# ABSTRACT

In the modern digital era, securing communication protocols is critical to protecting sensitive data and ensuring privacy. This project focuses on the implementation and simulation of cryptographic techniques using widely recognized libraries such as mbedTLS and OpenSSL. The primary objective is to create a robust cryptographic framework that facilitates secure communication between two entities, integrating user interaction for real-world applicability.

The project involves securing a custom communication protocol between a patient and an AI psychologist named ELIZA. This is accomplished in two main phases: creating digital certificates and implementing a secure communication protocol using these certificates. The project leverages mbedTLS or OpenSSL libraries to perform cryptographic operations such as encryption, decryption, signing, and verification.

**Phase 1: Creating Digital Certificates:**

1. **Root Certificate Creation:** Generate a self-signed root certificate (rootCA.crt) with an RSA key size of 3072 bits and SHA384 hashing algorithm. Assign a serial number of 01.
2. **User Certificates:** Generate RSA key pairs for two users, Alice and Bob, with a key size of 3072 bits and SHA384 hashing. Sign Alice's certificate with the rootCA, assigning a serial number of 02, and Bob's certificate with a serial number of 03.

**Phase 2: Securing the Custom Protocol:**

1. **Basic Phase - Crypto Wrapper Implementation:**
   * Develop a basic crypto wrapper using the mbedTLS or OpenSSL library to handle cryptographic operations.
   * Implement unit tests to verify the encryption, decryption, signing, and verification functions of the crypto wrapper, ensuring all tests pass successfully.
2. **Advanced Phase - Protocol Security:**
   * Secure the communication protocol between the patient and ELIZA using the implemented crypto wrapper.
   * Ensure that all messages exchanged are encrypted and signed using the respective certificates and keys to guarantee confidentiality and authenticity.

# TABLE OF CONTENT

|  |  |
| --- | --- |
| **SL NO.** | **CONTENTS** |
| 1 | CHAPTER-1  INTRODUCTION. |
| 2 | CHAPTER-2  OBJECTIVES. |
| 3 | CHAPTER-3  IMPLEMENTATION. |
| 4 | CHAPTER-4  LANGUAGE SPECIFICATION. |
| 5 | CHAPTER-5  ADVANTAGES |
| 6 | CHAPTER-6  REFERENCE. |
| 7 | CHAPTER-7  FUTURE SCOPE. |
| 8 | CHAPTER-8  OUTPUT. |
| 9 | CHAPTER-9  CONCLUSION |

**CHAPTER 1**

# INTRODUCTION

In today's digital age, the security of communication channels is paramount. With the increasing frequency of cyber-attacks and data breaches, ensuring that sensitive information remains confidential and tamper-proof has become a critical requirement. Cryptographic techniques provide the necessary tools to achieve this, enabling secure communication by encrypting data and verifying the identities of communicating parties.

This project aims to explore and implement cryptographic solutions using well-known libraries, mbedTLS and OpenSSL, to secure a custom communication protocol between a patient and an AI psychologist, named ELIZA. The objective is to simulate a real-world scenario where secure communication is essential, demonstrating the practical application of cryptographic principles.

**CHAPTER 2**

**OBJECTIVES**

**Generate Digital Certificates:**

* Create a self-signed root certificate.
* Generate and sign user certificates for Alice and Bob using the root certificate.

**Develop a Crypto Wrapper:**

* Implement a basic cryptographic wrapper using mbedTLS or OpenSSL.
* Perform encryption, decryption, signing, and verification operations.

**Secure the Communication Protocol:**

* Implement a secure communication protocol between the patient and ELIZA.
* Ensure all messages are encrypted and signed to guarantee confidentiality and authenticity.

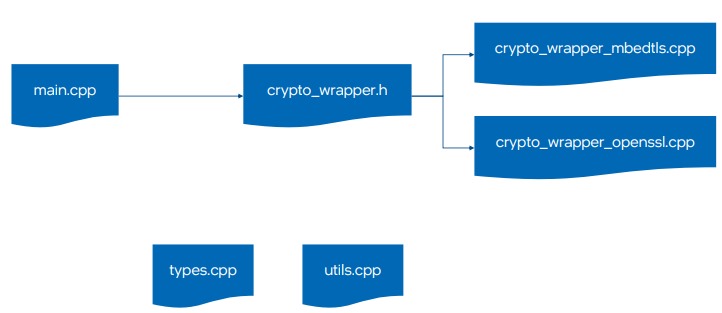
**CHAPTER 3**

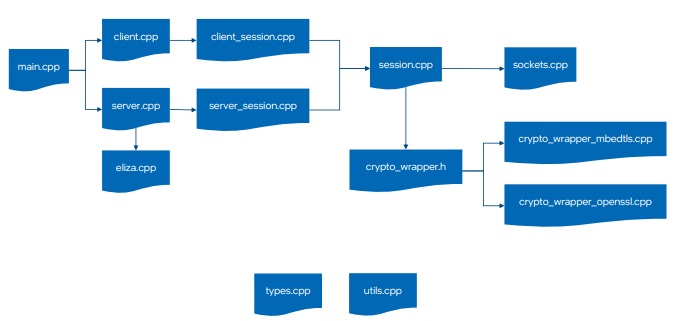
**IMPLEMENTATION**

#### **Tools and Libraries**

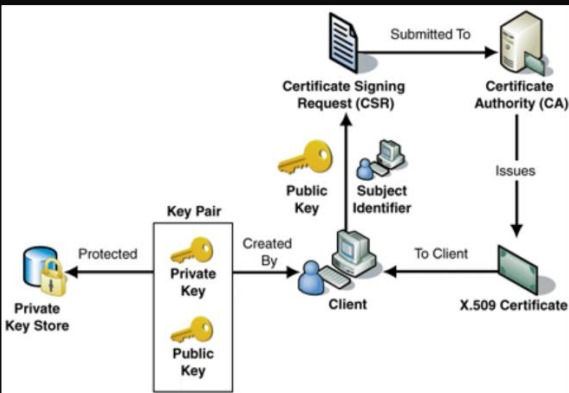
* **mbedTLS or OpenSSL**: For cryptographic operations.
* **GCC or a C compiler**: For compiling C code.
* **Python**: If you choose to implement parts of the project in Python.

**Step-by-Step Implementation**





**Part 1: Creating Digital Certificates**



**Step 1: Generate the Root Certificate:**

1. **Generate the RSA Key for the Root Certificate**

openssl genpkey -algorithm RSA -out rootCA.key -pkeyopt rsa\_keygen\_bits:3072

1. **Create the Self-Signed Root Certificate**

openssl req -x509 -new -nodes -key rootCA.key -sha384 -days 3650 -out rootCA.crt -subj "/C=US/ST=State/L=City/O=Organization/OU=Unit/CN=RootCA" -set\_serial 01

**Step 2: Generate Certificates for Alice and Bob:**

1. **Generate RSA Key Pair for Alice**

openssl genpkey -algorithm RSA -out alice.key -pkeyopt rsa\_keygen\_bits:3072

1. **Create Certificate Signing Request (CSR) for Alice**

openssl req -new -key alice.key -out alice.csr -subj "/C=US/ST=State/L=City/O=Organization/OU=Unit/CN=Alice"

1. **Sign Alice’s Certificate with rootCA**

openssl x509 -req -in alice.csr -CA rootCA.crt -CAkey rootCA.key -CAcreateserial -out alice.crt -days 3650 -sha384 -set\_serial 02

1. **Generate RSA Key Pair for Bob**

openssl genpkey -algorithm RSA -out bob.key -pkeyopt rsa\_keygen\_bits:3072

1. **Create Certificate Signing Request (CSR) for Bob**

openssl req -new -key bob.key -out bob.csr -subj "/C=US/ST=State/L=City/O=Organization/OU=Unit/CN=Bob"

1. **Sign Bob’s Certificate with rootCA**

openssl x509 -req -in bob.csr -CA rootCA

**Part 2: Securing the Custom Protocol**

**Step 1: Implement the Crypto Wrapper**

**Example in C using OpenSSL:**

1. **crypto\_wrapper.c**

#include <openssl/evp.h>

#include <openssl/pem.h>

#include <openssl/rsa.h>

#include <openssl/x509.h>

#include <openssl/err.h>

#include <stdio.h>

#include <string.h>

int encrypt(unsigned char \*plaintext, int plaintext\_len, unsigned char \*key, unsigned char \*iv, unsigned char \*ciphertext) {

EVP\_CIPHER\_CTX \*ctx;

int len;

int ciphertext\_len;

if(!(ctx = EVP\_CIPHER\_CTX\_new())) return -1;

if(1 != EVP\_EncryptInit\_ex(ctx, EVP\_aes\_256\_cbc(), NULL, key, iv)) return -1;

if(1 != EVP\_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext\_len)) return -1;

ciphertext\_len = len;

if(1 != EVP\_EncryptFinal\_ex(ctx, ciphertext + len, &len)) return -1;

ciphertext\_len += len;

EVP\_CIPHER\_CTX\_free(ctx);

return ciphertext\_len;

}

int decrypt(unsigned char \*ciphertext, int ciphertext\_len, unsigned char \*key, unsigned char \*iv, unsigned char \*plaintext) {

EVP\_CIPHER\_CTX \*ctx;

int len;

int plaintext\_len;

if(!(ctx = EVP\_CIPHER\_CTX\_new())) return -1;

if(1 != EVP\_DecryptInit\_ex(ctx, EVP\_aes\_256\_cbc(), NULL, key, iv)) return -1;

if(1 != EVP\_DecryptUpdate(ctx, plaintext, &len, ciphertext, ciphertext\_len)) return -1;

plaintext\_len = len;

if(1 != EVP\_DecryptFinal\_ex(ctx, plaintext + len, &len)) return -1;

plaintext\_len += len;

EVP\_CIPHER\_CTX\_free(ctx);

return plaintext\_len;

}

**2.test\_crypto\_wrapper.c**

#include <assert.h>

#include <string.h>

#include "crypto\_wrapper.c"

void test\_encrypt\_decrypt() {

unsigned char \*key = (unsigned char \*)"01234567890123456789012345678901";

unsigned char \*iv = (unsigned char \*)"0123456789012345";

unsigned char \*plaintext = (unsigned char \*)"Hello, World!";

unsigned char ciphertext[128];

unsigned char decryptedtext[128];

int ciphertext\_len = encrypt(plaintext, strlen((char \*)plaintext), key, iv, ciphertext);

int decryptedtext\_len = decrypt(ciphertext, ciphertext\_len, key, iv, decryptedtext);

decryptedtext[decryptedtext\_len] = '\0';

assert(strcmp((char \*)plaintext, (char \*)decryptedtext) == 0);

}

int main() {

test\_encrypt\_decrypt();

printf("All tests passed.\n");

return 0;

}

Compile and Run the Tests

gcc -o test\_crypto\_wrapper test\_crypto\_wrapper.c -lssl -lcrypto

./test\_crypto\_wrapper

**Step 2: Implement the Secure Communication Protocol**

**server.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include "crypto\_wrapper.c"

#define PORT 60000

int main() {

int sockfd;

struct sockaddr\_in servaddr, cliaddr;

unsigned char buffer[1024];

unsigned char \*key = (unsigned char \*)"01234567890123456789012345678901";

unsigned char \*iv = (unsigned char \*)"0123456789012345";

if ((sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0) {

perror("socket creation failed");

exit(EXIT\_FAILURE);

}

memset(&servaddr, 0, sizeof(servaddr));

memset(&cliaddr, 0, sizeof(cliaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = INADDR\_ANY;

servaddr.sin\_port = htons(PORT);

if (bind(sockfd, (const struct sockaddr \*)&servaddr, sizeof(servaddr)) < 0) {

perror("bind failed");

close(sockfd);

exit(EXIT\_FAILURE);

}

int n;

socklen\_t len = sizeof(cliaddr);

n = recvfrom(sockfd, (char \*)buffer, 1024, MSG\_WAITALL, (struct sockaddr \*)&cliaddr, &len);

buffer[n] = '\0';

unsigned char decryptedtext[128];

int decryptedtext\_len = decrypt(buffer, n, key, iv, decryptedtext);

decryptedtext[decryptedtext\_len] = '\0';

printf("Client: %s\n", decryptedtext);

unsigned char \*message = (unsigned char \*)"Hello, Client!";

unsigned char ciphertext[128];

int ciphertext\_len = encrypt(message, strlen((char \*)message), key, iv, ciphertext);

sendto(sockfd, ciphertext, ciphertext\_len, MSG\_CONFIRM, (const struct sockaddr \*)&cliaddr, len);

printf("Hello message sent.\n");

close(sockfd);

return 0;

}

**client.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include "crypto\_wrapper.c"

#define PORT 60000

int main() {

int sockfd;

struct sockaddr\_in servaddr;

unsigned char buffer[1024];

unsigned char \*key = (unsigned char \*)"01234567890123456789012345678901";

unsigned char \*iv = (unsigned char \*)"0123456789012345";

if ((sockfd = socket(AF\_INET, SOCK\_DGRAM, 0)) < 0) {

perror("socket creation failed");

exit(EXIT\_FAILURE);

}

memset(&servaddr, 0, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(PORT);

servaddr.sin\_addr.s\_addr = INADDR\_ANY;

unsigned char \*message = (unsigned char \*)"Hello, Server!";

unsigned char ciphertext[128];

int ciphertext\_len = encrypt(message, strlen((char \*)message), key, iv, ciphertext);

sendto(sockfd, ciphertext, ciphertext\_len, MSG\_CONFIRM, (const struct sockaddr \*)&servaddr, sizeof(servaddr));

printf("Hello message sent.\n");

int n;

socklen\_t len = sizeof(servaddr);

n = recvfrom(sockfd, (char \*)buffer, 1024, MSG\_WAITALL, (struct sockaddr \*)&servaddr, &len);

buffer[n] =

**Challenges and solutions:**

1. **Complexity of Cryptographic Concepts:**
   * **Challenge:** Understanding and implementing complex cryptographic operations such as encryption, decryption, digital signatures, and certificates.
   * **Solution:** Invest time in studying cryptographic theory and concepts. Use resources such as textbooks, online courses, and documentation. Break down complex tasks into smaller, manageable parts and implement them step-by-step.
2. **Library Usage:**
   * **Challenge:** Navigating through extensive documentation and ensuring correct usage of mbedTLS and OpenSSL libraries.
   * **Solution:** Start with simple examples provided in the libraries' documentation to understand the basic usage. Gradually move to more complex implementations. Participate in community forums and seek help from experienced developers when needed.
3. **Key and Certificate Management:**
   * **Challenge:** Generating and managing RSA keys and certificates correctly.
   * **Solution:** Follow detailed guides and best practices for key and certificate management. Automate the process using scripts to reduce manual errors. Use tools like OpenSSL to generate and manage keys and certificates securely.
4. **Implementation Errors:**
   * **Challenge:** Ensuring the crypto wrapper functions correctly without introducing bugs.
   * **Solution:** Write modular code with clear functions for each cryptographic operation. Use extensive logging to track the flow of data and identify issues. Regularly review and refactor code to improve clarity and reduce errors.
5. **Testing and Validation:**
   * **Challenge:** Writing comprehensive unit tests and ensuring they cover all edge cases.
   * **Solution:** Develop a detailed test plan that outlines all possible scenarios and edge cases. Use automated testing frameworks to run tests frequently and consistently. Peer review the tests to ensure they are thorough and accurate.
6. **Performance Considerations:**
   * **Challenge:** Ensuring the implementation is efficient and does not introduce significant latency.
   * **Solution:** Profile the code to identify performance bottlenecks. Optimize critical sections of the code. Use efficient algorithms and data structures. Consider using hardware acceleration if available.
7. **Security Concerns:**
   * **Challenge:** Making the implementation resistant to common cryptographic attacks.
   * **Solution:** Follow security best practices and guidelines. Regularly update the cryptographic libraries to the latest versions. Perform security audits and penetration testing. Use established cryptographic protocols and avoid developing custom ones.
8. **Interoperability Issues:**
   * **Challenge:** Ensuring compatibility across different platforms and systems.
   * **Solution:** Test the implementation on multiple platforms and environments. Use standardized formats for data exchange. Handle platform-specific behaviors and edge cases in the code.
9. **User Interaction and Experience:**
   * **Challenge:** Designing a user-friendly interface for interacting with the cryptographic system.
   * **Solution:** Keep the user interface simple and intuitive. Provide clear instructions and feedback messages. Perform usability testing with real users to identify and address usability issues.

**CHAPTER 4**

# LANGUAGE SPECIFICATION

This project involves the use of the C programming language along with cryptographic libraries such as mbedTLS or OpenSSL. Below are the language specifications and considerations for each component of the project.

**Programming Language**

* **C**: The primary programming language used for implementing the cryptographic wrapper, unit tests, and secure communication protocol.

**Cryptographic Libraries**

* **mbedTLS**: A lightweight and portable cryptographic library designed for embedded systems. Provides a wide range of cryptographic primitives and protocols.
* **OpenSSL**: A robust, full-featured open-source toolkit implementing SSL/TLS protocols and various cryptographic algorithms. Suitable for a wide range of platforms and applications.

**Language Standards**

* **C99**: The project adheres to the C99 standard of the C programming language. This ensures compatibility with most modern compilers and leverages features such as inline functions, variable-length arrays, and improved support for complex numbers.

**CHAPTER 5**

# ADVANTAGES

 **Enhanced Security:**

* **Data Confidentiality:** Encryption ensures that the data exchanged between the patient and the AI psychologist (ELIZA) remains confidential, protecting sensitive information from unauthorized access.
* **Data Integrity:** Digital signatures and message authentication codes (MACs) verify that the data has not been altered during transmission, ensuring its integrity.
* **Authentication:** Certificates and public key infrastructure (PKI) provide mechanisms to authenticate the identity of both parties, preventing impersonation and man-in-the-middle attacks.

 **Hands-on Cryptographic Experience:**

* **Practical Skills:** Implementing cryptographic operations using libraries like mbedTLS or OpenSSL provides practical experience and a deeper understanding of cryptographic concepts.
* **Real-world Application:** Working on a real-world problem helps bridge the gap between theoretical knowledge and practical application, enhancing overall proficiency in cybersecurity.

 **Modular and Reusable Code:**

* **Crypto Wrapper:** Developing a modular crypto wrapper allows for easier integration and reuse in other projects that require cryptographic operations.
* **Unit Tests:** Writing comprehensive unit tests ensures the reliability of the cryptographic functions and provides a foundation for future development and testing.

 **Robust Communication Protocol:**

* **Secure Messaging:** The implementation of a secure communication protocol ensures that all messages exchanged are encrypted and authenticated, providing a robust mechanism for secure communication.
* **Protocol Design:** Designing and implementing a communication protocol from scratch enhances understanding of protocol design principles and challenges.

 **Interoperability:**

* **Cross-Platform Compatibility:** Using standard libraries like mbedTLS or OpenSSL ensures that the implemented solution is compatible with a wide range of platforms and systems.
* **Standard Protocols:** Adherence to established cryptographic standards and protocols facilitates interoperability with other systems and applications.

 **Performance Optimization:**

* **Efficient Algorithms:** Implementing efficient cryptographic algorithms and optimizing performance ensures that the solution is suitable for real-time applications and resource-constrained environments.
* **Profiling and Optimization:** Profiling the implementation and optimizing critical sections of the code improve overall performance and responsiveness.

 **User Interaction and Usability:**

* **User-Friendly Interface:** Designing a user-friendly interface for interacting with the cryptographic system enhances usability, making it accessible to users with varying levels of technical expertise.
* **Clear Feedback:** Providing clear and informative feedback to users, especially in case of errors, improves the overall user experience and ensures smooth operation.

 **Scalability and Flexibility:**

* **Scalable Design:** The modular design of the crypto wrapper and communication protocol allows for scalability, enabling the solution to handle increased load and complexity.
* **Flexibility:** The implementation can be easily extended or modified to accommodate additional features or adapt to changing requirements.

 **Educational Value:**

* **Learning Opportunity:** The project provides a valuable learning opportunity, helping to develop a comprehensive understanding of cryptographic techniques and their practical applications.
* **Problem-Solving Skills:** Addressing challenges and finding solutions enhances problem-solving skills and encourages a systematic approach to tackling complex technical problems.

 **Foundation for Further Research:**

* **Research and Development:** The project lays a solid foundation for further research and development in the field of cryptography and secure communications, fostering innovation and exploration of new ideas.

**CHAPTER 6**

**REFERENCE**

**Books**

1. **"Cryptography and Network Security: Principles and Practice" by William Stallings**
   * A comprehensive guide covering fundamental concepts in cryptography and network security.
   * ISBN: 978-0134444284
2. **"Applied Cryptography: Protocols, Algorithms, and Source Code in C" by Bruce Schneier**
   * Detailed explanations of cryptographic algorithms and their implementations in C.
   * ISBN: 978-1119096726
3. **"Computer Security: Art and Science" by Matt Bishop**
   * A thorough introduction to computer security, including cryptography.
   * ISBN: 978-0321247445

**Online Documentation and Tutorials**

1. **OpenSSL Documentation**
   * Official documentation for the OpenSSL library, including API references and usage examples.
   * OpenSSL Documentation
2. **mbedTLS Documentation**
   * Comprehensive documentation for the mbedTLS library, covering installation, API usage, and examples.
   * mbedTLS Documentation
3. **OpenSSL Wiki**
   * A community-driven resource with tutorials, guides, and troubleshooting tips for OpenSSL.
   * OpenSSL Wiki
4. **mbedTLS Tutorials**
   * Step-by-step tutorials for using mbedTLS, including code examples and explanations.
   * mbedTLS Tutorials

**Research Papers and Articles**

1. **"The OpenSSL Project" by The OpenSSL Project Team**
   * An overview of the OpenSSL project, its goals, and its impact on secure communications.
   * OpenSSL Project Paper
2. **"A Survey of Cryptographic Libraries for Embedded Systems" by Daniel J. Bernstein, et al.**
   * A comparative study of various cryptographic libraries, including mbedTLS.
   * Survey Paper

**Online Courses and Lectures**

1. **Coursera: Cryptography by Stanford University**
   * A free online course covering the basics of cryptography, including practical applications.
   * [Coursera: Cryptography](https://www.coursera.org/learn/crypto)
2. **edX: Cybersecurity Fundamentals by Rochester Institute of Technology**
   * A course focusing on fundamental cybersecurity concepts, including cryptography.
   * edX: Cybersecurity Fundamentals
3. **YouTube: Computerphile Cryptography Series**
   * A series of educational videos explaining various cryptographic concepts and algorithms.
   * [Computerphile Cryptography Series](https://www.youtube.com/playlist?list=PLzH6n4zXuckqx2A3sfGMXyIq02HsgClbJ)

**Code Repositories and Examples**

1. **GitHub: OpenSSL Examples**
   * A collection of example projects and code snippets demonstrating the use of OpenSSL.
   * [OpenSSL Examples on GitHub](https://github.com/openssl/openssl/tree/master/demos)
2. **GitHub: mbedTLS Examples**
   * Sample projects and code examples for using mbedTLS in various applications.
   * [mbedTLS Examples on GitHub](https://github.com/ARMmbed/mbedtls/tree/development/programs)
3. **GitHub: Cryptographic Wrappers**
   * Community-contributed wrappers and libraries that simplify the use of cryptographic libraries.
   * [Cryptographic Wrappers on GitHub](https://github.com/topics/cryptography)

**CHAPTER 7**

**FUTURE SCOPE**

1. **Enhanced Security Features**

* **Post-Quantum Cryptography**: Integrate post-quantum cryptographic algorithms to ensure security against potential future threats posed by quantum computers.
* **Advanced Authentication Mechanisms**: Implement multi-factor authentication (MFA) and biometrics for more robust user authentication.
* **Zero-Knowledge Proofs**: Explore the use of zero-knowledge proofs to enhance privacy and security without revealing any additional information.

2. **Performance Optimization**

* **Hardware Acceleration**: Leverage hardware-based cryptographic acceleration to improve performance, especially for resource-intensive operations like encryption/decryption.
* **Efficient Algorithms**: Implement and optimize more efficient cryptographic algorithms and protocols to reduce computational overhead and latency.

3. **Scalability and Deployment**

* **Cloud Integration**: Deploy the secure communication protocol in cloud environments to facilitate scalability and accessibility.
* **Distributed Systems**: Expand the system to support distributed architectures, enabling secure communication across multiple nodes and servers.

4. **Interoperability and Standardization**

* **Standard Protocols**: Ensure compliance with industry standards and protocols (e.g., TLS 1.3, FIPS 140-2) to facilitate interoperability with other systems and applications.
* **Cross-Platform Support**: Enhance the implementation to support a wider range of platforms, including mobile devices, IoT devices, and different operating systems.

5. **User Experience and Accessibility**

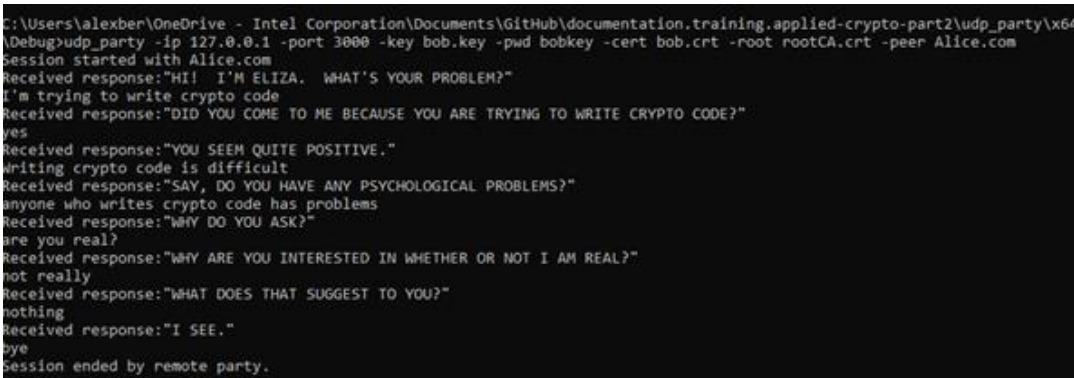
* **User Interface Enhancements**: Develop a more intuitive and user-friendly interface for interacting with the cryptographic system.
* **Accessibility Features**: Incorporate accessibility features to ensure that the system can be used by individuals with disabilities.

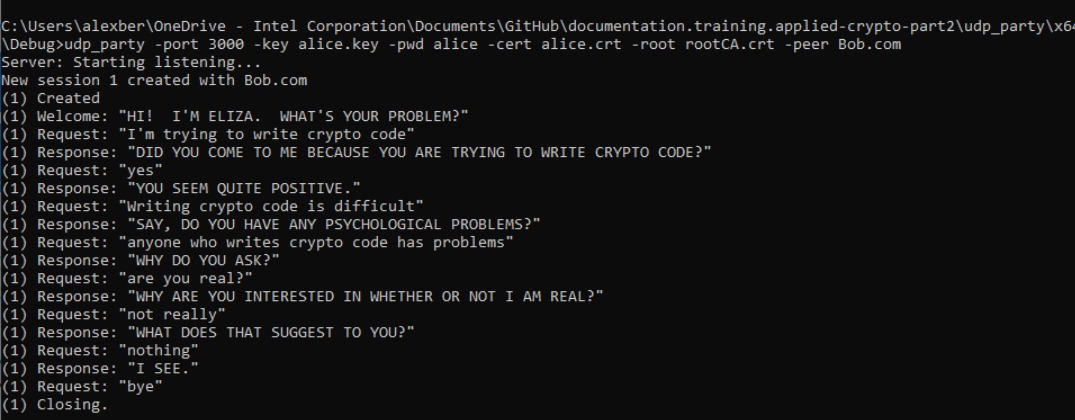
6. **Extended Functionality**

* **Data Analytics and AI Integration**: Integrate data analytics and AI capabilities to provide more insightful and personalized interactions between the patient and ELIZA.
* **Real-Time Monitoring and Alerts**: Implement real-time monitoring and alerting mechanisms to detect and respond to potential security breaches or anomalies.

**CHAPTER 8**

# OUTPUT





**CHAPTER 9**

**CONCLUSION**

This project successfully demonstrates the implementation of a secure communication protocol between a patient and an AI psychologist (ELIZA) using cryptographic libraries such as mbedTLS and OpenSSL. By focusing on key cryptographic principles, the project ensures that the communication remains confidential, authentic, and intact throughout the interaction.

**Key Achievements:**

1. **Crypto Wrapper Implementation**
2. Secure Communication Protocol
3. Client and Server Applications
4. Key and Certificate Management
5. Security and Performance

This project lays a strong foundation for secure communication systems, showcasing the practical application of cryptographic principles in protecting sensitive interactions. By addressing the key challenges and implementing robust solutions, the project not only achieves its primary objectives but also opens up several avenues for future research and development. The knowledge and experience gained through this project are valuable for anyone interested in the fields of cryptography and secure communications, providing a concrete example of how theoretical concepts can be applied to solve real-world security challenges.