

IERG4998I Final Year Project I

Privacy-Preserving Instant Messaging Android Application using Python

BY

TSOI, Ming Hon

1155175123

ZHANG, Yi Yao

1155174982

A FINAL YEAR PROJECT REPORT
SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF BACHELOR OF INFORMATION ENGINEERING
DEPARTMENT OF INFORMATION ENGINEERING
THE CHINESE UNIVERSITY OF HONG KONG

November, 2024

ABSTRACT

In today's digital landscape, ensuring the privacy and security of online communication has become paramount. Our project focuses on developing a Python-based instant messaging Android application that prioritizes user data protection through robust encryption techniques. By leveraging advanced encryption methods like Diffie-Hellman, AES, RSA, and Signal protocols, we aim to create a secure platform that guarantees the confidentiality, integrity, and anonymity of user conversations.

1. INTRODUCTION

1.1 BACKGROUND

In today's digital age, the security and privacy of communication over instant messaging platforms have become very important. Global cyber-attacks rose by 28% in Q3 2022 [1], underscoring the growing threat landscape that makes encryption essential. According to the UK government, the average cost of a data breach for a small-to-medium enterprise (SME) in the UK was around £4,180 in 2023 [2]. A study by the Ponemon Institute found that 60% of data breaches in 2021 involved unencrypted data, emphasizing the need for widespread encryption adoption. The global encryption software market is expected to grow from \$5.4 billion in 2020 to \$13.9 billion by 2025, at a CAGR of 21.0% during the forecast period, indicating the rising demand for encryption solutions [3].

These statistics clearly demonstrate the growing importance of data breaches, surveillance, and unauthorized access to personal information, we want to make sure that when we chat on messaging apps, our messages are safe from prying eyes, hackers (ensuring the confidentiality, integrity, and anonymity of user communications).

Two prominent examples of privacy-preserving instant messaging applications are Signal and Apple's iMessage with PQ3. Signal is an open-source messaging app that uses end-to-end encryption by default. Apple's iMessage also offers end-to-end encryption, and has recently introduced the PQ3 cryptographic protocol, which adds post-quantum cryptography [4][5].

This project focuses on addressing these challenges by implementing various cryptographic techniques and security measures to minimize the exposure of sensitive information to unauthorized parties so that user data and communication content can be protected (text, images, videos etc.) from eavesdroppers, hackers, service providers, and other malicious entities.

1.2 MESSAGING ENCRYPTION INDUSTRY & PROJECT SIGNIFICANCE

Messaging encryption is crucial for protecting sensitive information from unauthorized access. It ensures that only the intended recipients can read the messages,

safeguarding against various cyber threats such as data interception and man-in-the-middle attacks. As businesses increasingly rely on text messaging for communication, the need for secure messaging solutions has become paramount to maintain client trust and comply with legal regulations [6].

The GSM Association (GSMA) has announced plans to implement end-to-end encryption (E2EE) for Rich Communications Services (RCS) messaging, which will enable secure messaging between Android and iOS devices. This initiative is a major step towards standardizing messaging encryption across different platforms, addressing challenges such as key federation and group membership security. Tom Van Pelt, GSMA's technical director, emphasized the importance of this development for user protection in messaging services [7][8].

This announcement coincides with Apple's rollout of iOS 18, which includes support for RCS in its Messages app. The update introduces features like message reactions, typing indicators, and high-quality media sharing, but RCS currently lacks built-in E2EE. Google has previously implemented the Signal protocol to secure RCS messages on Android devices, while Apple's iMessage already supports E2EE.

1.3 MOTIVATION AND PROJECT IDEA

The reason we are working on a Privacy-Preserving Instant Messaging App is because we love cybersecurity and using tech to solve real problems. We want to get hands-on experience with advanced ways to keep messages safe, like end-to-end encryption, and find new ways to make communication really secure. This project fits my studies and my goal to work in cybersecurity of being a security engineer.

Creating this app allows us to use what we have learned in a practical way, getting better at coding, encryption, and keeping systems safe. The app could also help make communication safer for everyone online. It will definitely improve our tech skills and understand how tech, privacy, and security all work together.

The core goal of the project is to develop a Python-based application to protect user communications from threats such as eavesdropping and data leakage. By enhancing the security of instant messaging platforms, we aim to create a privacy-protecting instant messaging system that ensures the confidentiality, integrity and anonymity of shared communications. While past efforts have focused on traditional encryption algorithms, we will explore new ways to enhance chat security.

1.4 PROJECT PIPELINE

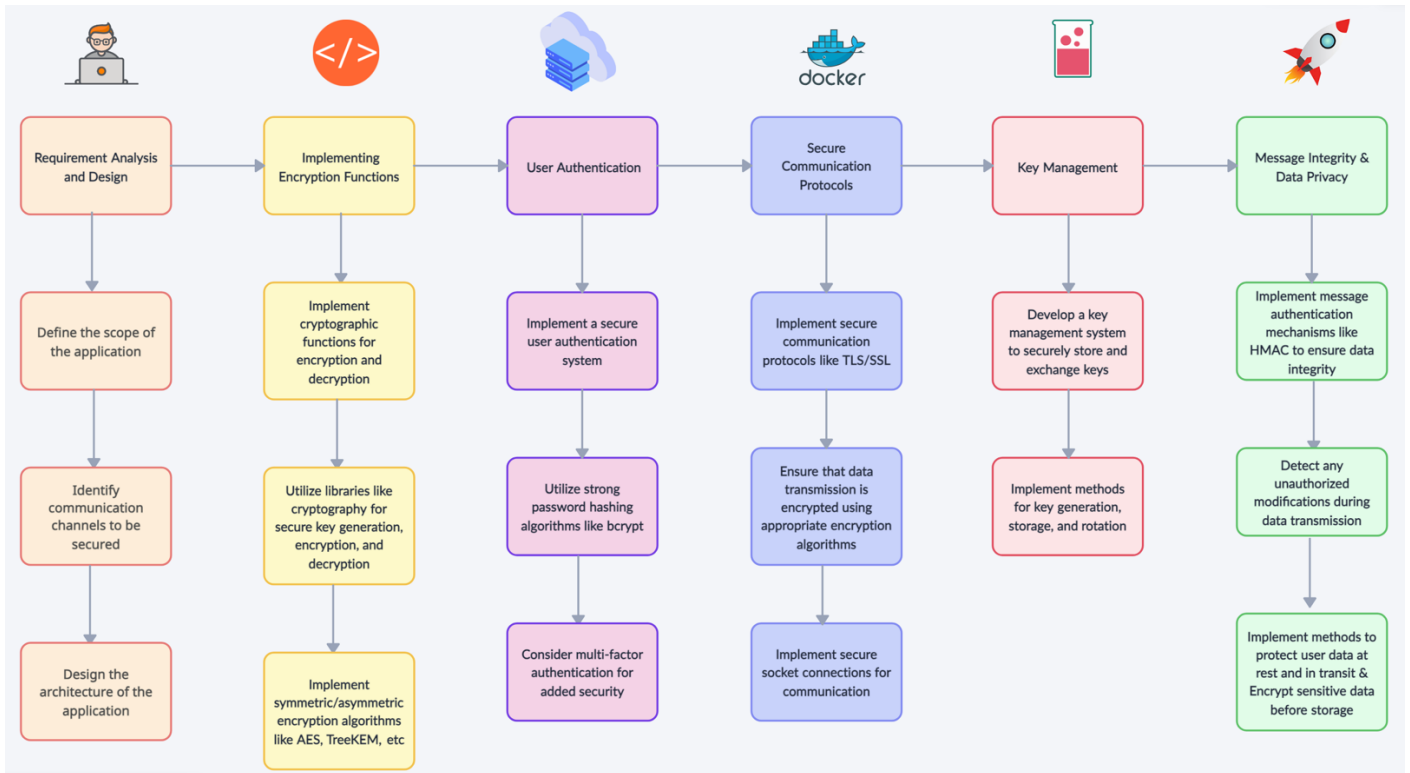


Figure 1: Flow chart of the project pipeline

Creating a secure communication app in Python involves several steps. First, we have to plan what the app will do and how it will work. We will figure out which ways of talking need protection and we draw a map of how the app will be built. Next, we make sure messages are scrambled and unscrambled safely by using special codes from libraries like cryptography. We will also set up ways for users to prove they are who they say they are by using strong password codes like bcrypt and maybe extra security steps. We will make sure that messages are sent in a safe way using special rules like TLS/SSL and that connections are secure. We will keep the keys, which are like secret codes, safe and change them regularly. Then we will check messages to make sure no one has changed

them without permission. After that, we will make sure that user information is kept secret both when it's saved and when it's moving around. Lastly, we will keep a close eye on what's happening, tracking user actions and looking out for any signs of trouble (Fig. 1).

2. RELATED OR PREVIOUS WORK

2.1 OVERVIEW

Protocols like Signal groups and Sender Keys offer simplicity but face scalability issues and potential vulnerabilities in ensuring post-compromise security (PCS). Matrix's protocol refines the Sender Keys concept with decentralized features and improved PCS mechanisms. The MLS protocol introduces TreeKEM for efficient key updates in large groups but relies on centralization and strict commit processing.

Protocol	Central server not needed	Broadcast messages	Update & remove costs: ¹		PCS & FS	PCS in face of concurrent updates
			Sender	Per recipient		
Signal groups	✓ ²	✗	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\frac{1}{2}$ (PCS issues) ³	[Optimal] ³
Sender Keys (WhatsApp)	✓ ²	✓	$\mathcal{O}(n)$	$\mathcal{O}(n)$	FS only ⁴	[Optimal] ⁴
Matrix	✓	✓	$\mathcal{O}(n)$	$\mathcal{O}(n)$	PCS only	Optimal ³
MLS (TreeKEM)	✗	✓	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\frac{1}{2}$ (FS issues)	Only one sequence heals
Re-randomized TreeKEM	✗	✓	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	✓	
Causal TreeKEM	✓ ²	✓	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\frac{1}{2}$ (severe FS issues)	Any sequence heals
Concurrent TreeKEM	✓ ²	✓	$\mathcal{O}(t + t \log(n/t))$	$\mathcal{O}(1)$	PCS only	After two rounds
Our DCGKA protocol	✓	✓	$\mathcal{O}(n)$	$\mathcal{O}(1)$	✓	All but last can be concurrent

¹ Costs are the number of public-key cryptographic operations performed. The total size of messages broadcast equals the "Sender" column except for Sender Keys and Matrix, which have total broadcast network cost $\mathcal{O}(n^2)$. t denotes the number of mutually concurrent messages.

² Does not specify how to determine group membership in the face of concurrent additions and removals.

³ Optimal PCS in the face of concurrent updates is possible by using a 2-party protocol with optimal PCS+FS in place of pairwise Signal.

⁴ Optimal PCS in the face of concurrent updates is possible at the given costs, but not used in practice.

Figure 2: Comparison of different messaging protocols. Table from Key Agreement for Decentralized Secure Group Messaging with Strong Security Guarantees - Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, Alastair R. Beresford (2020).

Available at: <https://eprint.iacr.org/2020/1281.pdf>.

Additionally, advancements like Re-randomized TreeKEM and Causal TreeKEM explore strengthening forward secrecy and handling dynamic groups in decentralized scenarios. Concurrent TreeKEM and DCGKA aim to enhance PCS update efficiency, with the latter providing advantages in post-compromise security and concurrent update handling. Each protocol presents unique trade-offs in terms of scalability, security guarantees, and decentralization, contributing to the evolving landscape of secure group messaging protocols (Fig. 2) [9].

2.2 RESEARCH PAPER ANALYSIS

The following five research papers have been reviewed, all of which are closely aligned with our project, offering insights into the implementations of several encryption algorithms. Each paper provides significant contributions to the field of cryptography and secure communication:

2.2.1 CSIDH: AN EFFICIENT POST-QUANTUM COMMUTATIVE GROUP ACTION [10]

This paper summarizes the key points discussed and the significance of the proposed CSIDH scheme. CSIDH scheme is introduced as an efficient commutative group action suitable for non-interactive key exchange in a post-quantum setting. The authors adapted the Couveignes-Rostovtsev-Stolbunov scheme to supersingular elliptic curves defined over a prime field F_p , providing a solution that addresses the inefficiency of previous schemes while maintaining security.

CSIDH offers a more efficient alternative to existing isogeny-based schemes, allowing for non-interactive key exchange with reduced computational costs. The scheme is designed to provide a conjectured post-quantum security level of 64 bits, matching NIST's post-quantum security category I, while offering a presumed classical security level of 128 bits.

Also, CSIDH achieves these security levels with public keys of only 64 bytes, making it suitable for resource-constrained environments. By adapting the Couveignes-Rostovtsev-Stolbunov scheme to supersingular elliptic curves over F_p , the authors demonstrate a novel approach to isogeny-based cryptography. There's also a proof-of-concept implementation is reported to carry out key exchange efficiently, highlighting the practical viability of the CSIDH scheme.

2.2.2 KEY AGREEMENT FOR DECENTRALIZED SECURE GROUP MESSAGING WITH STRONG SECURITY GUARANTEES

This paper introduces a Decentralized Continuous Group Key Agreement (DCGKA) security notion for

establishing shared symmetric keys in dynamic groups. An implemented protocol that achieves DCGKA, ensuring correctness and security. The development of a secure group messaging system based on the DCGKA protocol and the evaluation of a prototype implementation demonstrating practical efficiency suitable for deployment.

It has introduced that this protocol is designed for decentralized networks where there is no central authority, allowing for communication even in scenarios of network partitions or high latency links. That it provides forward secrecy, ensuring that compromised devices cannot decrypt past messages, and post-compromise security, enabling users to continue communication securely even after a compromise.

2.2.3 PARAKEET: PRACTICAL KEY TRANSPARENCY FOR END-TO-END ENCRYPTED MESSAGING [11]

This paper presents a comprehensive solution to the challenges of key transparency in encrypted messaging systems. It introduces Parakeet, a key transparency system designed to address scalability issues that limit previous academic works from practical implementation in real-world encrypted messaging applications.

It introduces an efficient Verified Key Directory (VKD) construction using an ordered zero-knowledge set (oZKS), providing significant storage improvements compared to existing solutions. This implementation is modular and flexible, supporting billions of users and offering storage optimizations. To overcome the limitations of append-only data structures in verifiable data systems, the paper introduces the concept of "compaction." This operation allows for the reduction of stored data by removing outdated entries, thereby easing the burden of ever-growing storage requirements.

The Parakeet proposes a lightweight consensusless consistency protocol to ensure that users can access small, shared commitments efficiently. This protocol provides the necessary guarantees with low performance overhead, addressing issues related to scalability and performance in large-scale systems. And it emphasizes the practical viability of Parakeet for real-world deployments, considering the challenges of handling large user bases and ensuring efficient and secure key management in end-to-end encrypted messaging applications.

2.2.4 SECURITY ANALYSIS OF THE MLS KEY DISTRIBUTION [12]

This paper delves deep into the security analysis of the MLS Key Distribution protocol. It discusses the challenges of group communication protocols, the importance of continuous key-exchange for dynamic groups, and the specific focus on the MLS continuous group key distribution (CGKD), including the MLS key schedule and TreeKEM design. The paper highlights the significance of achieving post-compromise security (PCS) and forward secrecy (FS) guarantees in modern messaging applications, emphasizing the role of key exchange protocols in ensuring secure communication.

It extensively studies the security aspects of the MLS key distribution, particularly the MLS continuous group key distribution (CGKD), comprising the MLS key schedule and TreeKEM. The analysis establishes the uniqueness and key indistinguishability properties of the MLS CGKD as computational security properties, crucial for ensuring secure group communication.

The paper also compares its findings with related security analyses of the MLS protocol, summarizing key proposals for enhancing security guarantees, such as Post-Compromise Forward Security (PCFS) and improvements to TreeKEM for stronger forward secrecy. Drawing inspiration from the analysis of TLS 1.3, the paper highlights the improved domain separation in the MLS key schedule, which simplifies security analysis by avoiding complications like evolved invariant proofs. And it provides a detailed cryptographic analysis of the MLS key distribution structure, focusing on the key derivation mechanisms of TreeKEM and the key schedule, while excluding authentication and the Secret Tree component in the key schedule.

2.2.5 HOW TO WATERMARK CRYPTOGRAPHIC FUNCTIONS [13]

This paper discusses a way to add hidden marks to special math rules that keep secrets safe. These marks help identify copies and protect the rules from being changed by bad actors. The paper explains that while adding these marks is crucial, there haven't been many studies on how to do it, especially for these secret math rules.

The authors introduce a new method to add these marks securely, making sure that the rules stay safe even if someone tries to remove the marks. The authors show that by using a special type of math problem called the decisional linear problem, they can add marks to these secret math rules in a way that's hard for others to tamper with. They use advanced techniques like dual system encryption and dual pairing vector spaces to create their marking system. This approach is a fresh idea in this area of research, and it could have important applications in protecting and tracing ownership of digital secrets.

3. METHODOLOGIES

In order to achieve the goal of protecting the privacy of instant messaging, the project will focus on advanced encryption technologies/algorithms and security measures for text, images, videos and other forms of communication through instant messaging platforms. By combining symmetric and asymmetric encryption technologies, as well as secure key exchange protocols such as Diffie-Hellman, AES, RSA and Signal protocols, our goal is to create a powerful framework that ensures the confidentiality and integrity of user communications that ensures end-to-end encryption and security key exchange.

3.1 END-TO-END ENCRYPTION

End-to-end encryption (E2EE) is a private communication system that only communication users can participate in. Therefore, no one else, including communications system providers, telecommunications providers, network providers or malicious actors, can access the encryption keys required for the conversation. End-to-end encryption is designed to prevent data from being read or secretly modified by people other than the true sender and recipient. Messages are encrypted by the sender, but third parties cannot decrypt them and store them in an encrypted manner. The recipient retrieves the encrypted material and decrypts it himself.

Signal and Apple's iMessage are two widely used messaging apps that use end-to-end encryption to protect user communications. Signal uses the Signal protocol, which combines Diffie-Hellman key exchange, symmetric key encryption, and identity keys to provide end-to-end encryption. Apple's iMessage, on the other hand, recently launched the PQ3 encryption protocol, which combines classical elliptic curve encryption with post-quantum Kyber encryption to provide protection against current and future quantum computing threats.

3.2 E2EE METHODS

Key algorithms used in E2EE include Diffie-Hellman key exchange, AES, RSA and the signal protocol.

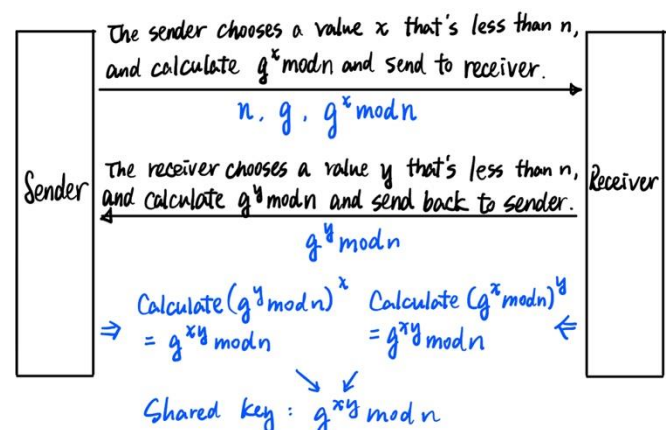


Figure 3: A simple demonstration of Diffie-Hellman key exchange protocol

Diffie-Hellman key exchange protocol: A shared secret key is established between communicating parties, ensuring secure communication channels (Fig. 3).

We have a shared secret key for Alice:

$$K_a = (Y_b)^A \bmod p = (g^B \bmod p)^A \bmod p = g^{B \times A} \bmod p$$

and a same shared secret key for Bob:

$$K_b = (Y_a)^B \bmod p = (g^A \bmod p)^B \bmod p = g^{A \times B} \bmod p$$

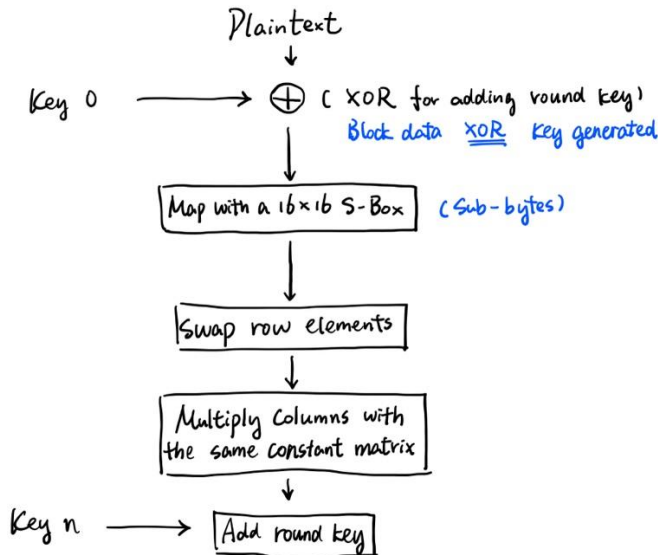


Figure 4: A simple demonstration of AES encryption. Table referring to Jena, B.K. (2023). What Is AES Encryption and How Does It Work? - Simplilearn. [online] Simplilearn.com. Available at: <https://www.simplilearn.com/tutorials/cryptography-tutorial/aes-encryption>.

AES: A symmetric-key algorithm that is used by WhatsApp and Signal (Fig. 4) [14]. AES operates on fixed-size blocks of data, typically 128 bits, and supports key sizes of 128, 192, or 256 bits. Symmetric encryption means that the same secret key is used for both encryption and decryption, necessitating secure key management between communicating parties. The algorithm processes data in blocks, with each block undergoing a specific number of rounds depending on the key size: 10 rounds for AES-128, 12 for AES-192, and 14 for AES-256. These rounds involve intricate operations such as substitution, permutation, mixing, and key expansion, ensuring a high level of security through a combination of confusion and diffusion techniques. AES encryption will be a major encryption method to be used in our project later. More details will be discussed in Part 4.2. Implementation of MLS Key Distribution.

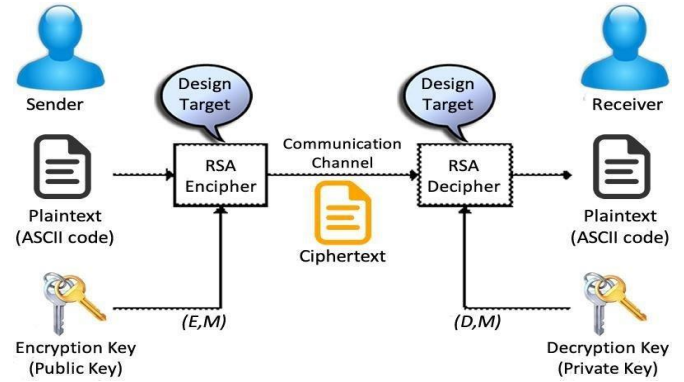


Figure 5: A simple demonstration of RSA encryption. Table from ResearchGate. (n.d.). Figure 2. RSA algorithm structure. [online] Available at: https://www.researchgate.net/figure/RSA-algorithm-structure_fig2_298298027.

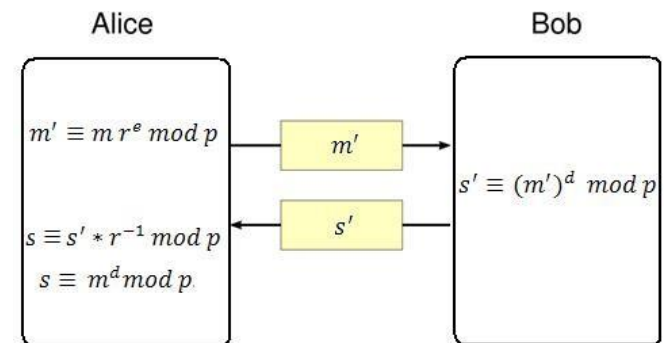


Figure 6: An overview of the RSA algorithm. Table from D. Şuteu, "RSA algorithm," Blogspot.com, Jan. 10, 2017. <https://trizenx.blogspot.com/2017/01/rsa-algorithm.html>

RSA: An asymmetric-key algorithm which the implementation of it makes heavy use of modular arithmetic, Euler's theorem, and Euler's totient function (Fig. 5 and Fig. 6) [15].

Signal protocol: An open-source encryption protocol developed by Open Whisper Systems which is used by apps like Signal, WhatsApp, and Google Allo etc. It combines several cryptographic primitives, including the Double Ratchet algorithm.

A formal security analysis of the Signal Messaging Protocol (Including the Double Ratchet Algorithm) will be looking like this (Fig. 7):

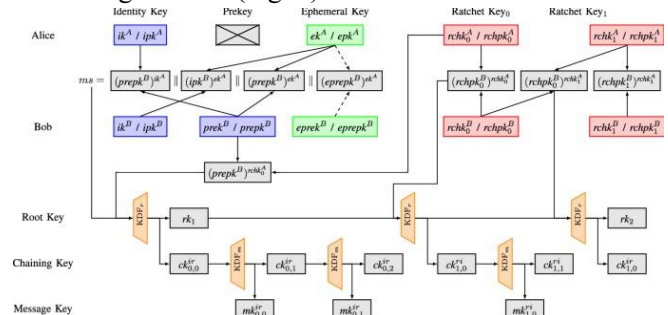


Figure 7: A formal security analysis of the Signal Messaging Protocol. Table from Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L. and Stebila, D. (2020). A Formal Security Analysis of the Signal Messaging Protocol. Journal of Cryptology, 33(4),

4. EXPERIMENT PROCESS AND RESULT ANALYSIS

4.1 AES ENCRYPTION AND CURVE25519 KEYS

```
1 #include <openssl/evp.h>
2 #include <openssl/pem.h>
3 #include <stdio.h>
4
5 int main() {
6     EVP_PKEY *pkey = EVP_PKEY_new();
7     if (!pkey) {
8         fprintf(stderr, "Error creating EVP_PKEY object\n");
9         return 1;
10    }
11
12    EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_X25519, NULL);
13    if (!ctx) {
14        fprintf(stderr, "Error creating context\n");
15        EVP_PKEY_free(pkey);
16        return 1;
17    }
18
19    if (EVP_PKEY_keygen_init(ctx) <= 0 || EVP_PKEY_keygen(ctx, &pkey) <= 0) {
20        fprintf(stderr, "Error generating key\n");
21        EVP_PKEY_CTX_free(ctx);
22        EVP_PKEY_free(pkey);
23        return 1;
24    }
25
26    EVP_PKEY_CTX_free(ctx);
27
28    FILE *privateKeyFile = fopen("private_key.pem", "wb");
29    if (!privateKeyFile) {
30        fprintf(stderr, "Error opening private key file\n");
31        EVP_PKEY_free(pkey);
32        return 1;
33    }
34
35    PEM_write_PrivateKey(privateKeyFile, pkey, NULL, NULL, 0, NULL, NULL);
36    fclose(privateKeyFile);
37
38    FILE *publicKeyFile = fopen("public_key.pem", "wb");
39    if (!publicKeyFile) {
40        fprintf(stderr, "Error opening public key file\n");
41        EVP_PKEY_free(pkey);
42        return 1;
43    }
44
45    PEM_write_PUBKEY(publicKeyFile, pkey);
46    fclose(publicKeyFile);
47
48    EVP_PKEY_free(pkey);
49    return 0;
50 }
```

Figure 8: A code snippet above aims to generate a Curve25519 key pair and save both the private and public keys in PEM format to files

The code snippet above aims to generate a Curve25519 key pair and save both the private and public keys in PEM format to files. This program leverages OpenSSL's API capabilities to facilitate key generation and file output, making it well-suited for cryptographic applications (Fig. 8).

To achieve this functionality, the program relies on specific OpenSSL header files:

- “<openssl/evp.h>”: This header includes definitions essential for the EVP (envelope) interface, primarily used for encryption and key generation.
- “<openssl/pem.h>”: Utilized for the reading and writing of keys in PEM (Privacy-Enhanced Mail) format.

Key Steps in the Code Execution:

1. “EVP_PKEY_new()”: This function initializes a new EVP_PKEY object, designated to hold the generated key.
2. “EVP_PKEY_CTX_new_id(EVP_PKEY_X25519, NULL)”: It creates a key generation context tailored for the Curve25519 algorithm.

3. “EVP_PKEY_keygen_init(ctx)”: This call initiates the key generation process within the specified context.
4. “EVP_PKEY_keygen(ctx, &pkey)”: Here, the key generation function is invoked to generate the key, storing it in the pkey object for further processing.

```
renako@LAPTOP-452HPI88:~$ cd ..
renako@LAPTOP-452HPI88:~/hom$ cd ..
renako@LAPTOP-452HPI88:~$ cd mt
renako@LAPTOP-452HPI88:~/mt$ cd d
renako@LAPTOP-452HPI88:~/mt/d$ cd fyp
renako@LAPTOP-452HPI88:~/mt/d/fyp$ gcc curve25519_keygen.c -o curve25519_keygen.exe -lssl -lcrypto
renako@LAPTOP-452HPI88:~/mt/d/fyp$ ./curve25519_keygen.exe
X25519 Private-Key:
priv:
38:22:69:5a:16:0a:2c:f2:7b:f7:6e:68:47:46:3d:
7d:89:ef:f6:3b:b5:c1:91:bc:b9:c8:28:a5:b5:26:
5d:6c
pub:
74:38:ea:91:33:ea:a7:98:14:92:97:99:86:5e:68:
93:fa:d1:7f:5b:ed:7d:8d:fa:17:a1:b8:d5:2e:f7:
fb:19
renako@LAPTOP-452HPI88:~/mt/d/fyp$ openssl pkey -in private_key.pem -pubin -text -noout
X25519 Public-Key:
pub:
74:38:ea:91:33:ea:a7:98:14:92:97:99:86:5e:68:
93:fa:d1:7f:5b:ed:7d:8d:fa:17:a1:b8:d5:2e:f7:
fb:19
renako@LAPTOP-452HPI88:~/mt/d/fyp$
```

Figure 9: A result showcase after generating a Curve25519 key pair and persistently stores the private and public keys in PEM format

By following this sequence of operations, the program effectively generates a Curve25519 key pair and stores the private and public keys in PEM format, showcasing a robust approach to cryptographic key management using OpenSSL's functionalities (Fig. 9).

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <sodium.h>
4 #include <openssl/evp.h>
5 #include <openssl/err.h>
6
7 void handleError() {
8     ERR_print_errors_fp(stderr);
9     abort();
10 }
11
12 void aes_encrypt(const unsigned char *key, const unsigned char *plaintext, unsigned char *ciphertext) {
13     EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
14     EVP_EncryptInit_ex(ctx, EVP_aes_128_ecb(), NULL, key, NULL);
15
16     int len;
17     int ciphertext_len;
18
19     EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, strlen((const char *)plaintext));
20     ciphertext_len = len;
21
22     EVP_EncryptFinal_ex(ctx, ciphertext + len, &len);
23     ciphertext_len += len;
24
25     EVP_CIPHER_CTX_free(ctx);
26 }
27
28 void aes_decrypt(const unsigned char *key, const unsigned char *ciphertext, unsigned char *plaintext) {
29     EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
30     EVP_DecryptInit_ex(ctx, EVP_aes_128_ecb(), NULL, key, NULL);
31
32     int len;
33     int plaintext_len;
34
35     EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext, strlen((const char *)ciphertext));
36     plaintext_len = len;
37
38     EVP_DecryptFinal_ex(ctx, plaintext + len, &len);
39     plaintext_len += len;
40
41     plaintext[plaintext_len] = '\0';
42     EVP_CIPHER_CTX_free(ctx);
43 }
44
45 int main() {
46     if (sodium_init() < 0) {
47         fprintf(stderr, "Failed to initialize libsodium\n");
48         return 1;
49     }
50
51     unsigned char private_key[crypto_box_SECRETKEYBYTES];
52     unsigned char public_key[crypto_box_PUBLICKEYBYTES];
53     unsigned char recipient_private_key[crypto_box_SECRETKEYBYTES];
54     unsigned char recipient_public_key[crypto_box_PUBLICKEYBYTES];
55
56     crypto_box_keypair(public_key, private_key);
57     crypto_box_keypair(recipient_public_key, recipient_private_key);
58
59     unsigned char shared_key[crypto_box_BEFORENMBYTES];
60     if (crypto_box_beforenm(shared_key, recipient_public_key, private_key) != 0) {
61         fprintf(stderr, "Failed to compute shared key\n");
62         return 1;
63     }
64
65     const unsigned char *message = (unsigned char *)"Hello, this message with AES and Curve25519!";
66     unsigned char ciphertext[128];
67     unsigned char decryptedtext[128];
68
69     aes_encrypt(shared_key, message, ciphertext);
70     aes_decrypt(shared_key, ciphertext, decryptedtext);
71
72     printf("Decrypted message: %s\n", decryptedtext);
73
74     return 0;
75 }
```

Figure 10: A code snippet that shows the utilization of both OpenSSL and libsodium libraries to conduct AES encryption/decryption and generate Curve25519 key pairs

The provided code snippet above illustrates the utilization of both OpenSSL and libsodium libraries to conduct AES encryption/decryption and generate Curve25519 key pairs, offering a secure mechanism for encrypting messages through a shared key derived from a public-private key exchange (Fig. 10) [16].

Key Libraries Used:

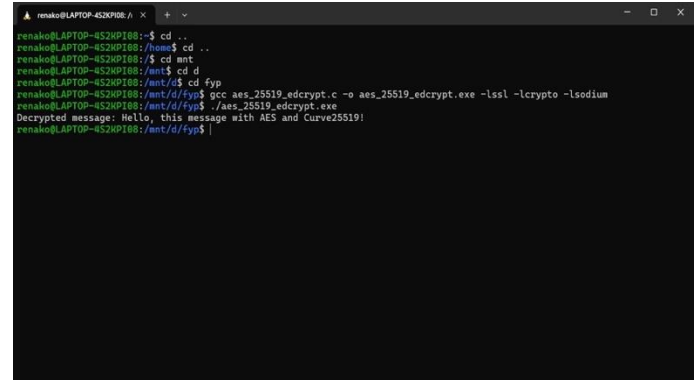
- “<sodium.h>”: This header encapsulates the functionalities of the libsodium library, renowned for its secure cryptographic operations.
- “<openssl/evp.h>”: Integral for the OpenSSL EVP interface, abstracting encryption and decryption functions.
- “<openssl/err.h>”: Essential for error handling within the OpenSSL framework.

Essential Steps for AES Encryption (also applicable for Decryption):

1. “EVP_CIPHER_CTX_new()”: Initiates a new cipher context tailored for encryption tasks.
2. EVP_EncryptInit_ex(ctx, EVP_aes_128_ecb(), NULL, key, NULL)”: Sets up the cipher context for AES-128 encryption in ECB mode, using the designated key.
3. “EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, strlen((const char *)plaintext))”: Encrypts the plaintext, storing the outcome in the ciphertext. The function tracks the number of bytes processed in len.
4. EVP_EncryptFinal_ex(ctx, ciphertext + len, &len)”: Finalizes the encryption process, manages padding requirements, and appends the final output to the ciphertext.

Incorporating libsodium:

- “sodium_init()”: Initializes the libsodium library, a crucial prerequisite for cryptographic operations. It returns a negative value upon initialization failure.
- “crypto_box_keypair(public_key, private_key)”: Generates a public-private key pair utilizing Curve25519, storing them in the provided arrays.
- “crypto_box_beforenm(shared_key, recipient_public_key, private_key)”: Computes a shared key via the recipient's public key and the local private key, facilitating secure key exchange operations.



```
renako@LAPTOP-452NP188: ~$ cd ..
renako@LAPTOP-452NP188: /home$ cd ..
renako@LAPTOP-452NP188: ~$ cd nnt
renako@LAPTOP-452NP188: /mnt$ cd d
renako@LAPTOP-452NP188: /mnt/d$ cd fyp
renako@LAPTOP-452NP188: /mnt/d/fyp$ gcc aes_25519_encrypt.c -o aes_25519_encrypt.exe -lssl -lcrypto -lsodium
renako@LAPTOP-452NP188: /mnt/d/fyp$ ./aes_25519_encrypt.exe
Encrypted message: Hello, this message with AES and Curve25519!
renako@LAPTOP-452NP188: /mnt/d/fyp$
```

Figure 11: A result showcase after conducting AES encryption/decryption and generate Curve25519 key pairs

This comprehensive approach integrates the strengths of both OpenSSL and libsodium libraries to deliver robust encryption capabilities and key pair generation, ensuring secure message transmission through shared key mechanisms (Fig. 11) [17].

4.2 IMPLEMENTATION OF MLS KEY DISTRIBUTION

Messaging Layer Security (MLS) key distribution is a process that involves securely distributing and managing encryption keys within a group of users to enable secure communication.

4.2.1 OVERVIEW

Here are some main functions and features of the MLS security protocol:

1. Group Key Establishment: MLS establishes a shared group key that is used for encrypting messages within a group of users. This key is securely distributed among group members to ensure confidentiality and integrity of communication.
2. Key Updates: MLS supports dynamic group membership changes, allowing for efficient key updates when members join or leave the group. Key updates are handled in a secure manner to maintain the security of the communication channel.
3. Tree Structure: MLS often utilizes a tree structure (such as TreeKEM) to manage key distribution within the group. The tree structure facilitates efficient key management and enables secure key updates as the group composition changes. The purpose of the key schedule is to combine existing key material into new key material and, in particular, to chain keys across epochs via the internal key sint, which is referred to as the init secret in MLS terminology.
4. End-to-End Encryption: MLS key distribution ensures end-to-end encryption of messages exchanged within the group. Only authorized

group members with the necessary keys can decrypt and access the communicated data.

Multi-Party Secure Communication (MPSC) protocols, such as the Messaging Layer Security (MLS) protocol, play a critical role in ensuring secure group communication by facilitating the distribution and management of encryption keys within a group setting. This process is essential for maintaining the confidentiality and integrity of messages exchanged among group members in a secure and efficient manner.

4.2.2 ENCRYPT.PY

```
1 from cryptography.hazmat.primitives.asymmetric import x25519
2 from cryptography.hazmat.primitives.kdf.hkdf import HKDF
3 from cryptography.hazmat.primitives import hashes
4 from cryptography.hazmat.primitives import serialization
5 import shlex
6 import os
7
8 # TreeKEM
9 class TreeNode:
10     def __init__(self):
11         self.children = []
12         self.group_key = None
13
14     def add_member_TreeNode(self, new_public_key):
15         new_node = TreeNode()
16         new_node.public_key = new_public_key
17         self.children.append(new_node)
18         self.update_key_TreeNode()
19
20     def remove_member_TreeNode(self, public_key_to_remove):
21         for child in self.children:
22             if child.public_key == public_key_to_remove:
23                 self.children.remove(child)
24                 self.update_key_TreeNode()
25                 break
26
27     def update_key_TreeNode(self):
28         self.group_key = self.generate_group_key_TreeNode()
29
30     def generate_group_key_TreeNode(self):
31         if not self.children:
32             return os.urandom(32)
33
34         combined_key = b''.join(child.generate_group_key_TreeNode() for child in self.children)
35         return HKDF(
36             algorithm=hashes.SHA256(),
37             length=32,
38             salt=os.urandom(16),
39             info=b'',
40         ).derive(combined_key)
41
42     def print_tree(self, level=0):
43         if not self.children:
44             print(" " * level + f"Leaf Node Level {level}: Public Key: {self.public_key.public_key.hex()}"
45             else:
46                 print(" " * level + f"Node Level {level}: Group Key: {self.group_key.hex()}")
47         for child in self.children:
48             child.print_tree(level + 1)
```

Figure 12: A code snippet that shows the basic TreeKEM function implemented inside “encrypt.py”

“encrypt.py” and “decrypt.py” are both designed to facilitate key distribution and encryption/decryption processes for secure messaging using the Messaging Layer Security (MLS) protocol. Here are the functions described inside “encrypt.py” (Fig. 12):

4.2.2.1 KEY DISTRIBUTION FUNCTIONS

- `aes_encrypt(data, key)`: Encrypts data using AES encryption with a given key.
- `save_derived_key(derived_key, filename)`: Saves a derived key to a file.
- `encrypt_long_message(message, derived_key)`: Encrypts a long message using a derived key.
- `encrypt_photo(photo_path, derived_key)`: Encrypts a photo file using a derived key.
- `encrypt_photo_with_watermark(photo_path, derived_key, sender_id, receiver_id)`: Encrypts a photo file with a watermark added, using AES encryption with a derived key.

- `encrypt_and_save_text(root, message)`: Encrypts and saves a text message using the group key from a given root node.
- `encrypt_and_save_photo(root, photo_path)`: Encrypts and saves a photo file using the group key from a given root node.

4.2.2.2 TREEKEM CLASS

- `TreeNode`: Represents a node in a tree structure where group keys are updated and maintained.
- `add_member_TreeNode(new_public_key)`: Adds a member with a new public key.
- `remove_member_TreeNode(public_key_to_remove)`: Removes a member based on their public key.
- `update_key_TreeNode()`: Updates the group key based on the current node structure.
- `generate_group_key_TreeNode()`: Generates a group key based on the children's keys.
- `print_tree(level)`: Prints the tree structure with group keys and public keys.

4.2.2.3 MAIN FUNCTION

1. **User Interaction**: The main function prompts the user to specify whether they want to encrypt text or a photo and the number of members in the group. For each member, it generates a random private key, computes the public key, and adds the member to the group represented by a tree structure.

2. **Key Distribution & Derivation**: The main function utilizes the TreeKEM class to manage the group key distribution within the tree structure. It also updates the group key based on the current structure of the tree whenever a member is added or removed. It also utilizes the HMAC-based Key Derivation Function (HKDF) to derive keys for encryption and decryption processes, ensuring secure key generation from a shared secret.

3. **AES Encryption and Decryption**: Both files implement functions for AES encryption and decryption, allowing data to be securely encrypted using a symmetric encryption algorithm.

4. **x25519 Key Exchange**: The files leverage the x25519 key exchange algorithm for secure key exchange between parties, enabling the sharing of secret keys for symmetric encryption.

Overall, “encrypt.py” initiates key distribution, encryption, and user interaction for text and photo encryption. It also handles adding or removing members from the group and re-encrypting data accordingly.

4.2.3 DECRYPT.PY

```
1 from cryptography.hazmat.primitives.asymmetric import rsa
2 from cryptography.hazmat.primitives.serialization import load_der_private_key
3 from cryptography.hazmat.primitives import serialization
4 from cryptography.hazmat.primitives.kdf.hkdf import HKDF
5 from cryptography.hazmat.primitives import hashes
6
7 def load_derived_key(filename):
8     with open(filename, "rb") as file:
9         derived_key = file.read()
10        return derived_key
11
12 def decrypt_message(encrypted_chunks, derived_key):
13     decrypted_message = b""
14     for encrypted_chunk in encrypted_chunks:
15         decrypted_chunk = bytes([char ^ derived_key[i % 32] for i, char in enumerate(encrypted_chunk)])
16         decrypted_message += decrypted_chunk
17     decrypted_message_text = decrypted_message.decode('latin-1')
18     return decrypted_message_text
19
20 def decrypt_photo(encrypted_chunks, derived_key):
21     decrypted_photo = b""
22     for encrypted_chunk in encrypted_chunks:
23         decrypted_chunk = bytes([char ^ derived_key[i % 32] for i, char in enumerate(encrypted_chunk)])
24         decrypted_photo += decrypted_chunk
25     return decrypted_photo
26
27 def decrypt_and_save_text():
28     derived_key = load_derived_key("derived_key.bin")
29     with open("encrypted_messages.txt", "rb") as file:
30         encrypted_chunks = []
31         while chunk := file.read(32):
32             encrypted_chunks.append(chunk)
33     decrypted_message = decrypt_message(encrypted_chunks, derived_key)
34     print("Decrypted message: ", decrypted_message)
35
36 def decrypt_and_save_photo():
37     derived_key = load_derived_key("derived_key.bin")
38     with open("encrypted_photo.bin", "rb") as file:
39         encrypted_chunks = []
40         while chunk := file.read(32):
41             encrypted_chunks.append(chunk)
42     decrypted_photo = decrypt_photo(encrypted_chunks, derived_key)
43     with open("decrypted_photo.png", "wb") as file:
44         file.write(decrypted_photo)
45     print("The photo has been decrypted and saved as 'decrypted_photo.png'.")
46
47 if __name__ == "__main__":
48     choice = input("What would you like to decrypt? Enter 'text' or 'photo': ").strip().lower()
49
50     if choice == 'text':
51         decrypt_and_save_text()
52     elif choice == 'photo':
```

Figure 13: A code snippet that shows the basic decryption function implemented inside “decrypt.py”

“decrypt.py” decrypts the encrypted data back to its original form, providing a secure messaging layer that ensures confidentiality and integrity of the communicated content. Here are some of the main functions inside “decrypt.py” (Fig. 13):

4.2.3.1 DECRYPTION FUNCTIONS

- `load_derived_key(filename)`: Loads a derived key from a file.
- `aes_decrypt(encrypted_data, key)`: Decrypts data encrypted with AES encryption using a given key.
- `decrypt_message(encrypted_chunks, derived_key)`: Decrypts and reconstructs an encrypted message.
- `decrypt_photo(encrypted_chunks, derived_key)`: Decrypts an encrypted photo.
- `extract_watermark(photo_path)`: Extracts a watermark from a photo file.

4.2.3.2 DECRYPTION AND SAVE FUNCTIONS

- `decrypt_and_save_text()`: Loads the derived key and encrypted text, then decrypts and displays the message.
- `decrypt_and_save_photo()`: Loads the derived key and encrypted photo, then decrypts and saves the photo.

4.2.3.3 MAIN FUNCTIONS

1. User Interaction: Prompts the user to choose whether they want to decrypt text or a photo.
2. Key Retrieval: Loads the derived key from the file saved during encryption.
3. AES Decryption Function: Decrypts data encrypted with AES using a provided key and initialization vector.
4. Output Handling: For text decryption, reconstructs the decrypted message and displays it. For photo decryption, reconstructs the decrypted photo and saves it as a PNG file.
5. Watermark Extraction: The file includes a function to extract watermarks from decrypted photo files, enabling additional information retrieval from watermarked images.

Overall, “decrypt.py” prompts the user to choose between decrypting text or a photo and calls the corresponding decryption function based on the user's choice.

4.2.4 WORKFLOW OVERVIEW & RESULTS

Encryption Process: Users choose the type of data (text or photo) they want to encrypt. Random private keys are generated for each member, and the group key is derived and updated accordingly in the tree structure. Data is chunked, encrypted using the derived key, and saved in encrypted form along with the derived key for future decryption.

Decryption Process: Users choose the type of data (text or photo) they want to decrypt. The script loads the derived key used for encryption. Encrypted data is read and decrypted using the derived key. Decrypted text or photo is reconstructed and displayed or saved based on the user's choice.

Overall, the result showcases demonstrate a secure text and photo encryption and decryption system with watermarking functionality, showcasing group key management, member addition and removal, and the extraction of watermark information during the decryption process (Fig. 14 and Fig. 15).

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(.venv) lifehater@LifedeMacBook-Pro FYP % python3 encrypt.py
What would you like to encrypt? Enter 'text' or 'photo': text
Number of members in the group: 3
Please enter the text message you want to encrypt: This is a test to see if encrypt.py is working perfectly.
Encrypted Message chunks:
Chunk 1: b"\x02J\xcd\x05P\x95c\xbc\x07\x07\x0a0N6iw\x89\x02\x09a\xfc*\xfde\x06.\x88H\x04cZ\x099U\x02\x08c\x94\x09f9)\x04e4\x09\x040\tJ'\x05=FJ\x0aa\x01\x0ea\x0dc\x0b\x01\x0df\x0bd\x011\x08b
\x09b\x0f8E\x083!\x01\x0dc\x04c!\x01\x05\x0ad\x0f6m\x05"
Node Level 0: Group Key: a61idd0e4c9e2181d119ff986cd4d3d385b7aa9400d2571d10abab5fa07cfff6
Leaf Node Level 1: Public Key: e6e17d7959397df52b2e35d138d5a1373faf05ca646705e7e92904a4e4d9b138
Leaf Node Level 1: Public Key: add67e533be4af9019b47953e66436076b20bf6bbae5977d0a875af00bb4c22b
Leaf Node Level 1: Public Key: 84ed37cea804c71861bda3e6469857e983c487bf4b01af0d39f81359d5b74716
Would you like to add or remove members? (add/remove/no): add
New member added.
Encrypted Message chunks:
Chunk 1: b"\x00\x02L\x0d9\xdf\x00 \x00\x02L\x05P\x95c\xbc\x07\x07\x0a0N6iw\x89\x02\x09a\xfc*\xfde\x06.\x88H\x04cZ\x099U\x02\x08c\x94\x09f9)\x04e4\x09\x040\tJ'\x05=FJ\x0aa\x01\x0ea\x0dc\x0b\x01\x0df\x0bd\x011\x08b
\x09b\x0f8E\x083!\x01\x0dc\x04c!\x01\x05\x0ad\x0f6m\x05"
Node Level 0: Group Key: bb482a7464acb203e0c635685e2b7af4d4408e65f9ba731a1324a24734e0547a9
Leaf Node Level 1: Public Key: e6e17d7959397df52b2e35d138d5a1373faf05ca646705e7e92904a4e4d9b138
Leaf Node Level 1: Public Key: add67e533be4af9019b47953e66436076b20bf6bbae5977d0a875af00bb4c22b
Leaf Node Level 1: Public Key: 84ed37cea804c71861bda3e6469857e983c487bf4b01af0d39f81359d5b74716
Leaf Node Level 1: Public Key: a7a4bd1f91e835ed0eab96d5f1c71d75e366dc24e1216ab1462ef1d6e3b8f50c
(.venv) lifehater@LifedeMacBook-Pro FYP % python3 decrypt.py
What would you like to decrypt? Enter 'text' or 'photo': text
Decrypted message: This is a test to see if encrypt.py is working perfectly.
(.venv) lifehater@LifedeMacBook-Pro FYP %
```

Figure 14: A sample run to test text encryption

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(.venv) lifehater@LifedeMacBook-Pro FYP % python3 encrypt.py
What would you like to encrypt? Enter 'text' or 'photo': photo
Number of members in the group: 2
Please enter the path of the photo you want to encrypt: '/Users/lifehater/Downloads/FYP/Bron.jpg'
Would you like to add a watermark? (yes/no): yes
Please enter the sender's identity: Sender1
Please enter the receiver's identity: Receiver1
The encrypted photo has been saved to 'encrypted_photo.bin'.
Node Level 0: Group Key: 6bb420d8ef611b965d4eaf8f03a079e52c4e013bf7a132595f405db0f09f2f7d
Leaf Node Level 1: Public Key: fdc58c94a9a11e787b3a54adbc8379e2405bf3e402d4fa7ee06b27be8cc7e37b
Leaf Node Level 1: Public Key: b0e8ea741d265bbac5b79587b90be43ca0af4337216aaaffb25750299870c17
Would you like to add or remove members? (add/remove/no): remove
Enter the index of the member to remove (starting from 0): 1
Member removed.
The encrypted photo has been saved to 'encrypted_photo.bin'.
Node Level 0: Group Key: 9ffaab2cbf73381390a0690fb344dbe1e85ecd5fc1668cb5da7f3f01fb9ed11
Leaf Node Level 1: Public Key: fdc58c94a9a11e787b3a54adbc8379e2405bf3e402d4fa7ee06b27be8cc7e37b
(.venv) lifehater@LifedeMacBook-Pro FYP % python3 decrypt.py
What would you like to decrypt? Enter 'text' or 'photo': photo
The photo has been decrypted and saved as 'decrypted_photo.png'.
Extracted watermark information: Sender1:Receiver1
(.venv) lifehater@LifedeMacBook-Pro FYP %
```

Figure 15: A sample run to test photo encryption

4.2.5 USE CASE

The MLS Key Distribution algorithm is ideal for scenarios where multiple parties need to exchange sensitive information securely. It provides a robust framework for secure messaging, allowing users to communicate confidentially while ensuring that only authorized parties can access the decrypted content. It has several advantages:

1. End-to-End Security: Ensures end-to-end encryption of messages and files, safeguarding data during transmission and storage.
2. Dynamic Key Management: Supports dynamic addition and removal of members while efficiently updating and distributing group keys.
3. User-Friendly Interface: Simple user prompts enable easy encryption and decryption of text and photo data, enhancing user experience.

To conclude, “encrypt.py” and “decrypt.py” showcases a comprehensive approach to secure messaging, presenting an encryption and decryption system that employs AES encryption for data security, key derivation mechanisms like HKDF for key generation, and x25519 key exchange for secure key sharing. These files also feature functions for reading and writing data to files, enabling the secure storage of encrypted information and derived keys. With support for encrypting both text messages and photo files, these scripts offer a versatile solution for safeguarding various data types. Additionally, the ability to extract

watermarks from decrypted photo files enhances their utility for data verification and integrity maintenance. Overall, these files showcase a robust framework for secure data handling, key management, and communication, incorporating essential encryption techniques and file operations to ensure data confidentiality and integrity in a range of applications.

5. CONCLUSION

Our final report highlights the significance of developing a privacy-preserving instant messaging application to address the escalating concerns surrounding data privacy and security. For the whole semester, my project partner and I have focused on privacy-enhancing cryptography for secure messaging. We developed a plan for a "Privacy-Preserving Instant Messaging" project, explored encryption methods, and finalized our proposal. We emphasize the importance of creating a secure instant messaging application that prioritizes user privacy in response to the growing concerns about data security. We have successfully implemented the Watermark encryption and MLS Key Distribution system to fortify the security of our messages. Currently, we are actively engaged in constructing a messaging application designed to guarantee the confidentiality of communications. Our goal is to enable secure communication by employing robust encryption techniques, safeguarding messages from unauthorized access. We are so excited about how our project can make a positive impact. We're dedicated

to improving our app to meet the ever-changing challenges of keeping data safe.

6. THE SUGGESTION OF FUTURE DIRECTION

Moving forward, our next steps involve incorporating the MLS Key Distribution system into the messaging app and extensively testing its functionality to guarantee secure message exchanges between users. Concurrently, we will embark on enhancing the user interface and experience of the app, concentrating on usability, visual appeal, and simplicity to cater to the needs of our users effectively.

With the MLS Key distribution algorithm and watermark now successfully integrated, our focus shifts towards refining the app's security features, optimizing user experience, and fortifying its overall resilience against potential threats. By prioritizing both usability and security, we are committed to delivering a messaging platform that not only looks great but also offers a safe and trustworthy environment for communication.

7. CONSOLIDATED LOGBOOK

Logbook Entry 2024-09-16

Today, our focus on privacy-enhancing cryptography deepened as we delved into various research avenues within this domain. My group member and I explored distinct facets of this field, aiming to enhance our understanding and identify potential research topics. Currently, we are honing in on the specifics of our research theme, which revolves around implementing a diverse array of cryptographic techniques and security protocols to safeguard messages. These messages encompass a wide range of media types, including text, photos, videos, and more, with the primary objective of restricting access solely to the intended recipient.

Logbook Entry 2024-09-23

This week, my project partner and I finished our plan for a "Privacy-Preserving Instant Messaging" project, focusing on using encryption to keep user chats safe. We investigated different ways to lock up text, images, and videos, from old-school methods like AES to newer tricks to make our messaging system super secure. Now, we're waiting for our supervisor to give us feedback on our plan. Next, we'll dive deeper into securing images and videos, figure out the technical stuff for keeping messages safe, and keep building our Privacy-Preserving Instant Messaging system.

Logbook Entry 2024-09-30

This week, my project partner and I are refining and finalizing our project proposal. We ensured that all key aspects of the project were clearly articulated and aligned

with our objectives. Also, we have done some more in-depth research on secure messaging which involved studying encryption methods, security protocols, and best practices in secure communication.

Logbook Entry 2024-10-07

This week, my project partner and I have finalized our project proposal with our academic advisor. This week was dedicated to an intensive exploration of secure messaging, where we immersed ourselves in the intricacies of encryption methodologies, security protocols, and the essential principles governing secure communication.

Logbook Entry: 2024-10-14

This week, my project partner and I delved into research papers focusing on secure messaging, aligning with our project proposal. Additionally, we kicked off the coding phase by establishing the foundational layer for our SMS app, marking the initial steps in message transmission implementation.

Logbook Entry: 2024-10-21

This week, my project partner and I focused on our secure messaging project. We learned more about encryption algorithms and secure ways to communicate, specifically diving into the MLS Key Distribution system. We continued our work on implementing the MLS Key Distribution system in our messaging app. Our goal was to ensure that messages are protected using strong encryption methods. This approach allows only the sender and receiver to access the messages, keeping them secure from unauthorized access.

Logbook Entry: 2024-10-28

This week, my project partner and I made progress on our secure messaging project by working on the MLS Key Distribution system. We continue our coding for both the MLS Key System and our SMS app. Our next step will be testing different situations to make sure the key exchange was strong and secure, also to check on the speed of the encryption methods to keep the user experience smooth.

Logbook Entry: 2024-11-4

This week, my project partner and I continue our coding for both the MLS Key System and our SMS app. I would say it's half done by now and we will test different situations to make sure the key exchange was strong and secure once we have completed the algorithm.

Logbook Entry: 2024-11-11

This week, my project partner and I continue our coding for both the MLS Key System and our SMS app. It's 70% completed by now and we will conduct various scenarios

to verify the robustness and security of the key exchange after finalizing the algorithm.

Logbook Entry: 2024-11-18

This week, my project partner and I have finished coding for the MLS Key System and now we will move on to the SMS app. We are conducting various scenarios to verify the robustness and security of the MLS key exchange algorithm. And so far no errors are found.

8. REFERENCE

[1] E. Carriere, “Why encryption is mandatory in the Digital age,” Acrobats, <https://acrobats.net/blog/use-cases/encryption-in-the-digital-age/> (accessed Nov. 26, 2024).

[2] Tony Hayes - IT Support and Telecomms, “the importance of data encryption in the Digital age,” LinkedIn, <https://www.linkedin.com/pulse/importance-data-encryption-digital-age-it-support-and-telecomms-g4epc> (accessed Nov. 26, 2024).

[3] “Benefits of encryption: Safeguarding your data in 2024,” Secure Konnect Digital Business & Cyber Security, <https://www.secureconnect.com/benefits-of-encryption-safeguarding-your-data-in-the-digital-world/> (accessed Nov. 26, 2024).

[4] “iMessage with PQ3: The New State of the art in quantum-secure messaging at scale,” Blog - iMessage with PQ3: The new state of the art in quantum-secure messaging at scale - Apple Security Research, <https://security.apple.com/blog/imessage-pq3/> (accessed Nov. 26, 2024).

[5] “iMessage with PQ3 Cryptographic Protocol: Hacker news,” iMessage with PQ3 Cryptographic Protocol | Hacker News, <https://news.ycombinator.com/item?id=39453660> (accessed Nov. 26, 2024).

[6] A. Paavola, “How secure is texting and how businesses can reinforce it,” Textline, <https://www.textline.com/blog/secure-text-messaging> (accessed Nov. 26, 2024).

[7] The Hacker News, “GSMA plans end-to-end encryption for cross-platform RCS Messaging,” The Hacker News, <https://thehackernews.com/2024/09/gsma-plans-end-to-end-encryption-for.html> (accessed Nov. 26, 2024).

[8] “GSMA moves forward with end-to-end encryption for RCS Messaging,” Campaign Registry,

<https://www.campaignregistry.com/gsma-end-to-end-encryption/> (accessed Nov. 26, 2024).

[9] Key Agreement for Decentralized Secure Group Messaging with Strong Security Guarantees - Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, Alastair R. Beresford (2020), <https://eprint.iacr.org/2020/1281.pdf> (accessed Nov. 25, 2024).

[10] CSIDH: An Efficient Post-Quantum Commutative Group Action - Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes (2018), - cryptology ePrint Archive, <https://eprint.iacr.org/2018/383.pdf> (accessed Nov. 25, 2024).

[11] Parakeet: Practical Key Transparency for End-to-End Encrypted Messaging - Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghoshk, Ercan Ozturk, Kevin Lewi, and Sean Lawlor (2023) <https://eprint.iacr.org/2023/081.pdf> (accessed Nov. 25, 2024).

[12] Security analysis of the MLS Key Distribution - Chris Brzuska, Eric Cornelissen, Konrad Kohbrok Aalto University, Finland (2021), <https://eprint.iacr.org/2021/137.pdf> (accessed Nov. 25, 2024).

[13] How to watermark cryptographic functions - Ryo Nishimaki (2013), <https://www.iacr.org/archive/eurocrypt2013/78810105/78810105.pdf> (accessed Nov. 25, 2024).

[14] B. K. Jena, “AES encryption: Secure Data with Advanced Encryption Standard,” Simplilearn.com, <https://www.simplilearn.com/tutorials/cryptography-tutorial/aes-encryption> (accessed Nov. 26, 2024).

[15] “RSA encryption: Brilliant math & science wiki,” Brilliant, <https://brilliant.org/wiki/rsa-encryption/> (accessed Nov. 26, 2024).

[16] Openssl, “Openssl/openssl: TLS/SSL and crypto library,” GitHub, <https://github.com/openssl/openssl> (accessed Oct. 23, 2024).

[17] jedisct1, “Jedisct1/Libsodium: A modern, portable, easy to use crypto library,” GitHub, <https://github.com/jedisct1/libsodium> (accessed Oct. 23, 2024).

The Chinese University of Hong Kong
Academic Honesty Declaration Statement

Submission Details

Student Name	TSOI Ming Hon (s1155175123)		
Year and Term	2024-2025 Term 1		
Course	IERG-4998-IJ01 Final Year Project I		
Assignment Marker	Professor CHOW Sze Ming		
Submitted File Name	1155175123_Final_Report.pdf		
Submission Type	Group		
Assignment Number	5	Due Date (provided by student)	2024-11-28
Submission Reference Number	4099453	Submission Time	2024-11-28 01:12:27

Agreement and Declaration on Student's Work Submitted to VeriGuide

VeriGuide is intended to help the University to assure that works submitted by students as part of course requirement are original, and that students receive the proper recognition and grades for doing so. The student, in submitting his/her work ("this Work") to VeriGuide, warrants that he/she is the lawful owner of the copyright of this Work. The student hereby grants a worldwide irrevocable non-exclusive perpetual licence in respect of the copyright in this Work to the University. The University will use this Work for the following purposes.

(a) Checking that this Work is original

The University needs to establish with reasonable confidence that this Work is original, before this Work can be marked or graded. For this purpose, VeriGuide will produce comparison reports showing any apparent similarities between this Work and other works, in order to provide data for teachers to decide, in the context of the particular subjects, course and assignment. In addition, the Work may be investigated by AI content detection software to determine originality. However, any such reports that show the author's identity will only be made available to teachers, administrators and relevant committees in the University with a legitimate responsibility for marking, grading, examining, degree and other awards, quality assurance, and where necessary, for student discipline.

(b) Anonymous archive for reference in checking that future works submitted by other students of the University are original

The University will store this Work anonymously in an archive, to serve as one of the bases for comparison with future works submitted by other students of the University, in order to establish that the latter are original. For this purpose, every effort will be made to ensure this Work will be stored in a manner that would not reveal the author's identity, and that in exhibiting any comparison with other work, only relevant sentences/ parts of this Work with apparent similarities will be cited. In order to help the University to achieve anonymity, this Work submitted should not contain any reference to the student's name or identity except in designated places on the front page of this Work (which will allow this information to be removed before archival).

(c) Research and statistical reports

The University will also use the material for research on the methodology of textual comparisons and evaluations, on teaching and learning, and for the compilation of statistical reports. For this purpose, only the anonymously archived material will be used, so that student identity is not revealed.

I confirm that the above submission details are correct. I am submitting the assignment for:

☒ [X] a group project on behalf of all members of the group. It is hereby confirmed that the submission is authorized by all members of the group, and all members of the group are required to sign this declaration.

We have read the above and in submitting this Work fully agree to all the terms. We declare that: (i) all members of the group have read and checked that all parts of the piece of work, irrespective of whether they are contributed by individual members or all members as a group, here submitted are original except for source material explicitly acknowledged; (ii) the piece of work, or a part of the piece of work has not been submitted for more than one purpose (e.g. to satisfy the requirements in two different courses) without declaration; and (iii) the submitted soft copy with details listed in the <Submission Details> is identical to the hard copy(ies), if any, which has(have) been/ is(are) going to be submitted. We also acknowledge that we are aware of the University's policy and regulations on honesty in academic work, and of the disciplinary guidelines and procedures applicable to breaches of such policy and regulations, as contained in the University website <http://www.cuhk.edu.hk/policy/academichonesty/>.

We are aware that all members of the group should be held responsible and liable to disciplinary actions, irrespective of whether he/she has signed the declaration and whether he/she has contributed, directly or indirectly, to the problematic contents.

We declare that we have not distributed/ shared/ copied any teaching materials without the consent of the course teacher(s) to gain unfair academic advantage in the assignment/ course.

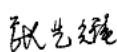
We declare that we have read and understood the University's policy on the use of AI for academic work. We confirm that we have complied with the instructions given by our teacher regarding the use of AI tools for this assignment and consent to the use of AI content detection software to review my submission.

We also understand that assignments without a properly signed declaration by all members of the group concerned will not be graded by the teacher(s).


Signature (TSOI Ming Hon, s1155175123)

28-11-2024
Date

Other Group Members (if any):

Name(s)	Student ID(s)	Signature(s)
ZHANG, Yi Yao	1155174982	

Instruction for Submitting Hard Copy / Soft Copy of the Assignment

This signed declaration statement should be attached to the hard copy assignment or submission to the course teacher, according to the instructions as stipulated by the course teacher. If you are required to submit your assignment in soft copy only, please print out a copy of this signed declaration statement and hand it in separately to your course teacher.