Annexure No : 1

Aim : Implement ceaser cipher and apply brute force attack to get original key.

Description: In this practical, you will learn and implement the ceaser cipher, a simple encryption algorithm where each letter is shifted by a fixed number of positions. You will then apply a brute force attack to systematically try all possible shifts and recover the original key.

Algorithm:

step-1: Choose a shift value (key): A number between 1 and 25 that determines how much each ~~other~~ letter will be shifted.

step-2: Shift each letter: For each letter in the plaintext, replace it with the letter that is shifted by the key positions in the alphabet. If the shift goes past 'Z', it wraps around to the beginning of the alphabet.

step-3: Encrypt: The resulting sequence of shifted letters forms the ciphertext.

Encryption Formula: For each ~~letter 'P' in the~~ plaintext:
ciphertext ~~(c) is~~ $\boxed{C = (P + Key) \% 26}$ [where, P = position of letter in alphabet]

Example:
plainText: "HELLO"
Key: 3
shift 'H' by 3 → 'K'
shift 'E' by 3 → 'H'
shift 'L' by 3 → 'O'
shift 'L' by 3 → 'O'
shift 'O' by 3 → 'R'

CipherText: "KHOOR"

Code:

```
#include<stdio.h>
#include<string.h>
void ceaser_encrypt(char*plaintext, int key, char *ciphertext){
    for(int i=0;plaintext[i]!='\0';i++){
        char ch=plaintext[i];
        if(ch>='A' && ch<='Z'){
            ciphertext[i] =((ch-'A'+key)%26)+'A';
            printf("Encrypting: plaintext[%d]= %c, ciphertext[%d]= %c\n",i,ch,i, ciphertext[i]);

        }
        else ciphertext[i]=ch;
    }
    ciphertext[strlen(plaintext)]='\0';
}
void ceaser_decrypt(char *ciphertext,int key, char *decrypt_text){
    int i=0;
    for(int i=0;ciphertext[i]!='\0';i++){
        char ch=ciphertext[i];
        if(ch>='A' && ch<='Z'){
            decrypt_text[i]=((ch-'A'-key+26)%26)+'A';
            printf("Decrypting: ciphertext[%d]= %c, Decrypted_Text[%d]=%c'\n",i,ch,i, decrypt_text[i]);

        }
        else decrypt_text[i]=ch;
        decrypt_text[i]='\0';
}
void brute_force_attack(char *ciphertext){
    char decrypt_text[100];
    for(int key =0; key<26;key++){
        ceaser_decrypt(ciphertext, key, decrypt_text);
        printf("key: %d : %s\n", key, decrypt_text);

    }
}
```

Annexure No :

```
int main(){
   char plaintext[100], ciphertext[100], decrypt_text[100];
   int key;
   printf("Enter the plaintext(Upper letters only): ");
   fgets(plaintext, sizeof(plaintext), stdin);
   printf("Enter the key(0-25): ");
   scanf("%d", &key);
   ceaser_encrypt(plaintext, key, ciphertext);
   printf("Attempting Brute Force Attack...\n");
   brute_force_attack(ciphertext);
   return 0;
}
```

Output:

```
Enter the plaintext(Upper letters only): CYBER
Enter the key(0-25):5
Encrypting: plaintext[0]='C', ciphertext[0]='H'
Encrypting: plaintext[1]='Y', ciphertext[1]='D'
Encrypting: plaintext[2]='B', ciphertext[2]='G'
Encrypting: plaintext[3]='E', ciphertext[3]='J'
Encrypting: plaintext[4]='R', ciphertext[4]='W'
Encrypted Text:HDGJW

Attempting Brute Force Attack...
Decrypting: ciphertext[0]='H', decrypt_text[0]='H'
Decrypting: ciphertext[1]='D', decrypt_text[1]='D'
Decrypting: ciphertext[2]='G', decrypt_text[2]='G'
Decrypting: ciphertext[3]='J', decrypt_text[3]='J'
Decrypting: ciphertext[4]='W', decrypt_text[4]='W'
key:0
```

Decrypting : ciphertext[0] = 'H', decrypt_text[0] = 'G'
Decrypting : ciphertext [1] = 'D', decrypt_text[1] = 'E'
Decrypting : ciphertext[2] = 'G', decrypt_text [2] = 'F'
Decrypting : ciphertext[3] = 'J', decrypt_text [3] = 'I'
Decrypting : ciphertext[4] = 'W', decrypt_text [4] = 'V'
key : 1

Decrypting : ciphertext[0] = 'H', decrypt_text[0] = 'C'
Decrypting : ciphertext[1] = 'D', decrypt_text[1] = 'Y'
Decrypting : ciphertext[2] = 'G', decrypt_text[2] = 'B'
Decrypting : ciphertext[3] = 'J', decrypt_text [3] = 'E'
Decrypting : ciphertext[4] = 'W', decrypt_text[4] = 'R'
key : 5

20 9
2/1/25

Decrypting : ciphertext[0] = 'H', decrypt_text[0] = 'I'
Decrypting : ciphertext[1] = 'D', decrypt_text[1] = 'E'
Decrypting : ciphertext[2] = 'G', decrypt_text [2] = 'H'
Decrypting : ciphertext[3] = 'J', decrypt_text[3] = 'K'
Decrypting : ciphertext[4] = 'W', decrypt_text [4] = 'X'
key : 25

Conclusion :

In conclusion, the Ceaser Cipher is vulnerable to Brute Force Attacks, as all possible key shifts can be easily text. This highlights it's weakness and shows it's not suitable for securing sensitive data.

Annexure No : 2

Aim : Apply attacks for cryptanalysis to decrypt the original message from a given cipher text using play Fair cipher. key = Parul

Description: In this practical, you will decrypt a cipher text encrypted with the playfair cipher using the key "Parul". By applying cryptanalysis, you will recover the original message from the ciphertext.

Algorithm:

Step-1: Create a 5x5 matrix: Use the key (eg: "KEYWORD"), remove duplicates and fill the matrix with the key followed by the remaining letters of the alphabet (excluding 'J').

Step-2: Prepare the plaintext: Split the message into diagraphs (pair of 2-letters). If a pair has identical letters (eg: "LL"), replace one with an 'X'. Add "X" if the number of characters is odd.

Step-3: Encrypt the Diagraphs:

(i.) Same Row: Replace with the letters to the right.

(ii.) Same Column: Replace with the letters below.

(iii.) Rectangle: Swap the corners.

Step-4: Form the ciphertext: Combine the encrypted diagraphs to form the final ciphertext.

Example:

keyword: "KEYWORD"

Matrix:

K	E	Y	W	O
R	D	A	B	C
F	G	H	I	L
M	N	P	Q	S
T	U	V	X	Z

Plain Text: "HELLO"

Break into pairs: HE LX LO

Encrypt each pair: HE → IG (rectangle rule)

LX → IZ (rectangle rule)

LO → CS (same column)

Cipher Text: IG IZ CS

Code:


```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define SIZE 5
char key[ ]= "Parul";
char matrix[SIZE][SIZE];
void prepareMatrix(char *x){
    int used[26]={0};
    used['J'-'A']=1 // J & I share same slot in PlayFair matrix
    int index=0;
    for(int i=0;x[i]!='\0';i++){
        char ch=toupper(x[i]);
        if(!used[ch-'A']&& isalpha(ch)){
            matrix[index/SIZE][index%SIZE]=ch;
            used[ch-'A']=1;
            index++;
        }
    }
    for(char ch='A';ch<='Z';ch++){
        if(! used[ch-'A']){
            matrix[index/SIZE][index%SIZE]=ch;
            used[ch-'A']=1;
            index++;
        }
    }
}
void displayMatrix(){
    printf("Play Fair Matrix:\n");
    for(int i=0;i<SIZE;i++){
        for(int j=0;j<SIZE;j++){
            printf("%c",matrix[i][j]);}
        printf("\n");
    }
```

```
void Find Position(char ch, int *row, int *col){
    if(ch=='J')ch='I';
    for(int i=0; i<SIZE; i++){
        for(int j=0;j<SIZE;j++){
            if(matrix[i][j]==ch){
                *row=i;
                *col=j;
                return;
            }
        }
    }
}
void decryptPlayfair(char *ciphertext, char *plaintext){
    int len=strlen(ciphertext);
    int i=0;
    if(len%2 !=0){
        ciphertext[len] = 'x';
        ciphertext[len+1] = '\0';
        len++;
    }
    for(i=0; i<len; i+=2){
        char a=toupper(ciphertext[i]);
        char b=toupper(ciphertext[i+1]);
        int row1, row2, col1, col2;
        FindPosition(a, &row1, &col1);
        FindPosition(b, &row2, &row2);
        if(row1 == row2){
            plaintext[i] =matrix[row1][(col1-1+SIZE)% SIZE];
            plaintext[i+1]= matrix[row2][(col2-1+SIZE)% SIZE];
        }
        elseif (col1 ==col2){
            plaintext[i]= matrix[(row1-1+SIZE)%SIZE][col1];
            plaintext[i+1]=matrix[(row2-1+SIZE)%SIZE][col2];
        }
```

```
    else{
        plaintext[i]=matrix[row1][col2];
        plaintext[i+1]=matrix[row2][col1];
    }
  }
  plaintext[len]='\0';
}
int main(){
   char ciphertext[100],plaintext[100];
   printf("Enter the ciphertext: ");
   scanf("%s",ciphertext);
   prepareMatrix(key);
   displayMatrix();
   decryptPlayfair(ciphertext,plaintext);
   printf("Encrypted Plaintext: %s\n",plaintext);
   return 0;
}
```

2/1/25 (9)

Output:

Enter the ciphertext: cryptography

playfair Cipher Matrix:

```
P A R U L
B C D E F
G H I K M
N O Q S T
V W X Y Z
```

Encrypted plaintext:DAVUSNIPPLKW

Conclusion:

By applying cryptanalytic techiniques to the playfair cipher with the key "Parul," we successfully decrypted the cipher text to real the original message.

Annexure No : 3

Aim: Implement Deffi Hellmn key exchange algorithm. Generate share secret without sharing the secret code.

Description:

The Diffie-Hellman key exchange allows two parties to securely share a secret key over a public channel without transmitting the secret itself. Each part generates a private and public key, then exchanges public keys to compute the shared secret. The security relies on the difficulty of deriving the secret from the public keys.

Algorithm:

(i.) Consider a large prime number 'q'.

(ii.) suppse 'α' is the primitive root of 'q'.

(iii.) Let X = private key and Y = public key of users.

$\boxed{Y_A = \alpha^{X_A} \bmod q}$ and $\boxed{Y_B = \alpha^{X_B} \bmod q}$ where $(X_A \,\&\, X_B) < q$

(iv.) If $\boxed{K_A = K_B}$ then key exchange was successful, where $\boxed{K_A = (Y_B)^{X_A} \bmod q}$ & $\boxed{K_B = (Y_A)^{X_B} \bmod q}$

Example:

$q = 7 ; \alpha = 5 ; X_A = 3 ; X_B = 4$

$Y_A = \alpha^{X_A} \bmod q$	$K_A = (Y_B)^{X_A} \bmod q$
$Y_A = 5^3 \bmod 7$	$K_A = 2^3 \bmod 7$
$\boxed{Y_A = 6}$	$\boxed{K_A = 1}$
↳ Private key of 'A'	$K_B = (Y_A)^{X_B} \bmod q$
$Y_B = \alpha^{X_B} \bmod q$	$K_B = 6^4 \bmod 7$
$Y_B = 5^4 \bmod 7$	$\boxed{K_B = 1}$
$\boxed{Y_B = 2}$	$\boxed{K_A = K_B}$ $\boxed{\therefore \text{key exchange is possible.}}$
↳ Private key of 'B'	

Annexure No :

Code:

```
#include<stdio.h>
#include<math.h>
int modular_exponent(int base, int exp, int mod){
    int result=1;
    base=base%mod;
    printf("Initial base: %d\n",base);
    while(exp>0){
        printf("Current exp: %d, Result:%d, Base:%d\n",exp,result,base);
        if(exp%2==1){
            result=(result*base)% mod;
            printf("Odd exp -> Result updated to %d\n",result);
        }
        else printf("Even exp->No change in result.\n");
        exp=exp/2;
        printf("Exp divided by 2->exp updated to %d\n",exp);
        base=(base*base)% mod;
        printf("Base square->Base updated to %d\n",base);
    }
    return result;
}
int main(){
    int p,g; //p=prime number & g=primitive root
    int private_key_A,private_key_B,public_key_A,public_key_B;
    printf("Enter a prime number: ");
    scanf("%d",&p);
    printf("Enter a primitive root modulo %d(g): ",p);
    scanf("%d",&g);
```

```c
printf("Enter Alice's private key: ");
scanf("%d",&private_key_A);
printf("Enter Bob's private key: ");
scanf("%d",&private_key_B);
printf("-------------\n");
printf("Calculating Alice's public key...");
public_key_A=modular_exponent(g,private_key_A,p);
printf("-------------\n");
printf("Alice's publickey: %d \n, public_key_A);
printf("-------------\n");
printf("Calculating Bob's public key...");
public_key_B=modular_exponent(g,private_key_B,p);
printf("-------------\n");
printf("Bob's public key: %d \n,public_key_B);
printf("-------------\n");
int shared_secret_A=modular_exponent(public_key_B,private_key_A,p);
int shared_secret_B=modular_exponent(public_key_A,private_key_B,p);
printf("\nShared secret calculated by Alice: %d\n",shared_secret_A);
printf("shared secret calculated by Bob: %d\n",shared_secret_B);
if(shared_secret_A==shared_secret_B){
    printf("\n-------------\n");
    printf("key exchange successful");
    printf("\n-------------\n");
}else{
    printf("\n-------------\n");
    printf("key exchange failed, secrets do not match...");
    printf("\n-------------\n");
}
return 0;
```

Output:

Annexure No :

Enter a prime number : 7
Enter a primitive root modulo 7 (g): 5
Enter Alice's private key: 6
Enter Bob's private key: 3
- - - - - - - - - - - - - - - - -
Calculating Alice's public key...
Initial base: 5
Current exp: 6, Result: 1, Base: 5
Even exp → No change in result.
Exp divided by 2 → exp updated to 3
Base squared → Base updated to 4
Current exp: 3, Result: 1, Base: 4
Odd exp → Result updated to 4
Exp divided by 2 → exp updated to 1
Base squared → Base updated to 2
Current exp: 1, Result: 4, Base: 2
Odd exp → Result updated to 1
Exp divided by 2 → exp updated to 0
Base squared → Base updated to 4
- - - - - - - - - - - - - - - - -
Alice's public key: 1
- - - - - - - - - - - - - - - - -
Calculating Bob's public key...
Initial base: 5
Current exp: 3, result: 1, base: 5
Odd exp → Result updated to 5
Exp divided by 2 → exp updated to 1
Base squared → Base updated to 4

current exp: 1, Result: 5, Base: 4
Odd exp -> result updated to 6
Exp divided by 2 -> exp updated to 0
Base squared -> Base updated to 2

Bob's public key: 6

Initial base: 6
current exp: 6, Result: 1, Base: 6
Even exp -> No change in result.
Exp divided by 2 -> exp updated to 3
Base squared -> base updated to 1
current exp: 3, Result: 1, Base: 1
Odd exp -> Result updated to 1
Exp divided by 2 -> exp updated to 1
current exp: 1, Result: 1, Base: 1
Odd exp -> result updated to 1
Exp divided by 2 -> exp updated to 0
Base squared -> base updated to 1
Initial base: 1
current exp: 3, Result: 1, Base: 1
odd exp -> result updated to 1
Exp divided by 2 -> exp updated to 1
Base squared -> base updated to 1
current exp: 1, Result: 1, Base: 1
odd exp -> result updated to 1
Exp divided by 2 -> exp updated to 0
Base squared -> base updated to 1

shared secret calculated by Alice: 1
shared secret calculated by Bob: 1

---

key exchange successful

Conclusion:

The Diffie Hellman key exchange algorithm provides a secure method for two parties to establish a shared secret over an insecure channel, without transmitting the secret itself. By leveraging mathematical properties of prime numbers and modular arithmetic, the protocol ensures that even if an attacker intercepts the exchanged public keys, they cannot easily derive the shared secret.

10
8/2/25

Annexure No : 4

Aim: Implement and analyze DES algorithm.

Description: The Data Encryption Standard (DES) is a symmetric-key block cipher that was widely used for data encryption. It operates on fixed-size blocks of data (64 bits) and uses a 56-bit key for encryption and decryption. DES is based on a series of transformations, including permutations, substitutions and the use of multiple subkeys, to securely encrypt plaintext into ciphertext and viseversa.

Algorithm:

(i) Initial Permutation (IP): The 64-bit block undergoes an initial permutation.

(ii) Key Schedule: A 56-bit key is used to generate 16 subkeys, each of 48 bits, which are used in each of the 16 rounds.

(iii) Rounds:

> For each round, right half of the block is expanded from 32 to 48 bits using the expansion table.

> The expanded block is XOR'ed with the current round subkey.

> The result is passed through 8 S-boxes, which reduce the result to 32-bits.

> The 32-bit output is then permuted (P_4).

> The left half is XOR'ed with the result, and the halves are swapped.

(iv) Final Permutation (FP): After 16 rounds, the resulting block undergoes a final permutation.

Example:

Key: 0x133457799BBCDFF1 (56 Bits)

Plain Text: 0x0123456789ABCDEF (64 Bits)

Cipher Text: 0x85E813540F0AB405

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BLOCK_SIZE 64
#define KEY_SIZE 64
int S_BOX[8][4][16]={
    // First S-Box
    {
     {14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7},{0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8},
     {4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0},{15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13}
    },
    //Other S-Boxes(7 more needed in real DES)
};
int IP[BLOCK_SIZE]{ 58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,
    //Remaining entries for full table
};
int FP[BLOCK_SIZE]={ 40,8,48,16,56,24,64,32,39,7,47,15,55,23,63,31,
    //Remaining entries for full table
};
void initialPermutation(unsigned char *block){
    unsigned char permutedBlock[BLOCK_SIZE];
    for(int i=0;i<BLOCK_SIZE;i++){
        permutedBlock[i]=block[IP[i]-1];
    }
    memcpy(block, permutedBlock, BLOCK_SIZE);

}
void finalPermutation(unsigned char *block){
    unsigned char permutedBlock[BLOCK_SIZE];
    for(int i=0;i<BLOCK_SIZE;i++){
        permutedBlock[i]=block[FP[i]-1];
    }memcpy(block,permutedBlock,BLOCK_SIZE);
}
```

```
unsigned char feistel(unsigned char right, unsigned char key){
        return (right ^ key);
}
void desEncrypt(unsigned char *block, unsigned char *key){
    unsigned char left[BLOCK_SIZE/2], right [BLOCK_SIZE/2];
    memcpy(left, block, BLOCK_SIZE/2);
    memcpy(right, block+BLOCK_SIZE/2, BLOCK_SIZE/2);
    for(int round=0; round<16; round++){
        unsigned char temp[BLOCK_SIZE/2];
        memcpy(temp, right, BLOCK_SIZE/2);
        for(int i=0; i< BLOCK_SIZE/2; i++){
            right[i] = left[i] ^ feistel(temp[i], key[round % 8]);
        }
        memcpy(left, temp, BLOCK_SIZE/2);
    }
    memcpy(block, left, BLOCK_SIZE/2);
    memcpy(block+BLOCK_SIZE/2, right, BLOCK_SIZE/2);
}
int main(){
    unsigned char block[BLOCK_SIZE] = "HELLODESBLOCK";
    unsigned char key[KEY_SIZE] = "SIMPLEKEYFORDES";
    printf("Original Block: %s \n", block);
    initialPermutation(block);
    desEncrypt(block, key);
    finalPermutation(block);
    printf("Encrypted Block: %s \n", block);
    return 0;
}
```

Output:

Original Block= HELLODESBLOCK

Encrypted Block=□□□H□H□□□□H□H□

Conclusion:

In conclusion, implementing the DES algorithm in C helps in understand the basics of symmetric encryption, key generation, and block cipher operations. While DES is now outdated, this practical provides valuable insight into cryptographic techniques that laid the foundation for modern encryption standards.

9

20/2/25

Annexure No : 5

Aim : Implement RSA cryptosystem.

Description: This practical involves creating a program to demonstrate RSA encryption and decryption. It covers key generation, modular arithmetic, and the use of public and private keys for secure communication, providing hands-on experience with cryptographic principles in 'C'.

Algorithm:

(i.) **Key Generation:**

⇒ Choose two large prime numbers 'p' & 'q'.

⇒ Calculate $\boxed{n = p \times q}$

⇒ Calculate the Euler's quotient function $\boxed{\phi(n) = (p-1)(q-1)}$.

⇒ Choose an integer 'e' (public exponent) such that $\boxed{1 < e < \phi(n)}$ and 'e' is co-prime with $\phi(n)$

⇒ Compute the private key 'd' such that $\boxed{d \times e \equiv 1 (\text{mod } \phi(n))}$.

(ii.) **Encryption:**

⇒ The message 'M' is converted to an integer 'm' such that $(m < n)$.

⇒ The encrypted message 'c' is calculated as $\boxed{c = m^e \text{ mod } n}$.

(iii.) **Decryption:**

⇒ The encrypted message 'c' is decrypted using the private key 'd' as $\boxed{m = c^d \text{ mod } n}$, which recovers the original message.

Example:

Let $p = 61$ & $q = 53$

$n = 61 \times 53$

$\boxed{n = 3233}$

$\phi(n) = 60 \times 52$

$\boxed{\phi(n) = 3120}$

$\boxed{e = 17}$

$d \times 17 \equiv 1 (\text{mod } 3120)$

$\boxed{d = 2753}$

$\boxed{\text{public key}(e, n) = (17, 3233)}$ $\boxed{\text{private key}(d, n) = (2753, 3233)}$

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int gcd(int a, int b){
    while(b!=0){
        int temp=b;
        b=a%b;
        a=temp;
    }
    return a;
}
long long mod_exp(long long base, long long exp, long long mod){
    long long result=1;
    base=base % mod;
    while(exp>0){
        if(exp%2==1) result=(result*base)% mod;
        exp=exp/2;
        base=(base *base)% mod;
    }
    return result;
}
int main(){
    int p=61, q=53;
    printf("Chosen primes p=%d and q=%d\n",p,q);
    int n=p*q;
    printf("Calculated n=p*q=%d\n",n);
    int phi=(p-1)*(q-1);
    printf("Calculated phi(n)=(p-1)*(q-1)=%d\n",phi);
    int e;
    for(e=2;e<phi;e++){
        if(gcd(e,phi)==1) break;
    }
    printf("Chosen encryption key e=%d\n",e);
```

```c
int d;
for(d=1; d< phi; d++){
    if((d*e) % phi == 1) break;
}
printf("Calculated decryption key d= %d \n", d);
printf("Public Key: (e= %d, n= %d)\n", e, n);
printf("Private Key: (d=%d, n= %d)\n", d, n);
int message;
printf(" Enter a message (as an integer) to encrypt: ");
scanf("%d", &message);
if(message <0 || message >=n){
    printf("Message must be in the range [0, %d).\n", n);
    return 1;
}
long long encrypted_message = mod_exp(message, e, n);
printf("Encryption message: %lld\n", encrypted_message);
long long decrypted_message = mod_exp(encrypted_message, d, n);
printf("Decrypted message: %lld\n", decrypted_message);
return 0;
}
```

10

26/12/25

Output: Chosen primes p=61 and q=53
Calculated n=p*q=3233
Calculated phi(n)=(p-1)*(q-1)= 3120
Chosen encryption key e=7
Calculated decryption key d= 1783
Public key: (e=7, n= 3233)
Private key: (d=1783, n= 3233)
Enter a message (as an integer) to encrypt: 45
Encrypted message: 1754
Decrypted message: 45

Conclusion:

In conclusion, RSA is a robust encryption method that uses public and private keys for secure communication. It's implementation highlights key generation, encryption, and decryption, emphasizing the role of cryptography in modern security.

Annexure No : 6

Aim : Implement message integrity using SHA-256 hashing function which creates chain of three blocks. Each block contains index, timestamp, data, previous hash value and current block of hash value. Test message integrity of program by modifying one of the hash value of block.

Description:

To implement message integrity using SHA-256, we create a chain of three blocks. Each block contains an index, timestamp, data, previous hash, and current hash. The blocks are linked by the previous hash, ensuring integrity. Modifying any block's hash or data will change the hash of the subsequent blocks, breaking the chain and indicating tampering. This demonstrates how SHA-256 ensures the integrity of the message.

Algorithm:

SHA-256 is a cryptographic hash function that generates a fixed 256-bit output from any input data.

(i) Padding: The message is padded to make it's length a multiple of 512-bit, ensuring the input data is properly aligned.

(ii) Message Parsing: The padded message is divided into 512-bit blocks.

(iii) Initialization: Eight 32-bit words (constants) are initialized as hash values.

(iv) Processing: Each 512-bit block is processed in 64 rounds using bitwise operations (AND, OR, XOR), shifting, and modular additions, modifying the hash values at each step.

(v) Finalization: After all blocks are processed, the resulting hash values are concatenated to produce a 256-bit output.

Code:

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <openssl/sha.h>
#define DATA_SIZE 256
typedef struct{
    int index;
    time_t timestamp;
    char data[DATA_SIZE];
    char prev_hash[65];
    char curr_hash[65];
}Block;
void to_hex(unsigned char *hash, char *output){
    for(int i=0; i<SHA256_DIGEST_LENGTH; i++){
        sprintf(output+(i*2), "%02x", hash[i]);
    }
    output[64]='\0';
}
void calc_hash(Block *block, char *output){
    char input[512];
    snprintf(input, sizeof(input), "%d%ld%s%s", block->index, block->timestamp, block->data,
                                                          block->prev_hash);
    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256_CTX sha256Ctx;
    SHA256_Init(&sha256Ctx);
    SHA256_Update(&sha256Ctx, input, strlen(input));
    SHA256_Final(hash, &sha256Ctx);
    to_hex(hash, output);
}
```

```
void init_block(Block *block, int index, const char *data, const char *prev_hash){
    block->index = index;
    block->timestamp = time (NULL);
    strncpy(block->data, data, DATA_SIZE - 1);
    block->data [DATA_SIZE - 1] = '\0';
    strncpy(block->prev_hash, prev_hash, 65);
    block->prev_hash [64] = '\0';
    calc_hash(block, block->curr_hash);
}
int verify_chain(Block *chain, int size){
    for(int i=1; i<size; i++){
        if(strcmp(chain[i].prev_hash, chain[i-1].curr_hash) != 0){
            return 0;
        }
    }
    return 1;
}
int main(){
    Block chain[3];
    const char *genesis_hash="0000000000000000000000000000000000000000000000000000000000000000";
    init_block(&chain[0], 0, "First block data", genesis_hash);
    init_block (&chain[1], 1, "Second block data", chain[0].curr_hash);
    init_block (& chain[2], 2, "Third block data", chain[1]. curr_hash);
    for (int i=0; i<3; i++){
        printf ("Block %d: Data= %s, prevHash = %s, currHash = %s\n",
            chain[i].index, chain[i].data, chain[i].prev_hash, chain[i].curr_hash);
    }
```

```
if(verify_chain(chain,3)){
    printf("Chain integrity verified\n");
}
else{
    printf("Chain integrity broken\n");
}
printf("\nTampering with block 1's Hash\n");
strcpy(chain[1].curr_hash,"1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef");
if(verify_chain(chain,3)){
    printf("chain integrity verified\n");
}
else{
    printf("Chain broken at block 2!\n");
}
return 0;
}
```

9

26/2/25

Output:

Block 0: Data = First block data, prevHash = 0000000000000000000000000000000000000000,
currHash = 7e8f4a2b....

Block 1: Data = Second block data, prevHash = 7e8f4a2b...., currHash = 4a1b2C3...

Block 2: Data = Third block data, prevHash = 4a1b2C3,...., currHash = 9e0f1a2b...

Chain integrity verified.

Tampering with block 1's Hash
Chain broken at block 2!

Conclusion:

This practical demonstrates a simple blockchain using SHA-256 hashing to ensure message integrity. The integrity of the chain is verified by checking if each block's previous hash matches the previous hash's current hash. Tampering with any block breaks the chain.

Annexure No : 7

Aim: Implement Elgamal Cryptosystem.

Description: This practical involves demonstrating Elgamal encryption and decryption. It covers key generation, modular exponentiation, and secure message transmission, helping students understand the core concepts of this public-key cryptosystem.

Algorithm:

(i.) Key Generation:

⇒ Choose a large prime number 'p' and a primitive root 'g' modulo 'p'.

⇒ Select a private key 'x', where $\boxed{1 < x < (p-1)}$.

⇒ Compute the public key $\boxed{y = g^x \bmod p}$.

The public key is (p, g, y) and the private key is 'x'.

(ii.) Encryption:

⇒ To encrypt a message 'm', select a random integer 'k' such that $\boxed{1 < k < (p-1)}$.

⇒ Compute:

$\boxed{C_1 = g^k \bmod p}$ $\boxed{C_2 = m \cdot y^k \bmod p}$

⇒ The ciphertext is (C_1, C_2).

(iii.) Decryption:

⇒ To decrypt the ciphertext (C_1, C_2), use the private key 'x' to compute:

$\boxed{s = C_1^x \bmod p}$ $\boxed{m = C_2 \cdot s^{-1} \bmod p}$, where '$s^{-1}$' is the modular inverse of 's'.

Example:

Let $p = 23$ & $g = 5$

$\boxed{x = 6}$ → Private key

public key $y = 5^6 \bmod 23$

$\boxed{y = 8}$ $\boxed{\text{public key } (p, g, y) = (23, 5, 8)}$

Let $m = 15$

Random integer $k = 10$

$C_1 = 5^{10} \bmod 23$

$\boxed{C_1 = 9}$

$C_2 = 15 \cdot 8^{10} \bmod 23$

$\boxed{C_2 = 17}$

CipherText $(C_1, C_2) = (9, 17)$

$s = 9^6 \bmod 23$

$\boxed{s = 4}$

Modular inverse of $s = 4^{-1} \bmod 23$

$= 6$

$m = 17 \cdot 6 \bmod 23 = 15$

$\boxed{m = 15}$ → Decrypted Text

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int mod_exp(int base, int exp, int mod){
    int result=1;
    while(exp>0){
        if(exp%2==1) result=(result*base)%mod;
        base=(base*base)%mod;
        exp/=2;
    }
    return result;
}
int main(){
    int p,g,x,y,k,m,c1,c2, decrypted, K, K_inv;
    printf("Enter a prime number (p): ");
    scanf("%d",&p);
    printf("Enter a primitive root (g): ");
    scanf("%d",&g);
    printf("Enter a private key(x): ");
    scanf("%d",&x);
    y=mod_exp(g,x,p);
    printf("Public key: (p=%d, g=%d, y=%d)\n", p,g,y);
    printf("Enter a message (m) to encrypt: ");
    scanf("%d",&m);
    printf("Enter random key: ");
    scanf("%d",&k);
    K=mod_exp(y,k,p);
    printf("Computed K value: %d\n", K);
    c1=mod_exp(g,k,p);
    c2=(m*K)%p;
```

```
printf("Ciphertext: (c1=%d, c2=%d)\n", c1, c2);
k=mod_exp(c1,x,p);
printf("Computed k value using decryption: %d\n", k);
k_inv=mod_exp(k, p-2, p);
printf("Computed k_inv value: %d\n", k_inv);
decrypted = (c2*k_inv) % p;
printf("Decrypted message: %d\n", decrypted);
return 0;
}
```

Output:

Enter a prime number(p): 17
Enter a primitive root (g): 3
Enter a private key (x): 15
Public Key: (p=17, g=3, y=06)
Enter message (m) to encrypt: 13
Enter random key (k): 10
Computed k value: 15
Ciphertext: (c_1 = 8, c_2 = 8)
Computed k value during decryption: 15
Computed k_inv value: 8
Decrypted message: 13

2P
26/2/25
10

Conclusion:

In conclusion, the Elgamal cryptosystem ensuring secure encryption and decryption using public and private keys based on modular arithmetic, providing confidentiality, and resistance to attacks.

Annexure No : 8

Aim: Implement Digital Signature.

Description:

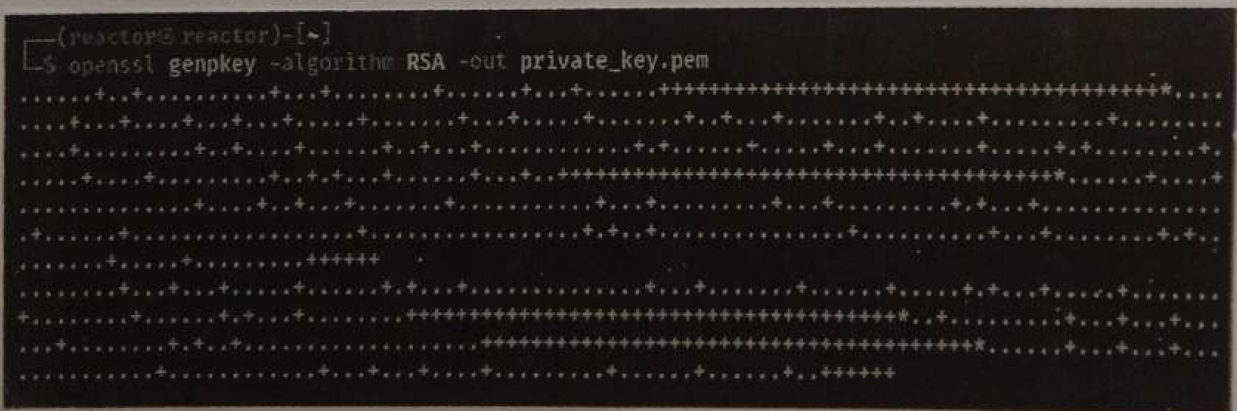
A digital signature is a cryptographic method used to verify the authencity and integrity of digital message or documents. It functions similarly to a handwritten signature but offers much stronger security. Digital signatures use a pair of keys - a private key, which is used to sign the document, and a public key, which others use to verify the signature. The process ensures that the document has not been altered after signing and confirms the identity of the signer. Digital signatures are widely used in secure communications, online transactions, and legal documents, ensuring both trust and security in the digital world.

Algorithms: RSA (Rivest-Shamir-Adleman); DSA (Digital Signature Algorithm); ECDSA (Elliptic Curve Digital Signature Algorithm)

Commands:

(A) `openssl genpkey -algorithm RSA -out private_key.pem`

This command generates an RSA private key and saves it in a file named "private_key.pem".



2. openssl rsa -pubout -in private_key.pem -out public_key.pem

This command takes an RSA private key from "private_key.pem" and extracts the corresponding public key, saving it to a file named "public_key.pem".

```
┌──(reactor㉿reactor)-[~]
└─$ openssl rsa -pubout -in private_key.pem -out public_key.pem
writing RSA key
```

3. openssl dgst -sha256 -sign "private_key.pem" -out signature.bin message.txt

This command generates a digital signature for the contents of "message.txt" using the SHA-256 hash of the message and signs it with the private key from "private_key.pem". The resulting signature is stored in "signature.bin".

```
┌──(reactor㉿reactor)-[~]
└─$ openssl dgst -sha256 -sign private_key.pem -out signature.bin message.txt
```

10
20/2/25

4. openssl dgst -sha256 -verify "public_key.pem" -signature signature.bin message.txt

This command verifies the digital signature (signature.bin) for the file "message.txt" using the public key stored in "public_key.pem". It checks that the signature matches the hash of the message and was signed with the corresponding private key.

```
┌──(reactor㉿reactor)-[~]
└─$ openssl dgst -sha256 -verify public_key.pem -signature signature.bin message.txt
Verified OK
```

Conclusion: In conclusion, the practical of digital signatures highlighted their crucial role in ensuring data integrity and authencity. By demonstrating cryptographic techniques, we showed how digital signatures verify identity and protect against tampering, emphasizing their importance in securing digital communications.

Aim: Study and configure SSH to authenticate machines, generate session key and transfer files using symmetric key and asymmetric key cryptography.

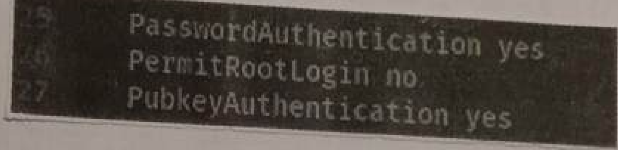
Description:

SSH (Secure Shell) is a protocol for securely authenticating and managing remote machines. It uses asymmetric key cryptography for authentication, where a public-private key pair verifies the client and server. After authentication, a symmetric session key is generated for secure data transfer. SSH enables secure file transfers using SCP or SFTP, with files encrypted via the session key, ensuring confidentiality and integrity during communication.

Procedure:

1. Edit the "/etc/ssh/ssh_config" file. check the following content;

PasswordAuthentication yes
PermitRootLogin no
PubkeyAuthentication yes



2. sudo systemctl enable ssh

This command is used to configure the SSH service to start automatically when the system boots up.

3. sudo systemctl start ssh

The command is used to start the SSH service immediately.

④ $\boxed{\text{sudo systemctl status ssh}}$

This command is used to check the current status of the SSH service. It checks whether the service is active (running) or inactive (stopped).

```
┌──(reactor㉿reactor)-[~/.ssh]
└─$ sudo systemctl status ssh
● ssh.service - OpenBSD Secure Shell server
     Loaded: loaded (/usr/lib/systemd/system/ssh.service; enabled; preset: disabled)
     Active: active (running) since Sun 2025-02-23 20:19:37 IST; 30min ago
 Invocation: 43ef1e4dd9a84bafaa38fd35301f7a85
       Docs: man:sshd(8)
             man:sshd_config(5)
    Process: 3325 ExecStartPre=/usr/sbin/sshd -t (code=exited, status=0/SUCCESS)
   Main PID: 3327 (sshd)
      Tasks: 1 (limit: 7519)
     Memory: 2.9M (peak: 6.9M)
        CPU: 291ms
     CGroup: /system.slice/ssh.service
             └─3327 "sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups"

Feb 23 20:29:06 reactor sshd-session[3631]: pam_unix(sshd:session): session closed for user re>
Feb 23 20:32:38 reactor sshd-session[3786]: Accepted publickey for reactor from 192.168.182.1 >
Feb 23 20:32:38 reactor sshd-session[3786]: pam_unix(sshd:session): session opened for user re>
Feb 23 20:32:38 reactor sshd-session[3786]: pam_unix(sshd:session): session closed for user re>
Feb 23 20:34:04 reactor sshd-session[3798]: Accepted publickey for reactor from 192.168.182.1 >
Feb 23 20:34:04 reactor sshd-session[3798]: pam_unix(sshd:session): session opened for user re>
Feb 23 20:34:04 reactor sshd-session[3798]: pam_unix(sshd:session): session closed for user re>
Feb 23 20:36:43 reactor sshd-session[3832]: Accepted publickey for reactor from 192.168.182.1 >
Feb 23 20:36:43 reactor sshd-session[3832]: pam_unix(sshd:session): session opened for user re>
Feb 23 20:36:43 reactor sshd-session[3832]: pam_unix(sshd:session): session closed for user re>
```

⑤ $\boxed{\text{ssh-keygen -t rsa -b 4096}}$ (Run this command in your local windows powershell)

This command will create a new RSA key pair with a key size of 4096 bits. ~~This~~ The private key is stored securely on your machine, and the public key can be placed on remote servers for authentication purposes, ensuring secure, passwordless SSH logins.

```
PS C:\Users\Dell\.ssh> ssh-keygen -t rsa -b 4096
Generating public/private rsa key pair.
Enter file in which to save the key (C:\Users\Dell/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in C:\Users\Dell/.ssh/id_rsa
Your public key has been saved in C:\Users\Dell/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:ubZzg+b3jqaMhvrQ54kPhzGcy3OIvpuG9+/CAk7Cw7M anand soni@ANAND-SONI
The key's randomart image is:
+---[RSA 4096]----+
|                 |
|                 |
|                 |
|                 |
|      . .        |
|o    =   S       |
|o* + *     .     |
|+o*.O.+ o.       |
|.E+o+O.=+.=.     |
| o*B+*Bo=*.+o    |
+----[SHA256]-----+
```

```
PS C:\Users\Dell\.ssh> type $env:USERPROFILE\.ssh\id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQDeB263VAv5DkliARQd7JVvT2amYVPbCuXmIOKsUaGvB6Yxn9AuujbagvgH3GWtePT/AN8FEJ5LeHzvSDjA
xi97WtDp+x57fC75Qu2qd7FzboVYU6F/4d6W7XM4UQPEz9rWH1vLRQrK4aVobeQcMjclcZVK6WAd98cwoXGRgLbpePkfTmcjZTd5jP8G17995USbJkVipkvG
mUzYLANfqYYRd/x5+F9k1mI1dyex+1v6Y45Qrxlrh/tXb3bVrgaH6FgxM/zn5siDLBQq/0V8h6cse4yw7p34NSP+uB65hqkJmCrpht3gTRQJiG1NiMsLVMfU
v7bWsH+6GX4oJTSAAfC+gWUzrg/AIivyTR39HZsk/hWUmaJHQThQ2gjOfQFBC0piVRkVSArKdLcPz7kESYFH+wwR5+vAWvhM+gDjeDvYik00t6ALOb7U0rJZ
PSHP2vrCUpJFhqZfqDGqIdFPMvr8DgWEY7+Mu4MwHBau3Wcf63vhbRbYIZeviX6nl9va09umACbq2XyFUw4LIvCDJMEuepReoQlcxxxlp0TujFxxsVG3by++
LG2/49/88J7UiSsZ596HBrmdsx8bqUmu+N7PudLEsouyJ3QGi6m5eAGM+t8Vl2T97iifZVAU9zlmFOmJj3Qa/LSrELsg4GRui8N8yXql3pMGM2+CsSzWNI5D
4Q== anand soni@ANAND-SONI
PS C:\Users\Dell\.ssh> |
```


(6) type $env: USERPROFILE\.ssh\id_rsa.pub

This command in powershell is used to display the contents of the SSH public key file "id_rsa.pub".

Copy the public key in your kali Linux, in the "authorized_keys" files of ".ssh" directory.

(7) openssl rand -base64 32 > my_symmetric_key.key

This command creates a random symmetric key, which is often used for encryption/decryption in symmetric-key cryptography, and saves it securely in the file "my_symmetric key.key"

(8) openssl enc -aes-256-cbc -salt -pbkdf2 -in secret.txt -out secret.enc -pass file:./my_symmetric_key.key

This command reads the content of "secret.txt", encrypts it using "AES-256-CBC" with the key from "my_symmetric_key.key", and stores the resulting encrypted data in "secret.enc". The "-salt" and "-pbkdf2" options add extra layers of security to the process.

(9) ssh reactor@192.168.182.130 "cat /home/reactor/.ssh/my_symmetric_key.key" > C:\Users\Dell\.ssh\my_symmetric_key.key

This command securely retrieves the symmetric key from the remote machine "192.168.182.130" and saves it to a file "my_symmetric_key.key" on your local machine "C:\Users\Dell\.ssh\".

(10) scp reactor@192.168.182.130:/home/reactor/.ssh/secret.enc C:\Users\Dell\.ssh

This command copies the "secret.enc" file from the remote server to the local machine.

```
PS C:\Users\Dell\.ssh> scp reactor@192.168.182.130:/home/reactor/.ssh/secret.enc C:\Users\Dell\
reactor@192.168.182.130's password:
secret.enc                                                   100%   48    9.4KB/s   00:00
PS C:\Users\Dell\.ssh> |
```

(11.)


```
openssl enc -d -aes-256-cbc -pbkdf2 -in "C:\Users\Dell\.ssh\secret.enc" -out
"C:\Users\Dell\.ssh\secret.txt"
```

This command decrypts the encrypted file "secret.enc" located in "C:\Users\Dell\.ssh" using the AES-256-CBC algorithm and saves the decrypted content into the file "secret.txt" in the same directory.

```
PS C:\Users\Dell\.ssh> openssl enc -d -aes-256-cbc -pbkdf2 -in "C:\Users\Dell\.ssh\secret.enc" -out "C:\Users\Dell\.ssh\
secret.txt" -pass file:"C:\Users\Dell\.ssh\my_symmetric_key.key"
PS C:\Users\Dell\.ssh> cat secret.txt
This is a Cryptographic Secret
PS C:\Users\Dell\.ssh> |
```

Conclusion:

In this practical, we used SSH and OpenSSL to secure communication and manage files. We generated SSH keys, encrypted and decrypted files with AES-256-CBC, and securely transfered keys between remote and local systems. This demonstrated how cryptographic tools ensure data confidentiality and integrity in real-world scenarios.

26/2/25

Aim: Demonstrate the use of different Testing Prepare report on any advanced cryptographic system, with comparision.

Post-Quantum Cryptography (PQC):

Post-Quantum Cryptography (PQC) refers to cryptographic algorithms designed to be secure against quantum computing attacks. As quantum computers become more powerful, traditional encryption methods like RSA and ECC may become obsolete. PQC algorithms, such as those proposed in the NIST PQC standardization Process, aim to provide long-term security.

Testing Methodologies for Cryptographic Systems:

1. Mathematical Analysis: Formal proofs and complexity analysis ensure the theoritical security of cryptographic algorithms.
2. Implementation Testing: Includes unit testing, integration testing, and functional testing of cryptographic modules.
3. Performance Testing: Measures encryption/decryption speed, memory usage, and computational overhead.
4. Side-Channel Analysis: Evaluates resistance against timing attacks, power analysis, and electromagnetic attacks.
5. Cryptanalysis: Tests the algorithm against known attack techniques, such as brute-force, differential cryptanalysis, and quantum attack.
6. Interoperability Testing: Ensures compatability between different cryptographic libraries and systems.

Comparision of Cryptographic Systems:

Feature	Post-Quantum Cryptography (PQC)	RSA (Rivest-Shamir-Adleman)	ECC (Elliptic Curve Cryptography)
Security Against Quantum Attacks	Resistant	Vulnerable	Vulnerable
Key Size	Larger (~1KB-10KB)	Large (~2048-4096 bits)	Small (~256-521 bits)
Speed	Varies by algorithm	Moderate	Fast
Implementation Complexity	High	Moderate	High
Adoption	Emerging	Widely Used	Widely Used

20/2/25 9

Conclusion:

Post-Quantum Cryptography represents the future of cryptographic security in the face of quantum computing advancements. While traditional methods like RSA and ECC remain widely used, they are vulnerable to quantum attacks, necessitating the development and adoption of PQC algorithms. Ongoing research, rigorous testing, and standardization efforts will be crucial for ensuring a secure digital landscape in the post-quantum era.