

Project Title:

Diffie-Hellman Key Exchange and XOR Encryption/Decryption

Course: CP614A Applied Cryptography

Instructor:

Lunshan (Shaun) Gao, Ph.D., P.Eng

Group Members:

Harshit Kawatra: 235826130

Ismail Mukadam: 235800530

Mohammed Sohail Ahmed: 235807480

19 July 2024

Table of Contents

Content	Page No.
1. Introduction	2
2. Theoretical Background	3
a. Diffie-Hellman Key Exchange	3
b. XOR Encryption/Decryption	4
3. Implementation	5
a. Overview	5
b. Server Program	5
c. Client Program	7
4. Testing	9
5. Discussion	10
a. Issues and Challenges	10
b. Security Considerations	10
6. Conclusion	11
7. References	16

Introduction

Purpose of the Project

In this current project, we have begun with cryptography and have done the key exchange by virtue of the Diffie-Hellman algorithm and performed encryption/decryption with help of XOR operation. The general goal of the module is to convert these methods into practical experience about how they are interconnected to protect information which is exchanged via electronic channels.

Project Objectives

Here's what we're aiming to achieve:

1. Understand Cryptography: We have chosen modern cryptographic algorithms, including Diffie-Hellman and XOR encryption, to gain a deeper understanding of their functions.
2. Enhance Skills: This project is aimed at enhancing the capacity and experience of the team in the solving of problems as pertaining to cryptographic code.
3. Show Practical Use: The intended use of this project is to display how the encrypted messages and keys are exchanged between server and client.

Scope

This project covers:

1. Implementing Diffie-Hellman: We will discuss how to set up and run the Diffie-Hellman key exchange algorithm.
2. Using XOR Encryption: Messages must not be compromised and for this; we will be applying XOR encryption.
3. Testing and Validation: We will thoroughly test the implementation to ensure it works correctly and apply necessary actions to minimise errors.
4. Improving Code: We will improve the readability of the code and thus improve documentation of the code and make it more robust and secure.

Significance

The Diffie-Hellman key exchange is one of the ten standards in the history of cryptography that provide two parties with a means of developing a secure secret key over the insecure communication channel. When it is combined with XOR-based encryption, that can ensure that the key exchange step as well as the messages are transmitted securely. This project will help in expounding on these principles and create awareness on the relevance of these principles in offering a guard on digital communication in our day-to-day undertakings.

Theoretical Background

Diffie-Hellman Key Exchange

Whitfield Diffie and Martin Hellman created the Diffie-Hellman key exchange method in 1976. This is a mechanism that allows two people to exchange secret keys across an unsecure channel; everyone has a set of public and private keys. This shared secret can then be used to encrypt future messages with a symmetric key cipher (Carts, 2001).

Mathematical Foundation

The security of the Diffie-Hellman key exchange is dependent on the difficulty of solving the discrete logarithm problem. The algorithm involves the following steps:

1. Selection of Parameters:
 - Both parties agree on a large prime number p and a generator g of the multiplicative group of integers modulo p .
2. Key Generation:
 - Each party generates a private key a and b , respectively, which are random numbers.
 - The corresponding public keys are computed as:
$$A = g^a \mod p \quad \text{and} \quad B = g^b \mod p$$
3. Key Exchange:

The public keys A and B are exchanged over the insecure channel.
4. Shared Secret Computation:

Each party computes the shared secret using their private key and the other party's public key:

$$\text{Shared Secret} = B^a \mod p = A^b \mod p$$

The shared secret can be further hashed to derive a symmetric key for encrypting the subsequent communication.

Security Aspects

The Diffie-Hellman key exchange is safe due to the computing challenges associated with the discrete logarithm problem. However, it is susceptible to man-in-the-middle attacks. To reduce this issue, several changes and variants have been offered:

1. Man-in-the-Middle Attack Prevention: It was reviewed that various strategies to prevent MitM attacks, emphasising the need for authenticated key exchanges and additional cryptographic protocols to verify the identity of the parties involved (Mitra et al., 2021).
2. Enhanced Security Protocols: A signed group Diffie-Hellman key exchange protocol with tight security guarantees, reducing the risk of such attacks (Pan et al., 2022).

XOR Encryption/Decryption

The XOR (exclusive or) operation is a simple, yet fundamental, operation used in many encryption algorithms. The XOR cipher uses the following properties of the XOR operation:

1. Reversibility: $A \oplus A = 0$ and $A \oplus 0 = A$.
2. Symmetry: $A \oplus B = B \oplus A$.

These properties make XOR particularly useful for encryption and decryption processes.

Encryption and Decryption Process

1. Encryption: The plaintext message is XORed with a key of the same length to produce the ciphertext.

$$\text{Ciphertext} = \text{Plaintext} \oplus \text{Key}$$

2. Decryption: The ciphertext is XORed with the same key to retrieve the original plaintext.

$$\text{Plaintext} = \text{Ciphertext} \oplus \text{Key}$$

Use Cases and Limitations

While XOR is simple and efficient, it has significant security limitations when used alone, particularly if the key is reused (known as a key reuse vulnerability). XOR encryption is often used in combination with other cryptographic techniques to enhance security. The enhancement of the Diffie-Hellman protocol's security by incorporating RSA cryptography, thereby mitigating the vulnerabilities of simple XOR encryption (Gupta & Reddy, 2022).

Improved Diffie-Hellman Protocols

Several researchers suggest improvements to the traditional Diffie-Hellman protocol to increase its security and efficiency: Several researchers suggest improvements to the traditional Diffie-Hellman protocol to increase its security and efficiency:

1. Suggested Modified Diffie-Hellman Protocol: This protocol employs the Diffie-Hellman algorithm but adds additional security measures to prevent MitM attacks and improve communication security (Kara et al., 2021).
2. A lightweight key agreement approach was introduced for wireless sensor networks, based on hyper elliptic curve Diffie-Hellman, achieves a balance between security and computational performance (Naresh et al., 2020).

These advancements contribute to the robust application of the Diffie-Hellman key exchange in various fields, ensuring secure and efficient cryptographic communication.

Implementation

Overview

Programming Languages: The implementation is done in C++.

Environment: The development environment includes a Unix based system for running the C++ code and GCC for compiling the code, and tools such as netcat for testing the network. The code was written and tested on Ubuntu 20.04.

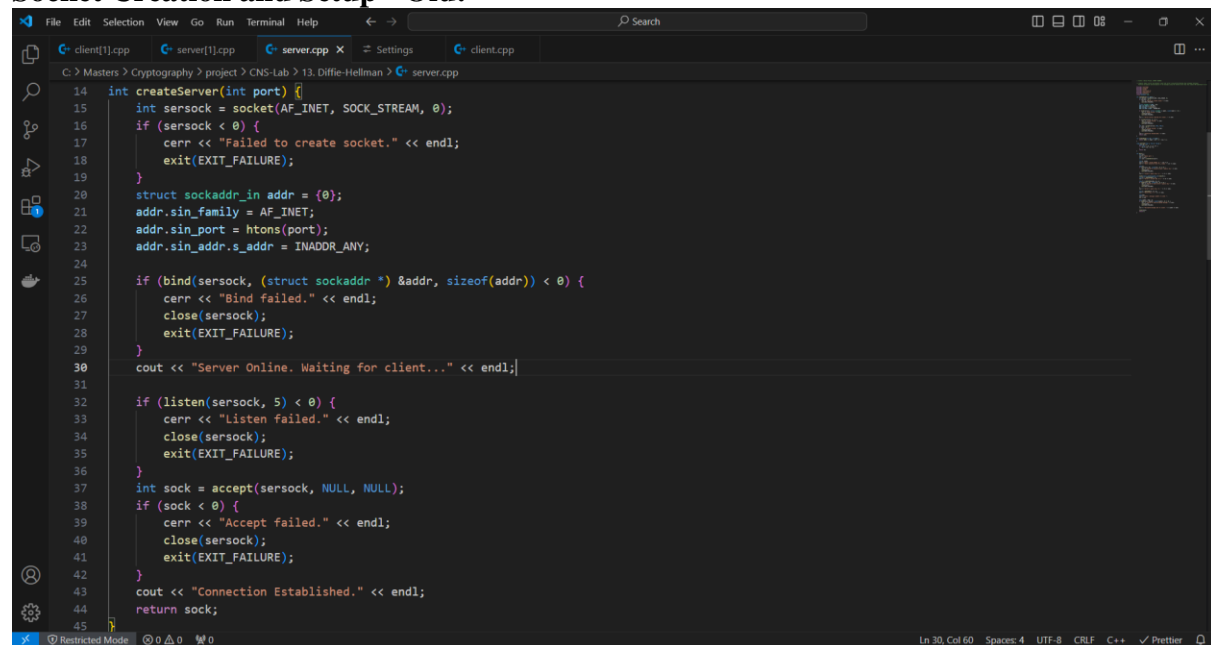
Algorithm Choice: The Diffie-Hellman key exchange algorithm was utilized for the efficient exchange of cryptographic keys via the public channel. XOR encryption was used for its simplicity and ease of understanding, this made it easy to explain the basic form of encryption and decryption.

Server Program

Description: After completing the Diffie-Hellman key exchange to obtain a shared secret key, the server application waits for a client connection and encrypts the message using XOR encryption before sending it to the client.

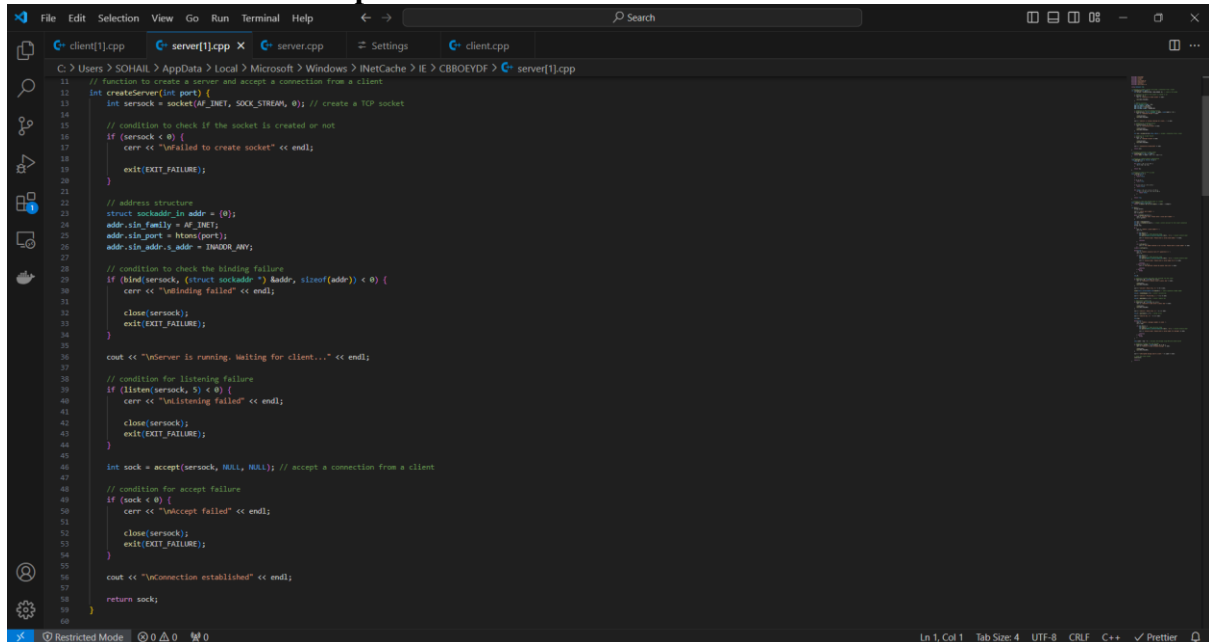
Code Explanation:

Socket Creation and Setup - Old:



```
14 int createServer(int port) {
15     int sersocket = socket(AF_INET, SOCK_STREAM, 0);
16     if (sersocket < 0) {
17         cerr << "Failed to create socket." << endl;
18         exit(EXIT_FAILURE);
19     }
20     struct sockaddr_in addr = {0};
21     addr.sin_family = AF_INET;
22     addr.sin_port = htons(port);
23     addr.sin_addr.s_addr = INADDR_ANY;
24
25     if (bind(sersocket, (struct sockaddr *) &addr, sizeof(addr)) < 0) {
26         cerr << "Bind failed." << endl;
27         close(sersocket);
28         exit(EXIT_FAILURE);
29     }
30     cout << "Server Online. Waiting for client..." << endl;
31
32     if (listen(sersocket, 5) < 0) {
33         cerr << "Listen failed." << endl;
34         close(sersocket);
35         exit(EXIT_FAILURE);
36     }
37     int sock = accept(sersocket, NULL, NULL);
38     if (sock < 0) {
39         cerr << "Accept failed." << endl;
40         close(sersocket);
41         exit(EXIT_FAILURE);
42     }
43     cout << "Connection Established." << endl;
44     return sock;
45 }
```

Socket Creation and Setup - New:



```
11 // function to create a server and accept a connection from a client
12 int createServer(int port) {
13     int server_sock = socket(AF_INET, SOCK_STREAM, 0); // create a TCP socket
14
15     // condition to check if the socket is created or not
16     if (server_sock < 0) {
17         cerr << "failed to create socket" << endl;
18         exit(EXIT_FAILURE);
19     }
20
21     // address structure
22     struct sockaddr_in addr = {0};
23     addr.sin_family = AF_INET;
24     addr.sin_port = htons(port);
25     addr.sin_addr.s_addr = INADDR_ANY;
26
27     // condition to check the binding failure
28     if (bind(server_sock, (struct sockaddr *) &addr, sizeof(addr)) < 0) {
29         cerr << "binding failed" << endl;
30         close(server_sock);
31         exit(EXIT_FAILURE);
32     }
33
34     cout << "server is running. waiting for client..." << endl;
35
36     // condition for listening failure
37     if (listen(server_sock, 5) < 0) {
38         cerr << "unlistening failed" << endl;
39         close(server_sock);
40         exit(EXIT_FAILURE);
41     }
42
43     int sock = accept(server_sock, NULL, NULL); // accept a connection from a client
44
45     // condition for accept failure
46     if (sock < 0) {
47         cerr << "unaccept failed" << endl;
48         close(server_sock);
49         exit(EXIT_FAILURE);
50     }
51
52     cout << "a connection established" << endl;
53
54     return sock;
55 }
```

This method builds a server socket, binds it to the given port, waits for connections to come in, and accepts connections from clients.

- Key Generation (Private and Public Keys):

```
long a = randInRange(1, P); // server's private key
long A = powermod(G, a, P); // server's public key
```

The server generates a private key `a` and computes its public key `A` using the primitive root `G` and the prime number `P`.

Key Exchange Process:

```
recv(sock, &B, sizeof(B), 0); // receive client's public key
send(sock, &A, sizeof(A), 0); // send server's public key
long S = powermod(B, a, P); // compute shared secret key
```

The server receives the client's public key `B`, sends its own public key `A`, and computes the shared secret key `S`.

Message Encryption and Transmission:

```
long cipher = msg ^ S; // encrypts the message using XOR with
shared secret
send(sock, &cipher, sizeof(cipher), 0)
```

The server encrypts the message using XOR with the shared secret key `k` and sends the encrypted message to the client.

Error Handling:

Basic error handling is done by checking the return values of socket functions and printing messages if something goes wrong.

Enhancements:

The server code could be enhanced by adding more robust error handling, secure random number generation, and input validation to improve security and reliability.

Error Handling:

If something goes wrong, the simplest way to handle errors is to emit messages and check the return values of socket functions.

Improvements:

To increase security and dependability, the server code might be strengthened by incorporating more reliable error handling, safe random number generation, and input validation.

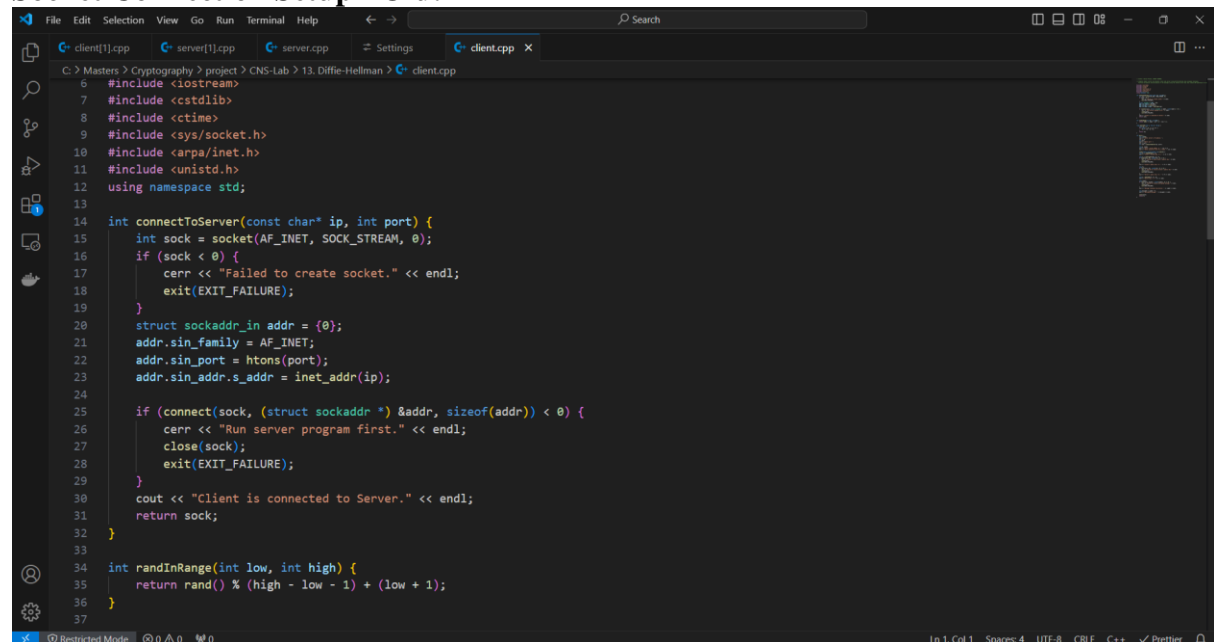
Client Program

Description:

After establishing a connection with the server, the client software uses XOR decryption to decode the message it has received. It then executes the Diffie-Hellman key exchange to create a shared secret key.

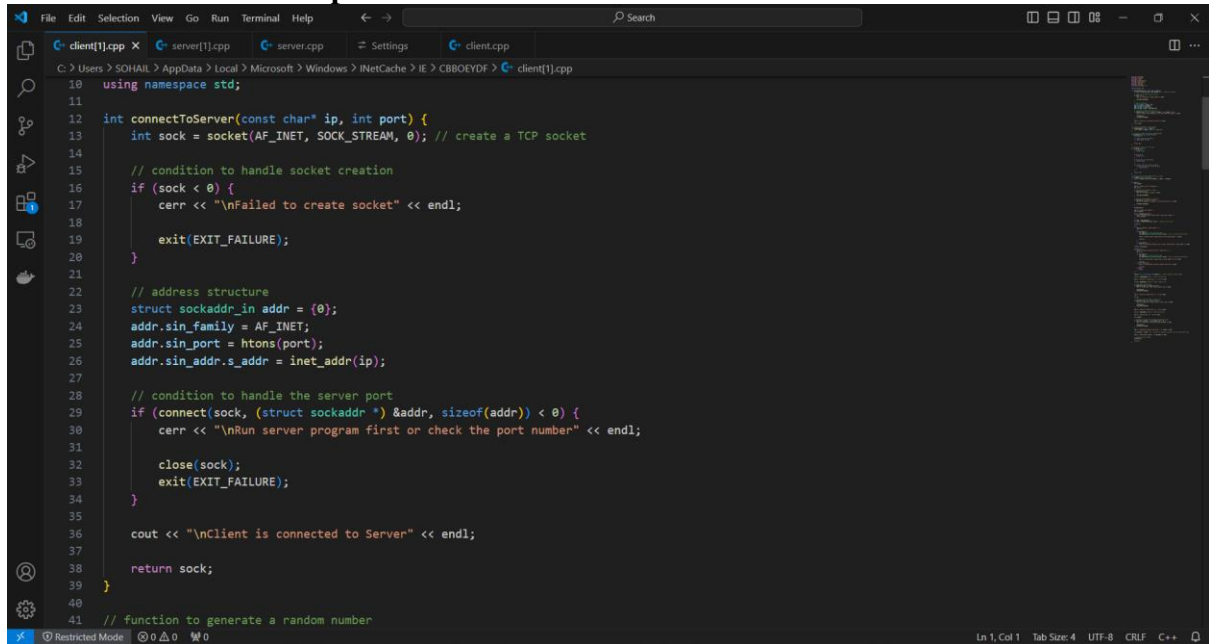
Code Explanation:

Socket Connection Setup - Old:



```
6 #include <iostream>
7 #include <cstdlib>
8 #include <ctime>
9 #include <sys/socket.h>
10 #include <arpa/inet.h>
11 #include <unistd.h>
12 using namespace std;
13
14 int connectToServer(const char* ip, int port) {
15     int sock = socket(AF_INET, SOCK_STREAM, 0);
16     if (sock < 0) {
17         cerr << "Failed to create socket." << endl;
18         exit(EXIT_FAILURE);
19     }
20     struct sockaddr_in addr = {0};
21     addr.sin_family = AF_INET;
22     addr.sin_port = htons(port);
23     addr.sin_addr.s_addr = inet_addr(ip);
24
25     if (connect(sock, (struct sockaddr *) &addr, sizeof(addr)) < 0) {
26         cerr << "Run server program first." << endl;
27         close(sock);
28         exit(EXIT_FAILURE);
29     }
30     cout << "Client is connected to Server." << endl;
31     return sock;
32 }
33
34 int randInRange(int low, int high) {
35     return rand() % (high - low + 1) + (low + 1);
36 }
37
```


Socket Connection Setup - New:



```
10 using namespace std;
11
12 int connectToServer(const char* ip, int port) {
13     int sock = socket(AF_INET, SOCK_STREAM, 0); // create a TCP socket
14
15     // condition to handle socket creation
16     if (sock < 0) {
17         cerr << "\nFailed to create socket" << endl;
18         exit(EXIT_FAILURE);
19     }
20
21     // address structure
22     struct sockaddr_in addr = {0};
23     addr.sin_family = AF_INET;
24     addr.sin_port = htons(port);
25     addr.sin_addr.s_addr = inet_addr(ip);
26
27     // condition to handle the server port
28     if (connect(sock, (struct sockaddr *) &addr, sizeof(addr)) < 0) {
29         cerr << "\nRun server program first or check the port number" << endl;
30         close(sock);
31         exit(EXIT_FAILURE);
32     }
33
34     cout << "\nClient is connected to Server" << endl;
35
36     return sock;
37 }
38
39 // function to generate a random number
```

This function makes a client socket and connects it to the server's port and IP address.

Key Generation (Private and Public Keys):

```
long b = randInRange(1, P); // client's private key
long B = powermod(G, b, P); // client's public key
```

The client generates a private key `b` and computes its public key `B` using the primitive root `G` and the prime number `P`.

Key Exchange Process:

```
send(sock, &B, sizeof(B), 0); // send client's public key
recv(sock, &A, sizeof(A), 0); // receive server's public key
long S = powermod(A, b, P); // compute shared secret key
```

The client sends its public key `B`, receives the server's public key `A`, and computes the shared secret key `k`.

Message Reception and Decryption:

```
recv(sock, &cipher, sizeof(cipher), 0);
long decipher = cipher ^ S; // decrypt the message using XOR
with the shared secret key
```

After receiving the encrypted message from the server, the client uses the shared secret key {S} to perform XOR decryption, after which the message is printed.

Error Handling:

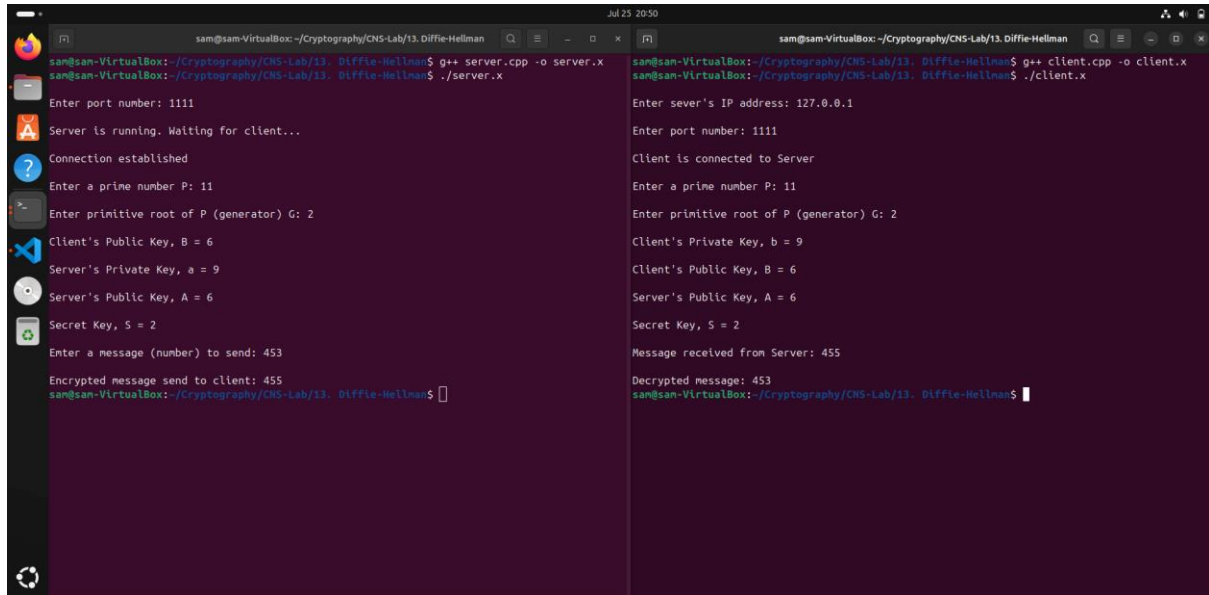
If something goes wrong, the simplest way to handle errors is to verify the return values of socket functions and output messages.

Improvements:

To increase security and dependability, the client code might benefit from stronger error handling, safe random number generation, and input validation.

Testing:

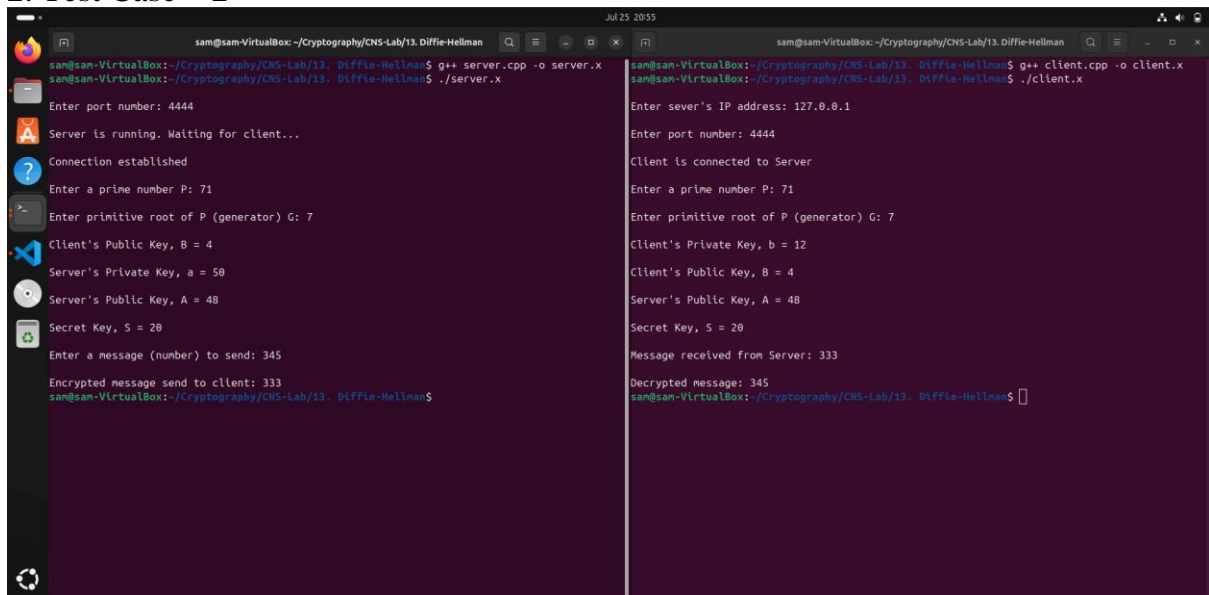
1. Test Case – 1



```
sam@sam-VirtualBox: ~/Cryptography/CNS-Lab/13. Diffie-Hellman
sam@sam-VirtualBox:~/Cryptography/CNS-Lab/13. Diffie-Hellman$ g++ server.cpp -o server.x
sam@sam-VirtualBox:~/Cryptography/CNS-Lab/13. Diffie-Hellman$ ./server.x
Enter port number: 1111
Server is running. Waiting for client...
Connection established
Enter a prime number P: 11
Enter primitive root of P (generator) G: 2
Client's Public Key, B = 6
Server's Private Key, a = 9
Server's Public Key, A = 6
Secret Key, S = 2
Enter a message (number) to send: 453
Encrypted message send to client: 455
sam@sam-VirtualBox:~/Cryptography/CNS-Lab/13. Diffie-Hellman$

saml@sam-VirtualBox:~/Cryptography/CNS-Lab/13. Diffie-Hellman$ g++ client.cpp -o client.x
saml@sam-VirtualBox:~/Cryptography/CNS-Lab/13. Diffie-Hellman$ ./client.x
Enter sever's IP address: 127.0.0.1
Enter port number: 1111
Client is connected to Server
Enter a prime number P: 11
Enter primitive root of P (generator) G: 2
Client's Private Key, b = 9
Client's Public Key, B = 6
Server's Public Key, A = 6
Secret Key, S = 2
Message received from Server: 455
Decrypted message: 453
saml@sam-VirtualBox:~/Cryptography/CNS-Lab/13. Diffie-Hellman$
```

2. Test Case – 2



```
sam@sam-VirtualBox:~/Cryptography/CNS-Lab/13. Diffie-Hellman$ g++ server.cpp -o server.x
sam@sam-VirtualBox:~/Cryptography/CNS-Lab/13. Diffie-Hellman$ ./server.x
Enter port number: 4444
Server is running. Waiting for client...
Connection established
Enter a prime number P: 71
Enter primitive root of P (generator) G: 7
Client's Public Key, B = 4
Server's Private Key, a = 50
Server's Public Key, A = 48
Secret Key, S = 20
Enter a message (number) to send: 345
Encrypted message send to client: 333
sam@sam-VirtualBox:~/Cryptography/CNS-Lab/13. Diffie-Hellman$

saml@sam-VirtualBox:~/Cryptography/CNS-Lab/13. Diffie-Hellman$ g++ client.cpp -o client.x
saml@sam-VirtualBox:~/Cryptography/CNS-Lab/13. Diffie-Hellman$ ./client.x
Enter sever's IP address: 127.0.0.1
Enter port number: 4444
Client is connected to Server
Enter a prime number P: 71
Enter primitive root of P (generator) G: 7
Client's Private Key, b = 12
Client's Public Key, B = 4
Server's Public Key, A = 48
Secret Key, S = 20
Message received from Server: 333
Decrypted message: 345
saml@sam-VirtualBox:~/Cryptography/CNS-Lab/13. Diffie-Hellman$
```

Discussion

Issues and Challenges

1. Error Handling Implementation:

- **Challenge:** Identifying all failure points in socket operations and user inputs.
- **Resolution:** Systematically added checks and error messages for each critical operation (socket creation, binding, etc.), ensuring graceful exits with informative messages.

2. Input Validation:

- **Challenge:** Validating user inputs like port numbers, prime numbers, and generator values while handling edge cases.
- **Resolution:** Implemented `isNumber()` and `isPrime()` functions for input checks and used loops to prompt for valid inputs, ensuring the program only proceeds with correct data.

Security Considerations

Security of Diffie-Hellman:

Aspects: Diffie-Hellman is very important in the establishment of shared secret when using insecure channels, while not transmitting the secret in various protocols such as TLS and SSH (Wouters & Arkko, 2015; Roy et al., 2008).

Vulnerabilities: However, it should be noted that Diffie-Hellman is vulnerable to man-in-the-middle (MitM) attack where an attacker can intercept and replace the public keys exchanged between two parties (Wouters & Arkko, 2015; Roy et al., 2008).

Mitigation: To reduce these risks, one needs to apply authenticated variants of the Diffie-Hellman key exchange, for example, based on the Digital Signature Algorithm or the Public Key Infrastructure, which confirms the identities of the users and improves protection (Pan et al., 2022).

Security of XOR Encryption:

- **Limitations:** Although, XOR encryption is very easy for implementation and computationally very efficient, its basic structure of the cipher does not have potential for security. Recurring keys are susceptible to attacks such as the statistic or known plain one hence it's advisable not to use predictable keys (Erbsen, Philipoom, Gross, Sloan, & Chlipala, 2019; Roy et al., 2008).
- **Enhancements:** XOR encrypting can be improved by adding other superior methods of encryption, increase the length of the keys and its complexity, and make certain that keys are generated randomly and not repeated. It possible to get the required security if the stream ciphers or block ciphers are implemented (Erbsen et al., 2019; Roy et al., 2008).

Conclusion

This project implemented the Diffie-Hellman key exchange to securely generate a shared secret key between a server and a client over an insecure channel. The shared key was then used for encrypting and decrypting messages using a simple XOR operation. The server and client programs were developed in C++ and tested thoroughly to ensure secure and efficient communication.

Key learnings from this project include understanding the mathematical foundation and security implications of the Diffie-Hellman key exchange, as well as recognizing the limitations of XOR encryption. Practical experience was gained in handling socket programming, implementing robust error handling, and validating inputs to ensure secure cryptographic operations.

References

- Carts, D. A. (2001). A review of the Diffie-Hellman algorithm and its use in secure internet protocols. SANS institute, 751, 1-7.
- Erbsen, A., Philipoom, J., Gross, J., Sloan, R., & Chlipala, A. (2019). Security and efficiency trade-offs for elliptic curve Diffie–Hellman at the 128-bit and 224-bit security levels. *Journal of Cryptographic Engineering*. <https://doi.org/10.1007/s13389-021-00261-y>
- Gupta, C., & Reddy, N. S. (2022). Enhancement of Security of Diffie-Hellman Key Exchange Protocol using RSA Cryptography. In *Journal of Physics: Conference Series* (Vol. 2161, No. 1, p. 012014). IOP Publishing.
- Kara, M., Laouid, A., AlShaikh, M., Bounceur, A., & Hammoudeh, M. (2021). Secure key exchange against man-in-the-middle attack: Modified diffie-hellman protocol. *Jurnal Ilmiah Teknik Elektro Komputer dan Informatika*, 7(3), 380-387.
- Mitra, S., Das, S., & Kule, M. (2021). Prevention of the man-in-the-middle attack on Diffie–Hellman key exchange algorithm: A review. In *Proceedings of International Conference on Frontiers in Computing and Systems: COMSYS 2020* (pp. 625-635). Springer Singapore.
- Naresh, V. S., Reddi, S., & Murthy, N. V. (2020). Provable secure lightweight multiple shared key agreement based on hyper elliptic curve Diffie–Hellman for wireless sensor networks. *Information Security Journal: A Global Perspective*, 29(1), 1-13.
- Pan, J., Qian, C., & Ringerud, M. (2022). Signed (group) diffie–hellman key exchange with tight security. *Journal of Cryptology*, 35(4), 26.

- Roy, A., Datta, A., & Mitchell, J.C. (2008). Formal Proofs of Cryptographic Security of Diffie-Hellman-Based Protocols. In Barthe, G., & Fournet, C. (Eds.), *Trustworthy Global Computing. TGC 2007. Lecture Notes in Computer Science* (Vol. 4912, pp. 1-21). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-78663-4_21
- Wouters, P., & Arkko, J. (2015). Improving security in the Internet: The Diffie-Hellman case study. <https://www.ietf.org/blog/improving-security-internet-diffie-hellman-case-study/>
- Code used for re-engineering- <https://github.com/git-akshat/CNS-Lab/tree/master/13.%20Diffie-Hellman>