is, *n* is less than 21024. . In RSA naming convention are Plaintext block *M* and ciphertext block *C*.

Some terms related to RSA are as below

*p, q*, two prime numbers

*n* = *p\*q*

*e*, with gcd(Ø (*n*), *e*) = 1; 1 < *e* < Ø (*n*)

*de* = 1(mod Ø (*n*))

The private key consists of {*d, n*} and the public key consists of {*e, n*}. Suppose that user A has published its public key and that user B wishes to send the message *M* to A. Then B calculates *C* = *Me* mod *n* and transmits *C*. On receipt

of this ciphertext, user A decrypts by calculating*M* = *Cd* mod *n*.

### Algorithm:

1. Select two prime numbers, *p* = 17 and *q* = 11.
2. Calculate *n* = *pq* = 17 × 11 = 187.

**3.** Calculate Ø(*n*) = (*p* - 1)(*q* - 1) = 16 × 10 = 160.

* 1. Select *e* such that *e* is relatively prime to Ø (*n*) = 160 and less than Ø (*n*); wechoose *e* = 7.
  2. Determine *d* such that *de=* 1 (mod n) and *d* < 160.The correct value is *d* = 23,because 23 × 7 = 161 = (1 × 160) + 1;

The resulting keys are public key *PU* = {7, 187} and private key *PR* = {23, 187}. The example shows the use of these keys for a plaintext input of *M*= 88. For encryption, we need to calculate*C* = 887 mod 187. we can evaluate as follows.

887 mod 187 = [(884 mod 187) x (882 mod 187) x (881 mod 187)] mod 187

881 mod 187 = 88

882 mod 187 = 7744 mod 187 = 77

884 mod 187 = 59,969,536 mod 187 = 132

887 mod 187 = (88 × 77 × 132) mod 187 = 894,432 mod 187 = 11

For decryption, we calculate *M* = 1123 mod 187:

1123 mod 187 = [(111 mod 187) \*(112 mod 187) × (114 mod 187) \* (118 mod 187)

\* (118 mod 187)] mod 187

111 mod 187 = 11

112 mod 187 = 121

114 mod 187 = 14,641 mod 187 = 55

118 mod 187 = 214,358,881 mod 187 = 33

1123 mod 187 = (11x121 x 55 x 33x 33) mod 187 = 79,720,245 mod 187 = 88

### Program:

/\* Aim: Program for encryption and decryption of message \*/

/\* Title: Program for encryption and decryption using RSA algorithm \*/ #include<stdio.h>

#include<string.h>

#include<math.h>

int main() {

    unsigned long p, q, n, k, e, d;

    unsigned long C, M;

    char message[80];

    unsigned char encrypt[80], decrypt[80]; // Changed to unsigned char

    unsigned int len, i, j;

    printf("\nEnter P and Q : ");

    scanf("%lu%lu", &p, &q);

    n = p \* q;

    k = (p - 1) \* (q - 1);

    printf("\nEnter the value of public key : ");

    scanf("%lu", &e);

    // Find d such that (d \* e) % k == 1

    d = 3;

    while(1) {

        if ((d \* e) % k == 1)

            break;

        d++;

    }

    printf("\nPublic Key = %lu, Private key = %lu", e, d);

    printf("\nEnter the message : ");

    fflush(stdin);

    // Use fgets instead of gets to avoid buffer overflow

    fgets(message, sizeof(message), stdin);

    len = strlen(message);

    for (i = 0; i < len; i++) {

        M = message[i];

        C = 1;

        for (j = 0; j < e; j++) {

            C = (C \* M) % n;

        }

        encrypt[i] = (unsigned char)C;

    }

    printf("\nEncrypted Message is : ");

    for (i = 0; i < len; i++)

        printf("%c", encrypt[i]);

    printf("\n");

    for (i = 0; i < len; i++) {

        M = 1;

        C = encrypt[i];

        for (j = 0; j < d; j++) {

            M = (M \* C) % n;

        }

        // Handle the case where the decrypted value is greater than 127

        if (M > 127)

            M -= 256;

        decrypt[i] = (unsigned char)M;

    }

    printf("\nDecrypted Message is : ");

    for (i = 0; i < len; i++)

        printf("%c", decrypt[i]);

    printf("\n");

    return 0;

}

**Outp****ut**

Certainly! The code you provided is a simple implementation of the RSA (Rivest–Shamir–Adleman) algorithm for public-key cryptography in C. Let me break down the code step by step:

Header Files:

c

Copy code

#include<stdio.h>

#include<string.h>

#include<math.h>

The code includes necessary header files for input/output, string manipulation, and mathematical operations.

Main Function:

c

Copy code

int main() {

// Function implementation

}

The main function is the entry point of the program.

Variable Declarations:

c

Copy code

unsigned long p, q, n, k, e, d;

unsigned long C, M;

char message[80];

unsigned char encrypt[80], decrypt[80]; // Changed to unsigned char

unsigned int len, i, j;

p and q: Prime numbers used to compute n.

n: Product of p and q, used as part of the public and private keys.

k: Euler's totient function, (p-1) \* (q-1).

e: Public exponent.

d: Private exponent.

C and M: Temporary variables used for encryption and decryption.

message: Array to store the input message.

encrypt and decrypt: Arrays to store the encrypted and decrypted messages.

len: Length of the input message.

i and j: Loop control variables.

Input of Prime Numbers and Public Key:

c

Copy code

printf("\nEnter P and Q : ");

scanf("%lu%lu", &p, &q);

n = p \* q;

k = (p - 1) \* (q - 1);

printf("\nEnter the value of public key : ");

scanf("%lu", &e);

The user is prompted to enter two prime numbers (p and q) and a public exponent (e).

Calculation of Private Key:

c

Copy code

d = 3;

while (1) {

if ((d \* e) % k == 1)

break;

d++;

}

The code calculates the private exponent (d) such that (d \* e) % k == 1. This loop finds the modular multiplicative inverse of e modulo k.

Printing Public and Private Keys:

c

Copy code

printf("\nPublic Key = %lu, Private key = %lu", e, d);

The program prints the calculated public and private keys.

Input Message:

c

Copy code

printf("\nEnter the message : ");

fflush(stdin);

fgets(message, sizeof(message), stdin);

len = strlen(message);

The user is prompted to enter a message. fgets is used to read the message, avoiding potential buffer overflow issues.

Encryption:

c

Copy code

for (i = 0; i < len; i++) {

M = message[i];

C = 1;

for (j = 0; j < e; j++) {

C = (C \* M) % n;

}

encrypt[i] = (unsigned char)C;

}

The code encrypts each character of the input message using the public key (e, n) and stores the result in the encrypt array.

Print Encrypted Message:

c

Copy code

printf("\nEncrypted Message is : ");

for (i = 0; i < len; i++)

printf("%c", encrypt[i]);

printf("\n");

The program prints the encrypted message.

Decryption:

c

Copy code

for (i = 0; i < len; i++) {

M = 1;

C = encrypt[i];

for (j = 0; j < d; j++) {

M = (M \* C) % n;

}

if (M > 127)

M -= 256;

decrypt[i] = (unsigned char)M;

}

The code decrypts each character of the encrypted message using the private key (d, n) and stores the result in the decrypt array. The additional condition handles cases where the decrypted value is greater than 127.

Print Decrypted Message:

c

Copy code

printf("\nDecrypted Message is : ");

for (i = 0; i < len; i++)

printf("%c", decrypt[i]);

printf("\n");

The program prints the decrypted message.

Return Statement:

c

Copy code

return 0;

The main function returns 0 to indicate successful program executio

# Experiment No: 4

**Title:** Write a Program to Simulate Diffie-Hellman Key Exchange.

**Aim:** To develop a Program for Diffie-Hellman Key Exchange Algorithm using C Language.

### Theory:

Diffie-Hellman was the first public-key algorithm ever invented, way back in 1976. It gets its security from the difficulty of calculating discrete logarithms in a finite field, as compared with the ease of calculating exponentiation in the same field. Diffie-Hellman can be used for key distribution—Alice and Bob can use this algorithm to generate a secret key—but it cannot be used to encrypt and decrypt messages. The math is simple. First, Alice and Bob agree on a large prime, *n* and *g,* such that *g* is primitive mod *n.* These two integers don’t have to be secret; Alice and Bob can agree to them over some insecure channel. They can even be common among a group of users. It doesn’t matter.

Then, the protocol goes as follows:

* 1. Alice chooses a random large integer *x* and sends Bob

*X* = *gx* mod *n*

* 1. Bob chooses a random large integer *y* and sends Alice

*Y* = *gy* mod *n*

* 1. Alice computes

*k* = *Yx* mod *n*

* 1. Bob computes

*k´* = *Xy* mod *n*

Both *k* and *k´* are equal to *g*xy mod *n.* No one listening on the channel can compute that value; they only know *n, g, X,* and *Y.* unless they can compute the discrete logarithm and recover *x* or *y,* they do not solve the problem. So, *k* is the secret

key that both Alice and Bob computed independently. The choice of *g* and *n* can have a substantial impact on the security of this system. The number (*n* - 1)/2 should also be a prime. And most important,*n* should be large: The security of the system is based on the difficulty of factoring numbers the same size as *n.* You can choose any *g,* such that *g* is primitive mod *n;* there’s no reason not to choose the smallest *g* you can—generally a one-digit number. (And actually, *g* does not have to be primitive; it just has to generate a large subgroup of the multiplicative group mod *n.*)

## DIFFIE-HELLMAN WITH THREE OR MORE PARTIES

The Diffie-Hellman key-exchange protocol can easily be extended to work with three or more people. In this example, Alice, Bob, and Carol together generate a secret key.

1. Alice chooses a random large integer *x* and sends Bob

*X* = *gx* mod *n*

1. Bob chooses a random large integer *y* and sends Carol

*Y* = *gy* mod *n*

1. Carol chooses a random large integer *z* and sends Alice

*Z* = *gz* mod *n*

1. Alice sends Bob

*Z´* = *Z*x mod *n*

1. Bob sends

Carol

*X´* = *X*y mod *n*

1. Carol sends Alice

*Y´* = *Yz* mod *n*

1. Alice computes

*k* = *Y*´*x* mod *n*

1. Bob computes

*k* = *Z*´*y* mod *n*

1. Carol computes

*k* = *X*´*z* mod *n*

The secret key, *k,* is equal to gxyz mod *n,* and no one else listening in on the communications can compute that value. The protocol can be easily extended to four or more people; just add more people and more rounds of computation.

**Program:** #include<stdio.h> #include<math.h> int alice(int,int,int); int bob(int,int,int);

int main()

{

int a,b,x,y,g,n,k1,k2;

printf("\nEnter the value of n & g:"); scanf("%d %d",&n,&g);

printf("\n Enter the value of x & y:"); scanf("%d %d",&x,&y);

a=alice(n,g,x);

printf("\nAlice generate end value is %d:",a);

b=bob(n,g,y);

printf("\nBob generate end value is %d:",b);

k1=alice(n,b,x);

printf("\nValue of k1 is %d :",k1);

k2=bob(n,a,y);

printf("\nValue of k2 is %d:",k2);

if(k1==k2)

{

printf("\nMessage is shared secretely between Alice and Bob");

}

else

{

printf("\nMessage is not shared");

}

return 0;

}

int alice(int n,int g,int x)

{

long int a,a1; a1=pow(g,x); a=a1%n;

return(a);

}

int bob(int n,int g,int y)

{

long int b,b1; b1=pow(g,y); b=b1%n; return(b);

}

### Output:

Enter the value of n & g:13 6 Enter the value of x & y:3 10 Alice generate end value is 8:

Bob generate end value is 4:

Value of k1 is 12 :

Value of k2 is 12:

Message is shared secretly between Alice and Bob

Certainly! The provided code simulates a simplified version of the Diffie-Hellman key exchange protocol between two parties, Alice and Bob. Here's a step-by-step explanation:

Header Files:

c

Copy code

#include<stdio.h>

#include<math.h>

These are standard C library header files for input/output and mathematical operations.

Function Prototypes:

c

Copy code

int alice(int, int, int);

int bob(int, int, int);

int mod\_exp(int, int, int);

Function prototypes for alice, bob, and mod\_exp are declared to inform the compiler about their signatures before they are defined.

Function Definitions:

c

Copy code

int alice(int n, int g, int x) {

int a = mod\_exp(g, x, n);

return a;

}

int bob(int n, int g, int y) {

int b = mod\_exp(g, y, n);

return b;

}

int mod\_exp(int base, int exp, int mod) {

// Function to perform modular exponentiation efficiently

// (base^exp) % mod

int result = 1;

base = base % mod;

while (exp > 0) {

if (exp % 2 == 1) {

result = (result \* base) % mod;

}

exp = exp >> 1;

base = (base \* base) % mod;

}

return result;

}

alice and bob functions calculate the value g^x mod n and g^y mod n, respectively. These represent the values that Alice and Bob publicly share.

mod\_exp is a helper function for efficient modular exponentiation.

Main Function:

c

Copy code

int main() {

// Variable declarations

int a, b, x, y, g, n, k1, k2;

// Input from the user

printf("\nEnter the value of n & g:");

scanf("%d %d", &n, &g);

printf("\nEnter the value of x & y:");

scanf("%d %d", &x, &y);

// Key generation steps

a = alice(n, g, x);

printf("\nAlice generate end value is %d:", a);

b = bob(n, g, y);

printf("\nBob generate end value is %d:", b);

// Shared key computation

k1 = alice(n, b, x);

printf("\nValue of k1 is %d:", k1);

k2 = bob(n, a, y);

printf("\nValue of k2 is %d:", k2);

// Key comparison

if (k1 == k2) {

printf("\nMessage is shared secretly between Alice and Bob");

} else {

printf("\nMessage is not shared");

}

return 0;

}

The main function initializes variables for public keys (n and g), private keys (x and y), and shared keys (k1 and k2).

Alice and Bob calculate their public values using alice and bob functions.

Both parties compute the shared secret keys (k1 and k2) using the other party's public value and their private key.

Finally, the code checks if the shared keys match, indicating successful key exchange.

### Experiment No: 5

**Title:** Write a Program to Implement Columnar Cipher Text **Aim:** Implementation of Columnar Transposition Technique. **Theory:**

A kind of mapping is achieved by performing some sort of permutation on the plaintext letters. This technique is referred to as a transposition cipher.

A complex scheme is to write the message in a rectangle, row by row, and read the message off, column by column, but permute the order of the columns. The order of the columns then becomes the key to the algorithm. For example

Key: 4 3 1 2 5 6 7

Plaintext: a t t a c k p

o s t p o n e d u n t i l t

w o a m x y z

Ciphertext: TTNAAPTMTSUOAODWCOIXKNLYPETZ

Thus, in this example, the key is 4312567.To encrypt, start with the column that is labeled 1, in this case column 3.Write down all the letters in that column. Proceed to column 4, which is labeled 2, then column 2, then column 1, then columns 5, 6,

and 7.

A pure transposition cipher is easily recognized because it has the same letter frequencies as the original plaintext. For the type of columnar transposition just shown, cryptanalysis is fairly straightforward and involves laying out the ciphertext in a matrix and playing around with column positions.

The transposition cipher can be made significantly more secure by performing more than one stage of transposition. The result is a more complex permutation that is not easily reconstructed. Thus, if the foregoing message is reencrypted using the same algorithm,

Key: 4 3 1 2 5 6 7

Input: t t n a a p t

m t s u o a o d w c o i x k n l y p e t z

Output: NSCYAUOPTTWLTMDNAOIEPAXTTOKZ

To visualize the result of this double transposition, designate the letters in the original plaintext message by the numbers designating their position. Thus, with 28 letters in the message, the original sequence of letters is

01 02 03 04 05 06 07 08 09 10 11 12 13 14

15 16 17 18 19 20 21 22 23 24 25 26 27 28

After the first transposition, we have

03 10 17 24 04 11 18 25 02 09 16 23 01 08

15 22 05 12 19 26 06 13 20 27 07 14 21 28

which has a somewhat regular structure. But after the second transposition, we have17 09 05 27 24 16 12 07 10 02 22

20 03 25

15 13 04 23 19 14 11 01 26 21 18 08 06 28

This is a much less structured permutation and is much more difficult to cryptanalyze.

**Program:**

#include<stdio.h>

#include<string.h>

void cipher(int i, int r);

void makeArray(int col, int row);

int findMin();

char arr[22][22], darr[22][22], emessage[111], retmessage[111], key[55], temp[55], temp2[55];

int k = 0;

int main() {

char message[111];

int i, j, klen, emlen, flag = 0;

int r, c, index, min, rows;

printf("Enter the key:\n");

fflush(stdin);

gets(key);

printf("\nEnter the message to be ciphered:\n");

fflush(stdin);

gets(message);

strcpy(temp, key);

klen = strlen(key);

k = 0;

for (i = 0;; i++) {

if (flag == 1)

break;

for (j = 0; key[j] != '\0'; j++) {

if (message[k] == '\0') {

flag = 1;

arr[i][j] = '-';

} else {

arr[i][j] = message[k++];

}

}

}

r = i;

c = j;

for (i = 0; i < r; i++) {

for (j = 0; j < c; j++) {

printf("%c ", arr[i][j]);

}

printf("\n");

}

k = 0;

for (i = 0; i < klen; i++) {

index = findMin();

cipher(index, r);

}

emessage[k] = '\0';

printf("\nEncrypted message is:\n");

for (i = 0; emessage[i] != '\0'; i++)

printf("%c", emessage[i]);

printf("\n\n");

// Deciphering

emlen = strlen(emessage);

strcpy(temp, key);

rows = emlen / klen;

j = 0;

for (i = 0, k = 1; emessage[i] != '\0'; i++, k++) {

temp2[j++] = emessage[i];

if ((k % rows) == 0) {

temp2[j] = '\0';

index = findMin();

makeArray(index, rows);

j = 0;

}

}

printf("\nArray Retrieved is:\n");

k = 0;

for (i = 0; i < r; i++) {

for (j = 0; j < c; j++) {

printf("%c ", darr[i][j]);

retmessage[k++] = darr[i][j];

}

printf("\n");

}

retmessage[k] = '\0';

printf("\nMessage retrieved is:\n");

for (i = 0; retmessage[i] != '\0'; i++)

printf("%c", retmessage[i]);

getchar();

return 0;

}

void cipher(int i, int r) {

int j;

for (j = 0; j < r; j++) {

emessage[k++] = arr[j][i];

}

}

void makeArray(int col, int row) {

int i;

for (i = 0; i < row; i++) {

darr[i][col] = temp2[i];

}

}

int findMin() {

int i, j, min, index;

min = temp[0];

index = 0;

for (j = 0; temp[j] != '\0'; j++) {

if (temp[j] < min) {

min = temp[j];

index = j;

}

}

temp[index] = 123;

return index;

}

### Output:

Enter the key hello

Enter the message to be ciphered how are you

h o w a r e y o u - - - -

Encrypted message is oe-hruw - y-ao-

Array Retrieved is h o w a

r e y o u - - - -

Message retrieved is how are you----

Certainly! The code you provided is an implementation of a transposition cipher, specifically a columnar transposition cipher. This cipher rearranges the characters of a message based on a provided key, and it can be used for both encryption and decryption.

Let's go through the code step by step:

Header Files:

c

Copy code

#include<stdio.h>

#include<string.h>

Standard C library headers for input/output and string manipulation.

Function Prototypes:

c

Copy code

void cipher(int i, int r);

void makeArray(int col, int row);

int findMin();

Prototypes for functions used in the main program.

Global Variables:

c

Copy code

char arr[22][22], darr[22][22], emessage[111], retmessage[111], key[55], temp[55], temp2[55];

int k = 0;

Arrays to store the matrix, messages, key, and temporary variables. The k variable is used to keep track of the current position in the messages.

Main Function:

c

Copy code

int main() {

// Function implementations

}

The main entry point of the program.

User Input:

c

Copy code

printf("Enter the key:\n");

fflush(stdin);

gets(key);

printf("\nEnter the message to be ciphered:\n");

fflush(stdin);

gets(message);

The user is prompted to enter a key and a message to be encrypted.

Matrix Initialization:

c

Copy code

for (i = 0;; i++) {

if (flag == 1)

break;

for (j = 0; key[j] != '\0'; j++) {

if (message[k] == '\0') {

flag = 1;

arr[i][j] = '-';

} else {

arr[i][j] = message[k++];

}

}

}

This loop initializes a 2D array (arr) with the message characters based on the key. If the message is shorter than the matrix, the remaining spaces are filled with the character '-'.

Printing the Matrix:

c

Copy code

for (i = 0; i < r; i++) {

for (j = 0; j < c; j++) {

printf("%c ", arr[i][j]);

}

printf("\n");

}

The program prints the initialized matrix.

Encryption:

c

Copy code

for (i = 0; i < klen; i++) {

index = findMin();

cipher(index, r);

}

The program encrypts the message using the transposition cipher. It iterates over the key characters, finds the minimum character in the key, and then calls the cipher function.

Printing Encrypted Message:

c

Copy code

emessage[k] = '\0';

printf("\nEncrypted message is:\n");

for (i = 0; emessage[i] != '\0'; i++)

printf("%c", emessage[i]);

The encrypted message is printed.

Decryption:

c

Copy code

// Deciphering

emlen = strlen(emessage);

strcpy(temp, key);

rows = emlen / klen;

j = 0;

for (i = 0, k = 1; emessage[i] != '\0'; i++, k++) {

temp2[j++] = emessage[i];

if ((k % rows) == 0) {

temp2[j] = '\0';

index = findMin();

makeArray(index, rows);

j = 0;

}

}

The program decrypts the message. It calculates the number of rows and uses the temp array to reconstruct the original matrix. The makeArray function is called to form the matrix.

Printing Decrypted Message:

c

Copy code

printf("\nMessage retrieved is:\n");

for (i = 0; retmessage[i] != '\0'; i++)

printf("%c", retmessage[i]);

The decrypted message is printed.

Helper Functions:

cipher(int i, int r): Rearranges characters in the arr matrix based on the column index i and the number of rows r.

makeArray(int col, int row): Reconstructs the original matrix (darr) based on the column and row indices.

findMin(): Finds the index of the minimum character in the temp array.

# Experiment No: 6

**Title:** Calculate the message digest of a text using the SHA-256 algorithm.

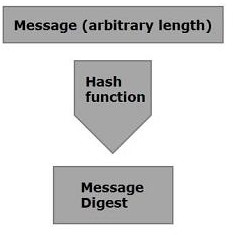
**Aim:** To develop a program for Calculate the message digest of a text using the SHA-256 algorithm using Java.

### Theory:

Hash functions are extremely useful and appear in almost all information security applications. A hash function is a mathematical function that converts a numerical input value into another compressed numerical value. The input to the hash function is of arbitrary length but output is always of fixed length.

Values returned by a hash function are called message digest or simply **hash**

values. The following picture illustrated hash function.



Java provides a class named **MessageDigest** which belongs to the package java.security. This class supports algorithms such as SHA-1, SHA 256, MD5 algorithms to convert an arbitrary length message to a message digest.

To convert a given message to a message digest, follow the steps given below –

### Step 1: Create a MessageDigest object

The MessageDigest class provides a method named **getInstance()**. This method accepts a String variable specifying the name of the algorithm to be used and returns a MessageDigest object implementing the specified algorithm.

Create MessageDigest object using the **getInstance()** method as shown below. MessageDigest md = MessageDigest.getInstance("SHA-256");

Step 2: Pass data to the created MessageDigest object

After creating the message digest object, you need to pass the message/data to it. You can do so using the **update()** method of the **MessageDigest** class, this method accepts a byte array representing the message and adds/passes it to the above created MessageDigest object.

md.update(msg.getBytes());

Step 3: Generate the message digest

You can generate the message digest using the **digest()** method od the MessageDigest class this method computes the hash function on the current object and returns the message digest in the form of byte array.

Generate the message digest using the digest method. byte[] digest = md.digest();

### Algorithm:

1. Read data from user.
2. Create the MessageDigest object.
3. Pass the data to created MessageDigest object.
4. Compute the message digest.
5. Convert the byte array in to HexString fromat.
6. Display the message digest.

### Program:

### import java.security.MessageDigest;

### import java.util.Scanner;

### public class MessageDigestExample {

### public static void main(String args[]) throws Exception {

### // Reading data from the user

### Scanner sc = new Scanner(System.in);

### System.out.println("Enter the message");

### String message = sc.nextLine();

### // Creating the MessageDigest object

### MessageDigest md = MessageDigest.getInstance("SHA-256");

### // Passing data to the created MessageDigest Object

### md.update(message.getBytes());

### // Compute the message digest

### byte[] digest = md.digest();

### // Converting the byte array to HexString format

### StringBuffer hexString = new StringBuffer();

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

### // Convert each byte to a two-digit hexadecimal representation

### String hex = Integer.toHexString(0xFF & digest[i]);

### if (hex.length() == 1) {

### hexString.append('0'); // Add leading zero if needed

### }

### hexString.append(hex);

### }

### System.out.println("Hex format: " + hexString.toString());

### }

### }

### Output:

Enter the message Hello how are you [B@55f96302

Hex format: 2953d33828c395aebe8225236ba4e23fa75e6f13bd881b9056a3295cbd64d3

Certainly! The provided Java code demonstrates the use of the MessageDigest class to compute the SHA-256 hash of a user-inputted message and then convert the hash to a hexadecimal string.

Let's break down the code:

Import Statements:

java

Copy code

import java.security.MessageDigest;

import java.util.Scanner;

Import statements to include necessary classes from the Java standard library.

Class Definition:

java

Copy code

public class MessageDigestExample {

The start of the class definition named MessageDigestExample.

Main Method:

java

Copy code

public static void main(String args[]) throws Exception {

The main method where the execution of the program begins.

User Input:

java

Copy code

Scanner sc = new Scanner(System.in);

System.out.println("Enter the message");

String message = sc.nextLine();

It creates a Scanner object to read user input from the console. The user is prompted to enter a message, and the entered line is stored in the message variable.

MessageDigest Initialization:

java

Copy code

MessageDigest md = MessageDigest.getInstance("SHA-256");

It creates an instance of the MessageDigest class using the SHA-256 algorithm. The getInstance method throws an exception (NoSuchAlgorithmException) if the specified algorithm is not available.

Updating Message Digest:

java

Copy code

md.update(message.getBytes());

The message is converted to bytes using getBytes() and then passed to the update method of the MessageDigest object. This updates the message digest with the input data.

Computing Message Digest:

java

Copy code

byte[] digest = md.digest();

The digest method is called to compute the final hash value, which is represented as an array of bytes.

Converting to Hexadecimal Format:

java

Copy code

StringBuffer hexString = new StringBuffer();

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

String hex = Integer.toHexString(0xFF & digest[i]);

if (hex.length() == 1) {

hexString.append('0'); // Add leading zero if needed

}

hexString.append(hex);

}

The digest array is iterated, and each byte is converted to a two-digit hexadecimal representation. The resulting hex digits are appended to the hexString buffer. Leading zeros are added if necessary.

Printing Hexadecimal Format:

java

Copy code

System.out.println("Hex format: " + hexString.toString());

The final hexadecimal string is printed to the console.

Closing Scanner:

java

Copy code

sc.close();

The Scanner object is closed to release associated resources.

End of Main Method and Class:

java

Copy code

}

}

Closing braces indicating the end of the main method and the MessageDigestExample class.

## EXPERIMENT NO: 7

**TITLE:** Demonstrate intrusion detection system (ids) using any tool.

**AIM:** Configure and implement IDS using SNORT.

## THEORY:

Intrusion detection system (ID) is a type of security system for computers and computer networks. Intrusion Detection basically helps in detecting outer and inner attacks performed by either user or hackers. An ID system collects information from various sources and analyzes information from various areas within a computer or a network to identify possible security breaches, which include both intrusions (attacks from outside the organization) and misuse (attacks from within the organization). ID uses vulnerability assessment (sometimes referred to as scanning), which is a technology developed to assess the security of a computer system or network.

Snort is a light-weight intrusion detection tool which logs the packets coming through the network and analyzes the packets. Snort checks the packets coming against the rules written by the user and generate alerts if there are any matches found. The rules are written by the user in a text file which is linked with snort.conf file where all the snort configurations are mentioned. There are few commands which is used to get snort running so that it can analyze network behavior.

**Configuration of SNORT**

Setting up Snort from the source code consists of a couple of steps: downloading the code, configuring it, compiling the code and lastly installing it. First up make a temporary download folder to your home directory and then move into it with the these commands

mkdir ~/snort\_src cd ~/snort\_src

After it is downloaded we have to configure the downloaded code.

Snort is a light-weight intrusion detection tool which logs the packets coming through the network and analyzes the packets. Snort checks the packets coming against the rules written by the user and generate alerts if there are any matches found. The rules are written by the user in a text file which

is linked with snort.conf file where all the snort configurations are mentioned. There are few commands which is used to get snort running so that it can analyze network behavior.

**Snort in packet sniffer mode**

If we want to print out the TCP/IP packet headers to the screen following command is used:

./snort -v

If we want to see the application data in transit, following command is used

./snort -vd

This instructs Snort to display the packet data as well as the headers.

If we want more descriptive display, showing the data link layer headers, following command need to be run.

./snort -vde

As an aside, notice that the command line switches can be

listed separately or in a combined form. The last command could also be typed out as:

./snort -d -v –e

This will produce the same result.

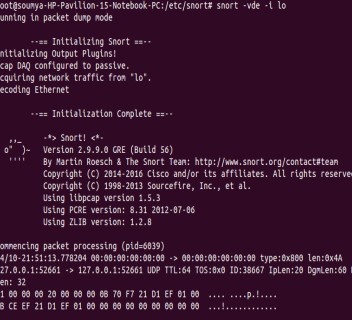


Figure: Running snort in packet sniffer mode

**Snort using snort.conf file**

Snort uses a configuration file at start up time. A sample configuration file snort. conf is included in the Snort distribution. You can use any name for the configuration file, however snort.conf is the conventional name. You use the - c command line switch to specify the name of the configuration file.

The following command uses /opt/snort/snort.conf as the configuration file.

We can also save the configuration file in our home directory as snortrc, but most commonly used method is specifying it on the command line. There are other advantages to using the configuration file name as a command line argument to Snort. It is possible to invoke multiple Snort instances on different network interfaces with different configuration.

$ sudo /usr/local/bin/snort -A console -q -u snort -g snort -c /etc/snort/snort.conf -i lo

This command should be run in our terminal to run snort using our snort configuration file. It can be modified according to the user suitability. Snort has various modes; few of them are listed here Description of the command: -

c: specifies the config file

-i : specifies the interface mode , if a loopback address is running then “lo” will be written , for Ethernet “eth0” or “eth1” will be written.

-A: It will print the output to the console

Once we run this command, then type $ ping 127.0.0.1

We should see that the snort logs this packet and displays it on the terminal. Here is the image of the terminal logging the ping packets.

**Writing rules**

Rules are written by the user, snort will log the packets and generate alert if there if finds any match with the rules that user defined in the rules file.

Here is an example of how to write rules.

1. alert ip $EXTERNAL\_NET any -> $HOME\_NET any (ip\_proto:igmp; rev:1000000)

For igmp traffic

1. alert tcp any any -> any 80 (content:"ABC"; content:"EFG"; http\_raw\_cookie;rev:1000001)

detects unnormalised cookie header

1. alert tcp any any -> any (msg:"exploit"; content:"|90|"; rev:1000002)
2. alert $EXATERNAL\_NET any -> $HOME\_NET any (flags: SF,12; msg:"SYN FIN scan"; rev:1000003)

installation

To install Suricata, an open-source Intrusion Detection System (IDS), you can follow the steps below on a Linux system, such as Ubuntu. These instructions assume you have administrative privileges on the machine. The example uses Ubuntu, but you can adapt the instructions for other Linux distributions as needed.

\*Step 1: Update Your System\*

Before installing any software, it's a good practice to update your system to ensure you have the latest package information and security updates. Open a terminal and run the following commands:

bash

sudo apt update

sudo apt upgrade

\*Step 2: Install Suricata\*

You can install Suricata from the default repositories on Ubuntu. Open a terminal and run the following command:

bash

sudo apt install suricata

\*Step 3: Verify Installation\*

Once the installation is complete, you can verify that Suricata has been installed correctly. Run the following command to check the Suricata version:

bash

suricata --version

This command should display the installed Suricata version.

\*Step 4: Configure Suricata\*

Suricata's configuration files are typically located in the /etc/suricata/ directory. You may need to customize these configurations to suit your specific network environment and monitoring needs.

\*Step 5: Start Suricata\*

To start Suricata, run the following command with administrative privileges:

bash

sudo suricata -c /etc/suricata/suricata.yaml -D

This command starts Suricata in daemon mode and loads the configuration from the specified file. The -D flag runs Suricata as a daemon in the background.

\*Step 6: Monitor Suricata Logs\*

Suricata logs its activities in various log files, including alerts and fast logs. You can monitor these logs to observe network traffic and detect potential intrusion attempts. For example, to monitor the fast.log file, you can use the following command:

bash

sudo tail -f /var/log/suricata/fast.log

This command will display log entries in real-time.

\*Step 7: Stop Suricata\*

To stop Suricata, you can run the following command:

bash

sudo suricata -q --pidfile /var/run/suricata.pid -x /var/run/suricata.pid

This command gracefully stops Suricata.

Please note that the actual configuration and usage of Suricata may vary depending on your specific use case and network environment. Be sure to consult the Suricata documentation and configure it according to your needs for intrusion detection and network monitoring.

Implementation

For the environment setup and configuration:

Implementing an Intrusion Detection System (IDS) experiment in an Information Security Lab is a comprehensive task that typically involves setting up a network environment, installing IDS software, generating network traffic, and analyzing the results. Below, I provide a high-level outline of the steps involved and some sample code for a basic network setup and using the Suricata IDS for experimentation.

\*Prerequisites:\*

- A Linux environment (e.g., Ubuntu or CentOS).

- Administrative privileges on the machine.

- A network interface card (NIC) for monitoring network traffic.

\*Step 1: Set Up the Environment\*

Ensure you have Suricata installed. You can install it on Ubuntu using:

bash

sudo apt-get install suricata

\*Step 2: Configure Suricata\*

You need to create a configuration file for Suricata. A basic configuration file may look like this:

yaml

af-packet:

- interface: eth0

- cluster-type: cluster\_flow

cluster-id: 99

outputs:

- fast:

enabled: yes

filename: fast.log

append: yes

\*Step 3: Start Suricata\*

You can start Suricata in the background using the following command:

bash

sudo suricata -c /etc/suricata/suricata.yaml -D

\*Step 4: Generate Network Traffic\*

You can use tools like ping, curl, or wget to generate network traffic. For example, to generate ICMP (ping) traffic:

bash

ping -c 10 google.com

\*Step 5: Monitor Suricata Logs\*

Suricata logs intrusion attempts in its log files. You can monitor these log files in real-time or after the experiment. For real-time monitoring, you can use the tail command:

bash

tail -f /var/log/suricata/fast.log

\*Step 6: Analyze the Results\*

Analyze the logs to detect and identify intrusion attempts. You can use custom scripts or tools to parse and analyze Suricata's log files.

Here's a simple Python script to parse Suricata's Fast Log and identify potential intrusion attempts:

python

import re

log\_file = "/var/log/suricata/fast.log"

def parse\_suricata\_log(log\_file):

with open(log\_file, 'r') as file:

for line in file:

if "ET" in line: # Example: ET MALWARE Possible Malicious Inbound to My SQL Server

print("Intrusion detected:", line.strip())

parse\_suricata\_log(log\_file)

\*Note\*: This is a very basic example, and IDS experimentation can be complex, depending on your lab's goals. More advanced experiments may involve setting up multiple hosts, using different attack scenarios, and analyzing the results more comprehensively.

Make sure to adapt and expand this code to meet the specific requirements and goals of your IDS experiment in your Information Security Lab.