Cryptography and Network Security Lab Manual

Index

Caesar Cipher

Theory

The Caesar Cipher is a classical substitution cipher in which each letter of the plaintext is shifted a certain number of positions down or up the alphabet. It is one of the simplest and most widely known encryption techniques, named after Julius Caesar, who is historically reported to have used it for military communications.

In this cipher, a fixed integer value known as the key is used to shift each alphabetic character in the plaintext. The operation wraps around the end of the alphabet so that after 'Z' comes 'A'. For decryption, the process is reversed by shifting in the opposite direction.

This cipher only affects alphabetic characters; all other characters (digits, punctuation, whitespace) remain unchanged. It supports both uppercase and lowercase characters, preserving their cases.

Algorithm

Input:

- A string containing the plaintext.
- An integer key representing the shift amount.

Encryption Steps:

1. Iterate through each character in the plaintext.
2. If the character is an uppercase letter:
    - Shift it forward by the key positions within the range 'A' to 'Z'.
3. If the character is a lowercase letter:
    - Shift it forward by the key positions within the range 'a' to 'z'.
4. If the character is non-alphabetic:
    - Leave it unchanged.
5. Concatenate the result to form the ciphertext.

Decryption Steps:

- Perform the same process but shift in the opposite direction by using (26 - key).

C++ Implementation

```
#include <iostream>
#include <string>
using namespace std;

// Function to encrypt plaintext using Caesar Cipher
string encrypt(const string& text, int key) {
    string result = "";
    key = key % 26; // Normalize the key to ensure it's within the range [0, 25]

    for (char ch : text) {
        if (isupper(ch)) {
            // Encrypt uppercase letters
            result += char((ch - 'A' + key) % 26 + 'A');
        } else if (islower(ch)) {
            // Encrypt lowercase letters
            result += char((ch - 'a' + key) % 26 + 'a');
        } else {
            // Leave non-alphabet characters unchanged
            result += ch;
        }
    }

    return result;
}

// Function to decrypt ciphertext using Caesar Cipher
string decrypt(const string& text, int key) {
    return encrypt(text, 26 - (key % 26)); // Reverse the shift for decryption
}

// Main driver function
int main() {
    string plaintext;
    int key;

    cout << "Enter the plaintext: ";
    getline(cin, plaintext);

    cout << "Enter the key (shift value): ";
    cin >> key;
```

```
    string cipherText = encrypt(plaintext, key);
    string decryptedText = decrypt(cipherText, key);

    cout << "\nEncrypted Text: " << cipherText << endl;
    cout << "Decrypted Text: " << decryptedText << endl;

    return 0;
}
```

Monoalphabetic Cipher

Theory

A **Monoalphabetic Cipher** is a type of **substitution cipher** where each letter in the plaintext is replaced by a corresponding letter from a fixed substitution alphabet. Unlike the Caesar cipher, where the shift is consistent and linear, monoalphabetic substitution uses an arbitrarily shuffled alphabet as the key.

For example, if the normal alphabet is:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
and the substitution alphabet is:
QWERTYUIOPASDFGHJKLZXCVBNM,
then the letter A is encrypted as Q, B as W, C as E, and so on.

This cipher is more secure than the Caesar cipher because it has 26! (factorial of 26) possible keys. However, it is still vulnerable to **frequency analysis**, especially if the ciphertext is long enough.

Algorithm

Input:

- A string (plaintext or ciphertext).
- A 26-character key string representing a shuffled alphabet (for encryption or decryption).

Encryption:

1. Create a mapping from each letter in the standard alphabet to a corresponding letter in the key.
2. For each character in the plaintext:
    - If it's a letter, substitute it with the mapped character from the key.

- Preserve the case of letters.
- Leave non-letter characters unchanged.

Decryption:

1. Reverse the mapping: map each character in the key back to the original alphabet.
2. Follow the same substitution process to retrieve the original text.

C++ Code

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;

// Function to create the encryption map
unordered_map<char, char> createEncryptMap(const string& key) {
    unordered_map<char, char> encMap;
    string alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    for (int i = 0; i < 26; ++i) {
        encMap[alphabet[i]] = toupper(key[i]);
        encMap[tolower(alphabet[i])] = tolower(key[i]);
    }

    return encMap;
}

// Function to create the decryption map
unordered_map<char, char> createDecryptMap(const string& key) {
    unordered_map<char, char> decMap;
    string alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    for (int i = 0; i < 26; ++i) {
        decMap[toupper(key[i])] = alphabet[i];
        decMap[tolower(key[i])] = tolower(alphabet[i]);
    }

    return decMap;
}

// Function to apply substitution using a given map
string substitute(const string& text, unordered_map<char, char>& subMap) {
    string result = "";
```

```
    for (char ch : text) {
        if (isalpha(ch)) {
            result += subMap[ch];
        } else {
            result += ch;
        }
    }
    return result;
}

int main() {
    string key;
    string plaintext;

    cout << "Enter a 26-letter key (shuffled alphabet): ";
    cin >> key;

    if (key.length() != 26) {
        cout << "Invalid key length. Key must be exactly 26 letters." << endl;
        return 1;
    }

    cin.ignore(); // Clear input buffer
    cout << "Enter plaintext: ";
    getline(cin, plaintext);

    // Encryption
    auto encryptMap = createEncryptMap(key);
    string ciphertext = substitute(plaintext, encryptMap);
    cout << "Encrypted Text: " << ciphertext << endl;

    // Decryption
    auto decryptMap = createDecryptMap(key);
    string decryptedText = substitute(ciphertext, decryptMap);
    cout << "Decrypted Text: " << decryptedText << endl;

    return 0;
}
```

Affine Cipher

Theory

The **Affine Cipher** is a type of monoalphabetic substitution cipher that applies a linear transformation to the position of each character in the alphabet. It uses the following formulas:

- **Encryption:**
  $E(x) = (a * x + b) \bmod 26$
- **Decryption:**
  $D(x) = a\_inv * (x - b) \bmod 26$

Where:

- x is the numerical equivalent of the plaintext character ($A=0$, $B=1$, ..., $Z=25$)
- a and b are keys of the cipher
- a_inv is the **modular inverse** of a modulo 26 (i.e., $(a * a\_inv) \% 26 = 1$)

The key a must be **coprime** with 26 to ensure the existence of the inverse. The Affine Cipher increases security by combining multiplication and addition, but it still remains vulnerable to frequency analysis.

Algorithm

Encryption Steps:

1. Convert each letter to its position x (0 to 25).
2. Compute $E(x) = (a * x + b) \bmod 26$.
3. Convert the result back to a character.
4. Keep non-alphabet characters unchanged.

Decryption Steps:

1. Compute the modular inverse a_inv of a.
2. Convert each letter to its numeric position y.
3. Compute $D(y) = a\_inv * (y - b + 26) \bmod 26$.
4. Convert the result back to a character.

C++ Code (Copy-Paste Ready)

```
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

// Function to compute modular inverse of a mod 26
int modInverse(int a, int m) {
    a = a % m;
    for (int x = 1; x < m; ++x) {
        if ((a * x) % m == 1) {
            return x;
```

```
        }
    }
    return -1; // No inverse if not found
}

// Encrypt function
string encrypt(const string& plaintext, int a, int b) {
    string result = "";

    for (char ch : plaintext) {
        if (isalpha(ch)) {
            char base = isupper(ch) ? 'A' : 'a';
            int x = ch - base;
            int enc = (a * x + b) % 26;
            result += (char)(enc + base);
        } else {
            result += ch;
        }
    }

    return result;
}

// Decrypt function
string decrypt(const string& ciphertext, int a, int b) {
    string result = "";
    int a_inv = modInverse(a, 26);

    if (a_inv == -1) {
        return "Invalid key: no modular inverse exists for a.";
    }

    for (char ch : ciphertext) {
        if (isalpha(ch)) {
            char base = isupper(ch) ? 'A' : 'a';
            int y = ch - base;
            int dec = (a_inv * (y - b + 26)) % 26;
            result += (char)(dec + base);
        } else {
            result += ch;
        }
    }

    return result;
}
```

```
// Main program
int main() {
    string text;
    int a, b;

    cout << "Enter the plaintext: ";
    getline(cin, text);

    cout << "Enter key a (coprime with 26): ";
    cin >> a;

    cout << "Enter key b: ";
    cin >> b;

    if (modInverse(a, 26) == -1) {
        cout << "Error: 'a' must be coprime with 26." << endl;
        return 1;
    }

    string encrypted = encrypt(text, a, b);
    string decrypted = decrypt(encrypted, a, b);

    cout << "Encrypted text: " << encrypted << endl;
    cout << "Decrypted text: " << decrypted << endl;

    return 0;
}
```

DES (Data Encryption Standard) – Single Round

Theory

The **Data Encryption Standard (DES)** is a symmetric-key block cipher that operates on 64-bit blocks of data using a 56-bit key. It involves **16 rounds** of complex operations, including substitution, permutation, and XOR with subkeys.

A **single round** of DES consists of the following steps:

1. **Initial input**: Split the 64-bit input block into two 32-bit halves: L and R.
2. **Expansion (E-box)**: Expand the 32-bit R to 48 bits using a predefined expansion table.
3. **Key Mixing**: XOR the expanded R with a 48-bit round key.
4. **Substitution (S-boxes)**: Divide the result into eight 6-bit segments. Each segment is input into a different S-box (S1 to S8) to get a 4-bit output.
5. **Permutation (P-box)**: Permute the 32-bit output from S-boxes using a fixed permutation table.

6. **Feistel Function**: XOR the result with the left half (L) to get the new R.
7. **Swap**: Swap the original L and R to prepare for the next round.

Simplified Algorithm for One Round

Input:

- 64-bit plaintext block (split into L and R)
- 48-bit subkey (K1)

Steps:

1. Expand R from 32 bits to 48 bits using the E-table.
2. XOR the expanded R with the subkey.
3. Apply S-box substitution on each 6-bit segment.
4. Apply P-box permutation.
5. Compute new R = L XOR P-box output.
6. Set new L = old R.

C++ Code (Single DES Round – Simplified)

```
#include <iostream>
#include <bitset>
#include <vector>
using namespace std;

// Dummy expansion table (E-box), S-box and P-box for simplification
int E[48] = {
    32, 1, 2, 3, 4, 5,
    4, 5, 6, 7, 8, 9,
    8, 9,10,11,12,13,
   12,13,14,15,16,17,
   16,17,18,19,20,21,
   20,21,22,23,24,25,
   24,25,26,27,28,29,
   28,29,30,31,32,1
};

// Dummy S-box (only 1 used for simplicity)
int S1[4][16] = {
    {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}
};

// Dummy P-box (Permutation)
int P[32] = {
    16,7,20,21,
    29,12,28,17,
    1,15,23,26,
    5,18,31,10,
    2,8,24,14,
    32,27,3,9,
    19,13,30,6,
    22,11,4,25
};

// Function to expand 32-bit R to 48-bit using E-box
bitset<48> expansion(bitset<32> R) {
    bitset<48> res;
    for (int i = 0; i < 48; i++) {
        res[47 - i] = R[32 - E[i]];
    }
    return res;
}

// S-box substitution (using only S1 for all blocks)
bitset<32> sboxSubstitution(bitset<48> input) {
    bitset<32> output;
    for (int i = 0; i < 8; i++) {
        int row = input[47 - (i * 6)] * 2 + input[47 - (i * 6 + 5)];
        int col = 0;
        for (int j = 1; j <= 4; j++) {
            col = col * 2 + input[47 - (i * 6 + j)];
        }
        int val = S1[row][col];
        for (int j = 0; j < 4; j++) {
            output[31 - (i * 4 + j)] = (val >> (3 - j)) & 1;
        }
    }
    return output;
}

// P-box permutation
bitset<32> permutation(bitset<32> input) {
    bitset<32> output;
    for (int i = 0; i < 32; i++) {
```

```
        output[31 - i] = input[32 - P[i]];
    }
    return output;
}

int main() {
    bitset<32> L("11001100110011001100110011001100");
    bitset<32> R("11110000111100001111000011110000");
    bitset<48> subkey("000110110000001011101111111111000111000001110010");

    cout << "Initial L: " << L << endl;
    cout << "Initial R: " << R << endl;

    // DES round begins
    bitset<48> R_expanded = expansion(R);
    bitset<48> R_xor_key = R_expanded ^ subkey;
    bitset<32> S_output = sboxSubstitution(R_xor_key);
    bitset<32> P_output = permutation(S_output);

    bitset<32> newR = L ^ P_output;
    bitset<32> newL = R;

    cout << "New L: " << newL << endl;
    cout << "New R: " << newR << endl;

    return 0;
}
```

Multiplicative Inverse using Extended Euclidean Algorithm

Theory

The **multiplicative inverse** of an integer a modulo m is an integer x such that:

$a \cdot x \equiv 1 \pmod{m}$

This inverse exists **only if a and m are coprime**, i.e., $\gcd(a, m) = 1$.

The **Extended Euclidean Algorithm** is used to compute not only the greatest common divisor (gcd) of a and m, but also the coefficients x and y such that:

$a \cdot x + m \cdot y = \gcd(a, m)$

If gcd(a, m) = 1, then x is the **modular inverse** of a modulo m. If x is negative, we can convert it to a positive equivalent using:

x=xmod mx = x \mod m

C++ Code

```
#include <iostream>
using namespace std;

// Function to implement the Extended Euclidean Algorithm
int extendedGCD(int a, int b, int &x, int &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }

    int x1, y1;
    int gcd = extendedGCD(b, a % b, x1, y1);

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

    return gcd;
}

// Function to find multiplicative inverse of a modulo m
int modInverse(int a, int m) {
    int x, y;
    int gcd = extendedGCD(a, m, x, y);

    if (gcd != 1) {
        // Inverse doesn't exist
        return -1;
    } else {
        // Ensure result is positive
        return (x % m + m) % m;
    }
}

int main() {
    int a, m;
```

```
    cout << "Enter a and m to compute inverse of a modulo m: ";
    cin >> a >> m;

    int inverse = modInverse(a, m);

    if (inverse == -1) {
        cout << "Multiplicative inverse does not exist (a and m are not coprime)." << endl;
    } else {
        cout << "Multiplicative inverse of " << a << " mod " << m << " is: " << inverse << endl;
    }

    return 0;
}
```

AES S-Box Creation

Theory

The **S-box** (Substitution box) in **AES (Advanced Encryption Standard)** is a crucial component of the SubBytes step. It provides **non-linearity** and **confusion**, which are essential for the cipher's security.

The AES S-box is a **16×16 lookup table** (256 entries), constructed in two main steps:

1. **Multiplicative Inverse** in GF(2^8):
    - For each byte value (0x00 to 0xFF), compute the multiplicative inverse in the finite field **GF(2^8)**.
    - The value 0x00 is mapped to itself.
2. **Affine Transformation**:
    - After the inversion, an **affine transformation** is applied over GF(2) using the formula:

      S(x)=A·x−1+bS(x) = A x^{-1} + b

    - Where:
        - x^{-1} is the inverse of x in GF(2^8)
        - A is a fixed 8×8 binary matrix
        - b is a fixed 8-bit vector: 0x63 in hexadecimal

This transformation is applied **bitwise** to the 8-bit input.

C++ Code to Generate AES S-Box

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

// Multiply two numbers in GF(2^8)
uint8_t gfMul(uint8_t a, uint8_t b) {
    uint8_t result = 0;
    while (b) {
        if (b & 1) result ^= a;
        bool hiBitSet = a & 0x80;
        a <<= 1;
        if (hiBitSet) a ^= 0x1B; // x^8 + x^4 + x^3 + x + 1
        b >>= 1;
    }
    return result;
}

// Compute multiplicative inverse in GF(2^8)
uint8_t gfInverse(uint8_t x) {
    if (x == 0) return 0;
    uint8_t inv = 1;
    for (uint8_t i = 1; i < 255; ++i) {
        if (gfMul(x, i) == 1) {
            inv = i;
            break;
        }
    }
    return inv;
}

// Affine transformation
uint8_t affineTransform(uint8_t x) {
    uint8_t result = 0;
    uint8_t c = 0x63;
    for (int i = 0; i < 8; i++) {
        uint8_t bit = (x >> i) & 1;
        uint8_t transformedBit = bit ^
            ((x >> ((i + 4) % 8)) & 1) ^
            ((x >> ((i + 5) % 8)) & 1) ^
            ((x >> ((i + 6) % 8)) & 1) ^
            ((x >> ((i + 7) % 8)) & 1) ^
            ((c >> i) & 1);
        result |= (transformedBit << i);
```

```
    }
    return result;
}

int main() {
    vector<uint8_t> sbox(256);

    for (int i = 0; i < 256; i++) {
        uint8_t inv = gfInverse(i);
        sbox[i] = affineTransform(inv);
    }

    // Print the S-box in 16x16 format
    cout << "AES S-Box:" << endl;
    for (int i = 0; i < 16; i++) {
        for (int j = 0; j < 16; j++) {
            cout << hex << setw(2) << setfill('0') << (int)sbox[i * 16 + j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Notes:

- The finite field **GF(2^8)** is used in AES for all byte-level operations.
- The irreducible polynomial used is:
  x^8+x^4+x^3+x+1x^8 + x^4 + x^3 + x + 1 → 0x11B or its reduction 0x1B for modular reduction.
- The affine transform ensures that the S-box is **non-linear** and **has no fixed points** or **opposite fixed points** — which increases its resistance to cryptanalysis.

Advanced Encryption Standard (AES)

Theory

AES is a symmetric block cipher standardized by NIST in 2001, designed to encrypt and decrypt data in blocks of **128 bits** (16 bytes). It supports key sizes of **128, 192, or 256 bits**, resulting in **10, 12, or 14 rounds**, respectively.

AES operates on a **4×4 byte matrix** known as the **state**. Each round (except the final one) includes four transformations:

1. **SubBytes** – Substitution using a non-linear S-box.
2. **ShiftRows** – Cyclically shift each row of the state.
3. **MixColumns** – Mix data within each column using matrix multiplication in GF(2^8).
4. **AddRoundKey** – XOR the state with a round key.

The first and last rounds differ slightly:

- **Initial Round**: Only AddRoundKey.
- **Final Round**: No MixColumns.

AES Encryption Algorithm (128-bit version)

Input:

- 16-byte plaintext block.
- 16-byte key.

Steps:

1. **Key Expansion** – Derive 11 round keys from the original key.
2. **Initial Round:**
    - AddRoundKey
3. **Rounds 1–9:**
    - SubBytes
    - ShiftRows
    - MixColumns
    - AddRoundKey
4. **Final Round:**
    - SubBytes

- ShiftRows
- AddRoundKey

5. **Output:** Ciphertext (16-byte block)

Simplified C++ Code (AES One Round Only)

The following code demonstrates a **single round** of AES (without full key expansion or full 10 rounds), suitable for learning the structure of AES.

```
#include <iostream>
#include <iomanip>
using namespace std;

const uint8_t sbox[256] = {
    // AES S-box values (only first few shown for brevity, fill rest if needed)
    0x63, 0x7c, 0x77, 0x7b, /* ... fill full S-box in real use ... */ 0x16
};

// Substitute bytes using the S-box
void subBytes(uint8_t state[4][4]) {
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
            state[i][j] = sbox[state[i][j]];
}

// ShiftRows transformation
void shiftRows(uint8_t state[4][4]) {
    uint8_t temp;

    // Row 1
    temp = state[1][0];
```

```
    for (int i = 0; i < 3; i++) state[1][i] = state[1][i + 1];
    state[1][3] = temp;

    // Row 2
    temp = state[2][0];
    uint8_t temp1 = state[2][1];
    state[2][0] = state[2][2];
    state[2][1] = state[2][3];
    state[2][2] = temp;
    state[2][3] = temp1;

    // Row 3
    temp = state[3][3];
    for (int i = 3; i > 0; i--) state[3][i] = state[3][i - 1];
    state[3][0] = temp;
}

// AddRoundKey (XOR with key)
void addRoundKey(uint8_t state[4][4], uint8_t key[4][4]) {
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
            state[i][j] ^= key[i][j];
}

// Print AES state matrix
void printState(uint8_t state[4][4]) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
```

```
            cout << hex << setw(2) << setfill('0') << (int)state[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    // Example state and key (for demonstration purposes)
    uint8_t state[4][4] = {
        {0x32, 0x88, 0x31, 0xe0},
        {0x43, 0x5a, 0x31, 0x37},
        {0xf6, 0x30, 0x98, 0x07},
        {0xa8, 0x8d, 0xa2, 0x34}
    };

    uint8_t roundKey[4][4] = {
        {0x2b, 0x28, 0xab, 0x09},
        {0x7e, 0xae, 0xf7, 0xcf},
        {0x15, 0xd2, 0x15, 0x4f},
        {0x16, 0xa6, 0x88, 0x3c}
    };

    cout << "Initial State:\n";
    printState(state);

    addRoundKey(state, roundKey);
    subBytes(state);
    shiftRows(state);
```

```
    cout << "\nState After One AES Round (without MixColumns):\n";
    printState(state);

    return 0;
}
```

Notes:

- This example includes only one AES round and omits MixColumns for simplicity.
- The **S-box** in the code is partially filled; for full functionality, you must complete it using standard AES S-box values.
- Proper AES implementation requires full **key expansion**, all **10 rounds**, and full **S-box/P-box tables**.

RSA Cryptosystem

Theory

RSA is a widely-used public-key cryptosystem for secure data transmission. It is based on the computational difficulty of factoring large prime numbers.

RSA involves two keys:

- **Public key (e, n)** – used for encryption
- **Private key (d, n)** – used for decryption

The security of RSA depends on:

- The product of two large primes p and q
- The difficulty of computing $\varphi(n)$ and its inverse modulo

Key Concepts

1. **Key Generation**:
    - Choose two distinct large prime numbers p and q.
    - Compute $n = p \times q$

- Compute Euler's totient function: $\varphi(n) = (p - 1)(q - 1)$
- Choose e such that $1 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$
- Compute $d \equiv e^{-1} \bmod \varphi(n)$ (modular inverse)

2. **Encryption**:
   - Ciphertext $c \equiv m^e \bmod n$
3. **Decryption**:
   - Plaintext $m \equiv c^d \bmod n$

RSA Algorithm

Input:

- Message m such that $0 < m < n$
- Public key (e, n)
- Private key (d, n)

Steps:

1. **Encrypt** using: $c = (m^e) \bmod n$
2. **Decrypt** using: $m = (c^d) \bmod n$

C++ Code (Simplified RSA)

```
#include <iostream>
using namespace std;

// Function to compute gcd
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

// Extended Euclidean Algorithm for modular inverse
int modInverse(int e, int phi) {
    int t = 0, newt = 1;
    int r = phi, newr = e;

    while (newr != 0) {
        int quotient = r / newr;
        int temp = newt;
        newt = t - quotient * newt;
        t = temp;
```

```
        temp = newr;
        newr = r - quotient * newr;
        r = temp;
    }

    if (r > 1) return -1; // No inverse
    if (t < 0) t += phi;
    return t;
}

// Modular exponentiation
long long modExp(long long base, long long exp, long long mod) {
    long long result = 1;
    base %= mod;

    while (exp > 0) {
        if (exp % 2 == 1)
            result = (result * base) % mod;

        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

int main() {
    // Choose small primes for demo
    int p = 61, q = 53;
    int n = p * q;            // n = 3233
    int phi = (p - 1) * (q - 1); // phi = 3120

    int e = 17; // Public exponent
    if (gcd(e, phi) != 1) {
        cout << "e and phi(n) are not coprime!" << endl;
        return 1;
    }

    int d = modInverse(e, phi); // Private exponent
    if (d == -1) {
        cout << "Modular inverse does not exist." << endl;
        return 1;
    }

    // Message to encrypt
    int m;
```

```
    cout << "Enter message (0 < m < " << n << "): ";
    cin >> m;

    // Encryption: c = m^e mod n
    int c = modExp(m, e, n);
    // Decryption: m = c^d mod n
    int decrypted = modExp(c, d, n);

    cout << "Public Key (e, n): (" << e << ", " << n << ")\n";
    cout << "Private Key (d, n): (" << d << ", " << n << ")\n";
    cout << "Encrypted: " << c << endl;
    cout << "Decrypted: " << decrypted << endl;

    return 0;
}
```

Example:

With p=61, q=53, e=17, and m=65, the output will be:

```
Encrypted: 2790
Decrypted: 65
```

Notes:

- This example uses **small primes** for clarity and demonstration.
- For **real-world RSA**, use large prime numbers (≥1024 bits) and libraries like GMP or OpenSSL for efficiency and security.

Diffie–Hellman Key Exchange (DHKE)

Theory

The **Diffie–Hellman Key Exchange** is a method that allows two parties to establish a **shared secret key** over an insecure channel, without transmitting the key directly. It relies on the hardness of the **Discrete Logarithm Problem** in modular arithmetic.

Mathematical Basis

Let:

- p = a large **prime number**

- g = a **primitive root modulo p** (also called generator)

Each party chooses:

- A **private key**: an integer a (Alice) or b (Bob)
- A **public key**: A=gamod p pA = g^a \mod p, B=gbmod p pB = g^b \mod p

They exchange public keys and compute the shared secret:

- Alice computes s=Bamod p ps = B^a \mod p
- Bob computes s=Abmod p ps = A^b \mod p

Since:

s=(gb)amod p=(ga)bmod p ps = (g^b)^a \mod p = (g^a)^b \mod p

Both arrive at the **same shared key**.

Diffie–Hellman Algorithm

Input:

- Prime number p
- Generator g
- Private keys a and b (chosen independently)

Steps:

1. Alice computes A=gamod p pA = g^a \mod p
2. Bob computes B=gbmod p pB = g^b \mod p
3. Exchange public keys
4. Compute shared secret:
    - Alice: s=Bamod p ps = B^a \mod p
    - Bob: s=Abmod p ps = A^b \mod p

C++ Code

```
#include <iostream>
using namespace std;

// Modular exponentiation (base^exp % mod)
long long modExp(long long base, long long exp, long long mod) {
    long long result = 1;
    base %= mod;

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

int main() {
    // Public parameters (agreed upon by both parties)
    long long p = 23;    // Prime number
    long long g = 5;     // Primitive root modulo p

    // Private keys (chosen secretly by each party)
    long long a = 6;     // Alice's private key
    long long b = 15;    // Bob's private key

    // Compute public keys
    long long A = modExp(g, a, p); // Alice's public key
    long long B = modExp(g, b, p); // Bob's public key

    // Exchange public keys and compute shared secret
    long long secretA = modExp(B, a, p); // Alice computes shared key
    long long secretB = modExp(A, b, p); // Bob computes shared key

    cout << "Public parameters:\n";
    cout << "p = " << p << ", g = " << g << "\n\n";

    cout << "Alice's private key: " << a << endl;
    cout << "Bob's private key: " << b << endl;

    cout << "\nAlice's public key (A): " << A << endl;
    cout << "Bob's public key (B): " << B << endl;
```

```
    cout << "\nShared secret (Alice computes): " << secretA << endl;
    cout << "Shared secret (Bob computes): " << secretB << endl;

    if (secretA == secretB)
        cout << "\nKey exchange successful. Shared secret: " << secretA << endl;
    else
        cout << "\nKey exchange failed." << endl;

    return 0;
}
```

Example Output

With $p = 23$, $g = 5$, $a = 6$, and $b = 15$:

```
Shared secret (Alice computes): 2
Shared secret (Bob computes): 2
Key exchange successful. Shared secret: 2
```

Notes

- In real-world applications, values of p are **very large primes** (e.g., 2048 bits), and g is a primitive root.
- The protocol is vulnerable to **Man-in-the-Middle attacks** unless authenticated with digital signatures.

Hash-Based Message Authentication Code (HMAC)

Theory

HMAC is a cryptographic technique used to verify both the **data integrity** and **authenticity** of a message. It combines a **cryptographic hash function** (like SHA-256) with a **secret key** to produce a fixed-length **authentication tag**.

HMAC Construction

HMAC is defined by the formula:

$$\text{HMAC}(K, m) = H((K' \oplus opad) \parallel H((K' \oplus ipad) \parallel m))$$

Where:

- H is a hash function (e.g., SHA-256)
- K is the secret key
- m is the message
- K' is the key padded to the block size
- ipad is the inner padding: 0x36 repeated
- opad is the outer padding: 0x5c repeated
- || denotes concatenation

The block size for SHA-256 is 64 bytes.

Applications

- Message authentication in TLS, IPsec, SSH
- Digital signatures
- Ensuring message integrity and authenticity

C++ Implementation of HMAC-SHA256

Note: Requires OpenSSL (or you can use any cryptographic library that provides SHA256).

```
#include <iostream>
#include <iomanip>
#include <openssl/hmac.h>
#include <openssl/evp.h>
#include <cstring>
using namespace std;

// Function to print HMAC in hexadecimal
void printHex(const unsigned char* data, int len) {
   for (int i = 0; i < len; ++i)
      cout << hex << setw(2) << setfill('0') << (int)data[i];
   cout << endl;
}

int main() {
   const char* key = "secretkey";
   const char* message = "This is a confidential message.";

   unsigned char* result;
   unsigned int len = 32; // SHA256 outputs 32 bytes
```

```
    result = HMAC(EVP_sha256(), key, strlen(key),
                  (unsigned char*)message, strlen(message), NULL, &len);

    cout << "HMAC (SHA-256) of message:\n";
    printHex(result, len);

    return 0;
}
```

Output Example:

For the message and key above, the output will look like:

HMAC (SHA-256) of message:
5d1cdbf5...<truncated>...

Notes:

- HMAC() is provided by OpenSSL and works with any digest algorithm (EVP_sha1(), EVP_md5(), etc.).
- Always use **constant-time comparison** to verify HMACs to prevent timing attacks.

OpenVAS – Open Vulnerability Assessment System

Overview

OpenVAS (Open Vulnerability Assessment System) is a **free and open-source vulnerability scanner** developed and maintained by **Greenbone Networks**. It is part of the **Greenbone Vulnerability Management (GVM)** framework and is widely used for detecting security issues in systems, networks, and applications.

Originally forked from the discontinued Nessus project when it became proprietary, OpenVAS has grown into a robust and actively maintained security toolset.

Key Features

- **Comprehensive Vulnerability Scanning**: Performs thousands of checks using regularly updated feeds (Greenbone Community Feed or Greenbone Security Feed).

- **Network-Based Scanning**: Scans remote systems for known vulnerabilities.
- **Protocol Support**: Supports numerous network protocols (HTTP, FTP, SSH, SNMP, SMB, etc.).
- **Custom Policies**: Allows users to create and manage scan policies tailored to specific environments.
- **Web Interface**: Accessible via **Greenbone Security Assistant (GSA)** — a browser-based UI.
- **Task Automation**: Supports scheduling, reporting, and email notifications.
- **Authenticated Scans**: Can log into systems for deeper analysis.

Architecture

OpenVAS is part of the **GVM framework**, which consists of several components:

- **openvas-scanner**: The core scanner that performs the vulnerability checks.
- **gvmd**: The Greenbone Vulnerability Manager daemon that handles tasks, configurations, and result management.
- **gsad**: Greenbone Security Assistant daemon providing the web interface.
- **GSA (Greenbone Security Assistant)**: A browser-based front end to manage scans and view results.

How It Works

1. **Setup**: Install OpenVAS and update the vulnerability feed.
2. **Configuration**:
    - Define target IPs/domains.
    - Create a scan configuration (port range, credentials, etc.).
3. **Scanning**: Launch scan tasks either manually or on a schedule.
4. **Analysis**: View and analyze results in the dashboard or export reports (PDF, HTML, XML).
5. **Remediation**: Use CVE references and suggested fixes to address vulnerabilities.

Installation (Debian)

```
sudo apt update
sudo apt install openvas
sudo gvm-setup
sudo gvm-check-setup
```

To start and access the web interface:

```
sudo gvm-start
```

Access the web UI at https://localhost:9392/

Use Cases

- Network vulnerability assessment
- Compliance auditing
- Penetration test preparation
- Continuous security monitoring
- Asset and service discovery

Advantages

- Open-source and free (community edition)
- Regular updates with public CVEs
- Integrates with enterprise tools via APIs
- Scalable and suitable for both small and large environments

---

Limitations

- Initial setup can be complex
- Resource-intensive during large scans
- Requires regular feed updates for accuracy
- Advanced features available only in the commercial edition (Greenbone Security Feed)

Alternatives

- **Nessus** (commercial)
- **Nmap** (for network discovery)
- **Qualys** (cloud-based)
- **Nexpose/InsightVM** by Rapid7
