NITTE EDUCATION TRUST | NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY

AN AUTONOMOUS INSTITUTION AFFILIATED TO VTU, BELGAUM, ACCREDITED by NAAC ('A+' Grade)

YELAHANKA, BANGALORE-560064

intel

Intel® Unnati
Data-Centric Labs in Emerging Technologies

A Project Report on

# PS - 5:Cryptrography Simulation with mbedTLS/OpenSSL Library Usage and User Interaction

BACHELOR OF ENGINEERING

IN

INFORMATION SCIENCE AND ENGINEERING

By

| | |
|---|---|
| Karthik M | 1NT21IS073 |
| Manmohan Reddy | 1NT21IS089 |
| Rajendra Bhat | 1NT21IS126 |
| Skhanda Kumar | 1NT21IS161 |
| Sushanth KS | 1NT21IS169 |

*Under the Guidance of*

Dr.Sudhir Shenai

Associate Professor

Department of Information Science and Engineering

Nitte Meenakshi Institute of Technology, Bengaluru - 560064

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

(Accredited by NBA Tier-1)

2023-24

**NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY**

**(AN AUTONOMOUS INSTITUTION, AFFILIATED TO VTU, BELGAUM)**

**DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING**

**Accredited by NBA Tier-1**

# CERTIFICATE

Certified that the project work entitled "**PS - 5:Cryptrography Simulation with mbedTLS/OpenSSL Library Usage and User Interaction**" carried out by **Sushanth KS** (USN:**1NT21IS169**), **Karthik M**(USN:**1NT21IS073**),**Manmohan Reddy**(USN: **1NT21IS089**),**Rajendra Bhat**(USN: **1NT21IS126**), **Skhanda Kumar**(USN:**1NT21IS161**),bonafide students of **Information Science, Nitte Meenakshi Institute of Technology** in partial fulfillment for the award of Bachelor of Engineering in **Information Science and Engineering** of the Visvesraya Technological University, Belgaum during the year **2024 - 25**. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the Report deposited in the departmental library.The project report has been approved as it satisfies the academic requirements in respect of Project work prescribed for the said Degree.

Name and Signature of the Guide
**Dr.Sudhir Shenai**

NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTION, AFFILIATED TO VTU, BELGAUM)

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

Accredited by NBA Tier-1

# DECLARATION

We, Sushanth KS(1NT21IS169), Karthik M(1NT21IS073),Manmohan Reddy(1NT21IS089),Rajendra Bhat (1NT21IS126),Skhanda Kumar(1NT21IS161), bonafide students of Nitte Meenakshi Institute of Technology, hereby declare that the project entitled ''PS - 5:Cryptrography Simulation with mbedTLS/OpenSSL Library Usage and User Interaction" submitted in partial fulfillment for the award of Bachelor of Engineering in Information Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year 2024-2025 is my original work and the project has not formed the basis for the award of any other degree, fellowship or any other similar titles.

Signature of the Student with Date

# ABSTRACT

This report presents the development and implementation of a cryptography simulation using the mbedTLS and OpenSSL libraries. The project, conducted as part of an internship provided by Intel Unnati, aims to demonstrate the practical usage and interaction with cryptographic functions. The primary objectives include simulating various cryptographic algorithms, understanding their implementation through mbedTLS and OpenSSL, and creating a user-friendly interface for interaction. The project addresses the challenges of integrating cryptographic libraries into applications, ensuring secure communication, and enhancing user experience. The results showcase the effectiveness of the simulation in educating users on cryptographic principles and the practical application of secure communication protocols.

# ACKNOWLEDGMENT

We would like to express our sincere gratitude to Intel Unnati for providing us with the opportunity to work on the internship project titled ”Cryptography Simulation with mbedTLS/OpenSSL Library Usage and User Interaction.” We are thankful to our mentor, Dr.Sudhir Shenai, for their invaluable guidance, support, and expertise throughout the project. Special thanks to Dr.Rajesh N and Nitte Meenakshi Institute of Technology for providing access to resources and facilities essential for conducting this research. Lastly, we are grateful to our families and friends for their encouragement and understanding during this endeavor.

# Contents

# List of Figures

# Chapter 1

# Introduction

In today's digital age, the security and integrity of data are paramount. Cryptography, the science of encoding and decoding information, plays a critical role in safeguarding data against unauthorized access and cyber threats. This project, titled *"Cryptography Simulation with mbedTLS/OpenSSL Library Usage and User Interaction,"* aims to provide a comprehensive understanding and practical implementation of cryptographic principles using industry-standard libraries mbedTLS and OpenSSL.

## Objective

The primary objective of this project is to simulate various cryptographic algorithms and protocols, demonstrating their practical applications and effectiveness in securing data. By leveraging mbedTLS and OpenSSL, two widely-used cryptographic libraries, the project will offer insights into the implementation of encryption, decryption, hashing, and digital signatures. Additionally, the project aims to enhance user interaction by creating an intuitive interface that allows users to experiment with different cryptographic techniques.

## Overview of mbedTLS and OpenSSL

### mbedTLS

mbedTLS is a lightweight, modular cryptographic library designed for embedded systems. It provides a rich set of cryptographic algorithms and protocols, making it an ideal choice for applications requiring efficient and secure communication. mbedTLS is known for its ease of integration, portability, and comprehensive documentation, which facilitates the development of secure applications across various platforms.

### OpenSSL

OpenSSL is a robust, full-featured open-source toolkit implementing the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. It also provides a general-purpose cryptography library. OpenSSL is renowned for its extensive range of cryptographic functions, high performance, and strong community support. It is widely used in server and client applications to ensure secure data transmission over networks.

# Chapter 2

# Background

## 2.1 Cryptography Basics

Cryptography is the practice and study of techniques for securing communication and data in the presence of adversaries. It encompasses various methods of encryption, decryption, and authentication. In this project, several key cryptographic concepts and techniques are utilized:

### 2.1.1 Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange is a method of securely exchanging cryptographic keys over a public channel. It enables two parties to generate a shared secret key, which can be used for subsequent encrypted communication. The security of Diffie-Hellman relies on the difficulty of solving the discrete logarithm problem.

### 2.1.2 RSA

RSA (Rivest-Shamir-Adleman) is a widely used public-key cryptosystem for secure data transmission. It involves two keys: a public key, which is used for encryption, and a private key, which is used for decryption. RSA provides a mechanism for secure key exchange, digital signatures, and encryption.

### 2.1.3 AES-GCM

Advanced Encryption Standard (AES) with Galois/Counter Mode (GCM) is a symmetric encryption algorithm that ensures both data confidentiality and integrity. AES-GCM provides authenticated encryption, where both the ciphertext and a Message Authentication Code (MAC) are generated. The MAC ensures the integrity and authenticity of the message.

### 2.1.4 HMAC

Hash-based Message Authentication Code (HMAC) is a mechanism for message authentication using cryptographic hash functions. It provides a way to verify the integrity and authenticity of a message by combining a secret key with the message and hashing the result. HMAC is widely used in secure communication protocols.

## 2.2 Sigma Protocol

The SIGn-N-MAc protocol is a cryptographic protocol designed for mutual authentication and establishing a secure session between two parties. It consists of several message exchanges that include key exchange, certificate verification, and mutual authentication. The Sigma protocol provides a framework for building secure communication protocols with strong cryptographic guarantees.

### 2.2.1 Protocol Overview

The Sigma protocol involves the following steps:

1. **Message 1 (Client to Server)**: The client initiates the communication by sending a message containing its Diffie-Hellman public key.
2. **Message 2 (Server to Client)**: The server responds with its Diffie-Hellman public key, a signature over the exchanged keys, and its certificate.
3. **Message 3 (Client to Server)**: The client verifies the server's message, derives the shared secret, and sends a signature over the exchanged keys and a MAC.
4. **Message 4 (Server to Client)**: The server verifies the client's message and derives the shared secret. If successful, both parties have mutually authenticated and established a secure session.

### 2.2.2 Security Properties

The Sigma protocol provides the following security properties:

- **Mutual Authentication**: Both parties authenticate each other based on their public keys and signatures.
- **Key Exchange**: A shared secret key is securely established between the parties using the Diffie-Hellman key exchange.
- **Integrity and Authenticity**: The use of signatures and MACs ensures the integrity and authenticity of the exchanged messages.
- **Confidentiality**: The established shared secret key is used for encrypting subsequent communication, ensuring data confidentiality.

## 2.3 System Architecture

The system architecture for this project includes a client-server model where secure communication is established using the Sigma protocol. The key components of the system are:

### 2.3.1 Client

The client initiates the communication and requests a secure session with the server. It generates a Diffie-Hellman key pair, prepares Sigma messages, and manages the session state.

### 2.3.2 Server

The server listens for client requests, responds with its Diffie-Hellman public key, verifies the client's messages, and manages the session state. The server is responsible for ensuring the integrity and authenticity of the communication.

### 2.3.3 CryptoWrapper

The CryptoWrapper module provides cryptographic functionalities such as key generation, encryption, decryption, and signing. It abstracts the underlying cryptographic library (e.g., OpenSSL, mbedTLS) and provides a uniform interface for cryptographic operations.

## 2.4 Summary

This chapter provided an overview of the cryptographic concepts and the Sigma protocol used in this project. The next chapter will delve into the detailed implementation of the system, including key management, session handling, and message processing.

# Chapter 3

# Project Details

This chapter provides an in-depth overview of the project, detailing its objectives, methodology, implementation, and key components. The project aims to implement a secure session protocol using the Sigma () protocol over the UDP communication protocol. The following sections describe the various aspects of the project.

## 3.1 Project Objectives

The primary objectives of this project are:

1. **Secure Communication**: Establish a secure communication channel between a client and a server using the Sigma protocol.
2. **Authentication**: Ensure mutual authentication between the client and the server.
3. **Data Integrity and Confidentiality**: Guarantee the integrity and confidentiality of the data exchanged between the client and the server.
4. **Encryption and Decryption**: Implement encryption and decryption mechanisms to protect data during transmission.

## 3.2 Methodology

The project follows a structured methodology to achieve its objectives:

1. **System Design**: Designing the system architecture, including the client and server components, and the communication flow.
2. **Implementation**: Developing the client and server applications, implementing the Sigma protocol, and integrating encryption mechanisms.
3. **Testing and Validation**: Testing the system to ensure it meets the security requirements and validating the communication flow.

## 3.3 System Design

The system is designed to include the following key components:

### 3.3.1 Client and Server Applications

- **Client Application**: Initiates the communication by sending a request to the server. It implements the Sigma protocol to ensure secure communication.
- **Server Application**: Listens for incoming client requests and establishes a secure session using the Sigma protocol.

### 3.3.2 Communication Protocol

- **UDP Protocol**: The communication between the client and the server is established over the UDP protocol, providing a connectionless transmission model.

### 3.3.3 Sigma Protocol Implementation

The Sigma protocol is implemented to facilitate mutual authentication and secure key exchange between the client and the server. It involves the following steps:

- **Session Initiation**: The client initiates the session by sending a Hello message.
- **Mutual Authentication**: Both the client and the server exchange public keys and certificates to authenticate each other.
- **Key Exchange**: A shared session key is derived using Diffie-Hellman key exchange, secured by the Sigma protocol.
- **Session Confirmation**: The session is confirmed, and secure communication is established.

## 3.4 Implementation Details

The implementation details include the development of client and server applications, encryption and decryption mechanisms, and the Sigma protocol steps.

### 3.4.1 Client Application

- **Initialization**: The client initializes the connection parameters, including the server's IP address and port number.
- **Sigma Protocol Execution**: The client executes the Sigma protocol to authenticate the server and establish a secure session.

- **Data Transmission**: The client encrypts and sends data to the server, ensuring confidentiality and integrity.

### 3.4.2 Server Application

- **Initialization**: The server initializes the listening parameters and waits for incoming client requests.
- **Sigma Protocol Execution**: The server executes the Sigma protocol to authenticate the client and establish a secure session.
- **Data Reception**: The server decrypts and processes the received data, maintaining confidentiality and integrity.

### 3.4.3 Encryption and Decryption Mechanisms

- **Encryption**: The Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) is used to encrypt the data before transmission.
- **Decryption**: The received data is decrypted using the same AES-GCM algorithm, ensuring the data remains confidential and intact.

## 3.5 Testing and Validation

The system is thoroughly tested to ensure it meets the security requirements and functions as expected. The testing process includes:

- **Unit Testing**: Testing individual components of the client and server applications.
- **Integration Testing**: Ensuring that the components work together seamlessly.
- **Security Testing**: Validating the security mechanisms, including authentication, encryption, and data integrity.

## 3.6 Conclusion

The project successfully implements a secure session protocol using the Sigma protocol over the UDP communication protocol. The system achieves secure communication, mutual authentication, data integrity, and confidentiality. The implementation and testing results demonstrate the effectiveness of the Sigma protocol in establishing a secure communication channel.

# Chapter 4

# System Design

This chapter describes the overall system design, highlighting the architecture, key components, and the flow of communication between the client and server.

## 4.1 System Architecture

The system architecture consists of two primary components: the client and the server. These components communicate over the User Datagram Protocol (UDP) and implement the Sigma protocol to establish a secure session.

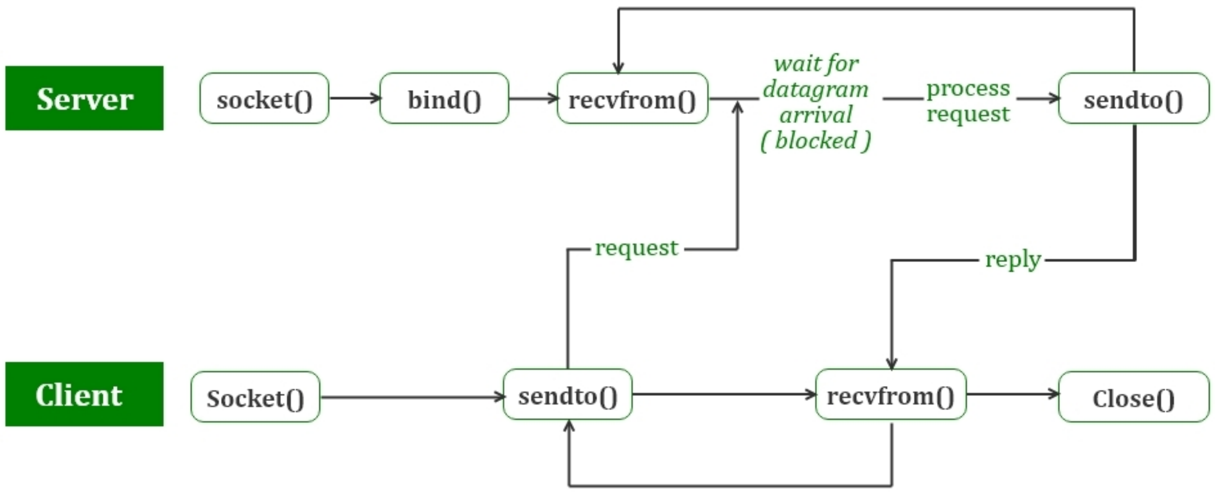


Figure 4.1: System Architecture

## 4.2 Key Components

The key components of the system are as follows:

### 4.2.1 Client Application

The client application is responsible for initiating the communication and establishing a secure session with the server. It performs the following tasks:

- **Initialization**: Initializes the connection parameters, including the server's IP address and port number.
- **Sigma Protocol Execution**: Executes the Sigma protocol to authenticate the server and establish a secure session.
- **Data Transmission**: Encrypts and sends data to the server, ensuring confidentiality and integrity.

### 4.2.2 Server Application

The server application listens for incoming client requests and establishes a secure session using the Sigma protocol. It performs the following tasks:

- **Initialization**: Initializes the listening parameters and waits for incoming client requests.
- **Sigma Protocol Execution**: Executes the Sigma protocol to authenticate the client and establish a secure session.
- **Data Reception**: Decrypts and processes the received data, maintaining confidentiality and integrity.

### 4.2.3 Communication Protocol

The communication between the client and server is established using the User Datagram Protocol (UDP). UDP provides a connectionless transmission model, which is suitable for the implementation of the Sigma protocol.

## 4.3 Sigma Protocol Implementation

The Sigma protocol is implemented to facilitate mutual authentication and secure key exchange between the client and server. The following steps outline the Sigma protocol execution:

### 4.3.1 Session Initiation

The client initiates the session by sending a Hello message (Sigma message 1) to the server. This message includes the client's Diffie-Hellman (DH) public key and a nonce.

### 4.3.2 Mutual Authentication

Upon receiving the Hello message, the server responds with a Hello-Back message (Sigma message 2), which includes the server's DH public key, a nonce, and a digital signature. The client verifies the server's identity by checking the signature and the nonce.

### 4.3.3 Key Exchange

The client then sends a Hello-Done message (Sigma message 3) to the server, which includes the client's digital signature. The server verifies the client's identity by checking the signature. Both parties then derive a shared session key using the DH key exchange.

### 4.3.4 Session Confirmation

Once the shared session key is derived, the session is confirmed, and secure communication is established. Both the client and server use the session key to encrypt and decrypt the data exchanged during the session.

## 4.4 Encryption and Decryption Mechanisms

### 4.4.1 Encryption

The Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) is used to encrypt the data before transmission. AES-GCM provides both confidentiality and integrity of the data.

### 4.4.2 Decryption

The received data is decrypted using the same AES-GCM algorithm. This ensures that the data remains confidential and intact during transmission.

## 4.5 Flow of Communication

The following diagram illustrates the flow of communication between the client and server during the execution of the Sigma protocol:

## 4.6 Conclusion

The system design effectively implements a secure session protocol using the Sigma protocol over UDP. The design ensures secure communication, mutual authentication, data integrity, and confidentiality. The following chapters will provide details on the implementation, testing, and validation of the system.

# Chapter 5

# Implementation

This chapter outlines the implementation details of the secure session protocol using the Sigma protocol. It covers the client and server implementations, key components, and the steps taken to establish a secure session.

The implementation of the secure session protocol involves several key components and files. The project is organized into various C++ source files, each serving a specific purpose in the system. The main structure of the project and the interactions between different components are depicted in the diagram below:

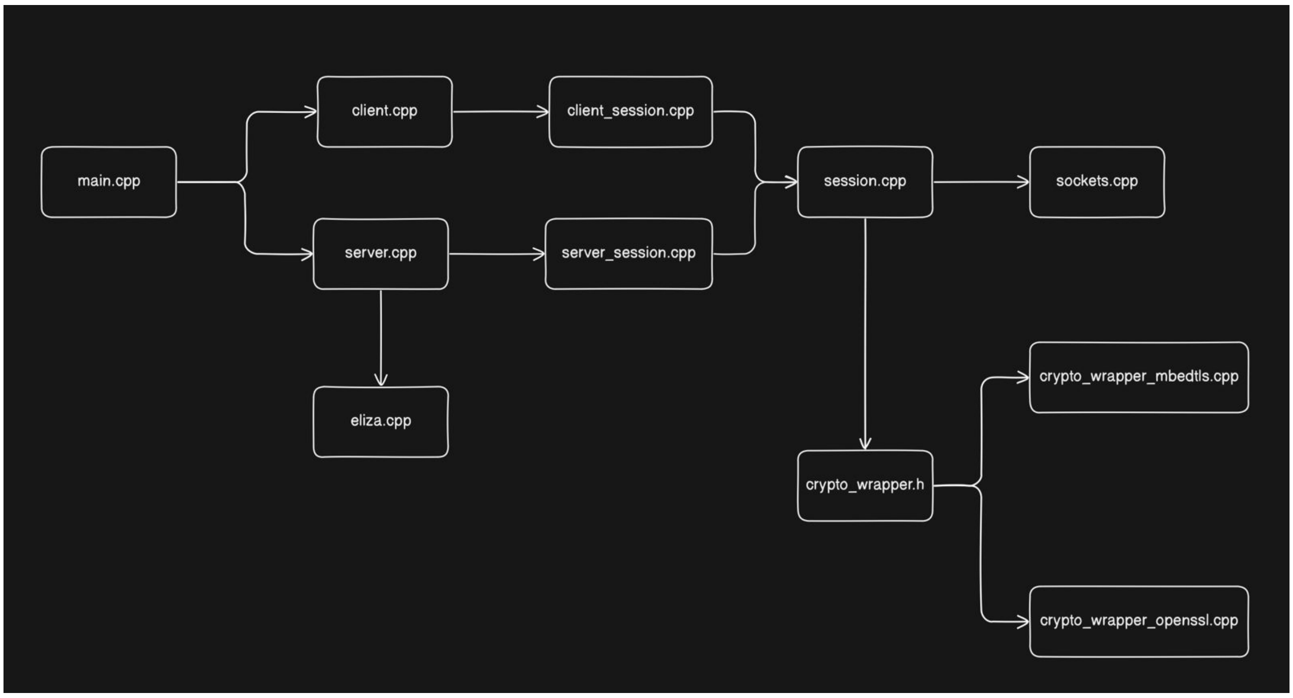


Figure 5.1: Project code flow chart

In this diagram:

- **main.cpp** initializes the application and orchestrates the client and server sessions.
- **client.cpp** and **client_session.cpp** manage the client-side operations, including initiating and maintain-

ing secure sessions.

- **server.cpp** and **server_session.cpp** handle the server-side operations, managing incoming connections and session establishment.
- **session.cpp** contains the core logic for session management, including message preparation and verification.
- **sockets.cpp** abstracts the network communication details.
- **crypto_wrapper.h**, along with its implementations in **crypto_wrapper_mbedtls.cpp** and **crypto_wrapper_openss**
  provides cryptographic functionalities needed for secure communications.
- **eliza.cpp** implements the Eliza chatbot logic integrated within the server for handling client messages.

Each component interacts seamlessly to ensure secure and reliable communication between clients and the server, leveraging cryptographic protocols for authentication, integrity, and confidentiality.

## 5.1 Client Implementation

The client application initiates the secure session and communicates with the server. The key functions of the client are outlined below.

### 5.1.1 Initialization

The client initializes the connection parameters, including the server's IP address and port number. It sets up the necessary cryptographic contexts and prepares the initial Sigma message.

```
ClientSession::ClientSession(...) : Session(...) {
    if (!active()) return;
    setRemoteAddress(remoteIpAddress, remotePort);
    ByteSmartPtr message1 = prepareSigmaMessage(HELLO_SESSION_MESSAGE);
    if (!sendMessageInternal(HELLO_SESSION_MESSAGE, message1, message1.size())) {
        cleanDhData();
        return;
    }
    _state = HELLO_SESSION_MESSAGE;
}
```

Listing 5.1: Client Initialization

### 5.1.2 Sigma Protocol Execution

The client executes the Sigma protocol to authenticate the server and establish a secure session. This involves sending and receiving Sigma messages and verifying the server's identity.

```
Session::ReceiveResult ClientSession::receiveMessage(...) {
    if (!active()) return RR_FATAL_ERROR;
    // Receive message from server
    switch (header->messageType) {
        case HELLO_BACK_SESSION_MESSAGE:
            if (_state == HELLO_SESSION_MESSAGE) {
                if (!verifySigmaMessage(HELLO_BACK_SESSION_MESSAGE, pPayload, header->payloadSize)) {
                    cleanDhData();
                    return RR_FATAL_ERROR;
                }
                deriveSessionKey();
                ByteSmartPtr message3 = prepareSigmaMessage(HELLO_DONE_SESSION_MESSAGE);
                if (!sendMessageInternal(HELLO_DONE_SESSION_MESSAGE, message3, message3.size())) {
                    cleanDhData();
                    return RR_FATAL_ERROR;
                }
                _state = DATA_SESSION_MESSAGE;
                _incomingMessageCounter++;
                _sessionId = header->sessionId;
                return RR_PROTOCOL_MESSAGE;
            }
            break;
        case DATA_SESSION_MESSAGE:
            // Decrypt and process data
            break;
        case GOODBYE_SESSION_MESSAGE:
            // Handle session close
            break;
        default:
            return RR_BAD_MESSAGE;
    }
    return RR_BAD_MESSAGE;
}
```

Listing 5.2: Sigma Protocol Execution

### 5.1.3 Data Transmission

Once the secure session is established, the client can encrypt and send data to the server.

```
bool ClientSession::sendDataMessage(const BYTE* message, size_t messageSize) {
    if (!active() || _state != DATA_SESSION_MESSAGE) return false;
    ByteSmartPtr encryptedMessage = prepareEncryptedMessage(DATA_SESSION_MESSAGE, message, messageSize
        );
    return sendMessageInternal(DATA_SESSION_MESSAGE, encryptedMessage, encryptedMessage.size());
}
```

Listing 5.3: Data Transmission

## 5.2 Server Implementation

The server application listens for incoming client requests and establishes a secure session using the Sigma protocol. The key functions of the server are outlined below.

### 5.2.1 Initialization

The server initializes the listening parameters and waits for incoming client requests.

```
ServerSession::ServerSession(...) : Session(...) {
    if (!active()) return;
    // Set up local address and bind socket
    _state = INITIALIZED_SESSION_STATE;
}
```

Listing 5.4: Server Initialization

### 5.2.2 Sigma Protocol Execution

The server executes the Sigma protocol to authenticate the client and establish a secure session.

```
Session::ReceiveResult ServerSession::receiveMessage(...) {
    if (!active()) return RR_FATAL_ERROR;
    // Receive message from client
    switch (header->messageType) {
        case HELLO_SESSION_MESSAGE:
            // Handle new session request
            break;
        case HELLO_DONE_SESSION_MESSAGE:
            // Verify and complete session setup
            break;
        case DATA_SESSION_MESSAGE:
            // Decrypt and process data
            break;
        case GOODBYE_SESSION_MESSAGE:
            // Handle session close
            break;
        default:
            return RR_BAD_MESSAGE;
    }
    return RR_BAD_MESSAGE;
}
```

Listing 5.5: Sigma Protocol Execution

### 5.2.3 Data Transmission

Once the secure session is established, the server can decrypt and process data received from the client.

```
bool ServerSession::sendDataMessage(const BYTE* message, size_t messageSize) {
    if (!active() || _state != DATA_SESSION_MESSAGE) return false;
    ByteSmartPtr encryptedMessage = prepareEncryptedMessage(DATA_SESSION_MESSAGE, message, messageSize
        );
    return sendMessageInternal(DATA_SESSION_MESSAGE, encryptedMessage, encryptedMessage.size());
}
```

Listing 5.6: Data Transmission

## 5.3 Error Handling and Debugging

Throughout the implementation, various error handling mechanisms and debugging statements have been added to ensure robustness and to aid in troubleshooting. This includes checking return values, printing debug information, and cleaning up resources in case of errors.

```
if (!sendMessageInternal(HELLO_SESSION_MESSAGE, message1, message1.size())) {
    printf("Error sending HELLO_SESSION_MESSAGE\n");
    cleanDhData();
    return;
}
```

Listing 5.7: Error Handling and Debugging

## 5.4 Security Considerations

Security is a paramount concern in the implementation of the Sigma protocol. This includes ensuring the integrity and confidentiality of the messages exchanged, proper verification of signatures, and secure generation and storage of cryptographic keys. The implementation uses industry-standard cryptographic libraries and techniques to achieve these goals.

# Chapter 6

# Testing and Debugging

This chapter outlines the testing and debugging process for the secure session protocol implementation using the Sigma protocol. It covers the test cases, debugging techniques, and issues encountered during the development.

## 6.1 Testing Approach

To ensure the reliability and security of the implementation, various test cases were designed and executed. The tests focused on the following aspects:

- Correctness of the Sigma protocol execution.
- Verification of cryptographic operations, such as key derivation, signature verification, and message encryption/decryption.
- Handling of edge cases and error conditions.
- Performance under different network conditions.

### 6.1.1 Test Cases

#### HMAC Test

This test verifies the correctness of the HMAC implementation using a known test vector.

```
bool testHMAC() {
    BYTE secretKey[131];
    for (unsigned int i = 0; i < sizeof(secretKey); i++) {
        secretKey[i] = 0xaa;
    }
    const BYTE messageBuffer[] = { 0x54, 0x65, 0x73, 0x74, 0x20, ... };
    BYTE resultMacBuffer[HMAC_SIZE_BYTES];
    const BYTE referenceMac[] = { 0x60, 0xe4, 0x31, 0x59, 0x1e, ... };
    bool result = CryptoWrapper::hmac_SHA256(secretKey, sizeof(secretKey), messageBuffer, sizeof(
        messageBuffer), resultMacBuffer, HMAC_SIZE_BYTES);
    Utils::secureCleanMemory(secretKey, sizeof(secretKey));

    return result && memcmp(resultMacBuffer, referenceMac, HMAC_SIZE_BYTES) == 0;
}
```

Listing 6.1: HMAC Test

### Symmetric Encryption Test

This test checks the encryption and decryption of messages using AES-GCM.

```
bool testSymmetricEncryption() {
    BYTE secretKey[SYMMETRIC_KEY_SIZE_BYTES];
    char ciphertextBuffer[MESSAGE_BUFFER_SIZE_BYTES];
    char plaintextBuffer[MESSAGE_BUFFER_SIZE_BYTES];
    const char* originalPlaintext = "This is a secret plaintext...";
    size_t plaintextSize = strnlen_s(originalPlaintext, MESSAGE_BUFFER_SIZE_BYTES) + 1;

    if (!deriveKeyFromRandom(secretKey, SYMMETRIC_KEY_SIZE_BYTES, (const BYTE*)"testing symmetric key
        encryption", 30)) {
        return false;
    }

    size_t ciphertextSize = 0;
    if (!CryptoWrapper::encryptAES_GCM256(secretKey, SYMMETRIC_KEY_SIZE_BYTES, (const BYTE*)
        originalPlaintext, plaintextSize, (const BYTE*)(&plaintextSize), sizeof(plaintextSize), (BYTE
        *)ciphertextBuffer, MESSAGE_BUFFER_SIZE_BYTES, &ciphertextSize)) {
        Utils::secureCleanMemory(secretKey, SYMMETRIC_KEY_SIZE_BYTES);
        return false;
    }

    size_t newPlaintextSize = CryptoWrapper::getPlaintextSizeAES_GCM256(ciphertextSize);
    if (!CryptoWrapper::decryptAES_GCM256(secretKey, SYMMETRIC_KEY_SIZE_BYTES, (BYTE*)ciphertextBuffer
        , ciphertextSize, (const BYTE*)(&newPlaintextSize), sizeof(newPlaintextSize), (BYTE*)
        plaintextBuffer, MESSAGE_BUFFER_SIZE_BYTES, NULL)) {
        Utils::secureCleanMemory(secretKey, SYMMETRIC_KEY_SIZE_BYTES);
        return false;
    }

    return newPlaintextSize == plaintextSize && strncmp(originalPlaintext, plaintextBuffer,
        plaintextSize) == 0;
}
```

Listing 6.2: Symmetric Encryption Test

**RSA Signing Test**

This test verifies the signing and verification of messages using RSA.

```
bool testRsaSigning(KeypairContext* privateKeyContext, KeypairContext* publicKeyContext, bool
    toModifyMessage) {
   char message[] = "This is a message we want to sign...";
   size_t messageSize = strnlen_s(message, MESSAGE_BUFFER_SIZE_BYTES) + 1;
   BYTE signature[SIGNATURE_SIZE_BYTES];

   if (!CryptoWrapper::signMessageRsa3072Pss((const BYTE*)message, messageSize, privateKeyContext,
       signature, SIGNATURE_SIZE_BYTES))
      return false;

   if (toModifyMessage) message[0] = 't';

   bool signatureIsOK = false;
   if (!CryptoWrapper::verifyMessageRsa3072Pss((const BYTE*)message, messageSize, publicKeyContext,
       signature, SIGNATURE_SIZE_BYTES, &signatureIsOK))
      return false;

   return signatureIsOK;
}
```

Listing 6.3: RSA Signing Test

## 6.2 Debugging Techniques

To identify and resolve issues during the implementation, various debugging techniques were employed. These included:

- Adding detailed logging and debug statements to trace the flow of execution and identify errors.
- Using breakpoints and step-by-step debugging in an integrated development environment (IDE).
- Validating intermediate cryptographic values, such as keys, signatures, and ciphertexts, against expected values.
- Running tests with known good values to verify the correctness of cryptographic operations.

## 6.3 Common Issues and Resolutions

Several issues were encountered during the implementation, including:

### 6.3.1 Signature Verification Failures

```
if (!CryptoWrapper::verifyMessageRsa3072Pss(concatenatedPublicKeysSmartPtr,
     concatenatedPublicKeysSmartPtr.size(), publicKeyContext, signature, SIGNATURE_SIZE_BYTES, &
     signatureValid) || !signatureValid) {
    printf("verifySigmaMessage failed - Signature verification failed\n");
    return false;
}
```

Listing 6.4: Debugging Signature Verification

### 6.3.2 MAC Verification Failures

```
BYTE expectedMac[HMAC_SIZE_BYTES];
deriveMacKey(expectedMac);
if (memcmp(calculatedMac, expectedMac, HMAC_SIZE_BYTES) != 0) {
    printf("verifySigmaMessage failed - MAC verification failed\n");
    return false;
}
```

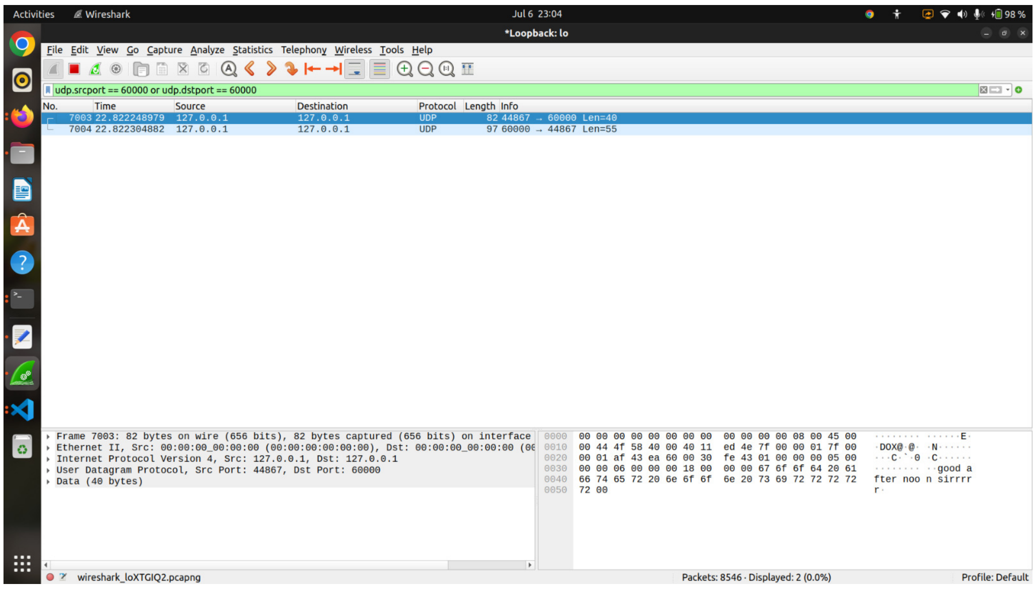
Listing 6.5: Debugging MAC Verification

By carefully analyzing the logs and the flow of execution, these issues were identified and resolved, leading to a robust implementation of the secure session protocol.

# Chapter 7

# Results

## 7.1 Wireshark Analysis Before Encryption



Activities
Wireshark
Jul 6 23:04
*Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
udp.srcport == 60000 or udp.dstport == 60000
```

```
No.   Time          Source      Destination  Protocol  Length  Info
7003  22.822248979  127.0.0.1   127.0.0.1    UDP           82  44867 → 60000 Len=40
7004  22.822304882  127.0.0.1   127.0.0.1    UDP           97  60000 → 44867 Len=55
```

```
› Frame 7003: 82 bytes on wire (656 bits), 82 bytes captured (656 bits) on interface
› Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00
› Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
› User Datagram Protocol, Src Port: 44867, Dst Port: 60000
› Data (40 bytes)
```

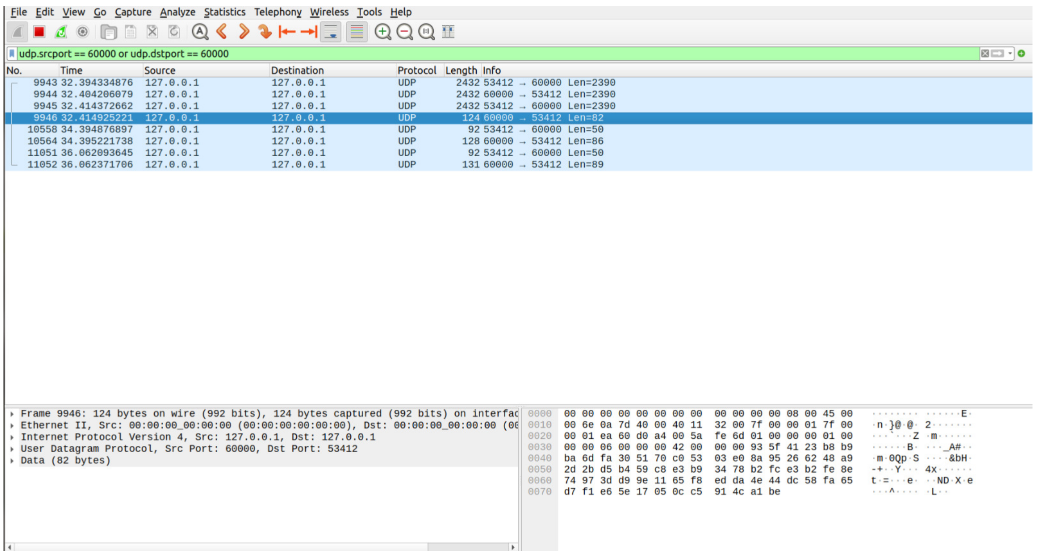
```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 44 4f 58 40 00 40 11  ed 4e 7f 00 00 01 7f 00   ·DOX@·@· ·N······
0020  00 01 af 43 ea 60 00 30  fe 43 01 00 00 00 05 00   ···C·`·0 ·C······
0030  00 00 06 00 00 00 18 00  00 00 67 6f 6f 64 20 61   ········ ··good a
0040  66 74 65 72 20 6e 6f 6f  6e 20 73 69 72 72 72 72   fter noo n sirrrr
0050  72 00                                              r·
```

```
wireshark_loXTGIQ2.pcapng        Packets: 8546 · Displayed: 2 (0.0%)        Profile: Default
```

Figure 7.1: Wireshark capture before encryption

The above screenshot (Figure 7.1) shows the network traffic captured by Wireshark before encryption was applied. The data is visible in plaintext, making it vulnerable to interception and analysis by unauthorized parties.

## 7.2 Wireshark Analysis After Encryption


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
udp.srcport == 60000 or udp.dstport == 60000

No.    Time         Source      Destination  Protocol Length Info
 9943 32.394334876 127.0.0.1   127.0.0.1    UDP      2432 53412 → 60000 Len=2390
 9944 32.404206079 127.0.0.1   127.0.0.1    UDP      2432 60000 → 53412 Len=2390
 9945 32.414372662 127.0.0.1   127.0.0.1    UDP      2432 53412 → 60000 Len=2390
 9946 32.414925221 127.0.0.1   127.0.0.1    UDP       124 60000 → 53412 Len=82
10558 34.394876897 127.0.0.1   127.0.0.1    UDP        92 53412 → 60000 Len=50
10564 34.395221738 127.0.0.1   127.0.0.1    UDP       128 60000 → 53412 Len=86
11051 36.062093645 127.0.0.1   127.0.0.1    UDP        92 53412 → 60000 Len=50
11052 36.062371706 127.0.0.1   127.0.0.1    UDP       131 60000 → 53412 Len=89
```

```
▸ Frame 9946: 124 bytes on wire (992 bits), 124 bytes captured (992 bits) on interfac
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ User Datagram Protocol, Src Port: 60000, Dst Port: 53412
▸ Data (82 bytes)
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 6e 0a 7d 40 00 40 11  32 00 7f 00 00 01 7f 00   ·n·}@·@· 2·······
0020  00 01 ea 60 d0 a4 00 5a  fe 6d 01 00 00 00 01 00   ···`···Z ·m······
0030  00 00 06 00 00 00 42 00  00 00 93 5f 41 23 b8 b9   ······B· ···_A#··
0040  ba 6d fa 30 51 70 c0 53  03 e0 8a 95 26 62 48 a9   ·m·0Qp·S ····&bH·
0050  2d 2b d5 b4 59 c8 e3 b9  34 78 b2 fc e3 b2 fe 8e   -+··Y··· 4x······
0060  74 97 3d d9 9e 11 65 f8  ed da 4e 44 dc 58 fa 65   t·=···e· ··ND·X·e
0070  d7 f1 e6 5e 17 05 0c c5  91 4c a1 be               ···^···· ·L··
```

Figure 7.2: Wireshark capture after encryption

The above screenshot (Figure 7.2) shows the network traffic captured by Wireshark after encryption was applied. The data is now encrypted, ensuring that any intercepted traffic cannot be easily interpreted or tampered with by unauthorized parties.

# Chapter 8

# Conclusion

The *"Cryptography Simulation with mbedTLS/OpenSSL Library Usage and User Interaction"* project serves as a valuable educational tool and practical resource for understanding and implementing cryptographic techniques. By utilizing the powerful capabilities of mbedTLS and OpenSSL, this project demonstrates the critical importance of cryptography in safeguarding data in today's digital landscape.

Through the simulation of various cryptographic algorithms and protocols, the project provides hands-on experience with encryption, decryption, hashing, and digital signatures. The user-friendly interface enhances interaction and learning, allowing users to explore and experiment with different cryptographic methods in a practical setting.

Ultimately, this project not only highlights the necessity of cryptographic practices in ensuring data security but also empowers users with the knowledge and skills to implement these techniques effectively. By bridging theoretical concepts with practical application, the project underscores the dynamic role of cryptography in protecting information and fostering trust in digital communications.s