The People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

Yahia Fares University of Medea

Faculty of Science

Department of Mathematics and Computer Science

Graduation Project

Presented for obtaining the Master's Degree in Computer Science

Option: Systems Engineering and Web Technology

THEME

Design and Implementation of a Secure Mobile Messaging Application Based on a Hybrid ECC/AES Cryptosystem

Prepared By:

Lachenani Abdelfetah
&
Kadik Salah Eddine

The jury consists of:

President Mr. HADJIR Ismail

Reviewer Mr. KHELDOUN Ahmed

Supervisor Mr. CHETOUANE Mohamed

Class of 2025/2026

Abstract

This thesis presents the design, implementation, and evaluation of **Payn**, a secure mobile messaging application that provides end-to-end encrypted communication through a hybrid ECC–AES cryptographic framework. The proposed system combines Curve25519 (X25519) for secure key exchange with AES-256-GCM for authenticated message encryption and incorporates a lightweight ratcheting mechanism to achieve forward secrecy and post-compromise security.

The application was developed as a native Android platform using Kotlin and Jetpack Compose and is supported by a server-assisted Node.js backend operating under limited trust assumptions. The proposed architecture aims to provide strong security guarantees while maintaining efficient performance on resource-constrained mobile devices.

Comprehensive security testing, performance evaluation, and comparative analysis against widely deployed secure messaging applications such as Signal and WhatsApp demonstrate the effectiveness of the proposed solution. Experimental results indicate that **Payn** successfully achieves confidentiality, integrity, and secure key management while preserving acceptable computational and communication overhead for real-world mobile environments.

Keywords: End-to-End Encryption, Hybrid Cryptography, Elliptic Curve Cryptography, AES-GCM, Forward Secrecy, Ratcheting Mechanism, Mobile Security, Secure Messaging.

Résumé

Cette thèse présente la conception, l'implémentation et l'évaluation de Payn, une application sécurisée de messagerie mobile qui fournit des communications chiffrées de bout en bout grâce à un framework cryptographique hybride ECC-AES. Le système proposé associe Curve25519 (X25519) pour l'échange de clés sécurisé à AES-256-GCM pour le chiffrement authentifié des messages, et intègre un mécanisme de "ratchet" (cliquet) léger afin de garantir la confidentialité persistante (forward secrecy) et la sécurité après compromission (post-compromise security).

L'application a été développée comme une plateforme Android native en utilisant Kotlin et Jetpack Compose. Elle s'appuie sur un backend Node.js assisté par serveur, fonctionnant sous des hypothèses de confiance limitée. L'architecture proposée vise à fournir de solides garanties de sécurité tout en maintenant des performances efficaces sur des appareils mobiles aux ressources limitées.

Des tests de sécurité complets, une évaluation des performances et une analyse comparative par rapport à des applications de messagerie sécurisée largement déployées telles que Signal et WhatsApp démontrent l'efficacité de la solution proposée. Les résultats expérimentaux indiquent que Payn assure avec succès la confidentialité, l'intégrité et la gestion sécurisée des clés, tout en préservant un coût computationnel et de communication acceptable pour des environnements mobiles réels.

Mots-clés: Chiffrement de bout en bout, Cryptographie hybride, Cryptographie sur les courbes elliptiques, AES-GCM, Confidentialité persistante, Mécanisme de Ratchet, Sécurité mobile, Messagerie sécurisée.

الملخص

تُقدّم هذه الأطروحة تصميم، وتنفيذ، وتقييم تطبيق *Payn*، وهو تطبيق آمن للمراسلة عبر الهاتف المحمول يوفر اتصالات مشفرة من طرف إلى طرف (*End-to-End Encryption*) بالاعتماد على إطار عمل تشفير هجين يجمع بين التشفير بمنحنيات إهليجية وتشفير المعيار المتقدم .(ECC–AES) يدمج النظام المقترح بين منحني Curve25519 (X25519)لتأمين تبادل المفاتيح، وخوارزمية AES-256-GCM لتشفير الرسائل بشكل موثوق، كما يتضمن آلية "مستندة إلى تروس مسننة (Ratcheting Mechanism) "خفيفة الوزن لضمان السرية الأمامية المستقبلية *Forward*) (*Secrecy*والأمان بعد الاختراق.(*Post-Compromise Security*)

تم تطوير التطبيق كمنصة أندرويد أصلية (Native) باستخدام لغة Kotlin وإطار العمل Jetpack Compose، وهو مدعوم بخادم خلفي (Backend) يعتمد على Node.js يعمل تحت فرضيات ثقة محدودة. تهدف هذه البنية المعمارية المقترحة إلى تقديم ضمانات أمنية قوية مع الحفاظ على كفاءة الأداء على الأجهزة المحمولة ذات الموارد المحدودة.

أظهرت نتائج اختبارات الأمن الشاملة، وتقييم الأداء، والتحليل المقارن للتطبيق مع تطبيقات المراسلة الآمنة واسعة الانتشار مثل Signal و WhatsApp فاعلية الحل المقترح. وتشير النتائج التجريبية إلى أن تطبيق *Payn* قد نجح في تحقيق السرية، وسلامة البيانات، والإدارة الآمنة للمفاتيح، مع الحفاظ على تكاليف حوسبة واتصالات مقبولة وفي المتناول بالنسبة لبيئات الهاتف المحمول في العالم الحقيقي.

الكلمات المفتاحية :التشفير من طرف إلى طرف، التشفير الهجين، التشفير بمنحنيات إهليجية، AES-GCM، السرية الأمامية، آلية الراشيت(Ratcheting) ، أمن الهاتف المحمول، المراسلة الآمنة .

Table of Contents

Table of Figures **10**

List of Tables **11**

List of Abbreviations **13**

General Introduction **16**

1. State of the Art and Background **20**

1.1 Introduction 20

Part I: Cryptographic Foundations **21**

1.2 Fundamental Concepts of Cryptography 21

1.2.1 Basic Cryptography Vocabulary 21

1.2.2 Security Objectives 21

1.3 Symmetric Cryptography 22

1.3.1 Principles of Symmetric Encryption 22

1.3.2 Modern Symmetric Algorithms 23

1.3.3 Message Authentication Codes (MAC) 26

1.4 Asymmetric (Public-Key) Cryptography 28

1.4.1 Principles of Asymmetric Cryptography 28

1.4.2 Modern Asymmetric Algorithms 29

1.4.3 Key Exchange Protocols 30

Part II: Hybrid Architectures and Secure Messaging Protocols **32**

1.5 Hybrid Encryption Architecture and Communication Stack 32

1.5.1 Concept of Hybrid Encryption 32

1.5.2 Detailed Hybrid Cryptosystem: ECC/AES 32

1.6 End-to-End Encryption Protocols 34
1.6.1 The Signal Protocol 34
1.6.2 Other E2EE Protocols (Brief Overview) 37
1.7 Communication Layer Protocols 38
1.7.1 TLS (Transport Layer Security): 38
1.7.2 HTTPS (HTTP over TLS): 39
1.7.3 WebSocket over TLS (WSS): 39
1.7.4 WebRTC (for real-time voice/video): 40
Part III: Analysis of Existing Systems and Their Limitations 41
1.8 Comparison of Major Secure Messaging Applications 41
1.8.1 Comparative Criteria 41
1.8.2 Comparison Table 42
1.8.3 Analysis of Key Findings 42
1.9 Security Analysis of E2EE Systems 44
1.9.1 Threat Model 44
1.9.2 Cryptographic Strength 46
1.9.3 Implementation Vulnerabilities 47
1.10 Limitations of Existing Messaging Systems 47
1.10.1 Metadata Exposure 48
1.10.2 Scalability Challenges 48
1.10.3 Trust in Servers (Key Distribution) 48
1.11 Performance Considerations for Mobile Platforms 48
1.11.1 Computational Load on Mobile Devices 49
1.11.2 Network Overhead 49

1.11.3 Scalability and Resource Usage 50

2. Design and Architecture of the Secure Messaging System 55

2.1 Introduction 55

2.2 Specifications and Requirements 57

2.2.1 Functional Requirements 57

2.2.2 Security Requirements 60

2.2.3 Mobile Platform Constraints 61

2.3 High-Level System Architecture 64

2.3.1 Overall Architecture and General System Diagram: 64

2.3.2 Client-Server Model: Roles of the Server and the Client: 65

2.3.3 Communication flow overview: 67

2.4 Cryptographic Protocol Design 68

2.4.1 Long-term Key Generation and Storage (ECC Identity Key): 68

2.4.2 Session Establishment via ECDH : 70

2.4.3 Message encryption/decryption using AES-GCM: 71

2.4.4 Verification of authentication tags: 73

2.5 Key Management and Ratcheting Mechanism 76

2.5.1 Need for Key Rotation: 76

2.5.2 Simple ratcheting mechanism inspired by the Double Ratchet: 77

2.6 Message Format 83

2.7 API Endpoints Design 86

2.7.1 Authentication 86

2.7.2 User Account Management 89

2.7.3 Chat Management 91

2.7.4 Contacts Managements 94

2.7.5 Attachment Management 97

2.8 Conclusion 98

3. Implementation of the Secure Messaging System 100

3.1 Introduction 100

3.2 Development Environment and Tools 101

3.2.1 Mobile Platform 101

3.2.2 Backend Stack 101

3.2.3 Cryptographic Libraries 102

3.3 Implementation of the Cryptographic Core 104

3.3.1 Key Pair Generation and Secure Storage 104

3.3.2 Hybrid Encryption Logic (ECDH + HKDF + AES-256-GCM) 105

3.3.3 Ratcheting Mechanism 105

3.3.4 Message Format Serialization 106

3.3.5 Multi Device Support 107

3.4 Client-Side Implementation 108

3.4.1 User Authentication and Identity Key Upload 108

3.4.2 Contact Management 110

3.4.3 Device Discovery and Public Key Retrieval 113

3.4.4 Sending and Receiving Messages 114

3.4.5 Local Encrypted Storage (SQLCipher) 115

3.5 Server-Side Implementation 117

3.5.1 Push Notifications (FCM) 117

3.5.2 Real-Time Communication 119

3.6 Security Hardening 119
3.6.1 Input Validation 119
3.6.2 No Plaintext Logging Policy 120
3.7 Links to the repositories 120
3.8 Conclusion 121
4. Security Analysis, Performance Evaluation, and Comparative Study 123
4.1 Introduction 123
4.2 Security Analysis 124
4.2.1 Revisiting the Threat Model 124
4.2.2 Attack Resistance Testing 124
4.2.3 Server Compromise Simulation 126
4.2.4 Limitations 126
4.3 Performance Evaluation 127
4.3.1 Cryptographic Latency 127
4.3.2 Resource Consumption (CPU) 128
4.4 Comparative Study 129
4.5 Limitations 131
4.6 Conclusion 132
General Conclusion 133
Bibliography 134

Table of Figures

Figure 1.1 Diagram illustrates Symmetric encryption. 22

Figure 1.2 Diagram illustrates Asymmetric encryption. 28

Figure 1.3 Diagram illustrates hybrid encryption (AES + ECC + ECDH). 33

Figure 2.1: System Components and Communication Architecture. 65

Figure 2.2: Communication Flow Overview 67

Figure 2.3: ECC Identity Key. 69

Figure 2.4: Session Establishment via ECDH. 71

Figure 2.5: Message encryption using AES-GCM. 72

Figure 2.6: Message decryption using AES-GCM. 73

Figure 2.7: Generation of authentication tags. 74

Figure 2.8: Verification of authentication tags. 75

Figure 2.9: General Diagram of the mechanism 82

Figure 3.1: User Register Steps 109

Figure 3.2: Login Screen 110

Figure 3.3: Adding a Contact Screen 111

Figure 3.4: List & Search Contact Screen 112

Figure 3.5: Edit & Delete Contact Screen 113

Figure 3.6: Chat Screen 115

Figure 3.7: Notifications settings 118

Figure 4.1: CPU Usage (Android Studio Profiler) 129

List of Tables

Table 1.1 E2E Protocol Comparison Table 42

Table 1.2 Comparison of key evaluation metrics 52

Table 2.1 Ratchet State Structure 78

Table 2.2 Ratchet Epoch Structure 79

Table 2.3 Message Header Structure 84

Table 2.4 Message Frame Structure 85

Table 2.5 Register API 86

Table 2.6 Login API 87

Table 2.7 Send OTP API 87

Table 2.8 OTP Verification API 88

Table 2.9 Register Device API 88

Table 2.10 Get Current User API 89

Table 2.11 Get User By ID API 89

Table 2.12 Profile Editing API 90

Table 2.13 Search Users API 90

Table 2.14 Update FCM Token API 91

Table 2.15 List Chats API 91

Table 2.16 List Messages API 92

Table 2.17 Get Chat By Id API 92

Table 2.18 Init Chat API 93

Table 2.19 Update Messages In Batch API 94

Table 2.20 Search Contact API 94

Table 2.21 Create Contact API 95

Table 2.22 List Contacts API 95

Table 2.23 Get Contact by Id API 96

Table 2.24 Edit Contact API 96

Table 2.25 Delete Contact API 97

Table 2.26 Upload Attachment API 97

Table 4.1 STRIDE Mapping 124

Table 4.2 E2E Protocol Security and Features Comparison 130

List of Abbreviations

AES Advanced Encryption Standard

AD Associated Data

API Application Programming Interface

E2EE End-to-End Encryption

ECC Elliptic Curve Cryptography

ECDH Elliptic Curve Diffie-Hellman

FCM Firebase Cloud Messaging

GCM Galois/Counter Mode

HKDF HMAC-based Key Derivation Function

JSON JavaScript Object Notation

JWT JSON Web Token

MITM Man-in-the-Middle

MQTT Message Queuing Telemetry Transport

PDU Protocol Data Unit

PFS Perfect Forward Secrecy

REST Representational State Transfer

TLS Transport Layer Security

UI User Interface

X25519 Elliptic Curve Diffie-Hellman over Curve25519

OTP One-Time Password

SQLCipher Encrypted SQLite Database

Keystore Android Keystore System

General Introduction

General Introduction

In today's interconnected world, instant messaging applications have become one of the primary means of communication for individuals, businesses, and organizations. Billions of messages containing personal information, financial data, and confidential communications are exchanged daily through mobile devices. As the volume of digital communication continues to increase, ensuring the confidentiality, integrity, and authenticity of transmitted information has become a critical security challenge.

Problem Statement

Despite the widespread adoption of messaging applications, concerns regarding privacy and security remain significant. Many platforms rely on centralized infrastructures that collect substantial amounts of user metadata, creating potential privacy risks. Furthermore, cyberattacks targeting communication systems continue to grow in sophistication, exposing users to threats such as eavesdropping, impersonation attacks, data breaches, and unauthorized access to sensitive information. Consequently, there is a need for secure messaging solutions capable of providing strong end-to-end encryption while maintaining usability and performance on mobile devices.

Challenges

The development of a secure mobile messaging system involves several technical challenges:

- Providing strong end-to-end encryption with minimal performance overhead on resource-constrained devices.
- Achieving forward secrecy and post-compromise security through efficient key management mechanisms.
- Supporting asynchronous communication and offline message delivery without weakening security guarantees.
- Maintaining acceptable battery consumption, memory usage, and response time.
- Enabling secure multi-device synchronization while preserving user privacy.

Purpose and Objectives

The primary objective of this thesis is to design, implement, and evaluate a secure mobile messaging application that ensures confidential communication through a hybrid ECC–AES cryptographic framework. The specific objectives are:

- To design a lightweight hybrid cryptographic protocol based on Elliptic Curve Cryptography and AES.
- To implement a native Android application supporting end-to-end encrypted messaging.
- To develop a server-assisted backend operating under limited trust assumptions.
- To evaluate the security and performance of the proposed solution.
- To compare the developed system with widely adopted secure messaging applications such as Signal and WhatsApp.

Organization of the Thesis

This thesis is organized into four chapters.

Chapter 1 introduces the background and motivation of the study, presents the research problem, identifies the main challenges, and defines the objectives and methodology.

Chapter 2 describes the design and architecture of the proposed secure messaging system, including the functional requirements, system architecture, hybrid cryptographic protocol, ratcheting mechanism, and message structure.

Chapter 3 details the implementation phase, including the Android client developed using Kotlin and Jetpack Compose, the cryptographic engine, and the backend infrastructure based on Node.js, PostgreSQL, and MQTT.

Chapter 4 presents the security and performance evaluation of the developed system. It includes attack-resistance analysis, performance measurements, comparative evaluation with Signal and WhatsApp, and a discussion of the obtained result.

Chapter I:

State of the Art and Background

1. State of the Art and Background

1.1 Introduction

End-to-end encryption (E2EE) has emerged as a fundamental security standard in modern messaging applications, driven by growing concerns over privacy, mass surveillance, and large-scale data breaches. Unlike traditional communication models where intermediaries—such as service providers—may have access to user data, E2EE ensures that only the communicating users can read the exchanged messages. This paradigm shifts trust away from centralized servers and places cryptographic protection directly at the endpoints, thereby significantly strengthening user privacy in mobile and internet-based communications.

In this chapter, we establish the foundational concepts required for the conception and implementation of a secure messaging application. We begin by introducing the core principles of cryptography, including essential terminology and the primary security objectives. The chapter then provides a comprehensive examination of both symmetric and asymmetric cryptographic systems, which constitute the building blocks of all modern secure communication protocols. A particular focus is placed on hybrid cryptosystems, such as the ECC/AES architecture central to this thesis, which leverages the efficiency of symmetric encryption and the secure key exchange of asymmetric cryptography.

Furthermore, this chapter presents a state-of-the-art review of existing end-to-end encryption protocols and messaging applications. By analyzing their cryptographic foundations, security models, and inherent limitations—particularly concerning metadata exposure and scalability—we establish the context and justification for the contributions of this work [1]. Finally, we consider the specific performance constraints of mobile platforms, which are critical to the design of any practical and user-centric secure messaging solution.

Part I: Cryptographic Foundations

1.2 Fundamental Concepts of Cryptography

1.2.1 Basic Cryptography Vocabulary

Cryptography: The science of securing information using mathematical techniques to protect data from unauthorized access [3].

Plaintext: The original readable message or data before it has been processed by an encryption algorithm [3].

Ciphertext: The encrypted, scrambled, and unreadable form of a message that results from the encryption process [3].

Encryption: The process of converting plaintext into ciphertext using a specific algorithm and a cryptographic key [3].

Decryption: The reverse process of encryption; converting ciphertext back into its original plaintext using the appropriate key [3].

Key: A secret value used by a cryptographic algorithm to govern the transformation of data. The security of the system relies on the secrecy of the key [3].

Session: A temporary and logical communication exchange established between two parties for a specific period of interaction [3].

Protocol: A set of predefined rules and procedures that govern how data is formatted, transmitted, and secured between communicating entities [3].

1.2.2 Security Objectives

Confidentiality: Ensures that sensitive information is accessed only by authorized parties. In your system, this is achieved by encrypting messages so that even if a database is breached, the content remains unreadable [8].

Integrity: Guarantees that data has not been altered, deleted, or tampered with during transmission or storage. The use of HMACs or AEAD ensures that any modification to a message is immediately detected [8].

Availability: Ensures that systems and data are consistently accessible to authorized users when needed. This involves protecting the backend against crashes or denial-of-service (DoS) attacks [8].

Authenticity: The assurance that a message, transaction, or party is who they claim to be. This is verified through digital signatures and the X3DH handshake [8].

Non-Repudiation: Provides mathematical proof that a specific user sent a specific message, preventing them from later denying their actions [8].

1.3 Symmetric Cryptography

1.3.1 Principles of Symmetric Encryption

Symmetric encryption uses a single shared secret key for both encrypting plaintext and decrypting ciphertext, enabling fast bulk data protection in secure channels, as shown in Figure 1.1.

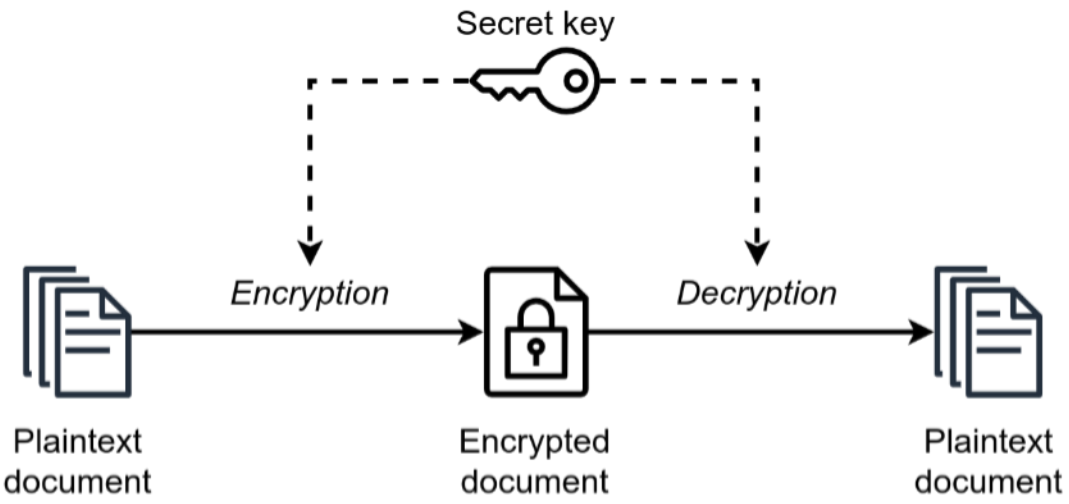


Figure 1.1 Diagram illustrates Symmetric encryption.

1.3.1.1 Core Principles:

The sender and receiver must securely exchange and protect this key beforehand, as its compromise exposes all communications. Algorithms like AES process data in blocks or streams efficiently, suiting high-volume applications[11].

1.3.1.2 Advantages:

Symmetric ciphers outperform asymmetric ones in speed and resource use, ideal for real-time encryption on constrained devices. They handle large data volumes with minimal overhead.

1.3.1.3 Disadvantages

Key distribution poses risks over insecure networks, scaling poorly with more parties.
Lacks built-in authentication or non-repudiation, needing extra mechanisms like MACs.

1.3.2 Modern Symmetric Algorithms

1.3.2.1 Advanced Encryption Standard (AES):

AES, standardized as FIPS 197 by NIST in 2001, is a symmetric block cipher that encrypts 128-bit data blocks using keys of 128, 192, or 256 bits through 10, 12, or 14 rounds of substitution-permutation operations[17].

Core Mechanism:

AES employs the Rijndael algorithm, featuring byte substitution (S-box), row shifting, column mixing, and round-key addition in its SPN structure, rigorously vetted against differential and linear cryptanalysis. This design ensures security margins exceeding 2^100 operations for brute force[11].

Advantages:

- Exceptional security: 256-bit keys resist quantum and classical attacks.
- Versatile efficiency across hardware (e.g., AES-NI instructions) and software[5].
- Ubiquitous in TLS, IPsec, and disk encryption per global standards.

Disadvantages:

- Key management challenges identical to all symmetric ciphers, requiring secure exchange.
- Side-channel vulnerabilities (timing, power analysis) demand constant-time implementations[11].

The modes of AES:

1. AES-GCM (Galois/Counter Mode):

The Modern Gold Standard. GCM is an "Authenticated Encryption with Associated Data" (AEAD) mode. It provides both **confidentiality** (encryption) and **integrity** (verification that the data wasn't tampered with)[12].

How it works: It combines CTR mode (for encryption) with a GHASH function (for authentication).

Best For: Secure web traffic (TLS 1.3), VPNs, and modern application data.

2. AES-CTR (Counter Mode)

The Performance Specialist. CTR mode turns a block cipher into a stream cipher. It generates a "keystream" by encrypting a sequence of incrementing numbers (counters)[13].

How it works: Each block is encrypted independently by XORing the plaintext with the encrypted counter.

Advantages: It is **highly parallelizable**. Since block 2 doesn't depend on block 1, a multi-core CPU can encrypt all blocks simultaneously.

3. AES-CBC (Cipher Block Chaining):

The Legacy Workhorse. CBC was the standard for decades. It uses a "chaining" mechanism where the previous ciphertext block is XORed with the next plaintext block before encryption[14].

How it works: It requires an **Initialization Vector (IV)** to ensure that the same plaintext doesn't produce the same ciphertext twice.

Disadvantages: It is **not parallelizable** (you must finish block 1 to start block 2). It is also vulnerable to "Padding Oracle Attacks" if not implemented with a separate MAC.

1.3.2.2 ChaCha20 Encryption Algorithm

The ChaCha20 algorithm was published in 2008 by Daniel J. Bernstein as a high-performance successor to Salsa20. While DES was a block cipher limited by a 56-bit key, ChaCha20 is a stream cipher that uses a 256-bit key, making it mathematically robust against brute-force attacks by modern supercomputers.

ChaCha20 is designed to provide high security without requiring specialized hardware. This led to its widespread adoption by Google in 2014 to secure Chrome and Android traffic, providing a faster and safer alternative to AES for devices without hardware acceleration[11].

Advantages of ChaCha20:

Software Efficiency: It is significantly faster than AES on mobile devices and low-power IoT hardware.

Side-Channel Resistance: Its "Add-Rotate-XOR" (ARX) design avoids memory lookup tables, protecting against timing attacks.

Standardized Security: It is a mandatory part of the TLS 1.3 protocol, ensuring it remains a cornerstone of modern web encryption.

Disadvantages of ChaCha20:

Nonce Fragility: Security fails completely if a "nonce" (number used once) is reused with the same key.

Lack of Integrity: On its own, it only encrypts; it must be paired with Poly1305 to ensure data hasn't been tampered with (creating an AEAD construction).

Note on Data Flow Type:

Both AES and DES are categorized as block ciphers, as they process data in fixed-size segments (typically 64 or 128 bits). In contrast, stream ciphers encrypt data bit-by-bit or byte-by-byte in a continuous flow, making them highly efficient for low-latency, real-time communication systems where buffering full blocks of data is impractical [5].

1.3.3 Message Authentication Codes (MAC)

1.3.3.1 HMAC (Hash-based Message Authentication Code)

Focus: Integrity & Authenticity:

HMAC uses a cryptographic hash function (like SHA-256) combined with a secret key. It ensures that the message has not been altered and that it truly came from someone who possesses the key. However, HMAC on its own does not provide confidentiality; the message is still "readable" unless encrypted separately (a process often called "Encrypt-then-MAC") [15].

How it works: It hashes the message and the key together twice to prevent "length-extension attacks."

1.3.3.2 AEAD (Authenticated Encryption with Associated Data)

Focus: Confidentiality + Integrity + Authenticity:

AEAD is the modern "all-in-one" solution. Instead of a developer manually combining encryption (like AES) with authentication (like HMAC), AEAD performs both simultaneously. This prevents common security flaws that occur when developers try to "roll their own" encryption schemes[16].

How it works: It encrypts the plaintext while producing a "Tag" (MAC) that verifies the ciphertext and any "Associated Data" (like headers) that must remain unencrypted but protected from tampering.

Popular Examples: AES-GCM (Galois/Counter Mode) and ChaCha20-Poly1305.

1.4 Asymmetric (Public-Key) Cryptography

1.4.1 Principles of Asymmetric Cryptography

Asymmetric cryptography, also known as public-key cryptography, is a cryptographic approach that uses two different but mathematically related keys: a public key for encryption and a private key for decryption. Unlike symmetric cryptography, the secret key does not need to be shared between communicating parties. This model improves security in open networks such as the Internet [5], as shown in Figure 1.2.

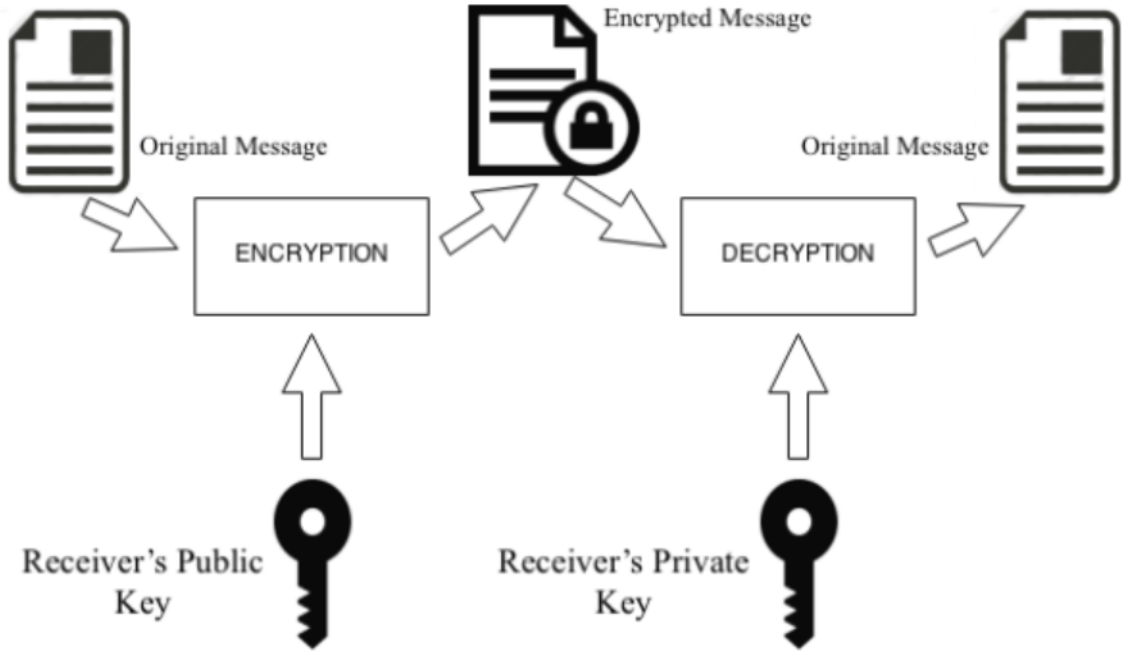


Figure 1.2 Diagram illustrates Asymmetric encryption.

Asymmetric cryptography is widely used for secure key exchange, digital signatures, authentication, and secure communication protocols.

Advantages of Asymmetric Cryptography:

- Solves the key distribution problem present in symmetric systems.
- Enables digital signatures and authentication.
- Provides non-repudiation.
- Suitable for secure communication over insecure networks[5].

Disadvantages of Asymmetric Cryptography:

- Slower than symmetric encryption.
- Requires higher computational resources.
- Not efficient for encrypting large amounts of data.
- More complex mathematical implementatio[5].

1.4.2 Modern Asymmetric Algorithms

1.4.2.1 RSA Algorithm

The RSA algorithm is one of the first and most widely used public-key cryptosystems. It was introduced in 1978 by Ron Rivest, Adi Shamir, and Leonard Adleman.

RSA is based on the mathematical difficulty of factoring large prime numbers. It uses a public key for encryption and a private key for decryption. RSA can be used for both encryption and digital signatures[18].

Advantages of RSA:

- Strong security with sufficiently large key sizes (2048+ bits).
- Widely implemented and standardized.
- Supports digital signatures.

Disadvantages of RSA:

- Large key sizes required for high security.
- Slower compared to modern alternatives like ECC.
- High computational cost on mobile devices.

1.4.2.2 Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC) is a modern public-key cryptographic approach based on the mathematics of elliptic curves over finite fields. ECC provides the same level of security as RSA but with significantly smaller key sizes. For example: 256-bit ECC ≈ 3072-bit RSA (in security level).

Because of its efficiency and lower computational requirements, ECC is particularly suitable for mobile devices and embedded systems[19].

Advantages of ECC:

- Smaller key sizes.
- Faster computations.
- Lower power consumption.
- High security level.

Disadvantages of ECC:

- More complex mathematical structure.
- Implementation errors may lead to vulnerabilities.
- Patent concerns existed in early adoption (mostly resolved now).

1.4.3 Key Exchange Protocols

Diffie-Hellman (DH), ECDH, and KDFs form the backbone of secure key exchange in modern protocols, as detailed in standard cryptography references[5] [11].

1.4.3.1 Diffie-Hellman (DH)

DH enables two parties to compute a shared secret over public channels without prior secrets, relying on discrete logarithm hardness in finite fields[5]. Alice selects private a, sends $g^a \bmod p$; Bob sends $g^b \bmod p$; both derive $(g^b)^a = (g^a)^b \bmod p$ Provides forward secrecy with ephemeral keys[5].

1.4.3.2 Elliptic Curve Diffie-Hellman (ECDH)

ECDH adapts DH to elliptic curves, replacing modular exponentiation with scalar multiplication: private (d), public $(Q = dG)$[19].

$$Shared\ Secret(Z = d_A.Q_B = d_B.Q_A = d_A.d_B.G)$$

A 256-bit curve matches 3072-bit DH security at 1/12th computational cost, ideal for mobile[11].

1.4.3.3 Key Derivation Functions (KDFs)

Raw DH/ECDH outputs require processing via KDFs like HKDF to yield uniform, cryptographically strong session keys[5].

HKDF extracts from low-entropy inputs, expands via HMAC: $HKDFExtract(salt, IKM) \rightarrow PRK \rightarrow Expand(info)$. Ensures domain separation and forward secrecy[5].

Part II: Hybrid Architectures and Secure Messaging Protocols

1.5 Hybrid Encryption Architecture and Communication Stack

Hybrid Encryption Architecture solves symmetric key distribution via asymmetric exchange while pres serving bulk encryption speed. Essential for mobile apps requiring low latency and strong ecurity guarantees[11].

1.5.1 Concept of Hybrid Encryption

Hybrid encryption uses public-key cryptography only to encrypt a small, randomly generated symmetric session key. The recipient recovers this key with their private key, then uses it for fast symmetric encryption of all actual message data[5].

1.5.2 Detailed Hybrid Cryptosystem: ECC/AES

Step 1 - Key Exchange: Sender performs ECDH with receiver's public key parameters to derive shared secret Z.

Step 2 - Key Generation: HKDF processes Z into cryptographically strong AES-256 key.

Step 3 - Data Encryption: AES-GCM encrypts message payloads with authentication tag. Ephemeral ECDH ensures forward secrecy [20]. See Figure 1.3.

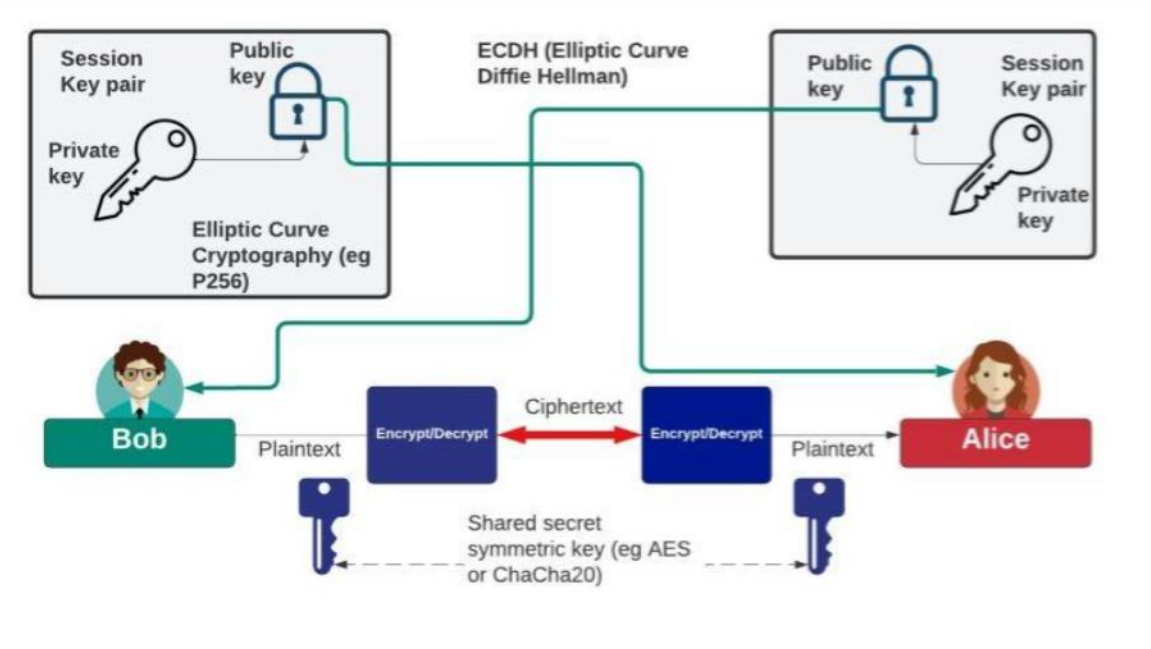


Figure 1.3 Diagram illustrates hybrid encryption (AES + ECC + ECDH).

1.6 End-to-End Encryption Protocols

End-to-End Encryption Protocols are cryptographic systems that secure data so only the sender and recipient can read it. They protect messages, files, or calls from being intercepted by anyone else, including service providers, hackers, or network operators.

1.6.1 The Signal Protocol

The Signal Protocol (formerly known as the TextSecure Protocol) is a non-federated cryptographic protocol that provides end-to-end encryption for voice and instant messaging conversations [21]. The protocol was developed by Open Whisper Systems in 2013 [21] and was introduced in the open-source TextSecure app, which later became Signal. Several closed-source applications have implemented the protocol, such as WhatsApp, which is said to encrypt the conversations of "more than a billion people worldwide [22]" or Google who provides end-to-end encryption by default to all RCS-based conversations between users of their Google Messages app for one-to-one conversations [23]. Facebook Messenger also say they offer the protocol for optional "Secret Conversations", as did Skype for its "Private Conversations"

1.6.1.1 X3DH (Extended Triple Diffie-Hellman):

X3DH allows two parties to establish a shared secret key even if one of them is offline. It "extends" the standard Diffie-Hellman exchange by using a set of pre-calculated keys stored on a server, ensuring that the initial handshake is mutually authenticated and provides "forward secrecy" from the very first message.

Now the following notations are used for further explanation of Extended Triple Diffie-Hellman.

IKA → Aditi’s Identity key

EKA → Aditi’s Ephemeral Key

IKB → Billy's Identity Key

SPKB → Billy's Signed Prekey

OPKB → Billy's One-Time Prekey

The above keys are public key notations, these keys also have their respective private keys, but only public keys are shared over the network. IKA and IKB are the long-term identity public keys. SPKBis Signed prekey which changes periodically when all the one-time pre keys have been deleted. With every protocol run, Aditi generates a new Ephemeral Key EKA and deletes it after forming the secret shared key SK. Extended Triple Diffie-Hellman has 3 phases. Billy publishes different public keys on the server i.e his Identity key IKB,his signed pre key SPKB, his pre key signature Sig(IKB , Encoded(SPKB)) and a set of his one-time pre keys (OPKB1, OPKB2, ...). Server then makes the prekey bundle by choosing one of the one-time keys if it exists. Aditi contacts the server and then she receives a bundle of prekey consisting of values like Billy's Identity key IKB, Billy's signed pre key SPKB, Billy's pre key signature Sig(IKB, Encode(SPKB)) and Billy's onetime prekey OPKB. After receiving the response from the server Aditi verifies Billy's prekey signature and if the signature does not match then she aborts the protocol but if the signature is correct, Aditi will generate a temporary public key EKA. Now below are the calculations for diffie Hellman exchange.

DHI = DeH(IKA , SPKB)

DHII = DeH(EKA , IKB)

DHIII = DeH(EKA , SPKB)

DHIV= DeH(EKA , OPKB)

Shared Key = KDF(DHI || DHII || DHIII || DHIV) where KDF → Key Derivation Function

calculating shared key, Aditi removes her Ephemeral keys and DH Values. Aditi then computes Associate Data which is equals to Encode(IKA) || Encode(IKB). Finally, Aditi sends Billy messages consisting of Aditi's Identity key IKA, Aditi's Ephemeral Key

EKA, Presigned key of Billy, Initiated Ciphertext encrypted with some encryption algorithm like ECC, RSA etc using shared key. Billy receives Aditi's message, he obtains her IKA, EKA, SPKB and encrypted Message, now by using these keys Billy also computes the shared key with the same procedure which was discussed above. Finally Billy tries to decrypt the ciphertext using a shared key, if the message can be decrypted the protocol is complete for Billy. Now Billy deletes a one-time private key that is used by Aditi to achieve perfect forward secrecy. Now E2EE is achieved using Extended Triple Diffie-Hellman and along with it also achieve forward secrecy along with mutual authentication. But still the signal protocol needs another algo. Suppose a scenario when the receiver is offline for a long time and the sender sends messages, these encrypted messages along with public keys stay at the server in order to send to the receiver when the receiver gets online. Since the keys stay there for a long time it gives very good time for the hacker to bruteforce those public keys, it makes the message at risk. There is a need to update those keys on a regular basis. Now to implement this we will use some post X3DH Algorithm i.e. Double Ratchet Algorithm [24]

1.6.1.2 Double Ratchet Algorithm

Double Ratchet Algorithm is used as post X3DH Algorithm which is used to manage session keys to provide E2EE for instant messaging. Trevor Perrin and Moxie Marlinspike developed this algorithm back in 2013 from the non-profit organization called Open Whisper Systems. This algorithm ensures perfect forward secrecy means changing of public keys at every new session and post compromise security. So these properties ensure that every message remains encrypted with a trapdoor function so even if an intruder decrypts a public key, He will not be able to find the early used keys and future public keys that means only a small portion of information leaked even if an attacker anyhow decodes the public keys. Basically it combines two types of properties i.e. symmetric key Ratchet and Diffie-Hellman Ratchet. Its initial name was Axolotl Ratchet, but the developers changed the name into Double Ratchet back in 2016. The main concept of this algorithm is KDF Chain which means Key Derivation Function Chain [24].

1.6.1.3 Cryptographic Primitives Used

In modern secure communication protocols (like the Signal Protocol), cryptographic primitives are the fundamental "building blocks" that provide specific security properties like secrecy, authenticity, and non-repudiation.

Asymmetric Primitives (Identity & Keys):

Curve25519 (X25519): Used for the Elliptic Curve Diffie-Hellman (ECDH) key exchanges in both X3DH and the Double Ratchet.

Ed25519 (EdDSA): Used for identity signatures.

XEdDSA: A special variation of EdDSA used by Signal that allows the same Curve25519 keypair to be used for both Diffie-Hellman and signing (normally you shouldn't mix them, but this math makes it safe).

Symmetric Primitives (The "Heavy Lifting"):

AES-256 (in CBC Mode): Interestingly, Signal traditionally uses AES-CBC rather than GCM for message encryption. Why? It combines it with an HMAC to create its own manual "AEAD" (Encrypt-then-MAC). This offers more flexibility in how they handle "Associated Data" (AD) during the Double Ratchet turns.

HMAC-SHA256: Used to provide the Integrity and Authenticity for the AES-encrypted blocks.

1.6.2 Other E2EE Protocols (Brief Overview)

1.6.2.1 OMEMO (XMPP Extension)

OMEMO is a new encrypted extension of XMPP protocol developed in 2015 that effectively copies the Signal Protocol and adopts it to XMPP. It has been presented to the XMPP Standards Foundation but not yet approved in any official manner.1110 OMEMO builds upon the work of the Signal Protocol as OTR is said to have “interclient mobility problems” and can only work when all conversation participants are online, while OpenPGP “does not provide any kind of forward secrecy and is vulnerable to replay

attacks” . The software implementations of OMEMO are growing such as conversations, an open-source application for Android that counts over 5000 downloads via Google Play Market, and an unknown number of installs via F-Droid[25].

1.6.2.2 Megolm

Megolm. Megolm is used to ensure group communicationsare encrypted. Originally introduced in 2016, before SenderKey (group messaging variant of Signal) was published,Megolm has very similar functionality to Sender Keys. How-ever, Megolm utilizes Olm as opposed to Signal as the securepeer-to-peer channel in order to share Megolm session infor-mation between peers. Additionally, Megolm utilizes a customratchet scheme that can be advanced an arbitrary amountforward while needing at most 1020 hash computations ,which is further described in the Appendix VIII-A. Aside fromthe details in cryptographic primitives between implementa-tions of Sender Keys and Megolm as shown in Table II, thesecure channel protocol, and the ratcheting scheme, all otherdetails between the two protocols remain the same[26].

1.7 Communication Layer Protocols

1.7.1 TLS (Transport Layer Security):

Transport Layer Security (TLS) is a cryptographic protocol designed to provide communications security over a computer network, such as the Internet. The protocol is widely used in applications such as email, instant messaging, and voice over IP, but its use in securing HTTPS remains the most publicly visible.

The TLS protocol aims primarily to provide security, including privacy (confidentiality), integrity, and authenticity through the use of cryptography, such as the use of certificates, between two or more communicating computer applications. It runs in the presentation layer and is itself composed of two layers: the TLS record and the TLS handshake protocols[27].

1.7.2 HTTPS (HTTP over TLS):

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990[28]. **Hypertext Transfer Protocol Secure (HTTPS)** is an extension of HTTP that incorporates **Transport Layer Security (TLS)** to provide confidentiality, integrity, and authentication for data exchanged between clients and servers.

1.7.3 WebSocket over TLS (WSS):

The WebSocket Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code. The security model used for this is the origin-based security model commonly used by web browsers. The protocol consists of an opening handshake followed by basic message framing, layered over TCP. The goal of this technology is to provide a mechanism for browser-based applications that need two-way communication with servers that does not rely on opening multiple HTTP connections (e.g., using XMLHttpRequest or <iframe>s and long polling) [29]. WebSocket Secure (WSS) extends the WebSocket protocol by operating over Transport Layer Security (TLS). By encrypting WebSocket traffic, WSS ensures secure bidirectional communication between clients and servers while providing confidentiality, integrity, and authentication.

1.7.4 WebRTC (for real-time voice/video):

With WebRTC, you can add real-time communication capabilities to your application that works on top of an open standard. It supports video, voice, and generic data to be sent between peers, allowing developers to build powerful voice- and video-communication solutions. The technology is available on all modern browsers as well as on native clients for all major platforms. The technologies behind WebRTC are implemented as an open web standard and available as regular JavaScript APIs in all major browsers. For native clients, like Android and iOS applications, a library is available that provides the same functionality. The WebRTC project is open-source and supported by Apple, Google, Microsoft and Mozilla, amongst others [30].

Part III: Analysis of Existing Systems and Their Limitations

1.8 Comparison of Major Secure Messaging Applications

1.8.1 Comparative Criteria

The comparison is structured around key technical and functional attributes that impact security, usability, and reliability:

- **E2E Encryption Protocol:** The cryptographic protocol used to implement E2E encryption (e.g., Signal Protocol, proprietary variants).
- **Supported Message Types:** Types of content protected by E2E encryption, including text, voice, images, video, and files.
- **Group Chat Support:** Availability of E2E encryption in group conversations, and the maximum supported group size.
- **Key Exchange Protocols:** Mechanisms used for securely exchanging encryption keys (e.g., Diffie-Hellman, X3DH, Double Ratchet).
- **Key Length and Algorithm:** Cryptographic key sizes and algorithms used (e.g., 256-bit AES, Curve25519 for asymmetric encryption).
- **Data Recovery Options:** Mechanisms for message backup or recovery, including encrypted cloud backups or recovery keys.
- **Transport Protocols:** Communication channels employed (e.g., WebSocket, WebRTC) and their implications for real-time messaging.
- **Platform Availability and Adoption:** Number of downloads, user ratings.
- **Additional Security Features:** Optional labels for extra security measures such as forward secrecy, device verification, or self-destructing messages.

1.8.2 Comparison Table

As shown in Table 1.1, Telegram and Facebook Messenger differ in several security and messaging.

Table 1.1 E2E Protocol Comparison Table

	Telegram	**Messenger**
E2E Encryption Protocol	MTProto 2.0	Signal Protocol (X3DH for session setup and Double Ratchet for send message)
Supported Message Types	text, images, videos, voice/video calls & files.	Text messages, Images, Videos, File attachments, Voice messages, Voice & Video calls, Reactions & delivery receipts
E2E Group Chat Support	Not Supported	Supported
Normal Group Chat Support	Supported	Supported
Key Exchange Protocols	Diffie-Hellman protocol	Diffie-Hellman protocol
Key Length and Algorithm	256 bytes with AES256	AES-256-GCM / AES-256-CBC variants
E2E Chat Data Recovery Options	Not supported	Recoverable using PIN or recovery key
Normal Chat Data Recovery Options	Supported	Recoverable using PIN or recovery key
Transport Protocols	MTProto 2.0	MQTT over TLS, WebRTC (for calls)
Platform Availability and Adoption	GooglePlayStore(16.6M reviews, 1B+ downloads)	Google Play Store (~5B+ installs, 100M+ reviews)
Additional Security Feature	Support Perfect Forward Secrecy	Support Forward Secrecy

1.8.3 Analysis of Key Findings

The comparison between Telegram and Facebook Messenger highlights several differences in their security design and messaging features.

First, the encryption protocols used by the two applications are different. Messenger uses the Signal Protocol, which combines X3DH for session establishment

and the Double Ratchet Algorithm for continuous key updates. This protocol is widely recognized for providing strong security and forward secrecy. In contrast, Telegram uses its own proprietary protocol called MTProto.

Another key difference is group chat encryption. Messenger supports end-to-end encryption in group chats, while Telegram only supports E2E encryption in “Secret Chats” between two users. Regular Telegram chats and groups are encrypted between the client and the server, not fully end-to-end.

Both applications use strong cryptographic algorithms such as AES-256 and rely on the Diffie-Hellman key exchange to establish secure keys between users.

Regarding data recovery, Messenger allows users to recover encrypted messages using a PIN or recovery key. Telegram does not allow recovery for Secret Chats, which improves privacy but may lead to permanent message loss if the device is lost.

Finally, Messenger has a much larger user base due to its integration with social media services from Meta Platforms, while Telegram is popular for its privacy-focused features and cloud-based messaging.

Overall, Messenger focuses on standardized and widely audited encryption protocols, while Telegram emphasizes speed, flexibility, and optional end-to-end encryption.

1.9 Security Analysis of E2EE Systems

End-to-end encryption (E2EE) places security at the core of modern digital communication systems by ensuring that only communicating endpoints can access transmitted data. Unlike traditional encryption models where intermediaries may decrypt or process information, E2EE prevents service providers, network operators, and potential attackers from reading message content during transmission or storage. This security model protects confidentiality, preserves data integrity, and strengthens user privacy even in untrusted network environments, making it a fundamental mechanism in contemporary secure communication architectures.

1.9.1 Threat Model

Threat modeling is an important technique used to support security and privacy by design. It helps developers identify potential risks by analyzing how data flows between different components of a system. One commonly used approach is STRIDE, adopted in Microsoft's secure development lifecycle, which systematically identifies security threats affecting system data flows. Since some privacy risks are different from traditional security threats, another methodology called LINDDUN focuses specifically on privacy-related risks such as user identification and loss of anonymity. Both methods analyze systems using data flows and trust boundaries, making them suitable for evaluating environments where sensitive information moves across devices.

1.9.1.1 STRIDE:

In STRIDE, the system is represented using data flow diagrams that describe processes, data storage, and communication paths. Each component and data flow is analyzed according to six main security threats:

- Spoofing
- Tampering
- Repudiation
- Information disclosure
- Denial of service
- Elevation of privilege

1.9.1.2 LINDDUN:

LINDDUN uses a similar modeling approach but focuses on privacy threats [33]. It evaluates system components and data flows according to seven privacy risks:

- Linkability
- Identifiability
- Non-repudiation (privacy risk)
- Detectability
- Information disclosure
- Content unawareness
- Policy and noncompliance

1.9.2 Cryptographic Strength

Protocol Robustness: The security of an end-to-end encryption system strongly depends on the robustness of the underlying cryptographic protocol. Modern E2EE applications typically rely on well-studied protocols such as the Signal Protocol or MTProto, which are designed to resist common cryptographic attacks. Secure protocols combine authenticated key exchange, strong encryption algorithms, and message authentication mechanisms to ensure confidentiality and integrity. Proprietary protocols may also be used, but their security depends on transparency, peer review, and formal analysis.

Key Management: Effective key management is essential for maintaining cryptographic security. Keys must be securely generated using strong randomness sources, safely exchanged between communicating parties, and stored in protected environments such as secure enclaves or encrypted storage. Modern systems also implement periodic key rotation and session keys to reduce long-term exposure in case a device or key becomes compromised.

Forward and Backward Secrecy: Forward secrecy ensures that previously exchanged messages remain secure even if long-term encryption keys are later compromised. Similarly, backward secrecy (also called future secrecy) protects upcoming communications if earlier session keys are exposed. These properties are typically achieved through continuous key updates and ephemeral key exchanges, limiting the amount of data an attacker can decrypt after a security breach.

1.9.3 Implementation Vulnerabilities

1.9.3.1 Client-side weaknesses:

Even when strong end-to-end encryption is used, security can be weakened at the client level. Compromised devices, malware infections, or insecure local storage may allow attackers to access decrypted messages directly on the user's device. Weak authentication mechanisms, poor key protection, or unencrypted backups can also expose sensitive data despite secure transmission.

1.9.3.2 Server-side issues:

Although servers in E2EE systems cannot access message content, they may still handle metadata such as user identities, timestamps, or communication patterns. Improper logging practices, insecure cloud backups, or excessive data retention policies can lead to unintended information disclosure. Metadata exposure may enable traffic analysis or user profiling even without decrypting message contents.

1.9.3.3 Protocol flaws:

Implementation errors or weaknesses in cryptographic protocols can introduce vulnerabilities that undermine overall system security. Incorrect use of encryption primitives, flawed randomness generation, or outdated protocol versions may enable attacks such as replay attacks, key compromise, or impersonation. Continuous security audits, updates, and public cryptographic review are therefore essential to maintain protocol resilience.

1.10 Limitations of Existing Messaging Systems

Although end-to-end (E2E) encryption significantly improves the privacy of mobile messaging, several limitations still exist in current systems.

1.10.1 Metadata Exposure

E2E encryption protects the content of messages but does not fully hide metadata. Metadata includes information such as sender and receiver identities, timestamps, and message frequency. Messaging applications such as WhatsApp, Telegram, and Facebook Messenger may still process this information to route and deliver messages. As a result, communication patterns and user relationships could potentially be inferred through metadata analysis.

1.10.2 Scalability Challenges

Implementing E2E encryption in large-scale systems presents scalability challenges. In group chats, encryption keys must be updated whenever members join or leave the conversation. Managing secure key distribution and synchronization across many users and devices increases system complexity and computational overhead.

1.10.3 Trust in Servers (Key Distribution)

Many E2E messaging systems rely on servers to distribute public keys during the initial communication setup. For example, the Signal Protocol uses servers to deliver users' public keys. If a server is compromised or malicious, it could potentially manipulate keys and enable interception unless users verify key authenticity through security codes or device verification mechanisms[34][35].

1.11 Performance Considerations for Mobile Platforms

End-to-end encryption systems must balance strong security guarantees with acceptable system performance. Encryption, key exchange, and authentication operations introduce computational overhead that can affect device resources, latency, and scalability. Evaluating performance is therefore essential to ensure secure communication without degrading user experience, especially on resource-constrained devices such as smartphones or IoT systems.

1.11.1 Computational Load on Mobile Devices

Computational load refers to the processing cost required to perform cryptographic operations within an end-to-end encrypted system. Encryption and decryption processes, digital signatures, and key exchange mechanisms consume CPU resources and memory, particularly during session establishment. Modern E2EE protocols commonly rely on efficient public-key cryptography such as Elliptic Curve Cryptography [36] and symmetric encryption algorithms like Advanced Encryption Standard to reduce computational overhead while maintaining strong security.

The computational impact is most noticeable during initial key negotiation and frequent key updates required for forward secrecy. Devices with limited processing power or battery capacity may experience increased energy consumption or delays when handling large volumes of encrypted messages. Consequently, optimized cryptographic libraries, hardware acceleration, and lightweight protocol design are important factors in maintaining efficient and scalable encrypted communication systems.

1.11.2 Network Overhead

Network overhead refers to the additional communication cost introduced by end-to-end encryption mechanisms during data transmission. Besides the encrypted message itself, E2EE protocols transmit extra information such as cryptographic headers, authentication tags, public keys, and key exchange parameters. These additional elements increase packet size and bandwidth consumption compared to unencrypted communication.

Secure messaging protocols also require periodic key exchanges and synchronization messages to maintain properties such as forward secrecy and session continuity. For example, protocols implementing mechanisms similar to the Signal Protocol frequently update session keys, which results in additional network traffic. While this overhead improves security, it may introduce higher latency or reduced efficiency in low-bandwidth or unstable network environments.

To minimize performance impact, modern E2EE systems employ compact message formats, efficient serialization techniques, and optimized cryptographic primitives. These optimizations help maintain secure communication while limiting bandwidth usage and ensuring acceptable performance across mobile and large-scale distributed networks.

1.11.3 Scalability and Resource Usage

Scalability in end-to-end encryption systems refers to how well the cryptographic mechanisms perform as the number of users, devices, or group members increases. In one-to-one communication, key management and encryption operations are relatively straightforward. However, in group chats or multi-device scenarios, the system must handle multiple encryption keys, maintain consistent session states, and ensure that all participants can decrypt messages securely.

As group size grows, key distribution and rekeying operations for forward secrecy can introduce significant computational and network overhead. Multi-device synchronization further complicates scalability, since each device requires up-to-date session keys, message histories, and state changes. Improper handling may lead to delays, increased resource consumption, or inconsistent message delivery across devices.

Modern E2EE protocols, like the Signal Protocol, use optimized group key management techniques such as sender keys and hierarchical key tree to reduce the per-device overhead while maintaining security properties. These approaches enable secure communication at scale, although trade-offs between performance, latency, and device resource usage remain important considerations for large or highly dynamic groups[32].

1.11.3.1 Benchmarks / Metrics:

Evaluating the performance of end-to-end encryption systems often involves using benchmarks and metrics to quantify computational efficiency, network overhead, and scalability. Common metrics include:

Encryption/Decryption latency: Time taken to encrypt or decrypt messages, which directly affects user experience.

CPU and memory usage: Resource consumption during cryptographic operations, particularly on mobile or low-power devices.

Bandwidth overhead: Additional network traffic introduced by encryption metadata, key exchanges, and authentication tags.

Message delivery time: Latency from sender to recipient, including delays caused by cryptographic processing or multi-device synchronization.

Scalability metrics: Performance impact as the number of users, devices, or group members increases, including rekeying delays and synchronization costs.

Benchmarks can be conducted in controlled experiments, simulations, or real-world deployments. They allow developers and researchers to identify performance bottlenecks, optimize cryptographic implementations, and balance security with usability. Protocols like the Signal Protocol often provide reference performance numbers that can guide implementation and system design choices, see Table 1.2 [37].

Table 1.2 Comparison of key evaluation metrics

Metrics	WhatsApp	Signal	Slack
Encryption Strength	AES-256 (Signal Prot.)	AES-256 (Signal Prot.)	AES-256 + TLS
Latency Overhead (ms)	50	75	30
User Satisfaction (1–10)	8.5	9.2	7.8
Reported Vulnerabilities	3 (2 years)	1 (2 years)	2 (3 years)
User Adoption Growth (%)	15% (Post-E2EE)	4,200% (2021 Surge)	12% (Post-EKM Launch)

Table Description:

The table presents a quantitative comparison of key evaluation metrics for WhatsApp, Signal, and Slack to demonstrate their security effectiveness, user adoption impact, and performance efficiency:

Encryption Strength:

- WhatsApp and Signal rely on AES-256 encryption combined with the Signal Protocol, ensuring robust end-to-end encryption.
- Slack integrates AES-256 for data at rest and TLS for data in transit, tailored for enterprise environments.

Latency Overhead:

- Signal experiences slightly higher latency (75 ms) due to rigorous cryptographic operations.
- WhatsApp maintains lower latency at 50 ms, balancing performance with encryption.
- Slack, designed for enterprise workflows, achieves the lowest latency at 30 ms.

User Satisfaction:

- Signal leads with a 9.2 satisfaction score, reflecting its privacy-centric features.
- WhatsApp scores 8.5, balancing usability and security.
- Slack achieves 7.8, influenced by its enterprise complexity.

Reported Vulnerabilities:

- Signal reports the fewest issues (1 in 2 years), underscoring its minimal attack surface.
- WhatsApp has resolved 3 issues, demonstrating a need for ongoing vigilance due to its scale.
- Slack recorded 2 vulnerabilities tied to third-party integrations.

User Adoption Growth:

- Signal saw a dramatic 4,200% surge in downloads during the 2021 privacy controversy.
- WhatsApp achieved a 15% growth post-E2EE implementation.
- Slack reported 12% growth following its Enterprise Key Management (EKM) rollout.

Chapter II:

Design and Architecture of the Secure Messaging System

2. Design and Architecture of the Secure Messaging System

2.1 Introduction

The objective of this chapter is to present the design and high-level architecture of the proposed secure messaging system. It aims to define the functional and non-functional requirements, describe the overall system architecture, detail the cryptographic protocol, explain the key management and ratcheting mechanisms, define the message format, and provide a security analysis of the proposed solution. This chapter lays the theoretical and technical foundation necessary before proceeding to the implementation phase.

The rapid increase in digital communications has highlighted critical concerns regarding user privacy and message security. Many existing messaging applications rely on centralized servers that can potentially access the content of users' messages, exposing them to risks such as mass surveillance, data breaches, and man-in-the-middle attacks. To address these issues, this project proposes a secure end-to-end encrypted messaging system based on a hybrid ECC/AES cryptographic approach. The system combines Elliptic Curve Cryptography (ECC), particularly Curve25519 (X25519) for key exchange and identity management, with AES-256-GCM for efficient and authenticated encryption of messages and attachments. This hybrid design provides a strong balance between security, performance, and energy efficiency, making it well-suited for resource-constrained mobile devices.

The proposed architecture adopts a client-server model with minimal trust in the server. The mobile client application performs all cryptographic operations locally on the user's device, including key generation, session establishment, message encryption, decryption, and ratcheting. The server is responsible only for lightweight tasks such as user registration, public key distribution, message routing, and temporary storage of encrypted messages for offline delivery. This design ensures that the server never has

access to plaintext messages or private keys, thereby guaranteeing true end-to-end encryption while maintaining practical usability features such as reliable message delivery even when users are offline.

2.2 Specifications and Requirements

2.2.1 Functional Requirements

2.2.1.1 User Account Management:

The system shall provide the following functionalities for user account creation, access, and basic profile management, as reflected in the application's onboarding and main screens:

Create an Account: The system shall prompt the user to select their country code and enter their phone number in international format. Upon submission, a one-time activation code shall be sent via SMS to the provided number then enter the received verification code. After successful phone number verification, the user shall be directed to the profile setup screen where they are required to enter their first name and last name. Users shall have the option to add a profile photo. Once the profile information is completed, the application shall generate the ECC identity key pair locally on the device and upload only the public identity key to the server, associating it with the verified phone number.

Profile Editing: Users shall be able to view and edit their profile information (first name, last name, and profile photo) from the Settings screen.

Login / Re-authentication: On subsequent app launches (or after logout), users shall re-enter their phone number and receive a new SMS verification code to regain access. After successful verification, the application shall restore the local encrypted chat history, sessions, and profile data.

Logout: Users shall be able to log out from the Settings screen. Upon logout, the current session shall be terminated and the user shall be redirected to the login screen.

Search Users:Users shall be able to search for other users by phone number, first name, last name..etc, the search include also users which aren’t in contact list or chat list of current user.

2.2.1.2 Sending and Receiving Messages (Text, Attachments):

The system shall support seamless sending and receiving of E2EE messages, including both text and various types of media attachments, as the core functionality of the **Payn** application.

Text Messaging: Users shall be able to send text messages to individual contacts. Real-time message delivery shall be supported when the recipient is online. When the recipient is offline, messages shall be temporarily stored on the server and delivered as soon as the recipient comes online.

Media and File Attachments: Users shall be able to attach and send different types of media and files, including: Images, Videos, Audio recordings, Documents and other files. Media attachments shall be compressed appropriately on the sender's device to optimize bandwidth and battery usage before encryption and transmission.

Message Encryption and Decryption: All text messages and attachments shall be encrypted locally on the sender's device using the hybrid ECC/AES cryptographic protocol before transmission. Only the intended recipient shall be able to decrypt and view the message content on their device. The server shall never have access to the plaintext content of messages or attachments.

Receiving Messages: Upon opening the chat, the received messages and attachments shall be automatically decrypted and displayed in chronological order. Users shall be able to view images and videos, and download other file types to their device storage.

Basic Message Management: Users shall be able to scroll through chat history and view previously exchanged messages (stored locally in encrypted form). The system shall support basic interactions such as copying message text or saving received attachments.

2.2.1.3 Contact management:

The contact management module defines the functionalities related to handling user contacts within the application. The system supports the following actions:

Add Contact: The user can add a new contact using a phone number or unique identifier.

List Contacts: The user can view a list of all saved contacts.

View Contacts: The user can access and browse the list of saved contacts.

Search Contacts: The user can use a search function to quickly find a specific contact.

Edit Contact: The user can modify contact information (e.g., display name).

Delete Contact: The user can remove a contact from the list.

2.2.1.4 Delivery and reading notifications:

This functionality allows the user to stay informed about message activity and interaction status.The user can:

Receive New Message Notifications: The user can receive a notification when a new message is received.

Receive Read Notifications: The user can be notified when a sent message has been read by the recipient.

View Message Status Indicators: The user can view the status of messages (e.g., sent, delivered, read) through visual indicators.

Manage Notification Settings: The user can enable, disable, or customize notifications according to their preferences.

This functionality ensures that the user is aware of message exchanges while maintaining control over notification behavior.

2.2.2 Security Requirements

2.2.2.1 E2EE, confidentiality, integrity, authenticity:

End-to-End Encryption (E2EE): All messages (text, images, videos, audio, and files) shall be encrypted on the sender's device and only decrypted on the recipient's device. The server shall never access plaintext data. Achieved using: Hybrid ECC + AES-GCM encryption with keys derived via ECDH.

Confidentiality: The content of all communications shall remain secret and accessible only to the intended participants, even if network traffic is intercepted or the server is fully compromised. Achieved using: AES-256 in GCM mode for symmetric encryption combined with Curve25519 for key agreement.

Integrity: The system shall ensure that messages and attachments cannot be modified in transit without detection. Any tampering shall cause the message to be rejected. Achieved using: AES-GCM authentication tag (16-byte MAC) and HMAC-SHA256 for additional integrity protection where needed.

Authenticity: Recipients shall be able to verify that a message truly originates from the claimed sender and has not been forged or modified by a third party. Achieved using: **AES-GCM authentication tags (Auth Tags)**, which provide cryptographic assurance of message authenticity and integrity.

2.2.2.2 Forward Secrecy and Future Secrecy:

Forward Secrecy: Compromise of long-term private keys shall not allow decryption of previously exchanged messages. Achieved using: Ephemeral ECDH key exchange (X25519) during session establishment, with frequent ratcheting of session keys.

Future Secrecy: Even if a current session key is compromised, all subsequent messages shall remain secure. Achieved using: A simplified Double Ratchet-inspired mechanism with symmetric-key ratcheting and periodic ECDH ratchet steps.

2.2.2.3 Participant authentication (MITM prevention):

This requirement ensures that communication occurs only between legitimate participants and prevents man-in-the-middle attacks.

Account Verification: The identity of each user must be verified during account creation using a **one-time passcode (OTP)** sent to their phone number.

Authenticated Communication: Only verified users are allowed to participate in message exchanges within the application.

End-to-End Encryption Enforcement: Communication between participants must be secured using end-to-end encryption, ensuring that only the intended parties can access message content.

Secure Key Exchange: Cryptographic mechanisms must be used to establish secure keys between participants, preventing interception or impersonation.

MITM Attack Prevention: The combination of user verification and E2EE techniques must ensure that no unauthorized entity can intercept or alter the communication.

This requirement guarantees that both the identity of users and the confidentiality of their communications are securely established and maintained.

2.2.3 Mobile Platform Constraints

The development of a secure messaging application for mobile devices must take into account the inherent limitations of smartphones, particularly in terms of resources and network conditions. The following subsections detail the main platform constraints considered in the design.

2.2.3.1 CPU and Battery Optimization:

Mobile devices operate on limited battery capacity and processing power. Cryptographic operations, if not carefully optimized, can lead to excessive CPU usage, rapid battery drain, and device overheating. To address these constraints, the system shall:

- Utilize lightweight elliptic curve cryptography, specifically Curve25519 (X25519) for key agreement and Ed25519 for signatures, which offer high security with significantly lower computational cost compared to traditional curves.
- Rely on hardware-accelerated AES-256-GCM for message encryption and decryption.
- Minimize the frequency of expensive operations such as full ECDH ratchet steps by implementing an efficient symmetric ratcheting mechanism for subsequent messages.
- Use HKDF-SHA256 for key derivation instead of more resource-intensive alternatives.
- Compress media attachments before encryption to reduce both CPU load and data transmission time.

These measures ensure that the application remains responsive and energy-efficient, providing acceptable battery consumption even during intensive messaging sessions.

2.2.3.2 Secure local storage:

The user’s sensitive data, such as messages and encryption keys, must be stored securely on the device.

- All data must be encrypted to protect it at rest.
- The local database (e.g., SQLite) shall be encrypted using SQLCipher with a key derived from the user's authentication secret.
- Cryptographic keys must be stored in secure platform storage (e.g., KeyStore on Android, Secure Enclave on iOS).
- Only the application and authenticated user can access stored data.
- Measures must protect data if the device is lost, stolen, or compromised.

This ensures that the user’s information stays safe and private on the device.

2.3 High-Level System Architecture

2.3.1 Overall Architecture and General System Diagram:

The proposed secure messaging system follows a client-server architecture. The system is composed of three main components:

- **Mobile Application (Client):** The core component running on the user's Android device.
- **Server:** A minimal-trust backend server.
- **Database:** A persistent storage layer on the server side.

As shown in Figure 2.1.

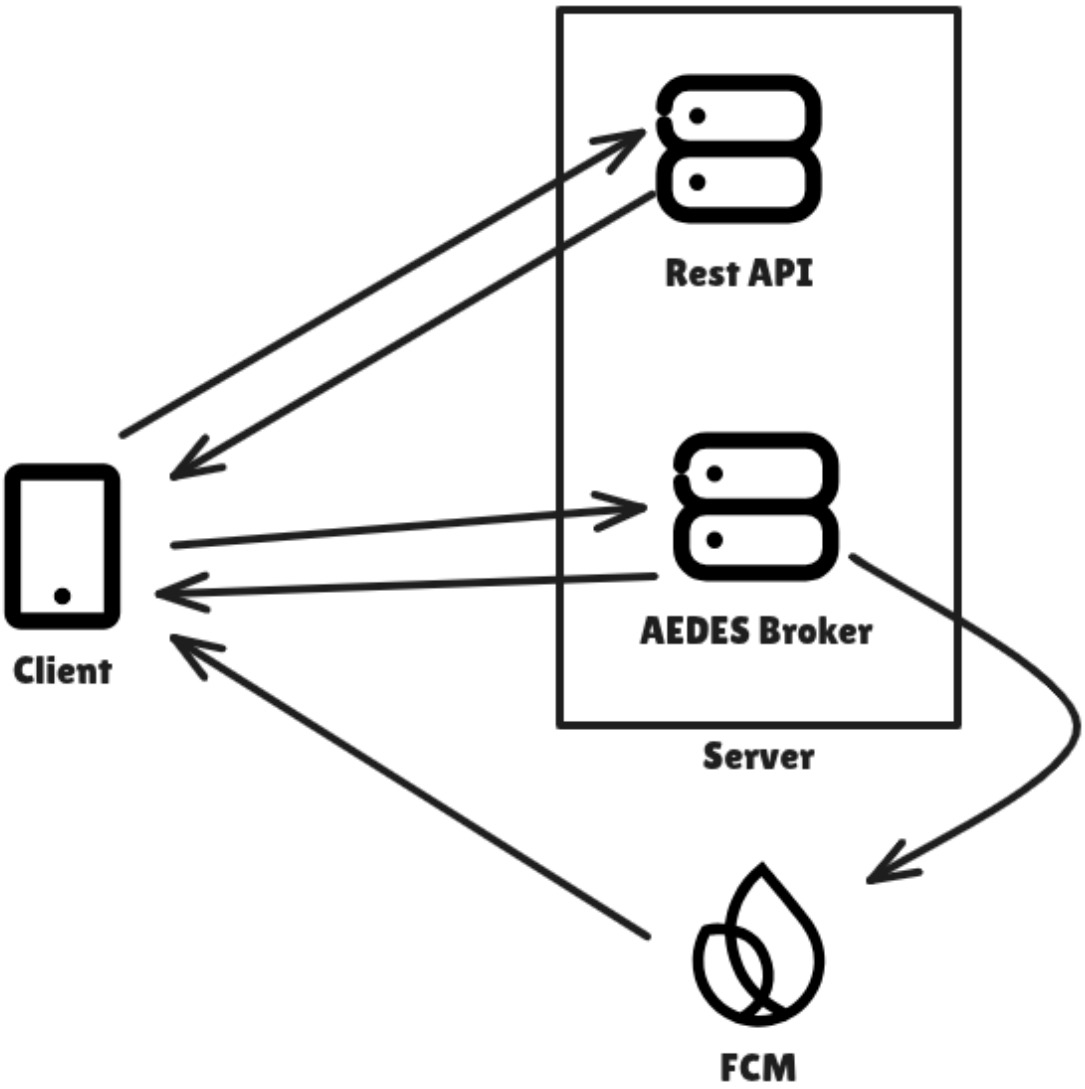


Figure 2.1: System Components and Communication Architecture.

2.3.2 Client-Server Model: Roles of the Server and the Client:

The secure messaging system adopts a **client-server model** where responsibilities are clearly divided to maximize security while ensuring reliable functionality.

2.3.3 Roles of the Client (Mobile Application):

- Generating and storing the long-term ECC identity key pair locally on the device.
- Performing ECDH key exchanges and managing session keys.
- Encrypting and decrypting all messages and attachments using the hybrid ECC/AES protocol.
- Handling the full Double Ratchet-inspired mechanism for forward and future secrecy.
- Managing secure local storage of private keys, session states, and encrypted chat history.
- Composing, sending, and displaying messages in the user interface.
- Verifying the authenticity of received messages and authentication tags.

2.3.4 Roles of the Server:

- Handling user registration and phone number verification (SMS OTP).
- Storing and distributing public identity keys only (never private keys).
- Routing encrypted messages between connected clients.
- Delivering push notifications to inform users of new incoming messages.
- Managing basic delivery receipts (sent, delivered, read) without accessing message content.
- Facilitating contact discovery by allowing users to find registered contacts based on phone numbers.

2.3.3 Communication flow overview:

The communication flow follows these steps:

- The user composes a message in the mobile application.
- The client encrypts the message using the appropriate cryptographic keys.
- The encrypted message is transmitted to the server over a secure connection.
- The server routes the encrypted message to the intended recipient without accessing its contents.
- The recipient's client receives the encrypted message and decrypts it locally.
- The recipient can read the message in plaintext, ensuring complete confidentiality of the communication.

See Figure 2.2, This architecture guarantees security, data privacy, and system scalability.

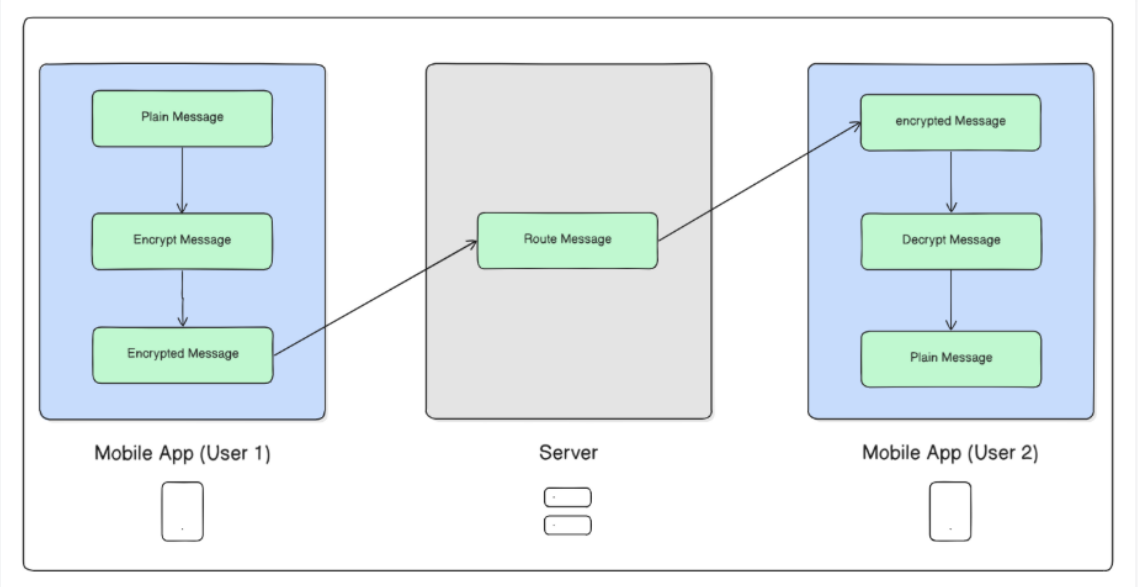


Figure 2.2: Communication Flow Overview

2.4 Cryptographic Protocol Design

The cryptographic protocol of the secure messaging system is based on a **hybrid ECC/AES** approach. It combines elliptic curve cryptography for key agreement and authentication with symmetric encryption for efficient message protection. This section details the core cryptographic mechanisms.

2.4.1 Long-term Key Generation and Storage (ECC Identity Key):

In the secure messaging system, a long-term ECC identity key pair on the X25519 elliptic curve serves as the backbone for user security. Each key pair includes both a public and a private key. The public key lets users authenticate and communicate securely, whereas the private key stays safely encrypted on the device.

For an extra layer of security, the system stores these keys in encrypted local storage. Before saving the private key, it gets encrypted once more with a secret key controlled by the Android Keystore system. This safeguard makes it way harder for anyone unauthorized to get access to that critical info, making the whole system much safer. See Figure 2.3.

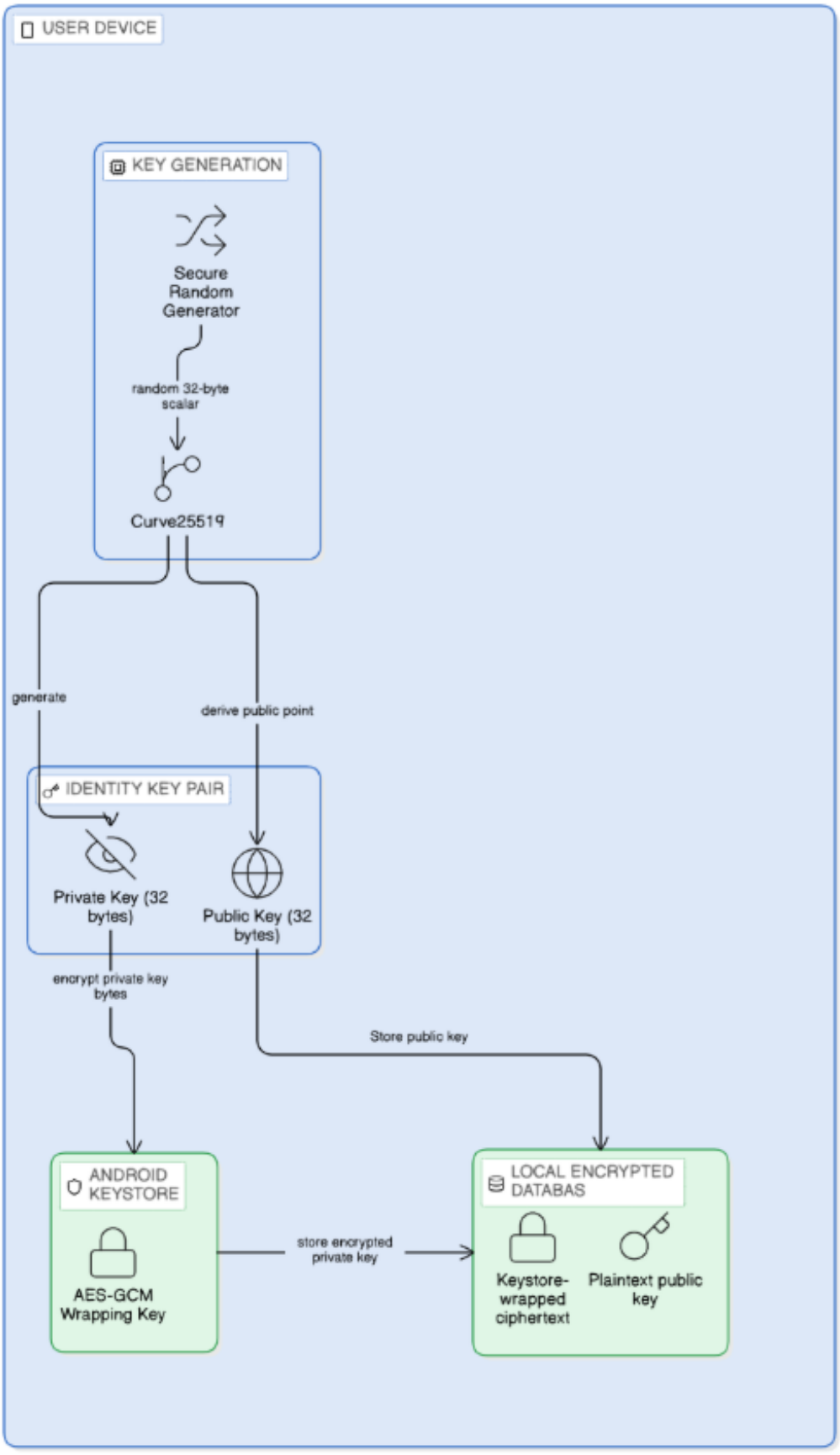


Figure 2.3: ECC Identity Key.

2.4.2 Session Establishment via ECDH :

Before two users can exchange encrypted messages, a secure shared session must be established between them using Elliptic Curve Diffie-Hellman (ECDH) combined with a Key Derivation Function (KDF). the device belonging to user A conducts an ECDH key exchange using X25519 curve, whereby it combines its own private identity key and the peer's public identity key to derive a 32-byte shared secret that can be independently calculated by both peers without the ability for the server to calculate it since the server does not possess any of the devices' private keys. The next step involves deriving the key from the shared secret to generate the Root Key (RK) which will form the basis for encryption for the session. One thing worth noting about this process is that the shared secret is never sent across the network; it is only generated independently by both sides through mathematics, making it impossible even for a completely compromised server to be able to decrypt the session. See Figure 2.4.

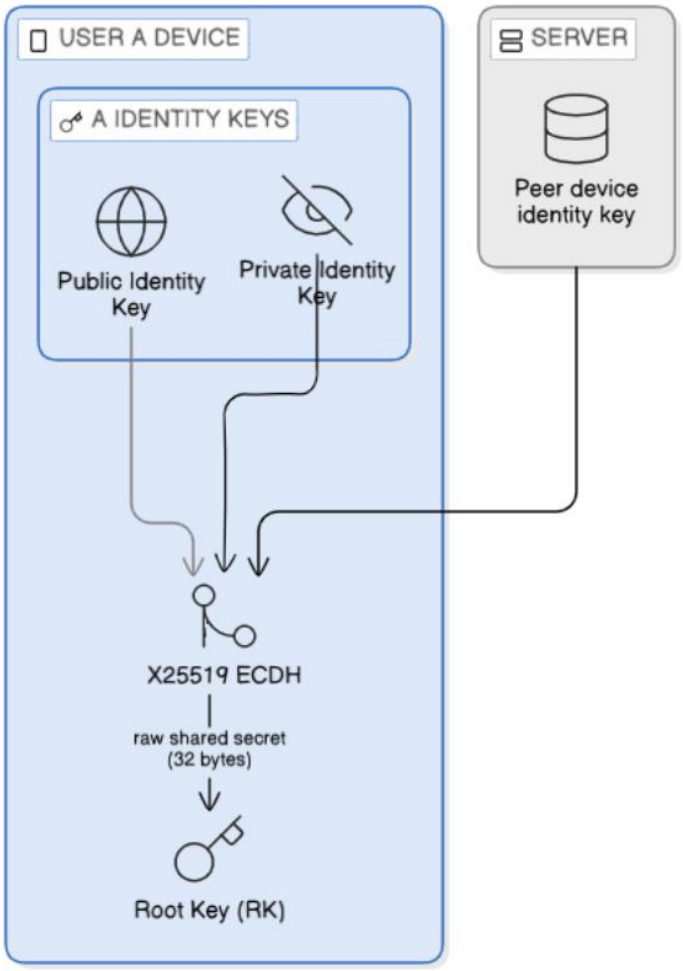


Figure 2.4: Session Establishment via ECDH.

Once the session is successfully established, both parties possess identical session keys, enabling secure message exchange using AES-GCM. This design provides forward secrecy through the use of ephemeral keys in the initial ECDH exchange.

2.4.3 Message encryption/decryption using AES-GCM:

Message encryption and decryption are performed using the AES-GCM algorithm, which relies on a symmetric session key (Ek) and a unique nonce (N). In this stage, only the encryption component (CTR mode) is considered, where the message (M) is processed block by block to produce ciphertext. The same parameters are used during decryption to recover the original message, ensuring confidentiality of the transmitted data.

Encryption Steps: (See Figure 2.5)

- Divide the message into blocks (Mi).
- Initialize a counter using the nonce (N).
- Generate keystream blocks (Ek(counter i)) using AES.
- Encrypt each block: (Ci=Mi⊕Ek(counter i)).
- Concatenate all (Ci) to obtain the ciphertext (C).
- For decryption, regenerate the same keystream and compute: (Mi=Ci⊕Ek(counteri)).

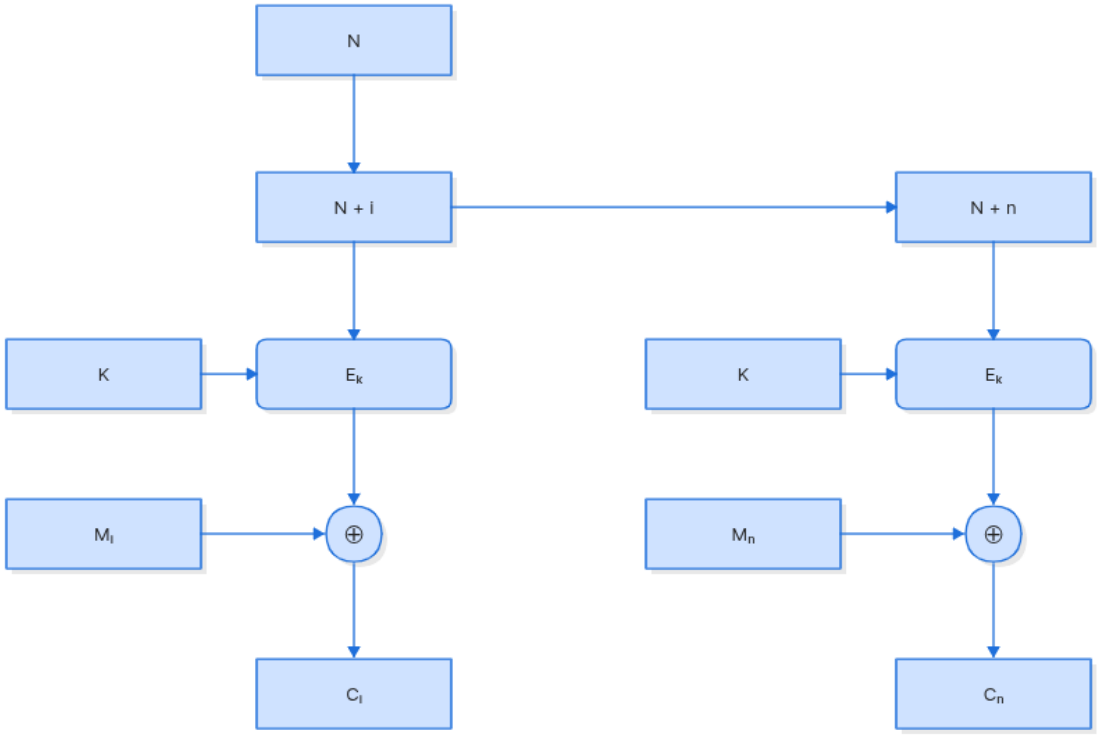


Figure 2.5: Message encryption using AES-GCM.

Decryption Steps: (See Figure 2.6)

- Receive the ciphertext (C) along with the same key (K) and nonce (N).
- Divide the ciphertext into blocks (Ci).
- Reconstruct the counter values using the nonce (N).
- Generate the keystream blocks (Ek(counteri)) using AES.
- Decrypt each block: (Mi=Ci⊕Ek(counteri)).
- Concatenate all (Mi) to recover the original message (M).

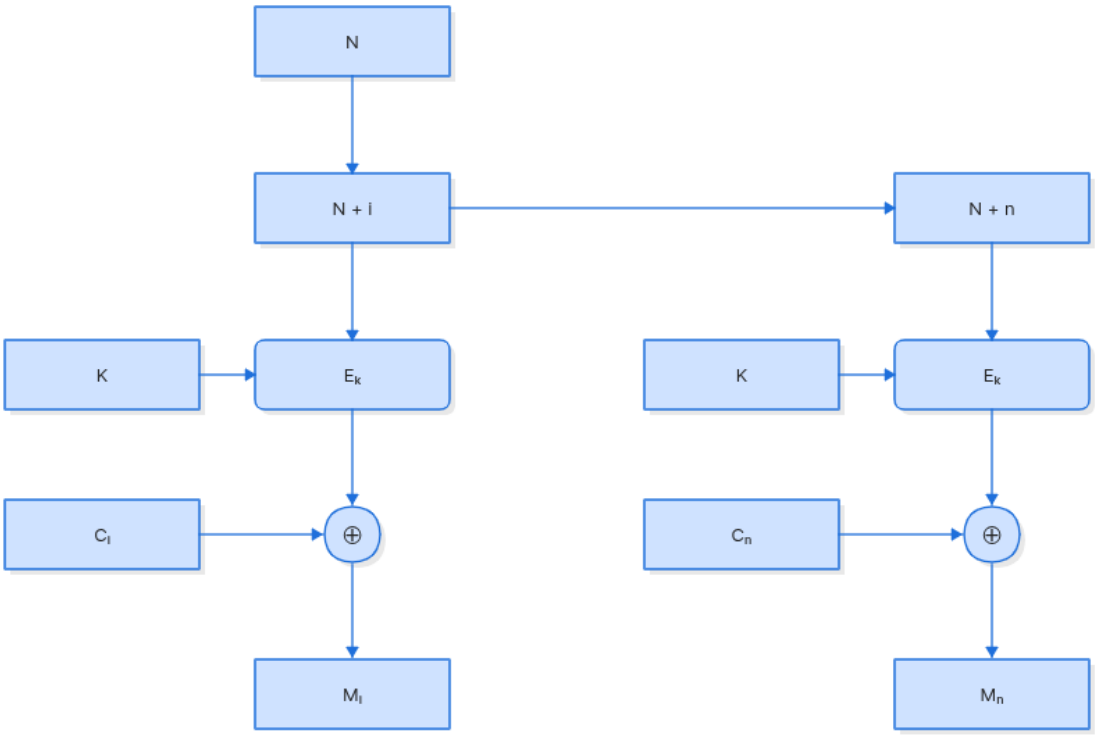


Figure 2.6: Message decryption using AES-GCM.

After presenting this simplified version of the algorithm, it becomes evident that **confidentiality** alone is insufficient, since there is no guarantee that the ciphertext has not been altered during transmission. To address this limitation, an additional mechanism known as an **authentication tag** is introduced to provide **integrity** and **authenticity**, ensuring that any modification to the data can be reliably detected.

2.4.4 Verification of authentication tags:

So far, we have seen how AES-GCM can encrypt and decrypt a message using a session key and a nonce, ensuring confidentiality. However, to guarantee integrity and authenticity of the transmitted data, AES-GCM also generates an authentication tag. The diagram below illustrates the full AES-GCM process, showing how the message, and nonce are used to produce both the ciphertext and the authentication tag, as shown in Figure 2.7 and Figure 2.8.

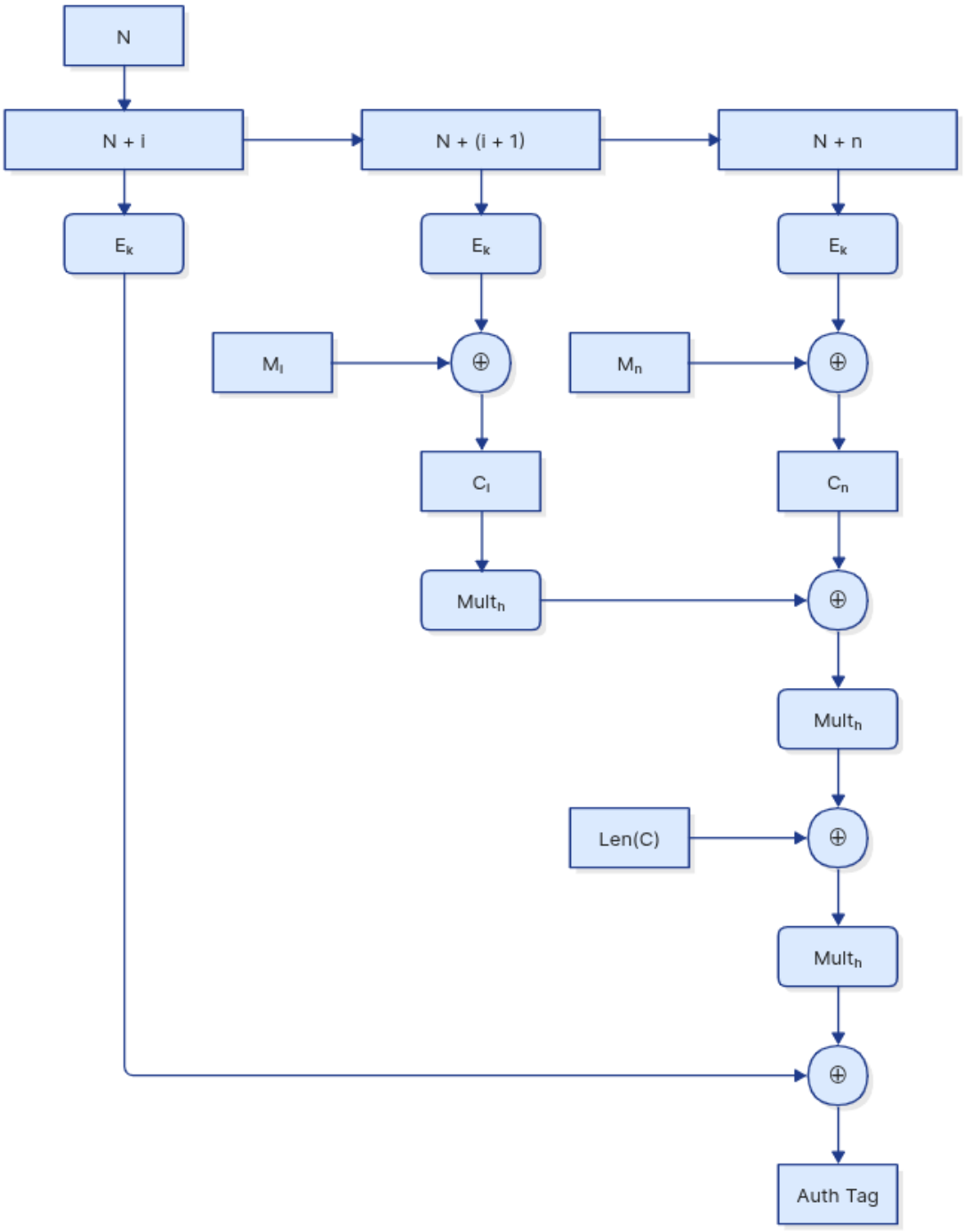


Figure 2.7: Generation of authentication tags.

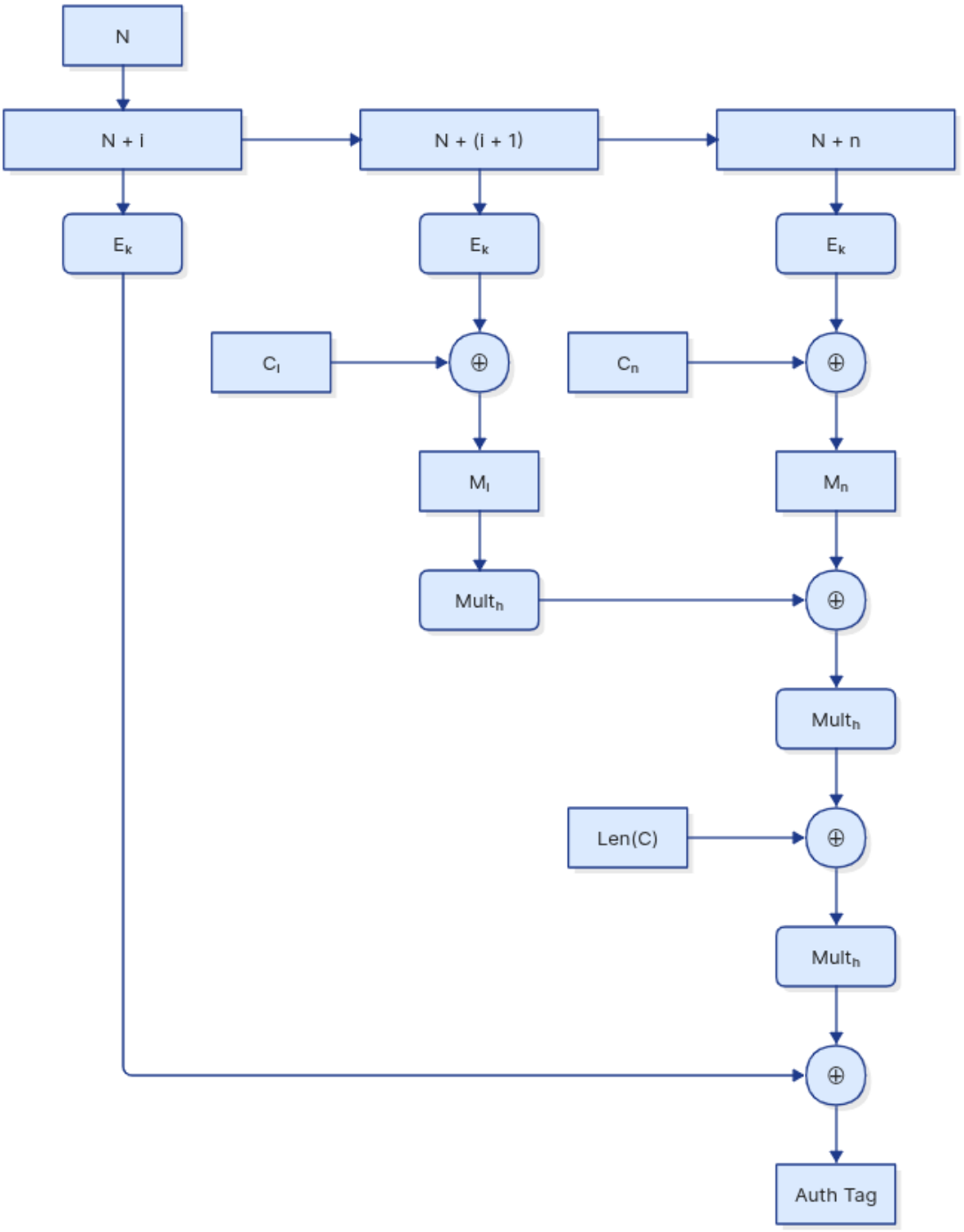


Figure 2.8: Verification of authentication tags.

2.5 Key Management and Ratcheting Mechanism

2.5.1 Need for Key Rotation:

In any secure messaging system, proper key management is essential to maintain long-term security. Even if the initial session keys are generated securely using ECDH and HKDF, using the same encryption keys for a long period of time introduces significant security risks. Therefore, the system must implement a mechanism for regular key rotation.

The main reasons for implementing key rotation are as follows:

Limiting the Impact of Key Compromise If a message key or chain key is compromised (for example, through a device breach or side-channel attack), an attacker could decrypt all subsequent messages if the same key continues to be used. Regular key rotation ensures that even if one key is leaked, only a limited number of messages can be decrypted.

Achieving Forward Secrecy Forward secrecy (also known as Perfect Forward Secrecy) guarantees that compromise of long-term keys or current session keys does not allow decryption of previously exchanged messages. Without key rotation, forward secrecy cannot be effectively maintained over long conversations.

Achieving Future Secrecy (Post-Compromise Security) In real-world scenarios, a user's device may be temporarily compromised (e.g., malware infection, physical access, or theft). After the compromise ends, the system should ensure that future messages remain secure. Key rotation allows the protocol to "heal" itself by generating new secure keys even after a breach.

To address these needs, the system implements a simplified ratcheting mechanism.

2.5.2 Simple ratcheting mechanism inspired by the Double Ratchet:

To provide **forward secrecy** and limited **post-compromise** security, we implemented a simplified ratcheting mechanism inspired by the Double Ratchet algorithm. This mechanism combines **per-message symmetric ratchets** with **periodic Diffie-Hellman (DH) rotations**. The ratchet operates independently of the initial session setup and long-term identity keys, focusing solely on the evolution of session keys for ongoing message exchanges.

2.5.3 Ratchet State:

Because the Double Ratchet protocol is inherently stateful, the system must continuously track the variables that dictate key derivation. To ensure session continuity across application restarts and prevent state loss, this cryptographic state is persisted locally on the client device within a structured relational database table.

Additionally, the implementation stores a ratchet epoch value. A new epoch is created every time a new remote ephemeral DH public key is received and a DH ratchet step is performed. This allows the application to logically separate key generations across DH rotations and simplifies synchronization and debugging of ratchet transitions.

The ratchet state structure is presented in Table 2.1, and the ratchet epoch structure is presented in Table 2.2.

Table 2.1 Ratchet State Structure

Field Name	Type	Description
id	String	Unique identifier for the ratchet state
deviceId	String	Identifier of the associated device or session
sendChainKey	String	Chain key used for encrypting outgoing messages
receiveChainKey	String	Chain key used for decrypting incoming messages
sendMessageCounter	String	Number of messages sent in the current chain
receiveMessageCounter	Integer	Number of messages received in the current chain
remoteEphemeralPublicKey	Integer	Public ephemeral key of the remote party
localSendEphemeralPublicKey	String	Local public ephemeral key used for sending
localSendEncryptedEphemeralPrivateKey	String	Encrypted local private ephemeral key used for sending
localReceiveEphemeralPublicKey	String	Local public ephemeral key used for receiving
localReceiveEncryptedEphemeralPrivateKey	String	Encrypted local private ephemeral key used for receiving
createdAt	Long	Timestamp indicating when the state was created

Table 2.2 Ratchet Epoch Structure

Field Name	Type	Description
id	String	Unique identifier for the epoch
deviceId	String	Identifier of the associated device or session
sendChainKey	String	Chain key used for sending during this epoch
receiveChainKey	String	Chain key used for receiving during this epoch
sendRootKey	String	Root key associated with the sending chain
receiveRootKey	String	Root key associated with the receiving chain
remoteEphemeralPublicKey	String	Public ephemeral key of the remote party
localSendEphemeralPublicKey	String	Local sending public ephemeral key
localSendEncryptedEphemeralPrivateKey	String	Encrypted local sending private ephemeral key
localReceiveEphemeralPublicKey	String	Local receiving public ephemeral key
localReceiveEncryptedEphemeralPrivateKey	String	Encrypted local receiving private ephemeral key
createdAt	Long	Timestamp indicating when the epoch was created

2.5.3 Symmetric Ratchet:

For each outgoing message, the sender advances the symmetric ratchet as follows:

$$sendChainKey, messageKey = HKDF(sendChainKey)$$

The message is then encrypted using **AES-GCM**:

$$ciphertext, tag = AESGCMEncrypt(plaintext, messageKey)$$

After sending, the sender increments the message counter:

$$messageCounter = messageCounter + 1$$

On the receiver side, the symmetric ratchet is advanced similarly. The message key is derived from the incoming chain key:

$$messageCounter = messageCounter + 1$$

$$recvChainKey, messageKey = HKDF(recvChainKey)$$

The ciphertext is decrypted:

$$plaintext = AESGCMDecrypt(ciphertext, tag, messageKey)$$

Integrity and authenticity are verified through the AES-GCM authentication tag.

2.5.4 Diffie-Hellman Ratchet:

To provide future secrecy, a DH ratchet is triggered periodically after receiving a new ephemeral public key.

Upon receiving a message containing a new DH public key, we advance the DH Ratchet which mean we generate a new ephemeral pair keys and the shared secret is computed:

$$ephemeralPrivateKey, ephemeralPublicKey = GenerateNewDHKeyPair()$$

$$sharedSecret = ECDH(ephemeralPrivateKey, ephemeralPublicKey, remoteEphemeralPublicKey)$$

The root key is that sharedSecret

New receiving symmetric chain key is derived from the new root key:

$$RecvChainKey = HKDF(rootKey)$$

Finally, the message counter is reset to zero.

$$messageCounter = 0$$

See Figure 2.9

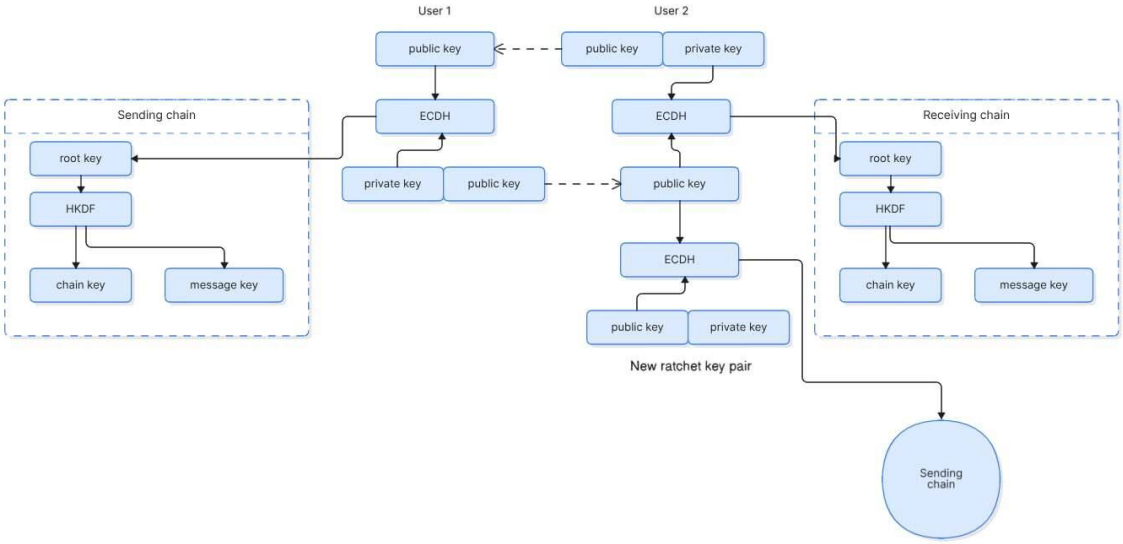


Figure 2.9: General Diagram of the mechanism

2.6 Message Format

To securely and reliably exchange messages in the hybrid ECC/AES system, a structured message format is used. This format defines how encrypted messages are organized and transmitted between users.

Each encrypted message contains three main parts:

- Header
- Payload (Ciphertext)
- Authentication Tag

The general structure is:

Encrypted Message = [Header] + [Payload] + [Authentication Tag]

For simplicity and easier data handling, messages are represented using JSON format.

2.6.1 Header:

The header contains the metadata required for the receiver to process and decrypt the message correctly. The header is not encrypted because this information is needed before decryption.

The header contains the following fields (See Table 2.3):

Table 2.3 Message Header Structure

Field Name	Type	Description
chatId	String	Unique identifier of the conversation
senderUserId	String	Identifier of the message sender
senderDeviceId	String	Identifier of the sender's device
recipientUserId	String	Identifier of the message recipient
recipientDeviceId	String	Identifier of the recipient's device
messageId	String	Unique identifier of the message
senderIdentityKey	String	Base64-encoded public identity key of the sender
senderEphemeralPublicKey	String	Base64-encoded ephemeral public key used for this message
messageCounter	Integer	Message sequence number used in the ratchet mechanism
messageType	String	The message type: text, voice, image, video, or file.
attachementObject	Object	Contains the attachment ID. For voice, image, video, and file messages, the attachment content is encrypted.

2.6.2 Payload (Ciphertext):

The payload contains the encrypted message content encoded in Base64 format. Only authorized recipients can decrypt and read this content using the derived message key.

2.6.3 Authentication Tag:

The authentication tag is generated by the AES-GCM encryption algorithm. It is used to verify message integrity and authenticity and is also encoded in Base64 format.

2.6.4 Complete Message Structure:

See Table 2.4.

Table 2.4 Message Frame Structure

Field Name	Type	Description
header	Object	Contains metadata required for message processing and decryption
ciphertext	String	Base64-encoded encrypted message content
authTag	String	Base64-encoded authentication tag generated by AES-GCM

2.7 API Endpoints Design

2.7.1 Authentication

Register:

Table 2.5 Register API

Path	/api/auth/register
Method	post
Query Parameters	none
Headers Parameters	none

Request Payload	{
	"username": "string",
	"firstname": "string",
	"lastname": "string",
	"password": "string",
	"phone_number": "string",
	"verification_token": "string"
	}

Response Payload	{
	"message": "User registered successfully",
	"user": {
	"id": "string",
	"username": "string",
	"firstname": "string",
	"lastname": "string",
	"phone_number": "string"
	}
	}

Login:

Table 2.6 Login API

Path	/api/auth/login
Method	post
Query Parameters	none
Headers Parameters	none

Request Payload	{
	"username": "string",
	"password": "string"
	}

Response Payload	{
	"message": "Login successful",
	"data": {
	"access_token": "string",
	"user": {
	"id": "string",
	"username": "string",
	"firstname": "string",
	"lastname": "string",
	"phone_number": "string"
	}
	}
	}

Send OTP:

Table 2.7 Send OTP API

Path	/api/auth/send-otp
Method	post
Query Parameters	none
Headers Parameters	none

Request Payload	{
	"phone_number": "string"
	}

Response Payload	{
	"message": "OTP sent via SMS ",
	}

OTP Verification:

Table 2.8 OTP Verification API

Path	/api/auth/verify-otp
Method	post
Query Parameters	none
Headers Parameters	none

Request Payload	{
	"phone_number": "string",
	"otp": "number"
	}

Response Payload	{
	"verification_token": "string",
	}

Register Device:

Table 2.9 Register Device API

Path	/api/auth/devices
Method	post
Query Parameters	none
Headers Parameters	none

Request Payload	{
	" base64_identity_key ": "string",
	" access_token ": "string",
	" fcm_token": "string"
	}

Response Payload	{
	"message": "Device registred successfully",
	"data": {
	"access_token": "string",
	"device": {
	"id": "string",
	"identity_key": "string",
	"fcm_token": "string",
	"user_id": "string",
	"created_at": "string"
	}
	}
	}

2.7.2 User Account Management

Get Current User:

Table 2.10 Get Current User API

Path	/api/users/me
Method	get
Query Parameters	none

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	none

Response Payload	{
	"message": "user feteched successfully",
	"data": {
	"id": "string",
	"username": "string",
	"firstname": "string",
	"lastname": "string",
	"phone_number": "string",
	"devices": "string",
	"profile_image_id": "string"
	}
	}

Get User By ID:

Table 2.11 Get User By ID API

Path	/api/users/:id
Method	get
Query Parameters	none

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	none

Response Payload	{
	"message": "user feteched successfully",
	"data": {
	"id": "string",
	"username": "string",
	"firstname": "string",
	"lastname": "string",
	"phone_number": "string",
	"devices": "string",
	"profile_image_id": "string"
	}
	}

Profile Editing:

Table 2.12 Profile Editing API

Path	/api/users
Method	post
Query Parameters	none

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	{
	"username": "string",
	"firstname": "string",
	"lastname": "string"
	}

Response Payload	{
	"message": "user updated successfully",
	"data": {
	"id": "string",
	"username": "string",
	"firstname": "string",
	"lastname": "string",
	"phone_number": "string",
	"devices": "string",
	"profile_image_id": "string"
	}
	}

Search Users:

Table 2.13 Search Users API

Path	/api/users
Method	get

Query Parameters	{
	" query": "string",
	}

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	none

Response Payload	{
	"message": "users list feteched successfully",
	"data": "list of users"
	}

Update FCM Token:

Table 2.14 Update FCM Token API

Path	/api/users/fcm-token
Method	post
Query Parameters	none

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	{
	"fcm_token": "string",
	}

Response Payload	{
	"message": "FCM token updated successfully"
	}

2.7.3 Chat Management

List Chats:

Table 2.15 List Chats API

Path	/api/chats
Method	get

Query Parameters	{
	"offset": "number",
	"limit": "number"
	}

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	none

Response Payload	{
	"message": "chats feteched successfully",
	"data": "list of chats"
	}

List Messages:

Table 2.16 List Messages API

Path	/api/chats/:id/messages
Method	get

Query Parameters	{
	"offset": "number",
	"limit": "number"
	}

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	none

Response Payload	{
	"message": "messages feteched successfully",
	"data": "list of encrypted messages"
	}

Get Chat By Id:

Table 2.17 Get Chat By Id API

Path	/api/chats/:id
Method	get
Query Parameters	none

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	none

Response Payload	{
	"message": "chat fetched successfully",
	"data": {
	"id": "string",
	"created_at": "string",
	"chatMembers": "list of chat members"
	}
	}

Init Chat:

Table 2.18 Init Chat API

Path	/api/chats/init
Method	post
Query Parameters	none

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	{
	"message_frames": [
	{
	"header": {
	"chat_id": "string",
	"sender_user_id": "string",
	"sender_device_id": "string",
	"recipient_user_id": "string",
	"recipient_device_id": "string",
	"message_id": "string",
	"sender_identity_key": "string",
	"sender_ephemeral_public_key": "string",
	"message_counter": "number",
	"attachment": {
	"id": "string",
	"bucket_name": "string",
	"object_name": "string",
	"original_file_name": "string",
	"original_file_size": "number",
	"created_at": "string"
	},
	"message_type": "one of 'file' | 'text' | 'image' | 'video' | 'voice'"
	},
	"ciphertext": "string",
	"auth_tag": "string"
	}
	]
	}

Response Payload	{
	"message": "chat created successfully",
	"data": {
	"id": "string",
	"created_at": "string"
	}
	}

Update Messages in Batch:

Table 2.19 Update Messages In Batch API

Path	/api/chats /:id/messages/batch
Method	patch
Query Parameters	none

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	{
	"message_ids": "list of ids",
	"status": "one of 'sent' | 'delivered' | 'seen'"
	}

Response Payload	{
	"message": "messages updated successfully"
	}

2.7.4 Contacts Managements

Search Contact:

Table 2.20 Search Contact API

Path	/api/contacts/search
Method	get

Query Parameters	{
	"keyword": "string"
	}

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	none

Response Payload	{
	"message": "Contacts searched successfully",
	"data": "list of user contacts"
	}

Create Contact:

Table 2.21 Create Contact API

Path	/api/contacts
Method	post
Query Parameters	none

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	{
	"phone_number": "string",
	"firstname": "string",
	"lastname": "string"
	}

Response Payload	{
	"message": "Contact added successfully",
	"data": {
	"id": "string",
	"firstname": "string",
	"lastname": "string",
	"user_id": "string",
	"contact_user_id": "string",
	"created_at": "string"
	}
	}

List Contacts:

Table 2.22 List Contacts API

Path	/api/contacts
Method	get

Query Parameters	{
	"offset": "string",
	"limit": "string"
	}

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	none

Response Payload	{
	"message": "Contacts fetched successfully",
	"data": "list of user contacts"
	}

Get Contact by Id:

Table 2.23 Get Contact by Id API

Path	/api/contacts/:id
Method	get
Query Parameters	none

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	none

Response Payload	{
	"message": "Contact fetched successfully",
	"data": {
	"id": "string",
	"firstname": "string",
	"lastname": "string",
	"user_id": "string",
	"contact_user_id": "string",
	"created_at": "string"
	}
	}

Edit Contact:

Table 2.24 Edit Contact API

Path	/api/contacts/:id
Method	patch
Query Parameters	none

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	{
	"firstname": "string",
	"lastname": "string"
	}

Response Payload	{
	"message": "Contact updated successfully",
	"data": {
	"id": "string",
	"firstname": "string",
	"lastname": "string",
	"user_id": "string",
	"contact_user_id": "string",
	"created_at": "string"
	}
	}

Delete Contact:

Table 2.25 Delete Contact API

Path	/api/contacts/:id
Method	delete
Query Parameters	none

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	none

Response Payload	{
	"message": "Contact deleted successfully",
	"data": {
	"id": "string",
	"firstname": "string",
	"lastname": "string",
	"user_id": "string",
	"contact_user_id": "string",
	"created_at": "string"
	}
	}

2.7.5 Attachment Management

Upload Attachment:

Table 2.26 Upload Attachment API

Path	/api/attachments/:id
Method	post
Query Parameters	none

Headers Parameters	{
	" Authorization": "string",
	}

Request Payload	FormData {
	"original_file_name": "string",
	"original_file_size": "number",
	"file": "binary"
	}

Response Payload	{
	"message": "attachment uploaded successfully",
	"data": {
	"id": "string",
	"bucket_name": "string",
	"object_name": "string",
	"original_file_name": "string",
	"original_file_size": "string",
	"created_at": "string"
	}
	}

2.8 Conclusion

In this chapter, we have presented the overall design and architecture of the proposed secure messaging system, highlighting the key technical and security-driven decisions. Starting from clearly defined functional and security requirements, the system was structured around a client-server model with minimal trust assumptions, ensuring that sensitive data remains protected at all stages of communication.

A hybrid cryptographic approach combining ECC for key exchange and AES-GCM for message encryption was adopted to achieve both strong security guarantees and efficiency on mobile platforms. The integration of mechanisms such as authenticated encryption, secure key management, and a simplified ratcheting process further strengthens confidentiality, integrity, and forward secrecy. Additionally, careful consideration was given to practical constraints such as intermittent connectivity, resource limitations, and secure local storage.

The next chapter will focus on the implementation of this architecture, detailing the technologies, tools, and development choices used to realize the system in practice.

Chapter III:

Implementation of the Secure Messaging System

3. Implementation of the Secure Messaging System

3.1 Introduction

This chapter presents the practical implementation of the secure messaging system designed in the previous chapter. this chapter details the actual development process, including the tools, technologies, and implementation decisions made to realize a functional and secure end-to-end encrypted messaging application.

The implementation follows the hybrid ECC/AES cryptographic approach proposed earlier, emphasizing strong security guarantees (end-to-end encryption, forward secrecy, and post-compromise security) while respecting the constraints of mobile devices, particularly limited CPU resources, battery life, and intermittent network connectivity.

This chapter is organized as follows: first, the development environment and selected tools are presented. Then, the cryptographic core is implemented in detail, followed by the client-side and server-side implementations. Finally, various security hardening measures applied to protect the system against common attacks are discussed.

3.2 Development Environment and Tools

3.2.1 Mobile Platform

The mobile client was developed as a native Android application using Kotlin as the primary programming language. Kotlin was chosen for excellent interoperability with Java libraries, which significantly improves development productivity and code reliability.

For the user interface, Jetpack Compose was used instead of the traditional XML-based layout system. Jetpack Compose offers a declarative and reactive approach to UI development.

This choice of technologies allows the application to deliver a smooth user experience while maintaining strong security standards and efficient resource usage (CPU and battery).

3.2.2 Backend Stack

The backend was built using a Node.js ecosystem to ensure high performance, scalability, and rapid development. The main components of the backend stack are as follows:

Node.js & Express.js: Node.js serves as the runtime environment, while Express.js is used as the web framework to build RESTful APIs. This combination provides excellent performance for handling a large number of concurrent connections with relatively low resource consumption.

PostgreSQL: The primary relational database used for storing persistent data such as users, devices, chats, contacts, and message metadata.

Aedes MQTT over WebSocket: For real-time communication, Aedes (a lightweight MQTT broker written in Node.js) is used over WebSocket. This setup enables efficient bidirectional real-time messaging, delivery receipts, typing indicators, and

online/offline status updates. The MQTT protocol over WebSocket was chosen for its lightweight nature and reliability in mobile environments with intermittent connectivity.

MinIO: Object storage server compatible with Amazon S3. MinIO is used for storing media attachments (images, videos, documents, and voice messages). It provides scalable, secure, and high-performance storage while allowing easy integration with the application through presigned URLs.

This backend architecture was designed to be lightweight, scalable, and aligned with the minimal trust principle — the server handles routing and storage of encrypted data but never has access to plaintext messages or private keys.

3.2.3 Cryptographic Libraries

To implement cryptographic functionalities in our system, we relied on several well-established libraries:

3.2.3.1 Bouncy Castle Library:

Bouncy Castle is a widely used collection of cryptographic APIs designed for secure application development. It supports both Java and C# programming languages and is maintained by the registered organization Legion of the Bouncy Castle Inc. This library provides a broad range of cryptographic algorithms and tools that facilitate secure data processing.

3.2.3.2 Android Keystore System:

The Android Keystore system is used to securely store cryptographic keys within a protected container on the device. Keys stored in the keystore cannot be easily extracted, which enhances security. Additionally, the system allows control over how and when keys are used, such as requiring user authentication or restricting their use to specific cryptographic operations.

3.2.3.3 Javax.crypto:

The javax.crypto package provides essential classes and methods for performing cryptographic operations. These include encryption, key generation, key agreement, and Message Authentication Code (MAC) generation. It supports various encryption types, such as symmetric and asymmetric encryption, as well as block and stream ciphers.

Furthermore, this package follows a provider-based architecture, where the interfaces are defined independently from their implementations. This allows developers to use different cryptographic providers without modifying the application code, offering flexibility and ease of integration.

3.3 Implementation of the Cryptographic Core

This section presents the practical implementation of the cryptographic core of the system. It focuses on how cryptographic operations were integrated into the application, including key management, encryption processes, key evolution, and message formatting.

3.3.1 Key Pair Generation and Secure Storage

In the implemented system, cryptographic key management is initialized during the first application launch after installation. At this stage, an identity key pair is generated using the X25519 algorithm through the Bouncy Castle library. This identity key pair is intended for long-term use and uniquely represents the user within the system.

In addition to the identity key pair, ephemeral key pairs are also generated to support secure session establishment and enhance forward secrecy. These keys are used temporarily during key exchange processes and are periodically renewed.

Before storage, all generated keys are encoded using Base64 format to ensure compatibility with the storage system and to simplify serialization. For enhanced protection, the private key is encrypted prior to storage using the AES-256-GCM algorithm. The encryption key used for this operation is securely obtained from the Android Keystore system, ensuring that the key material remains protected and non-exportable.

All cryptographic data, including public and encrypted private keys, are stored locally in an SQLite database configured with SQLCipher. This provides full database-level encryption, adding an additional layer of security to protect stored information against unauthorized access.

This multi-layered approach combining secure key generation, hardware-backed key storage, encryption of sensitive data, and encrypted database storage—ensures a high level of protection for cryptographic material within the application.

3.3.2 Hybrid Encryption Logic (ECDH + HKDF + AES-256-GCM)

The hybrid encryption process is implemented to ensure both secure key exchange and efficient message encryption. This process is executed each time a message is sent within the application.

First, a shared secret is established using the Elliptic Curve Diffie-Hellman (ECDH) protocol based on the X25519 algorithm. This operation is implemented using the Bouncy Castle library, where the sender's private key and the receiver's public key are used to compute the shared secret.

The resulting shared secret is then passed through a Key Derivation Function (HKDF), also implemented using Bouncy Castle, to derive a strong and uniform symmetric key. This step ensures that the generated key is suitable for encryption and independent of the raw shared secret.

For message encryption, the system uses the AES algorithm in Galois/Counter Mode (AES-256-GCM), implemented via the javax.crypto package. A unique initialization vector (IV) is generated for each encryption operation to guarantee security and prevent reuse attacks. The plaintext message is then encrypted using the derived symmetric key, producing both the ciphertext and an authentication tag.

This implementation combines asymmetric cryptography for secure key exchange with symmetric encryption for performance, while also ensuring data integrity and authenticity through the use of AES-GCM.

3.3.3 Ratcheting Mechanism

The ratcheting mechanism is implemented based on the principles of the Double Ratchet algorithm. A dedicated component, referred to as DoubleRatchetEngine, was developed to manage the evolution of cryptographic keys during secure communication. This component is responsible for handling both symmetric key updates and Diffie-Hellman key exchanges between communicating parties.

To ensure persistence and reliability, the ratchet state is stored locally in an SQLite database. The Android RoomDatabase library is used as an abstraction layer to simplify database operations and provide structured access to stored data. The stored state includes all parameters required to maintain synchronization between communicating parties, such as chain keys, message counters, and ephemeral key pairs. Sensitive key material, particularly private keys, is stored in encrypted form, and all values are encoded using Base64 before storage.

In addition to the current ratchet state, the system maintains a history of ratchet epochs. Each epoch represents a snapshot of the cryptographic state at a specific stage of communication. This allows the system to handle delayed or out-of-order messages by referencing previous states when necessary, ensuring reliable decryption while preserving forward secrecy.

This design ensures efficient key management, supports message synchronization, and maintains strong security guarantees through continuous key evolution and secure storage practices.

3.3.4 Message Format Serialization

The message serialization process is implemented to ensure efficient and reliable transmission of encrypted data between clients and the server. In this system, the **kotlinx.serialization** library is used to convert message objects into JSON format. JSON was selected due to its simplicity and seamless compatibility with the backend server, which is implemented using Node.js.

Each encrypted message is represented as a structured object composed of a header, ciphertext, and authentication tag. The header contains all metadata required for message identification and decryption, while the ciphertext represents the encrypted message content. The authentication tag, generated during AES-GCM encryption, is used to verify the integrity and authenticity of the message.

To avoid issues related to binary data handling during transmission, all cryptographic elements—including public keys and ciphertext—are encoded using Base64 before serialization. This ensures safe and consistent data exchange across different system components.

3.3.5 Multi Device Support

We implemented multi-device support by establishing a secure session between the user's current device and their additional devices. At the end-to-end encryption (E2EE) level, the system architecture is device-based rather than user-based. In other words, encryption and message exchange are handled between sender and receiver devices. However, at the chat application level, communication remains user-based, where interactions are represented as exchanges between sender and receiver users.

3.4 Client-Side Implementation

3.4.1 User Authentication and Identity Key Upload

This section describes how the application handles user authentication and the provisioning of cryptographic identity. The process is split across two distinct flows: registration, which creates the account and verifies the phone number, and login, which is where the ECC identity key pair is generated on the device and the public key is uploaded to the server. This separation ensures that key material only exists after the user's identity has been fully verified and an authenticated session has been established.

3.4.1.1 Registration: Account Creation and Phone Verification

Registration handles only account creation and phone number verification — no cryptographic key material is involved at this stage. The user enters their phone number, to send OTP code and server dispatches it via SMS provider. The raw OTP is never persisted in the database.

The user submits the received code and server will verify it. On success, it returns a short-lived verification token (5-minute validity). The user then completes their profile and submits a POST /api/auth/register request containing the profile fields, the verification token, and the account password. The server validates the payload hashes the password with bcrypt and creates the user record.

3.4.1.2 Login: Multi-Step Authentication and Device Registration

The application uses a multi-step login flow to support multiple user accounts on the same device. During authentication, the client first performs a standard login request using the user credentials. If authentication succeeds, the backend returns a temporary access token.

After receiving this access token, the application checks the local encrypted database to determine whether an identity key already exists for the authenticated account on the current device. If no identity key is found, such as during the first login for

that account on the device, the application generates a new X25519 identity key pair. If an identity key already exists, the application reuses the existing key pair.

The generated private key is securely stored locally using encrypted storage protected by the Android Keystore. The local encrypted database maintains the mapping between accounts and their associated identity keys, allowing multiple accounts to coexist securely on the same device.

Once the identity key is obtained, the application sends the public key to the backend using the temporary access token in order to register the current device. After successful device registration, the backend returns the final session token, which is then used for authenticated requests throughout the application.

On subsequent logins, the same identity key is reused for the corresponding account unless the local encrypted data is removed, such as after a factory reset, application reinstallation, or device migration. In such cases, the application generates a new identity key pair and registers the new public key with the backend as a new device. The user interface is shown in Figures 3.1 and 3.2.

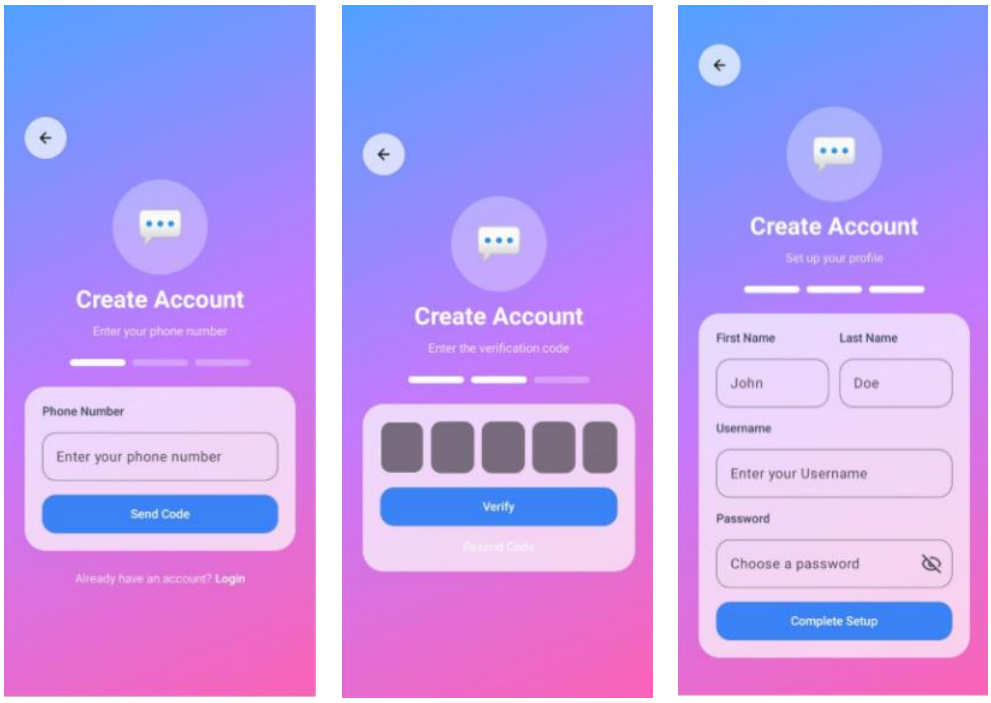


Figure 3.1: User Register Steps

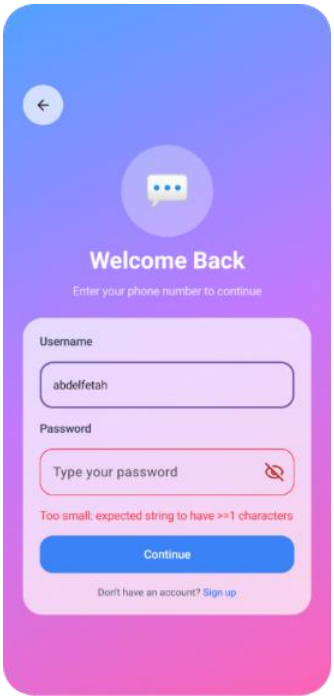


Figure 3.2: Login Screen

3.4.2 Contact Management

Contact management is the user's address book. Before a session can be established with another user, the current user searches for a specific user to add him to contacts. This section describes how contacts are added, stored.

3.4.2.1 Adding a Contact

A user adds a contact by entering a phone number and full name in the add-contact screen. Before creating the contact record, the server performs a lookup in the users table to check whether the provided phone number belongs to a registered user. The user interface is shown in Figures 3.3.

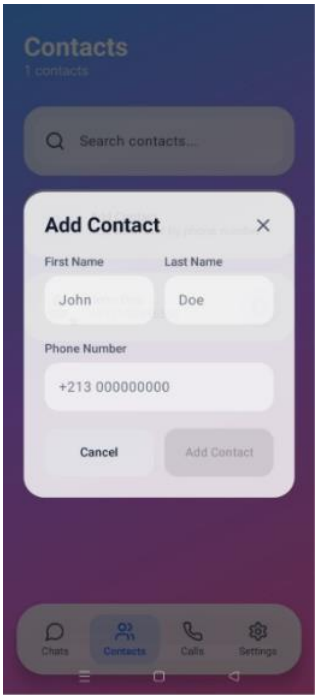


Figure 3.3: Adding a Contact Screen

3.4.2.2 Listing and Searching Contacts

The contacts list screen fetches the full contact list. The search function calls POST /api/contacts/search with a keyword string. The server performs a case-insensitive partial match against both the contact's full name and phone number. The user interface is shown in Figures 3.4.

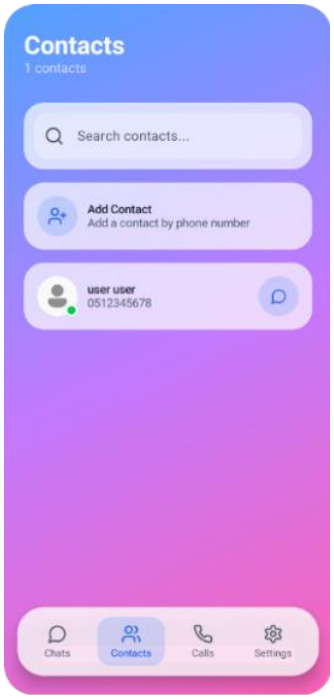


Figure 3.4: List & Search Contact Screen

3.4.2.3 Editing and Deleting Contacts

The contact details screen provides two actions: Edit Contact and Delete Contact. When the user presses the Edit Contact button, a modal form appears containing editable fields for the contact's display name.

When the user presses the Delete Contact button, the application remove the contact from the contact list. The user interface is shown in Figures 3.5.

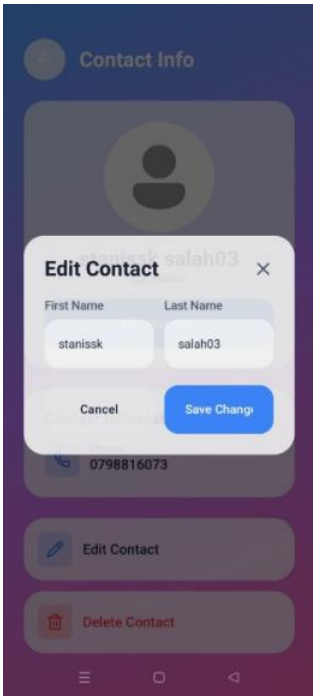


Figure 3.5: Edit & Delete Contact Screen

3.4.3 Device Discovery and Public Key Retrieval

Before two users can exchange encrypted messages, the sender's device must discover whether the intended recipient is registered on the system and obtain their public identity key. This step is a prerequisite for the ECDH-based session establishment. Device discovery in this system is server-mediated: the server acts as a public key directory, storing and serving only the public component of each user's identity key pair. The private key never leaves the owner's device.

3.4.3.1 User Lookup

Every time the sender opens a conversation, the application queries the server to fetch the recipient's current profile object, which always includes their public identity key as a Base64-encoded string. Fetching the key fresh on each conversation open ensures the application always works with the most up-to-date public key, naturally handling cases where a contact has re-registered or switched devices.

The server only exposes the public identity key no private key material, session state, or message history is ever returned by this endpoint.

3.4.3.2 Public Key Validation

Once the response is received, the client decodes the Base64 public key and performs a basic structural validation before using it, the decoded byte array must be exactly 32 bytes, which is the only valid length for an X25519 public key. Any response that fails this check is rejected and the conversation is not opened, preventing a malformed or tampered key from being fed into the ECDH computation.

3.4.4 Sending and Receiving Messages

Sending and receiving messages are implemented using MQTT over WebSocket (WS), enabling real-time communication.

For message sending, the plaintext is first encrypted using the **DoubleRatchetEngine**. A message key is derived from the current ratchet state and used with **AES-256-GCM** to generate the ciphertext and authentication tag. The encrypted message, together with its header, is then published to the appropriate MQTT topic over the established WS connection.

For message reception, subscription is made to the relevant MQTT topics. When a message is received:

If the conversation is currently open, the message is processed immediately Otherwise, a notification is triggered to inform the user

The received message is parsed into its components (header, ciphertext, authentication tag), and the DoubleRatchetEngine is used to derive the appropriate decryption key. Decryption is performed using AES-256-GCM, followed by integrity verification before displaying the message. The user interface is shown in Figures 3.6.

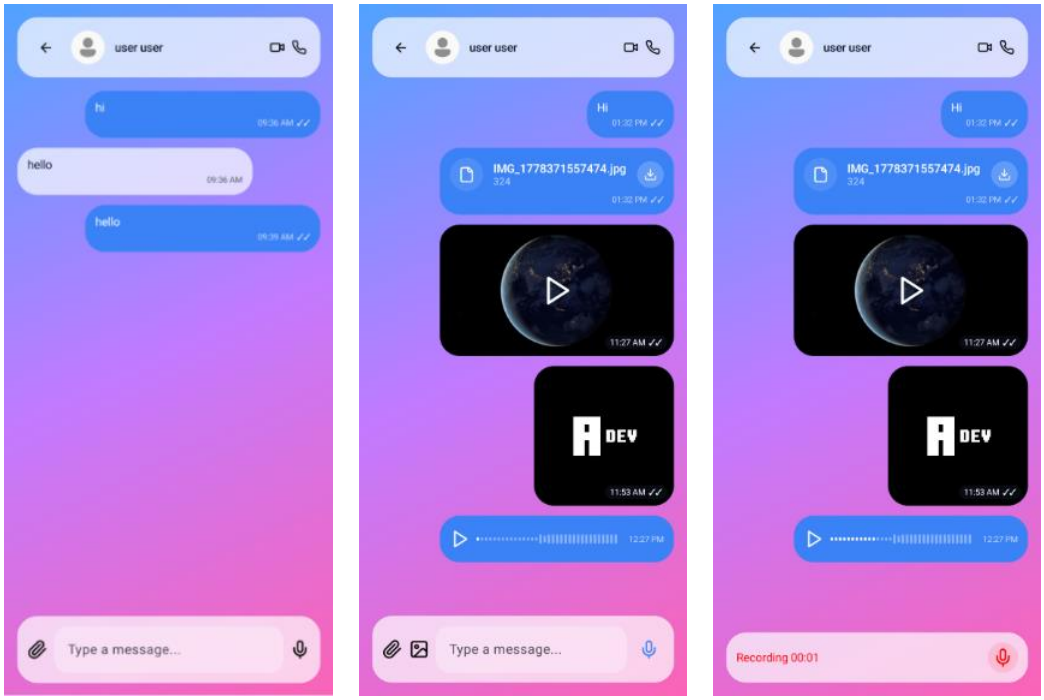


Figure 3.6: Chat Screen

3.4.5 Local Encrypted Storage (SQLCipher)

Local data persistence is implemented using SQLCipher, an encryption extension for SQLite that provides transparent database-level encryption. To simplify database interactions and avoid direct SQL query handling, the Android Room persistence library is used as an abstraction layer.

The database is encrypted using a symmetric key securely stored in the Android Keystore system, ensuring that the encryption key is protected by the device's hardware-backed security mechanisms.

Local storage is used to persist critical cryptographic state and application data, including:

- Ratchet states and ratchet epochs
- Identity keys and related key material
- User metadata required for session management

All stored data remains encrypted at rest, preventing unauthorized access even if the device storage is compromised.

3.5 Server-Side Implementation

3.5.1 Push Notifications (FCM)

Push notifications plays an essential part in providing a responsive user experience in mobile messaging apps. In order to accomplish this goal, Firebase Cloud Messaging (FCM) was selected as a push notification service because of its reliability and efficiency in delivery.

3.5.1.1 Approach to Implementation

While many traditional messaging apps send notification contents in plain text, the present approach focuses on E2EE. Thus, the following approach was used for push notifications:

- If a message comes when the receiver is offline, the server stores the message and triggers a lightweight notification by FCM to the relevant device.
- As for the notification payload delivered by the server to the device, it comprises metadata (chat_id, message_id, etc.) and cyphertext message.
- Finally, the message is decrypted with the help of respective session keys generated according to the ratcheting protocol.

3.5.1.2 Settings for Notifications Management

In order to provide complete control for the user over privacy settings and resources of his/her devices, a special screen has been implemented to configure the following parameters:

- Enable or disable push notifications entirely.
- Enable or disable notification sound.
- Enable or disable vibration on new messages.

3.5.1.3 Implementation Details

FCM Service: Custom FirebaseMessagingService was used to handle push notifications.

Notification Channel: Android Notification Channels were utilized for better handling of notifications on devices running Android version 8.0 or above.

Decryption in Background: Using Kotlin Coroutines and WorkManager, decryption of the messages takes place in the background upon receipt of a high-priority notification.

Security Issues: All operations related to decryption take place only after authenticating the message through the AES-GCM tag.

The user interface is shown in Figures 3.7.

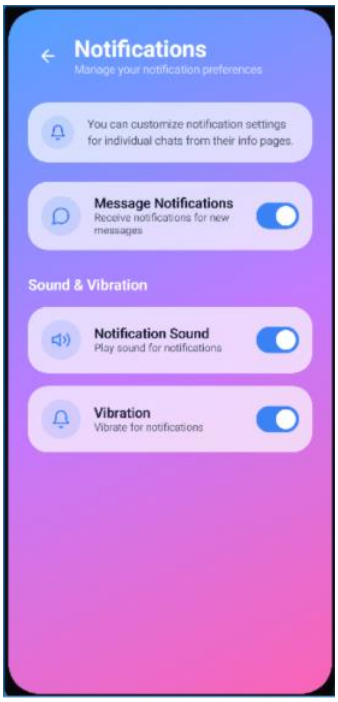


Figure 3.7: Notifications settings

3.5.2 Real-Time Communication

Real-time communication is implemented using MQTT over WebSocket (WS), combining the lightweight publish/subscribe model of MQTT with the persistent, full-duplex capabilities of WebSocket.

An MQTT broker is deployed using the Aedes library, configured to operate over WebSocket. This setup enables clients to maintain a persistent connection and exchange messages with low latency.

Communication is organized using topic-based routing. Each conversation is mapped to a dedicated topic of the form: **chat/<chat-id>**

Clients publish encrypted messages to the corresponding topic and subscribe to topics associated with their active conversations. The broker is responsible for distributing messages to all subscribed clients.

On the server side, message interception is implemented to persist messages in the database. Since all messages are end-to-end encrypted, only ciphertext is stored and processed, and no plaintext content is exposed at any stage.

This architecture ensures efficient, low-latency message delivery while preserving confidentiality through end-to-end encryption.

3.6 Security Hardening

3.6.1 Input Validation

For input validation, we used Zod, a TypeScript-first validation library. Zod allows us to define schemas for validating different types of data, from simple strings to complex nested objects. As a result, every request sent to the application is validated before processing, which helps improve security and data integrity.

3.6.2 No Plaintext Logging Policy

When logging information for debugging or error tracking purposes, sensitive data is excluded from the logs. This includes passwords, API keys, and user personal information. Avoiding plaintext logging helps protect user privacy and reduces the risk of sensitive data exposure.

3.7 Links to the repositories

- Mobile: https://github.com/payn-e2ee/payn-mobile-app
- Backend: https://github.com/payn-e2ee/payn-backend

Feel free to explore the repositories to gain insights into our project's architecture and implementation details.

3.8 Conclusion

This chapter presented the implementation of the secure messaging system, including the development environment, cryptographic mechanisms, client-side and server-side components, and security hardening measures. The system was built using modern technologies such as Kotlin for the Android application and Node.js with PostgreSQL for the backend infrastructure.

The implemented cryptographic core combines X25519, HKDF, and AES-256-GCM to ensure secure key exchange, confidentiality, and message integrity. Additional mechanisms such as encrypted local storage, offline message queuing, real-time communication, and push notifications improve both security and usability. Furthermore, security hardening techniques including input validation, rate limiting, and secure logging policies were integrated to strengthen the overall protection of the system.

Overall, the implementation demonstrates how modern cryptographic techniques and secure software engineering practices can be combined to build a practical and secure messaging application.

Chapter IV:

Security Analysis, Performance Evaluation, and Comparative Study

4. Security Analysis, Performance Evaluation, and Comparative Study

4.1 Introduction

This chapter evaluates the proposed end-to-end encrypted messaging application in terms of security and performance. The analysis focuses on the effectiveness of the implemented ECC and AES-based encryption mechanisms and their ability to protect user communication against common security threats.

The chapter first examines the resistance of the system against attacks such as Man-in-the-Middle (MITM), replay attacks, and key compromise scenarios. It also discusses the impact of server compromise and the limitations related to metadata exposure.

In addition, performance tests are presented to measure the execution time of cryptographic operations, resource consumption, and server scalability. Finally, a comparative study with popular messaging applications such as Signal, WhatsApp, and Telegram is provided to evaluate the proposed solution in terms of security features and cryptographic efficiency.

4.2 Security Analysis

4.2.1 Revisiting the Threat Model

We reassess the system using the STRIDE framework to ensure that all relevant threats are mitigated by the proposed design as shown in Table 4.1.

Table 4.1 STRIDE Mapping

Category	Threat	Mitigation in Proposed Design
Spoofing	Identity impersonation	Long-term identity keys + key verification (fingerprints)
Tampering	Data Integrity Violation	Auth Tags
Repudiation	User denies sending a message	Authenticated sessions, server-maintained metadata/logs, message timestamps
Information Disclosure	Message leakage	End-to-end encryption + forward secrecy
Denial of Service	Flooding / resource exhaustion	Server validation
Elevation of Privilege	Unauthorized access	Secure session management + key isolation

The design leverages mechanisms inspired by Signal Protocol to address these threats.

4.2.2 Attack Resistance Testing

The security of the proposed hybrid ECC-AES encryption model was validated by simulating attacks that test their claims. The following section outlines the methodologies adopted to carry out tests for attacks on the model.

4.2.2.1 MitM Attacks

MitM attacks happen when the attacker tries to interfere with communications between two parties. The security measures that are used against such attacks include the following:

Transport Layer Security (TLS 1.3): Communications between the mobile client and the server are secured with TLS 1.3 via HTTPS connections. Thus, both passive and active MITM attacks on the transport layer are prevented. Additionally, certificate pinning

was employed at the Android client side to prevent rogue certificate authorities from issuing certificates for our service.

End-to-End Encryption: Even if a man-in-the-middle can succeed in breaking the TLS layer by gaining access to the server or performing the attack on the transport layer level, he will not be able to read or change the message content. Messages are protected with end-to-end encryption and are encrypted on the device of the sending party with AES-256-GCM keys created through the ECDH protocol and key ratcheting.

4.2.2.2 Possible Scenarios for Compromised Keys

In this test, we will evaluate how well the system can protect messages from being decrypted in case some keys are compromised, we will validate Forward Secrecy and Post-compromise Security. Possible Test Scenarios:

Long-term Identity Key Compromise: The adversary gets access to the user's long-term private key of X25519 curve.

Outcome: No decryption possible because of session establishment with ephemeral key exchange (ECDH with ephemeral X25519 keys).

Message or Chain Key Compromise: An adversary compromises any message or chain key at one particular point in time.

Outcome: Symmetric ratcheting is used; future messages will not be compromised.

Temporary Device Compromise Recovery: The user is able to continue communicating using his application after the temporary device compromise.

Outcome: Using periodic Diffie-Hellman and symmetric ratcheting ensures Post-Compromise Security.

4.2.3 Server Compromise Simulation

A server compromise scenario was considered to evaluate the security of the proposed system. In this scenario, an attacker is assumed to gain unauthorized access to the messaging server and attempt to obtain user data or decrypt exchanged messages.

Because the application uses end-to-end encryption, message encryption and decryption are performed only on client devices. The server stores encrypted ciphertext and public keys required for communication, while private keys remain on the users' devices. Therefore, access to the server alone is insufficient to decrypt message content.

User passwords are also protected using **bcrypt** hashing before storage, which prevents attackers from obtaining plaintext passwords directly from the database. As a result, the attacker can only access limited plaintext information stored by the server, such as usernames, timestamps, and message routing data.

The use of **ECDH** key exchange and **AES-GCM** encryption further strengthens the confidentiality and integrity of communications by ensuring that encrypted data cannot be interpreted without the corresponding session keys.

4.2.4 Limitations

Although the system protects message content using end-to-end encryption, some metadata is still visible. This includes information such as sender and receiver IDs, timestamps, online status, and delivery details. While the message content remains secure, this information can still be observed.

This is a limitation of the current implementation and could be addressed in the future.

4.3 Performance Evaluation

4.3.1 Cryptographic Latency

This section assesses the performance of the primary crypto functions used within the Android app. The tests were conducted on a real device using release mode. Below is the time taken by each process when tested.

4.3.1.1 Key Pair Generation Time (X25519)

The generation of the X25519 key pair is a critical operation performed during user login and occasionally during ratcheting.

Average Time: 24.89 ms

This duration is considered acceptable, as key pair generation typically occurs only during device creation or when establishing new sessions, and does not affect real-time messaging performance.

4.3.1.2 ECDH Shared Secret Computation

This operation includes the execution of the X25519 Diffie-Hellman key exchange and subsequent key derivation using HKDF-SHA256 to generate the root key.

Average Time: 11.54 ms

The results show that the ECDH shared secret computation is efficient and suitable for mobile devices. These operations are performed mainly during session establishment and ratchet steps.

4.3.1.3 AES-GCM Encryption / Decryption Time

AES-256-GCM is used for encrypting and decrypting all messages and attachments. Tests were performed with different payload sizes to simulate real usage scenarios (text messages and media).

Average Encryption Time (small messages < 200 bytes): 0.85 ms

Average Decryption Time (small messages): 0.33 ms

Average Encryption Time (small messages ≈ 2 mb): 4.85 ms

Average Decryption Time (small messages ≈ 2 mb): 4.20 ms

Analysis: The results clearly demonstrate that AES-256-GCM is highly efficient, especially for typical message sizes (text and small media). Even when encrypting large files (over 2 MB), the operation remained under 5 milliseconds, which is negligible for the user experience. This excellent performance is largely due to hardware acceleration available on modern Android devices.

4.3.2 Resource Consumption (CPU)

CPU consumption was evaluated by interacting with the application and executing all computationally intensive operations, including key generation, ECDH key exchange, AES-GCM encryption and decryption, as well as various UI-related tasks. The objective was to observe the application's behavior under realistic usage conditions and measure its processing overhead. The following graph, obtained using the Android Studio CPU Profiler, illustrates the CPU utilization during these operations. As shown in the diagram, the application does not consume a significant amount of CPU resources, indicating efficient performance even during heavy cryptographic operations. See Figure 4.1.



Figure 4.1: CPU Usage (Android Studio Profiler)

4.4 Comparative Study

To better position the developed secure messaging system within the current landscape, a comparative analysis was conducted with two leading applications: Signal and WhatsApp.

The comparison focuses on key security and functional features. As shown in Table 4.2.

Table 4.2 E2E Protocol Security and Features Comparison

Feature	Payn (our project)	Signal	WhatsApp
End-to-End Encryption (E2EE)	Yes — X25519 + HKDF + AES-GCM, server stores / forwards ciphertext only,	Yes — Signal Protocol (X3DH + Double Ratchet),	Yes — Signal-derived protocol (Meta implementation)
Forward Secrecy	Partial — per-message chain keys + DH ratchet on new ephemerals, first epoch uses identity ECDH (no prekeys / X3DH); past messages weaker if long-term identity key is compromised	Yes — ephemeral keys + one-time prekeys; new keys per session/message epochs,	Yes — same protocol family as Signal
Ratcheting Mechanism	Yes — Double Ratchet (Signal-based)	Yes — full Double Ratchet (send/receive chains, DH ratchet, skipped-message handling)	Yes — Double Ratchet (Signal-based)
Multi-Device Support	Yes — linked devices, independent sessions per device,	Yes — linked devices, independent sessions per device,	Yes — companion / multi-device linked sessions
Metadata Protection	Low — server sees chat membership, sender/recipient user+device IDs, timestamps, message_counter, ephemeral_public_key, identity keys; FCM data push includes ciphertext + crypto metadata.	Medium–High — minimal server knowledge of content; still some metadata (registration, delivery),	Low–Medium — E2EE for content; Meta sees metadata (who, when, group membership, etc.)
Open Source	Yes (ourrepos in github) protocol is custom, not libsignal,	Yes — client + libsignal,	No — client and server closed source; protocol based on Signal's published work
Offline Message Delivery	Yes — encrypted messages stored on server per device. fetched via REST when user returns online,	Yes — encrypted messages queued on Signal servers until device syncs,	Yes — messages queued on Meta servers until all devices receive
Push Notifications	Yes — FCM data push. device decrypts locally for notification preview.	Yes — FCM/APNs with privacy-focused minimal payloads,	Yes — FCM/APNs; preview decrypted on device

This comparison demonstrates that the proposed system reaches a solid level of security while remaining lightweight and feasible for real-world deployment on Android devices.

4.5 Limitations

- End-to-end encryption (E2EE) is not fully supported for group messaging.
- End-to-end encryption (E2EE) is not implemented for voice and video calls.
- Message editing and deletion are not fully supported under encrypted communication.
- The system does not rely on an SMS provider for messaging or authentication.

4.6 Conclusion

This chapter presented a comprehensive evaluation of the implemented secure messaging system through security testing, performance analysis, and comparative study.

The attack resistance tests demonstrated that the system successfully resists common threats. Performance measurements conducted in release mode on real Android devices showed excellent results. Cryptographic operations proved to be efficient, These results confirm that the hybrid ECC/AES approach is well-suited for mobile environments with acceptable impact on CPU usage. The feature comparison with Signal and WhatsApp highlighted that the developed system achieves a solid level of security.

Overall, the evaluation validates the effectiveness of the proposed design and implementation. The system successfully meets its core security objectives while delivering satisfactory performance. However, some limitations remain, notably in full group messaging support, voice call and video call supports, which open promising directions for future enhancements.

General Conclusion

This thesis presented the design, implementation, and evaluation of a secure mobile messaging system based on a hybrid ECC/AES cryptographic architecture. The main objective was to create a practical and secure messaging application that guarantees true end-to-end encryption while maintaining good performance and usability on mobile devices.

Although the current version of Payn provides robust security for one-to-one messaging, several important enhancements are planned for future development. These include:

- Implementation of **End-to-End Encryption for group conversations** with proper group ratcheting mechanisms.
- Support for **secure voice and video calls** with end-to-end encrypted media streams.
- Secure cloud backup solutions that preserve end-to-end encryption.
- Comprehensive usability testing with real users to improve the overall experience.
- These future improvements will transform Payn into a more complete and competitive secure messaging solution while maintaining its core principle of minimal server trust and strong privacy protection.

This project contributes to the growing field of secure communication tools and demonstrates that it is possible to develop privacy-focused applications that balance security, performance, and usability.

Bibliography

[1] A. Rastogi and J. Hendler, “WhatsApp Security and Role of Metadata in Preserving Privacy,”1701.06817, 2017.

[2] S. Kamara, M. K. Reiter, and B. Ford, “Outside Looking In: Approaches to Content Moderation in End-to-End Encrypted Systems,”2202.04617, 2022.

[3] National Institute of Standards and Technology, Guideline for Using Cryptographic Standards in the Federal Government, NIST SP 800-175B Rev. 1, Mar. 2020. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-175B-rev1.pdf

[4] R. Shirey, “Internet security glossary, version 2,” RFC 4949, IETF, Aug. 2007. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4949

[5] W. Stallings, Cryptography and Network Security: Principles and Practice, 7th ed., Pearson, 2017.

[6] M. Handley, V. Jacobson, and C. Perkins, “SDP: Session description protocol,” RFC 8866, IETF, Jan. 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8866

[7] J. Postel, “Internet official protocol standards,” RFC 2500, IETF, Jun. 1999. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2500.html

[8] National Institute of Standards and Technology, An Introduction to Information Security, NIST SP 800-12 Rev. 1, Jun. 2017. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-12r1.pdf

[9] H. Krawczyk, M. Bellare, and R. Canetti, “HMAC: Keyed-hashing for message authentication,” RFC 2104, IETF, Feb. 1997. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2104

[10] International Organization for Standardization, ISO/IEC 27000:2018 Information technology—Security techniques—Information security management systems—Overview and vocabulary, 2018. [Online]. Available: https://www.iso.org/standard/73906.html

[11] J. Katz and Y. Lindell, Introduction to Modern Cryptography, 2nd ed. Chapman and Hall/CRC, 2014. [Online]. Available: https://data.ehealthireland.ie/textbooks/threads/HomePages/Katz_Lindell_Introduction_Modern_Cryptography_Solutions.pdf

[12] D. McGrew, “AES Galois counter mode (GCM) cipher suites for TLS,” RFC 5288, IETF, Aug. 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5288

[13] M. Dworkin, Recommendation for Block Cipher Modes of Operation, NIST SP 800-38A, 2001. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

[14] Y. S. Alslman, A. Ahmad, and Y. AbuHour, “Enhanced and authenticated cipher block chaining mode,” Bulletin of Electrical Engineering and Informatics, vol. 12, no. 4, pp. 2357–2362, Aug. 2023, doi: 10.11591/eei.v12i4.5113, [Online]. Available: https://doi.org/10.11591/eei.v12i4.5113

[15] H. Krawczyk, M. Bellare, and R. Canetti, “HMAC: Keyed-hashing for message authentication,” RFC 2104, IETF, Feb. 1997. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2104

[16] D. McGrew, “An interface and algorithms for authenticated encryption,” RFC 5116, IETF, Jan. 2008. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5116

[17] National Institute of Standards and Technology, Advanced Encryption Standard (AES), FIPS PUB 197, 2001. [Online]. Available: https://csrc.nist.gov/pubs/fips/197/final

[18] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM, vol. 21, no. 2, pp. 120–126, 1978. [Online]. Available: https://people.csail.mit.edu/rivest/Rsapaper.pdf

[19] National Institute of Standards and Technology, Digital Signature Standard (DSS), FIPS PUB 186-4, 2013. [Online]. Available: https://csrc.nist.gov/publications/detail/fips/186/4/final

[20] E. Rescorla, “The transport layer security (TLS) protocol version 1.3,” RFC 8446, IETF, Aug. 2018. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8446

[21] K. Ermoshina, F. Musiani, and H. Halpin, “End-to-end encrypted messaging protocols: An overview,” in Internet Science, LNCS 9934, Springer, 2016, pp. 244–254.

[22] Signal, “WhatsApp’s Signal protocol integration is now complete,” Signal Blog, 2016.

[23] D. Bohn, “Google is rolling out end-to-end encryption for RCS,” The Verge, Nov. 19, 2020.

[24] S. R. R. Reddy and P. Rajesh, “Secured end-to-end encryption using X3DH protocol with double ratchet algorithm and its limitations,” [Online]. Available: https://www.researchgate.net/publication/362387381_SECURED_END-TO-END_ENCRYPTION_USING_X3DH_PROTOCOL_WITH_DOUBLE_RATCHET_ALGORITHM_AND_ITS_LIMITATIONS

[25] K. Ermoshina, F. Musiani, and H. Halpin, “End-to-end encrypted messaging protocols: An overview.”. [Online]. Available: https://inria.hal.science/hal-01426845/document

[26] J. Ginesin and C. Nita-Rotaru, “A formal, symbolic analysis of the Matrix cryptographic protocol suite,”. [Online]. Available: https://www.researchgate.net/publication/383413113_A_Formal_Symbolic_Analysis_of_the_Matrix_Cryptographic_Protocol_Suite

[27] E. Rescorla, “The transport layer security (TLS) protocol version 1.3,” RFC 8446, IETF, Aug. 2018. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8446

[28] R. Fielding et al., “Hypertext transfer protocol—HTTP/1.1,” RFC 2616, IETF, Jun. 1999. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2616

[29] I. Fette and A. Melnikov, “The WebSocket protocol,” RFC 6455, IETF, Dec. 2011. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6455

[30] P. W. Saint-Andre, “Media type specifications and registration procedures,” RFC 8835, IETF, Sep. 2020. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8835.html

[31] Telegram FZ-LLC, "End-to-End Encryption," Telegram Core API Documentation. [Online]. Available: https://core.telegram.org/api/end-to-end.

[32] Meta Platforms, Inc., "Messenger End-to-End Encryption Overview," Engineering at Meta, Dec. 6, 2023. [Online]. Available: https://engineering.fb.com/wp-content/uploads/2023/12/MessengerEnd-to-EndEncryptionOverview_12-6-2023.pdf.

[33] K. Ermoshina, F. Musiani, and H. Halpin, “Threat models over space and time: A case study of end-to-end-encrypted messaging applications,”. [Online]. Available: https://www.researchgate.net/publication/380808774_Threat_models_over_space_and_time_A_case_study_of_end-to-end-encrypted_messaging_applications

[34] M. Marlinspike and T. Perrin, The Signal Protocol, Open Whisper Systems, 2016.

[35] European Union Agency for Cybersecurity (ENISA). Privacy and Data Protection in Mobile Applications.ENISA Report, 2017.

[36] “SUN ECC SSL performance,” Princeton University. [Online]. Available: https://www.princeton.edu/~rblee/ELE572Papers/ECCpapers/SUN_ECC_SSLperformance.pdf

[37] K. K. Nalla, “Securing chat applications: Strategies for end-to-end encryption and cloud data protection,” World Journal of Advanced Engineering Technology and Sciences, vol. 13, no. 2, pp. 528–540, 2024, doi: 10.30574/wjaets.2024.13.2.0634. [Online]. Available: https://wjaets.com/sites/default/files/fulltext_pdf/WJAETS-2024-0634.pdf

We would like to thank our supervisor, family, and colleagues for their valuable support and guidance during this research.

“Encryption works. Properly implemented strong crypto systems are one of the few things that you can rely on. Unfortunately, endpoint security is so terrifically weak that NSA can frequently find ways around it.” **Edward Snowden**

Thanks

2025/2026