BCSE309P - Cryptography & Network Security Lab

Lab Assignment 1

Kedar Shinde 22BCE1765

1 Aim

To implement the Caesar cipher, Vigenère cipher, Vernam cipher, Playfair cipher, Hill cipher, and Columnar transposition cipher.

2 Steps for Encryption and Decryption

2.1 Caesar Cipher

The Caesar cipher, named after Julius Caesar, is a substitution cipher that shifts each letter in the plaintext by a fixed number of positions in the alphabet.

2.1.1 Steps for Encryption

1. For each character in the plaintext message:
   a) Check if the character is alphabetic using `isalpha()`
   b) Determine the base value:
      - If lowercase: base = 'a' (97 in ASCII)
      - If uppercase: base = 'A' (65 in ASCII)
   c) Apply the shift transformation:

$$E(c) = (c - \text{base} + \text{shift}) \bmod 26 + \text{base} \tag{1}$$

   d) Non-alphabetic characters remain unchanged

2.1.2 Steps for Decryption

1. For each character in the ciphertext:
   a) Check if the character is alphabetic
   b) Determine the base value (same as encryption)
   c) Apply the reverse shift:

$$D(c) = (c - \text{base} - \text{shift} + 26) \bmod 26 + \text{base} \tag{2}$$

   d) The +26 ensures positive modulus result

2.2 Vigenère Cipher

The Vigenère cipher enhances the Caesar cipher by using a keyword to create multiple shift values, making it more resistant to frequency analysis.

2.2.1 Key Generation Process

1. Calculate required key length:

$$\text{required\_length} = \text{message\_length} \tag{3}$$

2. Generate repeating key:

   a) Calculate complete repetitions: reps $= \lfloor \frac{\text{message\_length}}{\text{key\_length}} \rfloor$

   b) Calculate remaining characters: remainder = message_length mod key_length

   c) Repeat key `reps` times

   d) Append first `remainder` characters of key

2.2.2 Encryption Steps

1. Generate the extended key to match message length

2. For each character position i:

   a) Get plaintext character p_i and key character k_i

   b) If p_i is alphabetic:

      - Determine character case and base
      - Apply Vigenère transformation:

$$E(p_i) = ((p_i - \text{base} + (k_i - \text{base})) \bmod 26) + \text{base} \tag{4}$$

   c) Preserve non-alphabetic characters

2.2.3 Decryption Steps

1. Generate the same extended key

2. For each character position i:

   a) Get ciphertext character c_i and key character k_i

   b) If c_i is alphabetic:

      - Determine character case and base
      - Apply reverse transformation:

$$D(c_i) = ((c_i - \text{base} - (k_i - \text{base}) + 26) \bmod 26) + \text{base} \tag{5}$$

2.3 Vernam Cipher

The Vernam cipher, also known as the one-time pad when used with a truly random key, provides perfect secrecy when implemented correctly.

2.3.1 Key Requirements

1. Key length must equal or exceed message length
2. Key should ideally be truly random
3. Key must never be reused

2.3.2 Encryption Steps

1. Validate key length against message length
2. For each character position i:
   a) Perform bitwise XOR operation:
   $$\text{temp} = p_i \oplus k_i \tag{6}$$
   b) Ensure printable ASCII result:
   $$E(p_i) = \text{temp}\&0x7F \tag{7}$$

2.3.3 Decryption Steps

1. For each character position i:
   a) Perform bitwise XOR with same key:
   $$\text{temp} = c_i \oplus k_i \tag{8}$$
   b) Ensure printable ASCII:
   $$D(c_i) = \text{temp}\&0x7F \tag{9}$$

2.4 Playfair Cipher

The Playfair cipher encrypts pairs of letters using a 5×5 matrix constructed from a keyword.

2.4.1 Key Matrix Generation

1. Process the keyword:
   a) Remove duplicate characters
   b) Convert to lowercase
   c) Replace 'j' with 'i'
2. Create the alphabet string:
   a) Use "abcdefghiklmnopqrstuvwxyz" (no 'j')

b) Remove letters already in processed key

3. Construct 5×5 matrix:

   a) Fill with processed key first
   b) Fill remaining positions with remaining alphabet

2.4.2 Plaintext Preprocessing

1. Remove non-alphabetic characters
2. Convert to lowercase
3. Replace 'j' with 'i'
4. Split into digraphs (pairs):

   a) If pair has same letters, insert 'x' between them
   b) If text length is odd, append 'x'

2.4.3 Encryption Rules

1. For each letter pair (a, b):

   a) Find positions (row_1, col_1) and (row_2, col_2) in matrix
   b) Apply transformation rules:
      - Same row: $E(a) = M[row_1][(col_1+1) \bmod 5]$, $E(b) = M[row_2][(col_2+1) \bmod 5]$
      - Same column: $E(a) = M[(row_1+1) \bmod 5][col_1]$, $E(b) = M[(row_2+1) \bmod 5][col_2]$
      - Rectangle: $E(a) = M[row_1][col_2]$, $E(b) = M[row_2][col_1]$

2.4.4 Decryption Rules

1. For each letter pair (a, b):

   a) Find positions in matrix
   b) Apply reverse transformations:
      - Same row: $D(a) = M[row_1][(col_1-1) \bmod 5]$, $D(b) = M[row_2][(col_2-1) \bmod 5]$
      - Same column: $D(a) = M[(row_1-1) \bmod 5][col_1]$, $D(b) = M[(row_2-1) \bmod 5][col_2]$
      - Rectangle: $D(a) = M[row_1][col_2]$, $D(b) = M[row_2][col_1]$

2.5 Hill Cipher

The Hill cipher uses linear algebra for encryption, representing text using matrices and performing matrix multiplication.

2.5.1 Matrix Operations

1. Calculate matrix determinant:

$$\det(K) = k_{11}k_{22} - k_{12}k_{21} \tag{10}$$

2. Find modular multiplicative inverse:

$$\det(K)^{-1} \bmod 26 \text{ where } \det(K) \times \det(K)^{-1} \equiv 1 \pmod{26} \tag{11}$$

3. Calculate adjugate matrix:

$$adj(K) = \begin{bmatrix} k_{22} & -k_{12} \\ -k_{21} & k_{11} \end{bmatrix} \tag{12}$$

2.5.2 Encryption Steps

1. Preprocess plaintext:
   a) Remove non-alphabetic characters
   b) Convert to lowercase
   c) Pad with 'x' if needed
2. Convert text to numbers (a=0, b=1, etc.)
3. For each block of n letters (n = matrix size):
   a) Create column vector P
   b) Compute $C = KP \bmod 26$
   c) Convert result back to letters

2.5.3 Decryption Steps

1. Calculate inverse key matrix K^{-1}:

$$K^{-1} = \det(K)^{-1} \times adj(K) \bmod 26 \tag{13}$$

2. For each block of n letters:
   a) Create column vector C
   b) Compute $P = K^{-1}C \bmod 26$
   c) Convert result back to letters

2.6 Rail Fence Cipher (Columnar Transposition)

The columnar transposition cipher rearranges characters based on the alphabetical ordering of a keyword.

2.6.1 Encryption Steps

1. Calculate dimensions:
   a) Number of columns = key length
   b) Number of rows = $\lceil \frac{\text{message\_length}}{\text{key\_length}} \rceil$
2. Create grid:
   a) Fill grid row by row with plaintext
   b) Pad incomplete final row with spaces
3. Determine column order:
   a) Number columns based on keyword letter positions
   b) Create mapping of column numbers to positions
4. Read off columns:
   a) Read columns in order determined by key
   b) Ignore padding characters

2.6.2 Decryption Steps

1. Calculate dimensions (same as encryption)
2. Calculate column lengths:
   a) Full columns: $\lfloor \frac{\text{message\_length}}{\text{key\_length}} \rfloor$
   b) Extra characters: message_length mod key_length
3. Reconstruct columns:
   a) Determine original column order from key
   b) Place appropriate number of characters in each column
4. Read plaintext:
   a) Read grid row by row
   b) Combine characters to form plaintext

3 Implementation

3.1 Caesar Cipher Implementation

```cpp
std::string caesar_encrypt(const std::string &message, int shift) {
    std::string encrypted;
    for (char c : message) {
        if (std::isalpha(c)) {
            char base = std::islower(c) ? 'a' : 'A';
            encrypted += (c - base + shift) % 26 + base;
        } else {
            encrypted += c;
        }
    }
    return encrypted;
}

std::string caesar_decrypt(const std::string &message, int shift) {
    std::string decrypted;
    for (char c : message) {
        if (std::isalpha(c)) {
            char base = std::islower(c) ? 'a' : 'A';
            decrypted += (c - base - shift + 26) % 26 + base;
        } else {
            decrypted += c;
        }
    }
    return decrypted;
}
```

3.2 Vigenère Cipher Implementation

```cpp
std::string generate_repeating_key(const std::string &key, int message_length) {
    std::string actual_key;
    int k_len = key.length();

    int reps = message_length / k_len;
    int rem = message_length % k_len;

    for (int i = 0; i < reps; i++) {
        actual_key += key;
    }
    actual_key += key.substr(0, rem);
    return actual_key;
```

```
}

std::string vigenere_encrypt(const std::string &message, const std::string &key) {
    std::string encrypted;
    std::string actual_key = generate_repeating_key(key, message.length());

    for (int i = 0; i < message.length(); i++) {
        char c = message[i];
        char k = actual_key[i];
        if (std::isalpha(c)) {
            char base = std::islower(c) ? 'a' : 'A';
            encrypted += ((c - base + (k - base)) % 26) + base;
        } else {
            encrypted += c;
        }
    }
    return encrypted;
}

std::string vigenere_decrypt(const std::string &message, const std::string &key) {
    std::string decrypted;
    std::string actual_key = generate_repeating_key(key, message.length());

    for (int i = 0; i < message.length(); i++) {
        char c = message[i];
        char k = actual_key[i];
        if (std::isalpha(c)) {
            char base = std::islower(c) ? 'a' : 'A';
            decrypted += ((c - base - (k - base) + 26) % 26) + base;
        } else {
            decrypted += c;
        }
    }
    return decrypted;
}
```

3.3 Vernam Cipher Implementation

```
std::string vernam_encrypt(const std::string &message, const std::string &key) {
    if (key.length() < message.length()) {
```

```cpp
        throw std::invalid_argument("Key length must be at least equal to the message length.");
    }

    std::string encrypted;
    for (size_t i = 0; i < message.length(); i++) {
        encrypted += (message[i] ^ key[i]) & 0x7F;
    }
    return encrypted;
}

std::string vernam_decrypt(const std::string &encrypted, const std::string &key) {
    std::string decrypted;
    for (size_t i = 0; i < encrypted.length(); i++) {
        decrypted += (encrypted[i] ^ key[i]) & 0x7F;
    }
    return decrypted;
}
```

3.4 Playfair Cipher Implementation

```cpp
std::string removeDuplicates(const std::string& str) {
    std::string result;
    for (char ch : str) {
        if (result.find(ch) == std::string::npos) {
            result += ch;
        }
    }
    return result;
}

std::vector<std::vector<char>> generateKeyMatrix(const std::string& key) {
    std::string processedKey = removeDuplicates(key);
    processedKey.erase(std::remove(processedKey.begin(), processedKey.end(), 'j'),
                       processedKey.end());
    std::string alphabet = "abcdefghiklmnopqrstuvwxyz";

    for (char ch : alphabet) {
        if (processedKey.find(ch) == std::string::npos) {
            processedKey += ch;
        }
    }

    std::vector<std::vector<char>> keyMatrix(5, std::vector<char>(5));
```

```
    int index = 0;
    for (int i = 0; i < 5; ++i) {
        for (int j = 0; j < 5; ++j) {
            keyMatrix[i][j] = processedKey[index++];
        }
    }
    return keyMatrix;
}

std::string preprocessPlaintext(std::string plaintext) {
    plaintext.erase(std::remove_if(plaintext.begin(),
        plaintext.end(),
                       [](char ch) { return !std::isalpha(ch);
                           }), plaintext.end());
    std::transform(plaintext.begin(), plaintext.end(),
        plaintext.begin(), ::tolower);
    std::replace(plaintext.begin(), plaintext.end(), 'j', 'i
        ');

    std::string processedText;
    for (size_t i = 0; i < plaintext.length(); ++i) {
        processedText += plaintext[i];
        if (i + 1 < plaintext.length() && plaintext[i] ==
            plaintext[i + 1]) {
            processedText += 'x';
        }
    }

    if (processedText.length() % 2 != 0) {
        processedText += 'x';
    }
    return processedText;
}

std::string playfair_encrypt(const std::string& plaintext,
    const std::string& key) {
    std::vector<std::vector<char>> keyMatrix =
        generateKeyMatrix(key);
    std::string processedText = preprocessPlaintext(
        plaintext);

    std::string ciphertext;
    for (size_t i = 0; i < processedText.length(); i += 2) {
        ciphertext += encryptPair(keyMatrix, processedText[i
            ], processedText[i + 1]);
    }
    return ciphertext;
}

```

```cpp
std::string playfair_decrypt(const std::string& ciphertext, const std::string& key) {
    std::vector<std::vector<char>> keyMatrix = generateKeyMatrix(key);

    std::string plaintext;
    for (size_t i = 0; i < ciphertext.length(); i += 2) {
        plaintext += decryptPair(keyMatrix, ciphertext[i], ciphertext[i + 1]);
    }
    return plaintext;
}
```

3.5 Hill Cipher Implementation

```cpp
int determinant(const std::vector<std::vector<int>>& matrix, int n) {
    if (n == 1) return matrix[0][0];

    int det = 0;
    std::vector<std::vector<int>> submatrix(n - 1, std::vector<int>(n - 1));
    for (int x = 0; x < n; x++) {
        int subi = 0;
        for (int i = 1; i < n; i++) {
            int subj = 0;
            for (int j = 0; j < n; j++) {
                if (j == x) continue;
                submatrix[subi][subj] = matrix[i][j];
                subj++;
            }
            subi++;
        }
        det += (x % 2 == 0 ? 1 : -1) * matrix[0][x] * determinant(submatrix, n - 1);
    }
    return det;
}

std::vector<std::vector<int>> matrixInverse(const std::vector<std::vector<int>>& matrix, int n) {
    int det = determinant(matrix, n);
    int detModInverse = modularInverse(det, 26);

    std::vector<std::vector<int>> adjoint(n, std::vector<int>(n));
    if (n == 1) {
        adjoint[0][0] = 1;
```

```
        return adjoint;
    }

    // Calculate adjoint matrix
    std::vector<std::vector<int>> temp(n - 1, std::vector<
        int>(n - 1));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            int subi = 0;
            for (int x = 0; x < n; x++) {
                if (x == i) continue;
                int subj = 0;
                for (int y = 0; y < n; y++) {
                    if (y == j) continue;
                    temp[subi][subj] = matrix[x][y];
                    subj++;
                }
                subi++;
            }
            adjoint[j][i] = (determinant(temp, n - 1) * ((i
                + j) % 2 == 0 ? 1 : -1));
        }
    }

    // Multiply adjoint by modular multiplicative inverse of
        determinant
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            adjoint[i][j] = mod(adjoint[i][j] *
                detModInverse, 26);
        }
    }
    return adjoint;
}

std::string hill_encrypt(const std::string& plaintext,
                        const std::vector<std::vector<int>>&
                            key) {
    int n = key.size();
    std::string processedText = plaintext;
    processedText.erase(
        std::remove_if(processedText.begin(), processedText.
            end(),
                      [](char c) { return !std::isalpha(c);
                          }),
        processedText.end()
    );
    std::transform(processedText.begin(), processedText.end
        (),
                  processedText.begin(), ::tolower);
```

```

    while (processedText.size() % n != 0) {
        processedText += 'x';
    }

    std::string ciphertext;
    for (size_t i = 0; i < processedText.size(); i += n) {
        for (int row = 0; row < n; ++row) {
            int sum = 0;
            for (int col = 0; col < n; ++col) {
                sum += key[row][col] * (processedText[i + col] - 'a');
            }
            ciphertext += (sum % 26) + 'a';
        }
    }
    return ciphertext;
}

std::string hill_decrypt(const std::string& ciphertext,
                         const std::vector<std::vector<int>>& key) {
    int n = key.size();
    std::vector<std::vector<int>> inverseKey = matrixInverse(key, n);

    std::string plaintext;
    for (size_t i = 0; i < ciphertext.size(); i += n) {
        for (int row = 0; row < n; ++row) {
            int sum = 0;
            for (int col = 0; col < n; ++col) {
                sum += inverseKey[row][col] * (ciphertext[i + col] - 'a');
            }
            plaintext += (mod(sum, 26)) + 'a';
        }
    }
    return plaintext;
}
```

3.6 Rail Fence Cipher (Columnar Transposition) Implementation

```
std::string columnar_encrypt(const std::string& plaintext, const std::string& key) {
    int numCols = key.size();
    int numRows = (plaintext.size() + numCols - 1) / numCols;
```

```cpp

    std::vector<std::string> grid(numRows, std::string(numCols, ' '));
    for (size_t i = 0; i < plaintext.size(); ++i) {
        grid[i / numCols][i % numCols] = plaintext[i];
    }

    std::vector<int> columnOrder(numCols);
    for (size_t i = 0; i < key.size(); ++i) {
        columnOrder[i] = i;
    }

    std::sort(columnOrder.begin(), columnOrder.end(),
              [&key](int a, int b) { return key[a] < key[b]; });

    std::string ciphertext;
    for (int col : columnOrder) {
        for (int row = 0; row < numRows; ++row) {
            if (grid[row][col] != ' ') {
                ciphertext += grid[row][col];
            }
        }
    }

    return ciphertext;
}

std::string columnar_decrypt(const std::string& ciphertext, const std::string& key) {
    int numCols = key.size();
    int numRows = (ciphertext.size() + numCols - 1) / numCols;

    std::vector<int> columnLengths(numCols, numRows);
    int extraChars = ciphertext.size() % numCols;
    for (int i = 0; i < numCols; ++i) {
        if (i >= extraChars) {
            columnLengths[i]--;
        }
    }

    std::vector<int> columnOrder(numCols);
    for (size_t i = 0; i < key.size(); ++i) {
        columnOrder[i] = i;
    }

    std::sort(columnOrder.begin(), columnOrder.end(),
              [&key](int a, int b) { return key[a] < key[b]; });
```

```cpp

    std::vector<std::string> grid(numCols);
    int index = 0;
    for (int col : columnOrder) {
        grid[col] = ciphertext.substr(index, columnLengths[
            col]);
        index += columnLengths[col];
    }

    std::string plaintext;
    for (int row = 0; row < numRows; ++row) {
        for (int col = 0; col < numCols; ++col) {
            if (row < grid[col].size()) {
                plaintext += grid[col][row];
            }
        }
    }

    return plaintext;
}
```

3.7 Server Code

```cpp
    // Server code
#include
#include
#include
#include
#include
#include
#include "encrypt.h"
#define PORT 8080
#define BUFFER_SIZE 256
int main() {
        int encryption = 0;
        std::cout << "Select Algorithm:\n";
        std::cout << "1) Caesar\n";
        std::cout << "2) Vigenere\n";
        std::cout << "3) Vernam\n";
        std::cout << "4) Playfair\n";
        std::cout << "5) Hill\n";
        std::cout << "6) Rail Fence\n";
        std::cin >> encryption;
        int server_fd, client_fd;
        struct sockaddr_in server_addr, client_addr;
        socklen_t client_len;
        char buffer[BUFFER_SIZE];
        // Create TCP socket
```

```
        server_fd = socket(AF_INET, SOCK_STREAM, 0);
        if (server_fd == -1) {
                perror("Socket creation failed");
                return 1;
        }
        const int enable = 1;
        if(setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0)
    perror("setsockopt(SO_REUSEADDR) failed");
        // Set up server address
        memset(&server_addr, 0, sizeof(server_addr));
        server_addr.sin_family = AF_INET;
        server_addr.sin_addr.s_addr = INADDR_ANY;
        server_addr.sin_port = htons(PORT);
        // Bind the socket
        if (bind(server_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) == -1) {
                perror("Bind failed");
                close(server_fd);
                return 1;
        }
        // Listen for connections
        if (listen(server_fd, 5) == -1) {
                perror("Listen failed");
                close(server_fd);
                return 1;
        }
        std::cout << "Server listening on port " << PORT << std::endl;
        // Accept a client connection
        client_len = sizeof(client_addr);
        client_fd = accept(server_fd, (struct sockaddr*)&client_addr, &client_len);
        if (client_fd == -1) {
                perror("Accept failed");
                close(server_fd);
                return 1;
        }
        std::cout << "Client connected." << std::endl;
        // Communicate with client
        while (true) {
                memset(buffer, 0, BUFFER_SIZE);
                ssize_t bytes_read = read(client_fd, buffer, BUFFER_SIZE - 1);
                if (bytes_read <= 0) {
                        if (bytes_read == 0)
                                std::cout << "Client disconnected." << std::endl;
                        else
```

```
                perror("Read error");
            break;
        }
        std::cout << "Received:\t" << buffer << std::endl;
        std::cout << "Decrypted:\t" << decrypt(buffer, encryption) << std::endl;
        std::string message = "";
        std::cout << "Enter Message:\t" ;
        std::cin >> message;
        std::string encrypted_message = encrypt(message, encryption);
        std::cout << "Encrypted message:\t" << encrypted_message << std::endl;
        if (write(client_fd, encrypted_message.c_str(), encrypted_message.length()) == -1) {
            perror("Write error");
            break;
        }
    }
    close(client_fd);
    close(server_fd);
    return 0;
}
```

3.8 Client Code

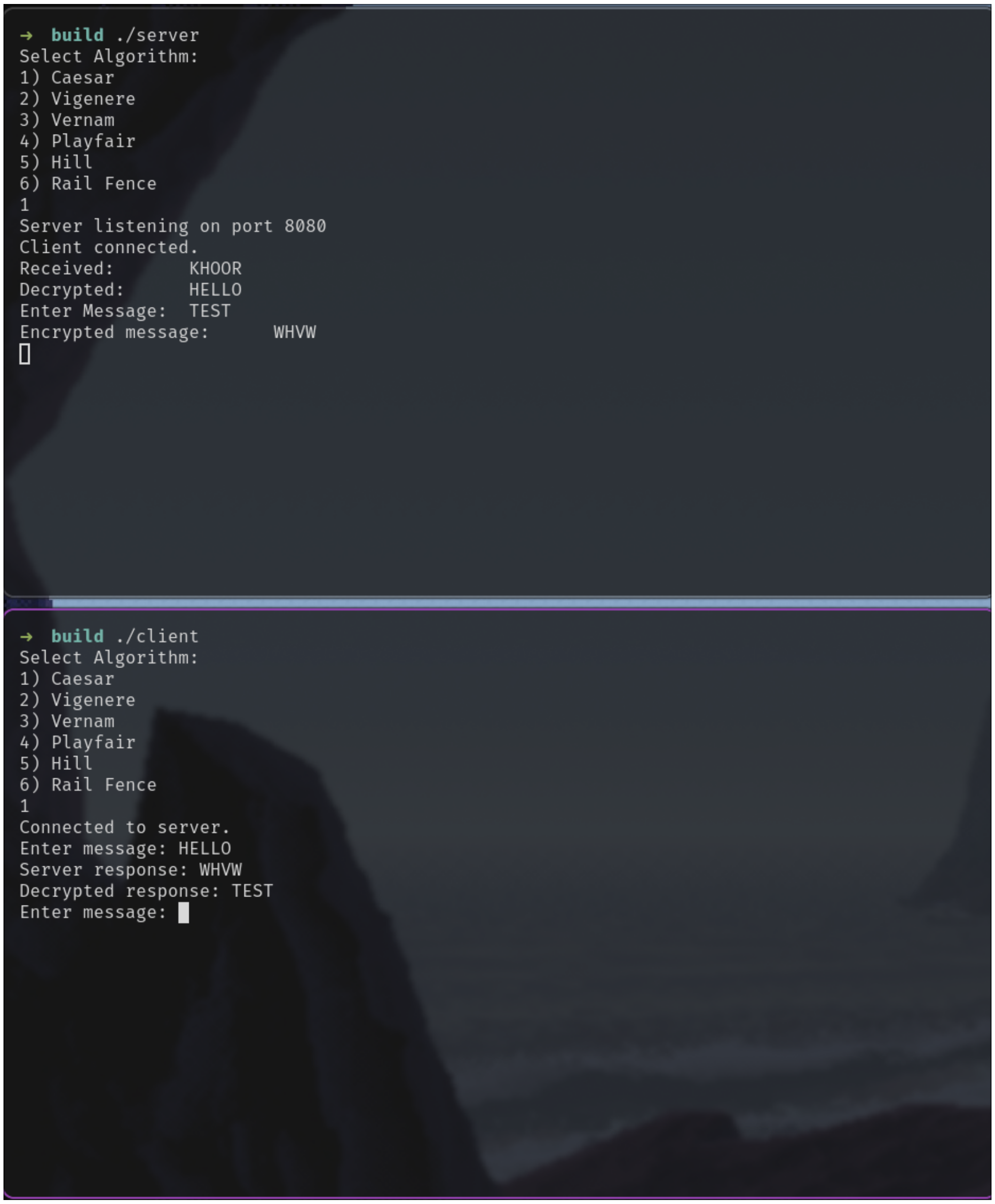
```

// Client code
#include
#include
#include
#include
#include
#include "encrypt.h"
#define PORT 8080
#define BUFFER_SIZE 256
int main() {
    int encryption = 0;
    std::cout << "Select Algorithm:\n";
    std::cout << "1) Caesar\n";
    std::cout << "2) Vigenere\n";
    std::cout << "3) Vernam\n";
    std::cout << "4) Playfair\n";
    std::cout << "5) Hill\n";
    std::cout << "6) Rail Fence\n";
    std::cin >> encryption;
    int client_fd;
```

```
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];
    // Create TCP socket
    client_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (client_fd == -1) {
        perror("Socket creation failed");
        return 1;
    }
    // Set up server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);
    // Connect to server
    if (connect(client_fd, (struct sockaddr*)&
        server_addr, sizeof(server_addr)) == -1) {
        perror("Connect failed");
        close(client_fd);
        return 1;
    }
    std::cout << "Connected to server." << std::endl;
    // Communicate with server
    while (true) {
        std::cout << "Enter message: ";
        std::cin >> buffer;
        std::string encrypted_message = encrypt(
            buffer, encryption);
        strcpy(buffer, encrypted_message.c_str());
        if (std::strcmp(buffer, "exit") == 0) {
            break;
        }
        // Send message to server
        if (write(client_fd, buffer, strlen(buffer))
             == -1) {
            perror("Write error");
            break;
        }
        // Read response from server
        memset(buffer, 0, BUFFER_SIZE);
        ssize_t bytes_read = read(client_fd, buffer,
             BUFFER_SIZE - 1);
        if (bytes_read <= 0) {
            if (bytes_read == 0)
                std::cout << "Server
                    disconnected." << std::
                    endl;
            else
                perror("Read error");
            break;
        }
```

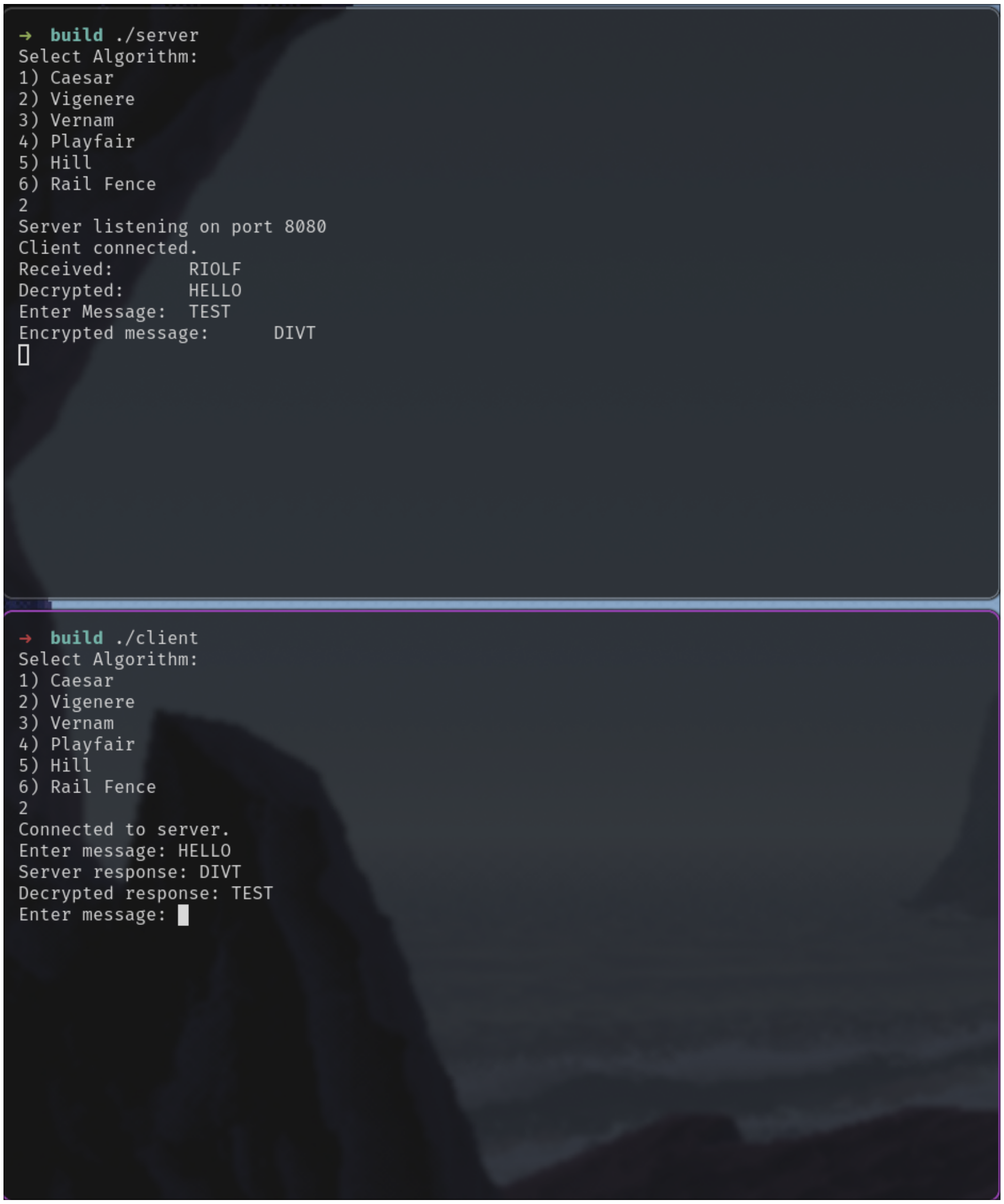
```
                    std::cout << "Server response: " << buffer
                        << std::endl;
                    std::cout << "Decrypted response: " <<
                        decrypt(buffer, encryption) << std::endl;
          }
          close(client_fd);
          return 0;
}
```

4 Outputs

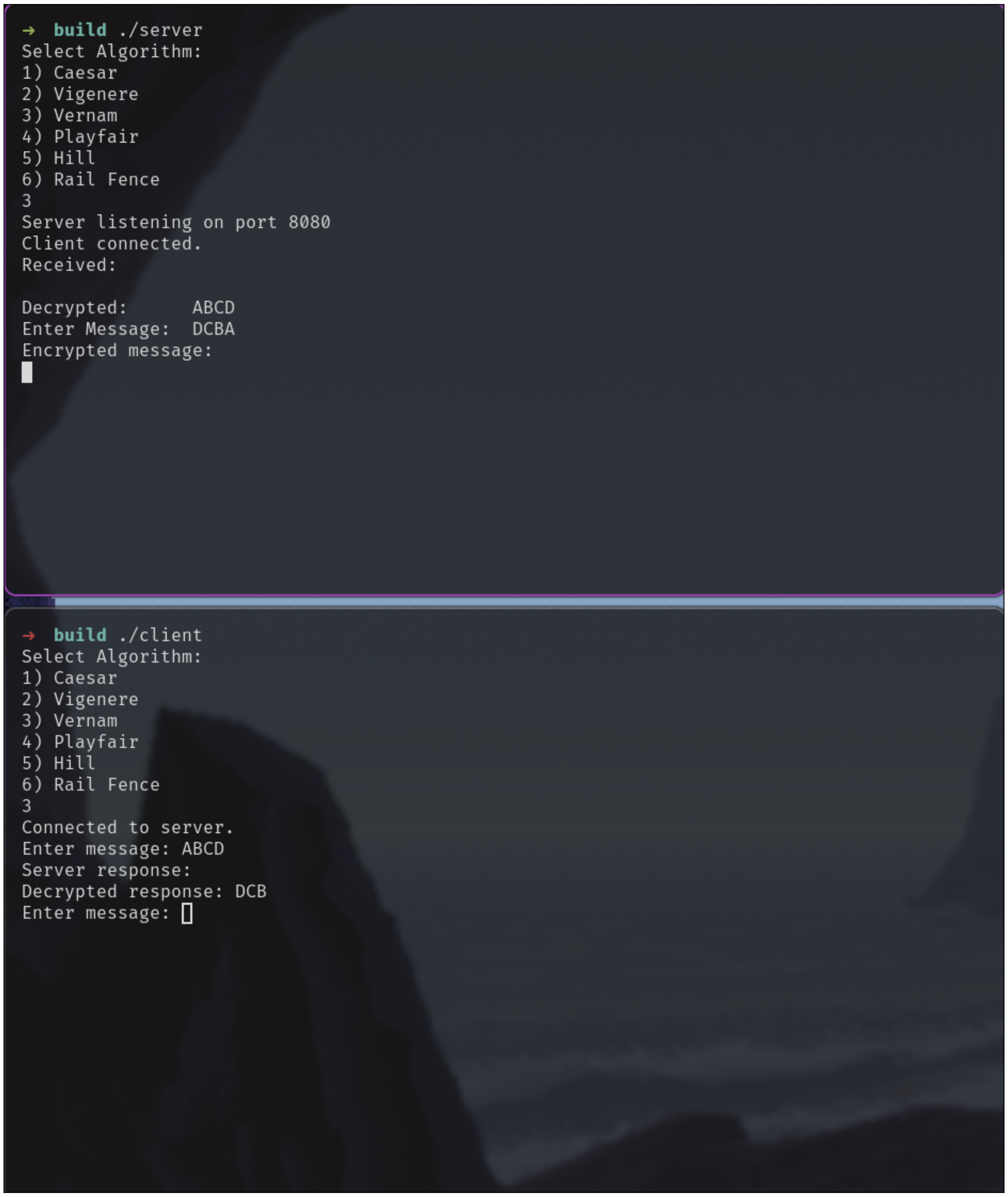
4.1 Caesar Cipher



4.2 Vigenère Cipher



4.3 Vernam Cipher

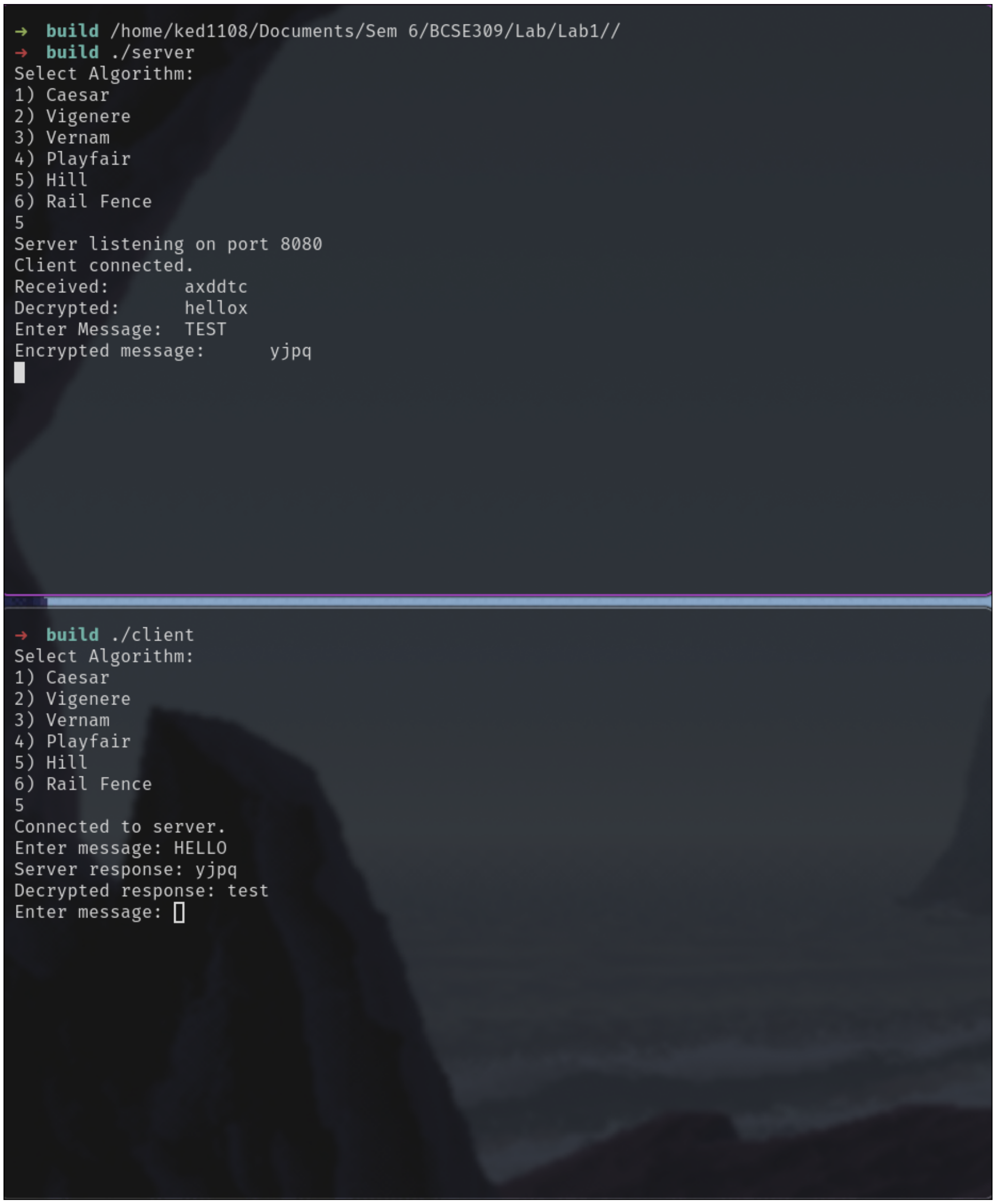


4.4 Playfair Cipher

```
→ build ./server
Select Algorithm:
1) Caesar
2) Vigenere
3) Vernam
4) Playfair
5) Hill
6) Rail Fence
4
Server listening on port 8080
Client connected.
Received:       lckyog
Decrypted:      m i n u a
b c d e f
g h k l o
p q r s t
v w x y z
helxlo
Enter Message:  yellow
m i n u a
b c d e f
g h k l o
p q r s t
v w x y z
Encrypted message:      ulkyogxy
```

```
→ build ./client
Select Algorithm:
1) Caesar
2) Vigenere
3) Vernam
4) Playfair
5) Hill
6) Rail Fence
4
Connected to server.
Enter message: hello
m i n u a
b c d e f
g h k l o
p q r s t
v w x y z
Server response: ulkyogxy
Decrypted response: m i n u a
b c d e f
g h k l o
p q r s t
v w x y z
yelxlowx
Enter message:
```

4.5 Hill Cipher



4.6 Rail Fence Cipher (Columnar Transposition Cipher)