Lab Manual: Cryptographic Algorithms and Security Practices

Experiment 1: Implementing the Extended Euclidean Algorithm

Title:

Implementation of the Extended Euclidean Algorithm in C++

Objectives:

1. To understand the principles behind the Extended Euclidean Algorithm.

2. To implement the algorithm in C++ for computing the greatest common divisor (GCD) and modular inverses.

3. To explore its applications in cryptography.

Theory:

The **Extended Euclidean Algorithm (EEA)** is an enhancement of the Euclidean algorithm for finding the GCD of two integers, aaa and bbb. It not only computes the GCD but also finds coefficients xxx and yyy such that:

ax+by=gcd(a,b)ax + by = \text{gcd}(a, b)ax+by=gcd(a,b)

This representation is crucial in number theory, particularly in applications involving modular arithmetic. The EEA is fundamental for computing the **modular inverse** of an integer aaa modulo mmm, which exists only if aaa and mmm are coprime (i.e., gcd(a,m)=1\text{gcd}(a, m) = 1gcd(a,m)=1).

The modular inverse a−1a^{-1}a−1 is defined as:

aa−1≡1 (mod m)aa^{-1} \equiv 1 \ (\text{mod} \ m)aa−1≡1 (mod m)

The Extended Euclidean Algorithm works recursively and can be expressed as follows:

1. If b=0b = 0b=0, then gcd(a,b)=a\text{gcd}(a, b) = agcd(a,b)=a.
2. Otherwise, recursively call the function with parameters (b,amod b)(b, a \mod b)(b,amodb).

In cryptographic applications, the modular inverse is used for tasks such as decryption in RSA and digital signature

The **Extended Euclidean Algorithm (EEA)** is an enhancement of the Euclidean algorithm for finding the GCD of two integers, \(a \) and \(b \). It not only computes the GCD but also finds coefficients \(x \) and \(y \) such that:

\[ax + by = \text{gcd}(a, b)\]

This representation is crucial in number theory, particularly in applications involving modular arithmetic. The EEA is fundamental for computing the **modular inverse** of an integer \(a \) modulo \(m \), which exists only if \(a \) and \(m \) are coprime (i.e., \(\text{gcd}(a, m) = 1 \)).

The modular inverse \(a^{-1} \) is defined as:

\[aa^{-1} \equiv 1 \ (\text{mod} \ m)

\]

The Extended Euclidean Algorithm works recursively and can be expressed as follows:

1. If \(b = 0 \), then \(\text{gcd}(a, b) = a \).

2. Otherwise, recursively call the function with parameters \((b, a \mod b) \).

In cryptographic applications, the modular inverse is used for tasks such as decryption in RSA and digital signatures.

Time Complexity: O(log N)
Auxiliary Space: O(log N)

Procedure:

1. Open your C++ IDE and create a new file named `ExtendedEuclidean.cpp`.

2. Implement the algorithm as shown in the code section below.

Code:

```cpp
#include <iostream>


void extendedGCD(int a, int b, int &x, int &y) {
    if (a == 0) {
        x = 0;
        y = 1;
        return;
    }
    int x1, y1;
    extendedGCD(b % a, a, x1, y1);

    x = y1 - (b / a) * x1;
    y = x1;
}

int modInverse(int a, int m) {
    int x, y;
    extendedGCD(a, m, x, y);
    return (x % m + m) % m; // Ensure positive result
}

int main() {
    int a, m;
    std::cout << "Enter two integers (a and m): ";
    std::cin >> a >> m;

    int inverse = modInverse(a, m);
    if (inverse == 0) {
        std::cout << "Inverse doesn't exist." << std::endl;
    } else {
        std::cout << "Modular Inverse of " << a << " mod " << m << " is: " << inverse << std::endl;
    }

    return 0;
}
```

Conclusion:

The Extended Euclidean Algorithm has been successfully implemented to find modular inverses. This forms a foundational understanding for more complex cryptographic algorithms.

Experiment 2: Implementing the Diffie-Hellman Key Exchange Algorithm

Title:

Implementation of the Diffie-Hellman Key Exchange Algorithm in C++

Objectives:

1. To understand the principles of the Diffie-Hellman key exchange method.

2. To implement the algorithm in C++.

3. To illustrate how secure keys can be exchanged over an insecure channel.

Theory:

The **Diffie-Hellman key exchange** algorithm allows two parties to securely share a secret key over a public channel. The key exchange relies on modular arithmetic and the difficulty of the discrete logarithm problem.

Basic Steps:

1. **Public Parameters**: Two parties agree on a large prime number \(p \) and a generator \(g \).

2. **Private Keys**: Each party selects a private key \(a \) (for Alice) and \(b \) (for Bob).

3. **Public Keys**: Each party computes their public key:

   - Alice computes \(A = g^a \mod p \)

   - Bob computes \(B = g^b \mod p \)

4. **Shared Secret**: They exchange their public keys and compute the shared secret:

   - Alice computes \(s = B^a \mod p \)

   - Bob computes \(s = A^b \mod p \)

Example:
Step 1: Alice and Bob get public numbers P = 23, G = 9
Step 2: Alice selected a private key a = 4 and
Bob selected a private key b = 3
Step 3: Alice and Bob compute public values
Alice: x =(9^4 mod 23) = (6561 mod 23) = 6
Bob: y = (9^3 mod 23) = (729 mod 23) = 16
Step 4: Alice and Bob exchange public numbers
Step 5: Alice receives public key y =16 and
Bob receives public key x = 6
Step 6: Alice and Bob compute symmetric keys
Alice: ka = y^a mod p = 65536 mod 23 = 9
Bob: kb = x^b mod p = 216 mod 23 = 9
Step 7: 9 is the shared secret.

The security of this method is based on the computational difficulty of the discrete logarithm problem, where it is easy to compute the public keys but hard to derive the private keys.

Procedure:

1. Create a new C++ file named `DiffieHellman.cpp`.

2. Implement the algorithm as shown in the code section below.

Code:

```cpp
#include <iostream>
#include <cmath>

long long power(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

int main() {
    long long p, g, a, b;
    std::cout << "Enter a prime number (p) and a generator (g): ";
    std::cin >> p >> g;

    std::cout << "Enter private key for User A: ";
    std::cin >> a;
    std::cout << "Enter private key for User B: ";
    std::cin >> b;

    long long A = power(g, a, p); // Public key of User A
    long long B = power(g, b, p); // Public key of User B

    long long secretA = power(B, a, p); // Shared secret for User A
    long long secretB = power(A, b, p); // Shared secret for User B

    std::cout << "User A's public key: " << A << std::endl;
    std::cout << "User B's public key: " << B << std::endl;
    std::cout << "Shared secret key (User A): " << secretA << std::endl;
    std::cout << "Shared secret key (User B): " << secretB << std::endl;

    return 0;
}
```

Conclusion:

The Diffie-Hellman key exchange algorithm successfully computes shared keys, demonstrating secure key exchange over an insecure channel.

Experiment 3: Implementation of Simplified DES (S-DES)

Title:

Implementation of Simplified DES (S-DES) in C++

Objectives:

1. To understand the structure and operation of S-DES.

2. To implement the encryption and decryption processes in C++.

Theory:

Simplified DES (S-DES) is an educational version of the Data Encryption Standard (DES). S-DES provides a simplified approach to learning the principles of symmetric key cryptography.

Components of S-DES:

1. **Key Generation**: The key used in S-DES is 10 bits long. It is permuted and split into two subkeys for use in the encryption process.

2. **Rounds of Encryption**: S-DES performs two rounds of processing on the plaintext:

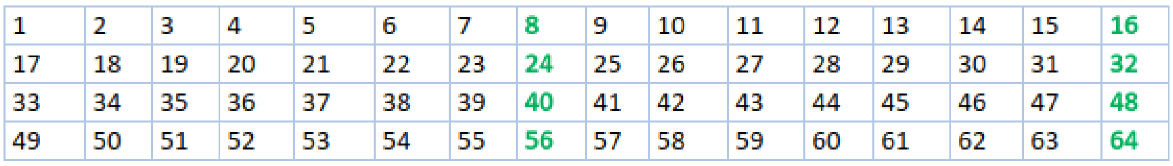
- Each round involves the use of a function that includes permutation, substitution using S-boxes, and XOR operations.

3. **Final Permutation**: After the rounds, a final permutation is applied to produce the ciphertext.

The encryption and decryption processes are symmetrical; however, the order of the keys used is reversed in decryption.

The basic idea is shown below:

We have mentioned that DES uses a 56-bit key. Actually, The initial key consists of 64 bits. However, before the DES process even starts, every 8th bit of the key is discarded to produce a 56-bit key. That is bit positions 8, 16, 24, 32, 40, 48, 56, and 64 are discarded.

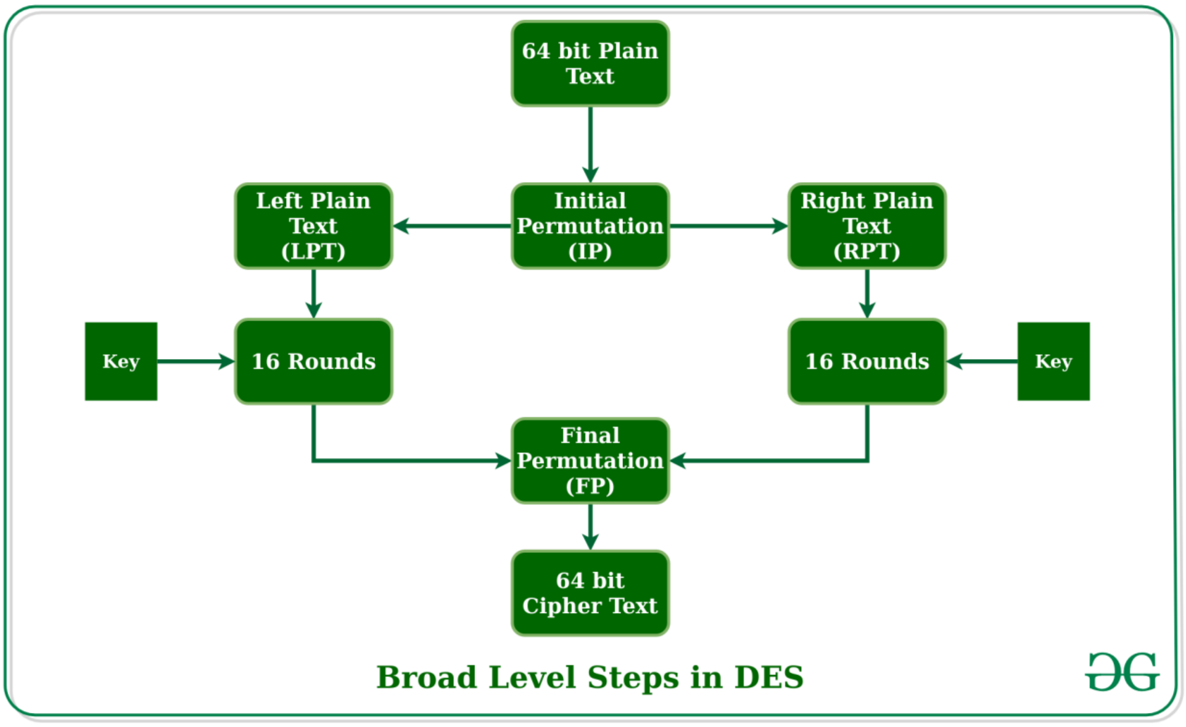


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64

Figure - discording of every 8th bit of original key

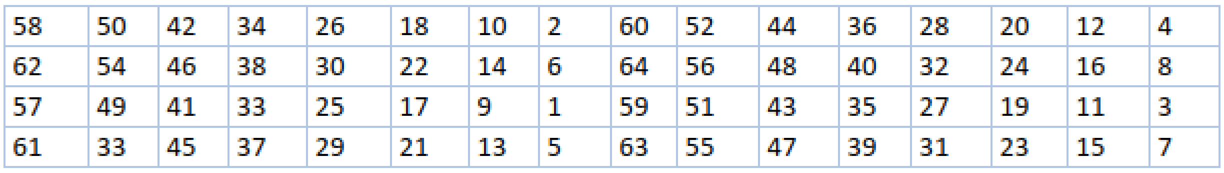
Thus, the discarding of every 8th bit of the key produces a **56-bit key** from the original **64-bit key**.
DES is based on the two fundamental attributes of cryptography: substitution (also called confusion) and transposition (also called diffusion). DES consists of 16 steps, each of which is called a round. Each round performs the steps of substitution and transposition. Let us now discuss the broad-level steps in DES.

- In the first step, the 64-bit plain text block is handed over to an initial Permutation (IP) function.
- The initial permutation is performed on plain text.
- Next, the initial permutation (IP) produces two halves of the permuted block; saying Left Plain Text (LPT) and Right Plain Text (RPT).
- Now each LPT and RPT go through 16 rounds of the encryption process.
- In the end, LPT and RPT are rejoined and a Final Permutation (FP) is performed on the combined block
- The result of this process produces 64-bit ciphertext.



nitial Permutation (IP)

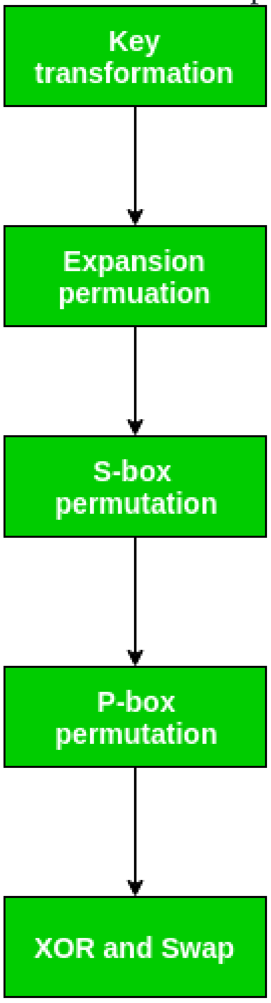
As we have noted, the initial permutation (IP) happens only once and it happens before the first round. It suggests how the transposition in IP should proceed, as shown in the figure. For example, it says that the IP replaces the first bit of the original plain text block with the 58th bit of the original plain text, the second bit with the 50th bit of the original plain text block, and so on.
This is nothing but jugglery of bit positions of the original plain text block. the same rule applies to all the other bit positions shown in the figure.



58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	33	45	37	29	21	13	5	63	55	47	39	31	23	15	7

Figure - Initial permutation table

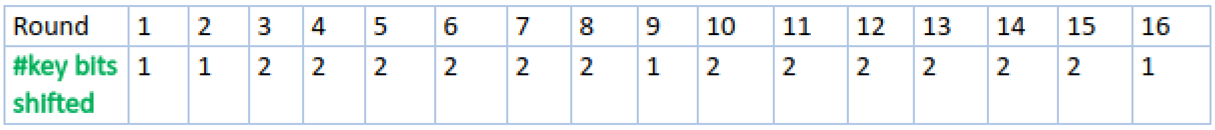
As we have noted after IP is done, the resulting 64-bit permuted text block is divided into two half blocks. Each half-block consists of 32 bits, and each of the 16 rounds, in turn, consists of the broad-level steps outlined in the figure.



Step 1: Key transformation

We have noted initial 64-bit key is transformed into a 56-bit key by discarding every 8th bit of the initial key. Thus, for each a 56-bit key is available. From this 56-bit key, a different 48-bit Sub Key is generated during each round using a process called key transformation. For this, the 56-bit key is divided into two halves, each of 28 bits. These halves are circularly shifted left by one or two positions, depending on the round.

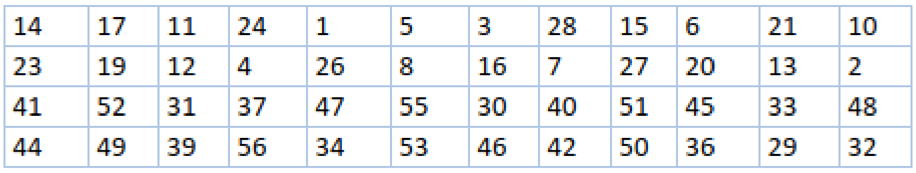
For example: if the round numbers 1, 2, 9, or 16 the shift is done by only one position for other rounds, the circular shift is done by two positions. The number of key bits shifted per round is shown in the figure.



Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
#key bits shifted	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Figure - number of key bits shifted per round

After an appropriate shift, 48 of the 56 bits are selected. From the 48 we might obtain 64 or 56 bits based on requirement which helps us to recognize that this model is very versatile and can handle any range of requirements needed or provided. for selecting 48 of the 56 bits the table is shown in the figure given below. For instance, after the shift, bit number 14 moves to the first position, bit number 17 moves to the second position, and so on. If we observe the table , we will realize that it contains only 48-bit positions. Bit number 18 is discarded (we will not find it in the table), like 7 others, to reduce a 56-bit key to a 48-bit key. Since the key transformation process involves permutation as well as a selection of a 48-bit subset of the original 56-bit key it is called Compression Permutation.



14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

Figure - compression permutation

Because of this compression permutation technique, a different subset of key bits is used in each round. That makes DES not easy to crack.

Step 2: Expansion Permutation

Recall that after the initial permutation, we had two 32-bit plain text areas called Left Plain Text(LPT) and Right Plain Text(RPT). During the expansion permutation, the RPT is expanded from 32 bits to 48 bits. Bits are permuted as well hence called expansion permutation. This happens as the 32-bit RPT is divided into 8 blocks, with each block consisting of 4 bits. Then, each 4-bit block of the previous step is then expanded to a corresponding 6-bit block, i.e., per 4-bit block, 2 more bits are added.

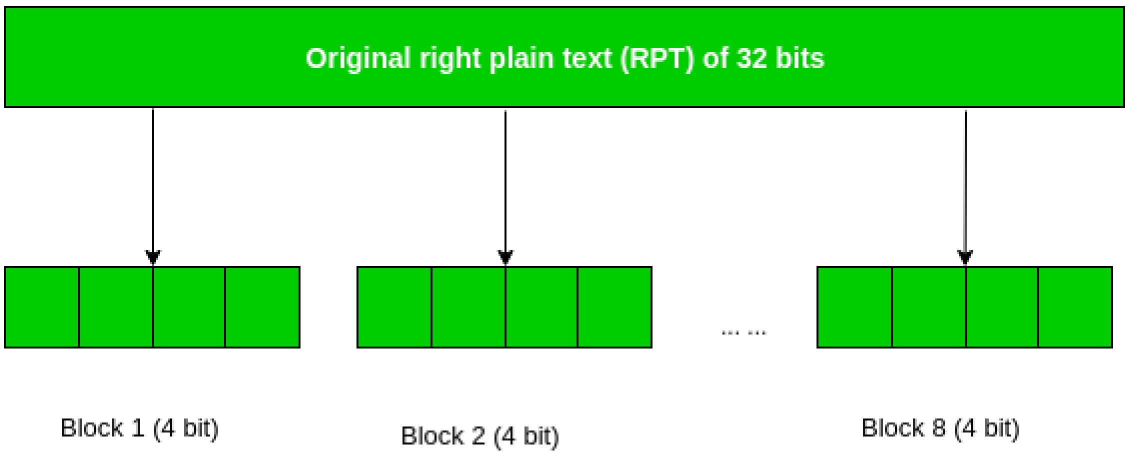


Figure - division of 32 bit RPT into 8 bit blocks

This process results in expansion as well as a permutation of the input bit while creating output. The key transformation process compresses the 56-bit key to 48 bits. Then the expansion permutation process expands the **32-bit RPT** to **48-bits**. Now the 48-bit key is XOR with 48-bit RPT and the resulting output is given to the next step, which is the **S-Box substitution**.

Procedure:

1. Create a new C++ file named `SDES.cpp`.

2. Implement the S-DES algorithm as shown in the code section below.

Code:

```cpp
#include <iostream>

#include <bitset>


using namespace std;


// Functions for key generation, encryption, and decryption go here...


int main() {

    int key, plaintext;
    cout << "Enter key (10 bits): ";
    cin >> key;
    cout << "Enter plaintext (8 bits): ";
    cin >> plaintext;

    // Call encryption and decryption functions
    // Display results

    return 0;
}
```

Conclusion:

The implementation of S-DES illustrates the principles of symmetric key encryption, including key generation, encryption, and decryption.

Experiment 4: Implementation of S-AES

Title:

Implementation of Simplified AES (S-AES) in C++

Objectives:

1. To understand the structure and operation of S-AES.

2. To implement the encryption and decryption processes in C++.

Theory:

Simplified AES (S-AES) serves as a simplified model of the Advanced Encryption Standard (AES), which is widely used in secure data transmission.

What is Advanced Encryption Standard (AES)?

Advanced Encryption Standard (AES) is a highly trusted **encryption algorithm** used to secure data by converting it into an unreadable format without the proper key. Developed by the National Institute of Standards and Technology (NIST), **AES encryption** uses various **key lengths** (128, 192, or 256 bits) to provide strong protection against unauthorized access. This **data security** measure is efficient and widely implemented in securing **internet communication**, protecting **sensitive data**, and encrypting files. AES, a cornerstone of modern cryptography, is recognized globally for its ability to keep information safe from cyber threats.

Points to Remember

- AES is a Block Cipher.
- The key size can be 128/192/256 bits.
- Encrypts data in blocks of 128 bits each.

That means it takes 128 bits as input and outputs 128 bits of encrypted cipher text. AES relies on the substitution-permutation network principle, which is performed using a series of linked operations that involve replacing and shuffling the input data.

Working of The Cipher

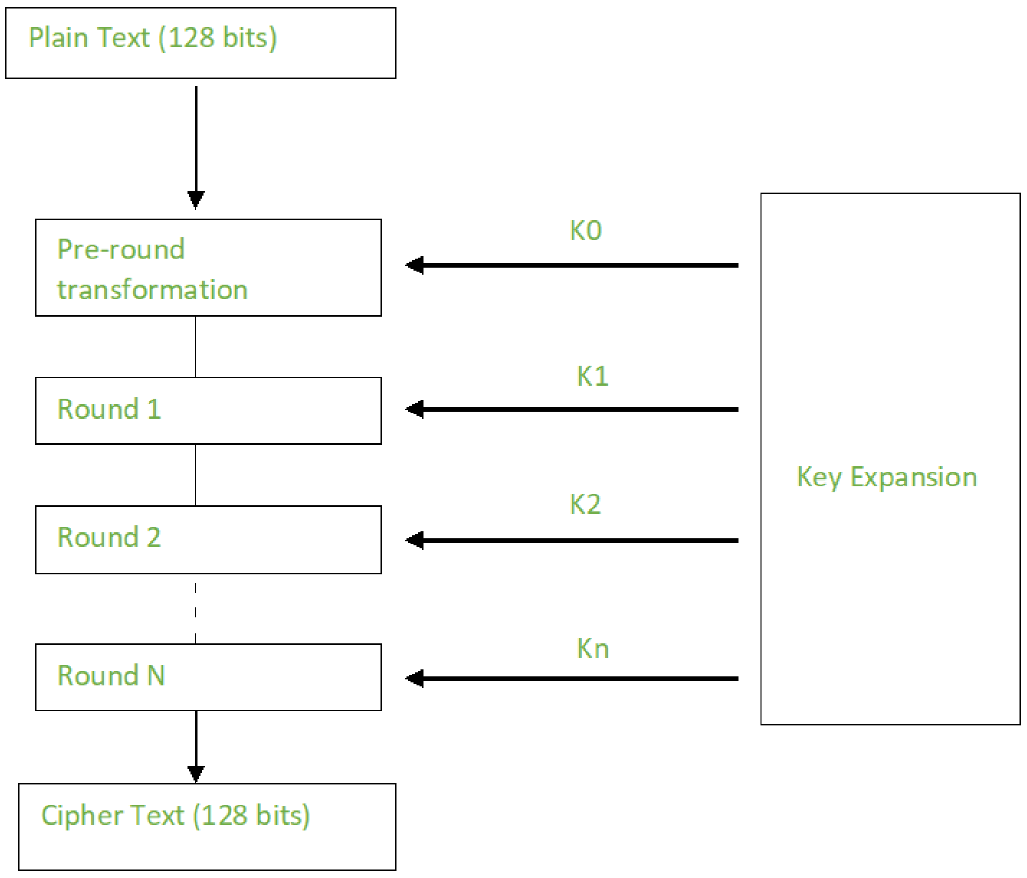
AES performs operations on bytes of data rather than in bits. Since the block size is 128 bits, the cipher processes 128 bits (or 16 bytes) of the input data at a time.

The number of rounds depends on the key length as follows :

- 128-bit key – 10 rounds
- 192-bit key – 12 rounds
- 256-bit key – 14 rounds

Creation of Round Keys

A Key Schedule algorithm calculates all the round keys from the key. So the initial key is used to create many different round keys which will be used in the corresponding round of the encryption.



Creation of Round Keys (AES)

Encryption

AES considers each block as a 16-byte (4 byte x 4 byte = 128) grid in a column-major arrangement.

[b0 | b4 | b8 | b12 |
| b1 | b5 | b9 | b13 |
| b2 | b6 | b10| b14 |
| b3 | b7 | b11| b15]

Each round comprises of 4 steps :

- SubBytes
- ShiftRows
- MixColumns
- Add Round Key

The last round doesn’t have the MixColumns round.

The SubBytes does the substitution and ShiftRows and MixColumns perform the permutation in the algorithm.

Sub Bytes

This step implements the substitution.

In this step, each byte is substituted by another byte. It is performed using a lookup table also called the S-box. This substitution is done in a way that a byte is never substituted by itself and also not substituted by another byte which is a compliment of the current byte. The result of this step is a 16-byte (4 x 4) matrix like before.

The next two steps implement the permutation.

Shift Rows

This step is just as it sounds. Each row is shifted a particular number of times.

- The first row is not shifted

- The second row is shifted once to the left.
- The third row is shifted twice to the left.
- The fourth row is shifted thrice to the left.

(A left circular shift is performed.)

```
[ b0  | b1  | b2  | b3  ]         [ b0  | b1  | b2  | b3  ]
| b4  | b5  | b6  | b7  |   ->  | b5  | b6  | b7  | b4  |
| b8  | b9  | b10 | b11 |         | b10 | b11 | b8  | b9  |
[ b12 | b13 | b14 | b15 ]        [ b15 | b12 | b13 | b14 ]
```

Mix Columns

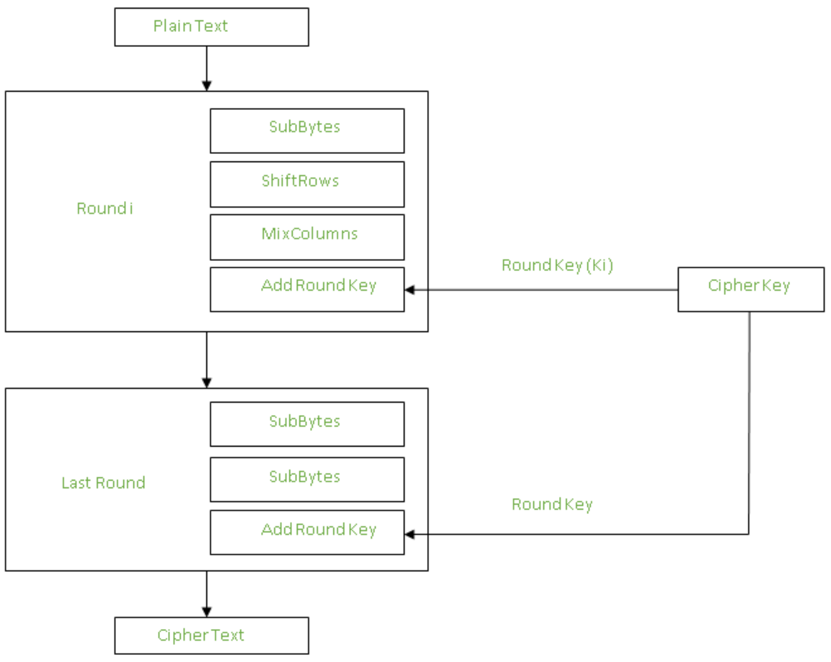
This step is a matrix multiplication. Each column is multiplied with a specific matrix and thus the position of each byte in the column is changed as a result.

This step is skipped in the last round.

```
[ c0 ]       [ 2  3  1  1 ]  [ b0 ]
| c1 |  =    | 1  2  3  1 |   | b1 |
| c2 |     | 1  1  2  3 |    | b2 |
[ c3 ]     [ 3  1  1  2 ]    [ b3 ]
```

Add Round Keys

Now the resultant output of the previous stage is XOR-ed with the corresponding round key. Here, the 16 bytes are not considered as a grid but just as 128 bits of data.



Added Round Keys (AES)

After all these rounds 128 bits of encrypted data are given back as output. This process is repeated until all the data to be encrypted undergoes this process.

Decryption

The stages in the rounds can be easily undone as these stages have an opposite to it which when performed reverts the changes. Each 128 blocks goes through the 10,12 or 14 rounds depending on the key size.

The stages of each round of decryption are as follows :

- Add round key
- Inverse MixColumns
- ShiftRows

- Inverse SubByte

The decryption process is the encryption process done in reverse so I will explain the steps with notable differences.

Inverse MixColumns

This step is similar to the Mix Columns step in encryption but differs in the matrix used to carry out the operation.

Mix Columns Operation each column is mixed independent of the other.

Matrix multiplication is used. The output of this step is the matrix multiplication of the old values and a constant matrix

[b0] = [14 11 13 9] [c0]
[b1]=[9 14 11 13] [c1]
[b2] =[13 9 14 11] [c2]
[b3]=[11 13 9 14] [c3]

Inverse SubBytes

Inverse S-box is used as a lookup table and using which the bytes are substituted during decryption. Function Substitute performs a byte substitution on each byte of the input word. For this purpose, it uses an S-box.

Applications

AES is widely used in many applications which require secure data storage and transmission. Some common use cases include:

- **Wireless security:** AES is used in securing wireless networks, such as Wi-Fi networks, to ensure data confidentiality and prevent unauthorized access.
- **Database Encryption:** AES can be applied to encrypt sensitive data stored in databases. This helps protect personal information, financial records, and other confidential data from unauthorized access in case of a data breach.
- **Secure communications:** AES is widely used in protocols such as internet communications, email, instant messaging, and voice/video calls. It ensures that the data remains confidential.
- **Data storage:** AES is used to encrypt sensitive data stored on hard drives, USB drives, and other storage media, protecting it from unauthorized access in case of loss or theft.
- **Virtual Private Networks (VPNs):** AES is commonly used in VPN protocols to secure the communication between a user’s device and a remote server. It ensures that data sent and received through the VPN remains private and cannot be deciphered by eavesdroppers.
- **Secure Storage of Passwords:** AES encryption is commonly employed to store passwords securely. Instead of storing plaintext passwords, the encrypted version is stored. This adds an extra layer of security and protects user credentials in case of unauthorized access to the storage.
- **File and Disk Encryption:** AES is used to encrypt files and folders on computers, external storage devices, and cloud storage. It protects sensitive data stored on devices or during data transfer to prevent unauthorized access.

.

S-AES serves as an educational tool to introduce the core concepts of AES without the complexity of full AES.

Procedure:

1. Create a new C++ file named `SAES.cpp`.

2. Implement the S-AES algorithm as shown in the code section below.

Code:

```cpp
#include <iostream>
#include <vector>

// Functions for key expansion, encryption, and decryption go here...

int main() {
    std::vector<uint8_t> plaintext = { /* initialize with 16 bytes */ };
    std::vector<uint8_t> key = { /* initialize with 16 bytes */ };

    // Call encryption and decryption functions
    // Display results

    return 0;
}
```

Conclusion:

The S-AES implementation provides insights into the operations of block ciphers and the importance of secure encryption methods.

Experiment 5: RSA Algorithm for Key Generation and Cipher Verification

Title:

Implementation of RSA Algorithm in C++

Objectives:

1. To understand the RSA algorithm and its components.

2. To implement RSA for key generation, encryption, and decryption in C++.

Theory:

The **RSA algorithm** is a widely used public-key cryptosystem that relies on the mathematical properties of large prime numbers.

Key Generation:

1. Choose two distinct large prime numbers \(p \) and \(q \).

2. Compute \(n = p \times q \) (modulus for both public and private keys).

3. Calculate Euler's totient function \(\phi(n) = (p-1)(q-1) \).

4. Choose an integer \(e \) such that \(1 < e < \phi(n) \) and \(\text{gcd}(e, \phi(n)) = 1 \) (public exponent).

5. Compute \(d \) as the modular inverse of \(e \) modulo \(\phi(n) \) (private exponent).

Encryption and Decryption:

- To encrypt a message \(m \):

  c = m^e \mod n

- To decrypt the ciphertext \(c \):

  m = c^d \mod n

The mechanism behind the RSA algorithm : >> Generating Public Key:

Select two prime no's. Suppose **P = 53 and Q = 59.**

Now First part of the Public key : n = P*Q = 3127.

We also need a small exponent say **e :**

But e Must be An integer.

Not be a factor of Φ(n).
1 < e < Φ(n) [Φ(n) is discussed below],
Let us now consider it to be equal to 3.
Our Public Key is made of n and e

>> Generating Private Key:
We need to calculate Φ(n) :
Such that **Φ(n) = (P-1)(Q-1)**
so, Φ(n) = 3016
Now calculate Private Key, **d :**
d = (k*Φ(n) + 1) / e for some integer k
For k = 2, value of d is 2011.
Now we are ready with our – Public Key (n = 3127 and e = 3) and Private Key(d = 2011) Now we will encrypt **"HI"**:
Convert letters to numbers : H = 8 and I = 9
Thus **Encrypted Data c = (89e)mod n**
Thus our Encrypted Data comes out to be 1394
Now we will decrypt **1394** :
Decrypted Data = (cd)mod n
Thus our Encrypted Data comes out to be 89
8 = H and I = 9 i.e. "HI".

The security of RSA is based on the difficulty of factoring large composite numbers.

Procedure:

1. Create a new C++ file named `RSA.cpp`.

2. Implement the RSA algorithm as shown in the code section below.

Code:

```cpp

#include <iostream>

#include <cmath>


long long gcd(long long a, long long b) {

    return (b == 0) ? a : gcd(b, a % b);

}


long long modInverse(long long a, long long m) {

    // EEA to find modular inverse...

}
```

```
void generateKeys(long long &e, long long &d, long long &n) {
    // Key generation logic...
}

long long encrypt(long long plaintext, long long e, long long n) {
    // Encryption logic...
}

long long decrypt(long long ciphertext, long long d, long long n) {
    // Decryption logic...
}

int main() {
    long long e, d, n;
    generateKeys(e, d, n);

    long long plaintext;
    std::cout << "Enter plaintext: ";
    std::cin >> plaintext;

    long long ciphertext = encrypt(plaintext, e, n);
    std::cout << "Ciphertext: " << ciphertext << std::endl;

    long long decrypted = decrypt(ciphertext, d, n);
    std::cout << "Decrypted text: " << decrypted << std::endl;

    return 0;
}
```

Conclusion:

The RSA algorithm has been implemented successfully, demonstrating the principles of public-key cryptography, including key generation, encryption, and decryption.

Experiment 6: Generation and Use of Digital Signatures

Title:

Implementation of Digital Signatures in C++

Objectives:

1. To understand the concept of digital signatures and their importance.

2. To implement digital signature generation and verification in C++.

Theory:

Digital signatures provide a means to authenticate the origin and integrity of a message. They combine a cryptographic hash function with asymmetric encryption.

Process:

1. **Hashing the Message**: A hash function generates a fixed-size output from a variable-length input.

2. **Signing the Message**:

   - The sender hashes the message to create a digest.

   - The digest is encrypted using the sender’s private key to create the digital signature.

3. **Verifying the Signature**:

   - The recipient decrypts the signature using the sender’s public key to retrieve the hash.

   - The recipient generates their own hash of the received message.

   - If the two hashes match, the signature is valid, confirming the message’s authenticity.

Digital signatures are widely used in software distribution, financial transactions, and legal documents to ensure integrity and non-repudiation.

Procedure:

1. Create a new C++ file named `DigitalSignature.cpp`.

2. Implement the digital signature process as shown in the code section below.

Code:

```cpp
#include <iostream>
#include <openssl/sha.h>

std::string hashFunction(const std::string& message) {
    unsigned char hash[SHA_DIGEST_LENGTH];
    SHA1(reinterpret_cast<const unsigned char*>(message.c_str()), message.size(), hash);
    std::string hashedMessage;
    for (int i = 0; i < SHA_DIGEST_LENGTH; i++) {
        char buffer[3];
        sprintf(buffer, "%02x", hash[i]);
        hashedMessage += buffer;
    }
    return hashedMessage;
}

std::string signMessage(const std::string& message, long long privateKey) {
    std::string hashed = hashFunction(message);
    // Signing logic using private key...
    return hashed; // Placeholder
}

bool verifySignature(const std::string& message, const std::string& signature, long long publicKey) {
    std::string hashed = hashFunction(message);
    return hashed == signature; // Placeholder
}

int main() {
    std::string message;
    long long privateKey = 12345; // Example private key
    long long publicKey = 67890;   // Example public key
    std::cout << "Enter message: ";
    std::cin >> message;

    std::string signature = signMessage(message, privateKey);
    std::cout << "Signature: " << signature << std::endl;

    bool isVerified = verifySignature(message, signature, publicKey);
    std::cout << "Signature verification: " << (isVerified ? "Valid" : "Invalid") << std::endl;

    return 0;
}
```

Conclusion:

The implementation of digital signatures illustrates their role in ensuring data integrity and authenticity.

Experiment 7: Implementing SHA-1 Algorithm

Title:

Implementation of SHA-1 Algorithm in C++

Objectives:

1. To understand the SHA-1 hashing algorithm.

2. To implement SHA-1 using libraries in C++.

Theory:

SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function that produces a 160-bit hash value. It is commonly used for ensuring data integrity and authenticity.

Properties of SHA-1:

1. **Deterministic**: The same input will always produce the same hash.

2. **Fast Computation**: It can compute the hash value quickly.

3. **Collision Resistance**: It should be computationally infeasible to find two different inputs that yield the same hash.

4. **Pre-image Resistance**: Given a hash, it should be difficult to find the original input.

Despite its usefulness, SHA-1 has been found vulnerable to certain attacks, leading to recommendations for stronger algorithms like SHA-256

How SHA-1 Works

The block diagram of the SHA-1 (Secure Hash Algorithm 1) algorithm. Here’s a detailed description of each component and process in the diagram:

Components and Process Flow:

1. **Message (M)**:
   - The original input message that needs to be hashed.
2. **Message Padding**:
   - The initial step where the message is padded to ensure its length is congruent to 448 modulo 512. This step prepares the message for processing in 512-bit blocks.
3. **Round Word Computation (WtW_tWt)**:
   - After padding, the message is divided into blocks of 512 bits, and each block is further divided into 16 words of 32 bits. These words are then expanded into 80 32-bit words, which are used in the subsequent rounds.
4. **Round Initialize (A, B, C, D, and E)**:

- Initialization of five working variables (A, B, C, D, and E) with specific constant values. These variables are used to compute the hash value iteratively.

5. **Round Constants (KtK_tKt)**:
   - SHA-1 uses four constant values (K1K_1K1, K2K_2K2, K3K_3K3, K4K_4K4), each applied in a specific range of rounds:
     - K1K_1K1 for rounds 0-19
     - K2K_2K2 for rounds 20-39
     - K3K_3K3 for rounds 40-59
     - K4K_4K4 for rounds 60-79
6. **Rounds (0-79)**:
   - The main computation loop of SHA-1, divided into four stages (each corresponding to one of the constants K1K_1K1 to K4K_4K4). In each round, a combination of logical functions and operations is performed on the working variables (A, B, C, D, and E) using the words generated in the previous step.
7. **Final Round Addition**:
   - After all 80 rounds, the resulting values of A, B, C, D, and E are added to the original hash values to produce the final hash.
8. **MPX (Multiplexing)**:
   - Combines the results from the final round addition to form the final message digest.

Procedure:

1. Create a new C++ file named `SHA1.cpp`.

2. Implement the SHA-1 algorithm using the OpenSSL library as shown in the code section below.

Code:

```cpp
#include <iostream>

#include <openssl/sha.h>


std::string sha1Hash(const std::string& input) {

    unsigned char hash[SHA_DIGEST_LENGTH];

    SHA1(reinterpret_cast<const unsigned char*>(input.c_str()), input.size(), hash);

    std::string output;

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

        char buffer[3];

        sprintf(buffer, "%02x", hash[i]);

        output += buffer;
    }
    return output;
}

int main() {
    std::string input;
    std::cout << "Enter input string: ";
    std::cin >> input;

    std::string hash = sha1Hash(input);
    std::cout << "SHA-1 Hash: " << hash << std::endl;

    return 0;
}
```

Conclusion:

The SHA-1 implementation demonstrates the use of cryptographic hashing for data integrity, showcasing how hashes are generated from input data.

Experiment 8: Identifying Software Vulnerabilities

Title:

Identifying Software Vulnerabilities Using Tools

Objectives:

1. To explore techniques and tools for identifying software vulnerabilities.

2. To analyze the vulnerabilities discovered.

Theory:

Identifying software vulnerabilities is crucial for ensuring security. Vulnerabilities can lead to unauthorized access, data breaches, and various forms of cyberattacks.

Types of Vulnerabilities:

1. **Code Vulnerabilities**: Bugs or flaws in the code that can be exploited (e.g., buffer overflow, SQL injection).

2. **Configuration Vulnerabilities**: Incorrect settings or configurations that expose systems (e.g., open ports).

3. **Dependency Vulnerabilities**: Using outdated libraries or frameworks with known vulnerabilities.

Vulnerability Assessment Techniques:

1. **Static Analysis**: Analyzing code without executing it to find vulnerabilities.

2

. **Dynamic Analysis**: Analyzing running applications to identify vulnerabilities.

3. **Fuzz Testing**: Providing random data inputs to test application robustness.

Procedure:

1. Choose a tool such as OWASP ZAP, Burp Suite, or a static analysis tool like SonarQube.

2. Set up a sample application with known vulnerabilities (e.g., DVWA - Damn Vulnerable Web App).

3. Use the selected tool to scan the application and identify vulnerabilities.

Conclusion:

The experiment highlights the importance of identifying software vulnerabilities and demonstrates how to use tools for effective vulnerability assessment. Addressing these vulnerabilities is essential for maintaining secure applications.