Università degli studi di Cagliari

Facoltà di Ingegneria e Architettura

Computer Engineering, Cybersecurity and Artificial Intelligence

Secure messaging application

Network Security

Giancarlo Pederiva, Nicola Scema

Contents

1 Introduction

In the context of digital communications, security has become a fundamental priority, especially in messaging applications where the transmission of confidential and sensitive data is becoming more widespread. The objective of this project is to develop a chat system that provides secure end-to-end communication, allowing two or more users to communicate with each other while ensuring the confidentiality, authenticity, and integrity of exchanged messages.

The idea behind our project consists of an instant messaging client that uses a central server to establish communications between the various users connected to the server. To ensure the security of communications, each client is equipped with a pair of persistent cryptographic keys and an ephemeral key, per session, used to establish an encrypted channel created with a customised handshake protocol.

The main objectives of the project are:

- Design and implement a chat client with an easy and intuitive, and accessible graphical interface capable of managing multiple users simultaneously.
- Securely manage user identities by generating and storing public/private key pairs.
- Realize a secure handshake protocol to negotiate temporary session keys and establish crypted end-to-end communications between users.
- Encrypt and decrypt messages exchanged using modern cryptographic algorithms (X25519 for key agreement, AES-GCM for message encryption).
- Avoid man-in-the-middle (MITM) attacks through the use of digital signatures and public key fingerprints.

2 Scenario

The system is designed to simulate secure instant messaging between two or more users connected over potentially insecure networks such as public Wi-Fi or untrusted LANs. Communication takes place through a central server that acts purely as a message relay and is considered untrusted, thus it cannot decrypt, store, or alter the content of exchanged messages.

This setup reflects realistic conditions in which users might be geographically distributed, connecting from different devices and locations, while still requiring strong guarantees of confidentiality, integrity, and authentication.

2.1 Example User Flow

1. User A opens the chat client, connects to the server via a TLS-secured WebSocket channel, and announces their presence.
2. User B connects to the same server and initiates a session with User A.
3. Both clients exchange their public identity keys and ephemeral session keys using a secure handshake protocol.
4. Once the session is established, messages are encrypted end-to-end before being sent to the server for relay.
5. Even if intercepted, the messages cannot be decrypted by the server or any third party.

This scenario models a secure peer-to-peer conversation mediated by an untrusted relay, a pattern often used in modern secure messaging applications.

In addition to one-to-one sessions, the system also supports a **broadcast chat mode**, where multiple clients share a common group state. A fresh epoch secret is distributed among members whenever someone joins or leaves, ensuring that only current participants can access group messages. Messages are encrypted end-to-end and relayed to all members by the server, which remains oblivious to their content. This enables secure multi-user conversations with forward secrecy and minimal trust in the server.

3 System Architecture

3.1 Overview

The proposed system is built upon a **client-server** architecture in which the server acts as a message relay rather than a trusted intermediary. This choice comes from the security goal of end-to-end encryption (E2EE): all encryption and decryption occur exclusively on the client side, ensuring that the server is unable to read, modify, or forge messages.

The architecture supports multiple clients connected simultaneously over **secure WebSocket connections** (`wss://`), which are themselves protected by TLS. TLS provides confidentiality and integrity at the transport layer, preventing network-level attackers from interfering with the WebSocket channel. However, TLS alone is not sufficient for our threat model, because it protects only the channel between a client and the server, not between two clients. For this reason, the application implements an application-layer encryption protocol on top of TLS.

Our threat model assumes that:

- The network may be insecure (e.g., public Wi-Fi with potential man-in-the-middle attackers).
- The server may be honest-but-curious or even compromised, and thus cannot be trusted with message contents.
- Attackers may attempt to impersonate other users, replay messages, or alter them in transit.

The design therefore, ensures that even if the server or the network is compromised, no plaintext communication or private keys are exposed.

3.2 Components

The system is composed of three main components, each with its own security responsibilities:

Client: The client represents the end user inside the system. While the user simply types and reads messages, the client transparently manages key generation, session establishment, and message encryption to ensure communication remains private and authenticated, hiding the complexity of cryptographic operations from the user. Client's main responsibilities include:

- A graphical user interface (GUI) for user interaction.
- Generation and secure storage of a persistent `Ed25519` identity key pair.
- On-demand generation of ephemeral `X25519` keys for each new session.
- Execution of the handshake protocol to authenticate peers and derive shared session keys.
- Encryption and decryption of all outgoing and incoming messages using `AES-256-GCM`.

Server: The server is designed to be minimal and unintrusive. Instead of interpreting or modifying the data it handles, its main purpose is to coordinate connections and deliver ciphertext messages between participants. This architecture keeps trust requirements low: the system remains secure even if the server is compromised. Its tasks are limited to:

- Maintaining a list of connected users and their advertised public keys.
- Acting solely as a relay for encrypted messages.
- Never storing or processing plaintext messages.
- Operating over TLS-secured WebSockets to protect against network-level interference.

In order to support TLS, the server requires a certificate-key–key pair. Rather than relying on external tools such as OpenSSL to generate and provision these files, our implementation creates them automatically at startup if none are present in the `certs/` directory. The procedure generates a 2048-bit RSA private key and a self-signed X.509 certificate, valid for ten years, with both `localhost` and `127.0.0.1` specified as subject alternative names (SANs). The resulting files `server.key` and `server.crt` are stored for reuse across sessions.

This design choice simplifies deployment in a development or testing environment: TLS is always available without manual configuration steps. Compared to manually invoking OpenSSL, our approach eliminates external dependencies, ensures consistency across different machines, and integrates directly into the server's startup routine. While self-signed certificates naturally trigger verification warnings in clients, they are sufficient for localhost experimentation and classroom use. In a production setting, however, these certificates should be replaced with ones issued by a trusted Certificate Authority (CA).

Cryptographic Primitives: The cryptographic layer is the foundation of the system's security. Instead of relying on a single algorithm, it combines multiple well-established primitives, each addressing a specific aspect of confidentiality, integrity, and authenticity. Together, they form a robust protocol that balances efficiency with modern security guarantees. The primitives used are:

- `Ed25519` for identity authentication and public key verification.
- `X25519` for elliptic-curve Diffie–Hellman (ECDH) key exchange.
- HKDF with SHA-256 for deriving independent symmetric keys from the shared secret.
- `AES-256-GCM` for authenticated encryption of messages.

3.3 Diagram

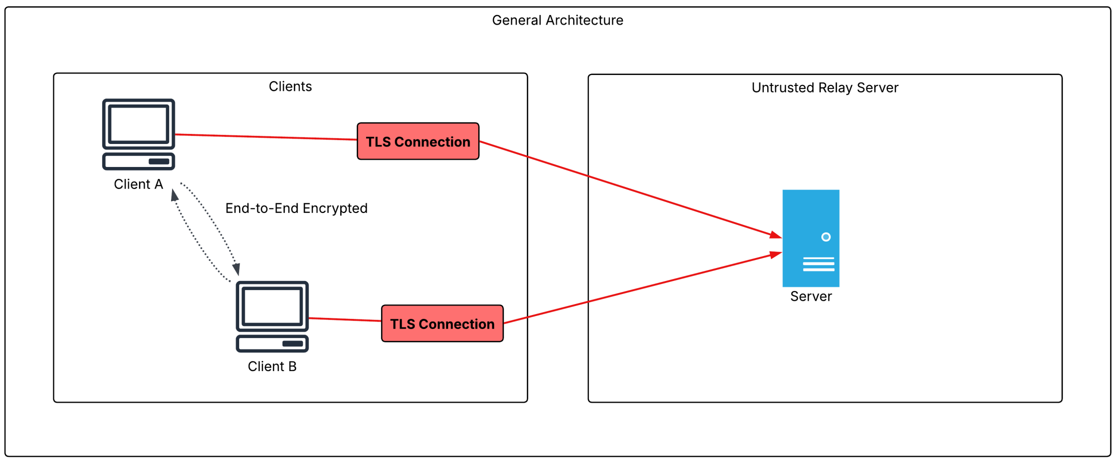


Figure 1: Logical system architecture showing secure end-to-end communication over an untrusted relay server.

In the diagram:

- Each client establishes a **TLS-protected WebSocket** connection to the relay server. This prevents passive eavesdropping and message tampering between the client and the server.
- During the handshake, clients exchange:
    1. Persistent identity keys (Ed25519) to authenticate themselves.
    2. Ephemeral Diffie–Hellman keys (X25519) to establish session-specific shared secrets.

  These exchanges are signed to prevent man-in-the-middle attacks and verified against known key fingerprints.
- Once the handshake is complete, all messages are encrypted end-to-end using `AES-256-GCM` with keys derived from the ECDH output via HKDF.
- The server merely forwards the ciphertext without the ability to decrypt it.

This layered approach (TLS at the transport layer, E2EE at the application layer) ensures that:

1. Network attackers cannot intercept or modify WebSocket traffic.
2. The server cannot read or alter messages, even if compromised.
3. Compromise of one session's keys does not endanger past or future sessions (**forward secrecy**).

3.4 Communication Flow

The high-level communication process works as follows:

1. Each client connects to the server over a TLS-secured WebSocket channel.
2. The initiating client requests the peer's public identity key and begins the handshake by sending its signed ephemeral key.
3. The peer responds with its own signed ephemeral key.
4. Both clients compute a shared secret using X25519 and derive two symmetric keys via HKDF (one for sending, one for receiving).
5. Encrypted messages are transmitted through the server as opaque ciphertexts.

Each step of this process is designed to ensure one or more core security properties:

- **Authentication** — guaranteed by the use of persistent Ed25519 identity keys and signature verification.
- **Confidentiality** — ensured by AES-256-GCM encryption of all message payloads.
- **Integrity** — provided by AES-GCM authentication tags and TLS channel protections.
- **Forward secrecy** — achieved by using ephemeral keys for each session.

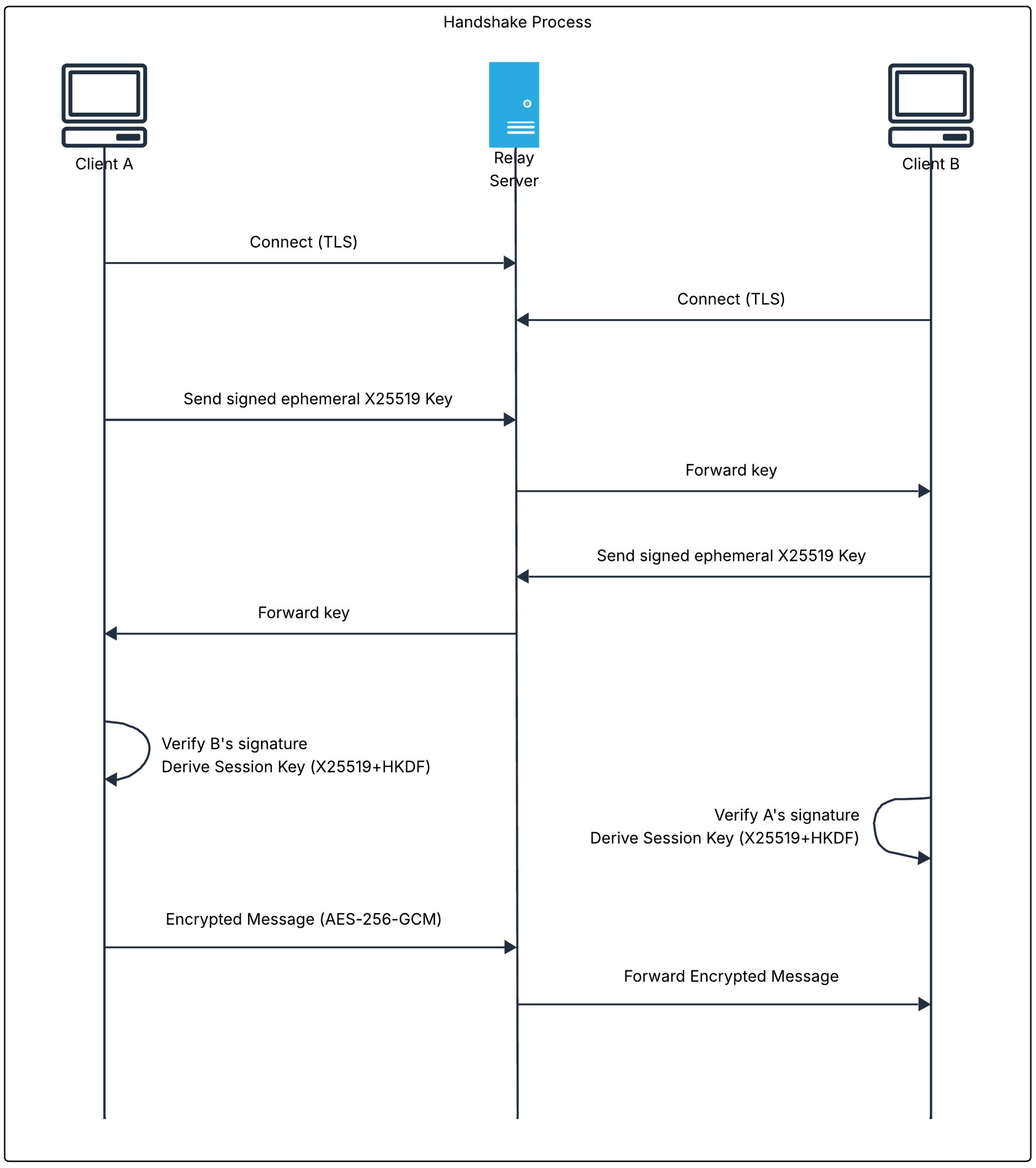

Handshake Process
Client A
Relay Server
Client B
Connect (TLS)
Connect (TLS)
Send signed ephemeral X25519 Key
Forward key
Send signed ephemeral X25519 Key
Forward key
Verify B's signature
Derive Session Key (X25519+HKDF)
Verify A's signature
Derive Session Key (X25519+HKDF)
Encrypted Message (AES-256-GCM)
Forward Encrypted Message

Figure 2: Handshake and key exchange: authenticated ephemeral key exchange using Ed25519 signatures and X25519 Diffie–Hellman, producing session keys via HKDF.

4 Design and Implementation

This chapter describes in detail how the cryptographic components, communication logic, and supporting libraries are integrated to achieve a secure end-to-end chat application. The focus is on the practical aspects of implementing the design while adhering to modern security best practices.

4.1 Key Management

Secure key management is the cornerstone of any end-to-end encrypted system. In our implementation, keys are generated and stored locally by each client, never leaving the device in unencrypted form.

- **Persistent Identity Key** Each user generates an `Ed25519` key pair upon first launch of the client. The **private key** is stored locally in a secure format (not transmitted to the server or other clients), while the **public key** acts as the user's cryptographic identity. This persistent key allows peers to authenticate each other in future sessions, preventing impersonation attacks.
- **Ephemeral Key Exchange** For every new chat session, a fresh `X25519` key pair is generated. Ephemeral keys provide **forward secrecy**: if a session key is ever compromised, past and future conversations remain secure because they use different ephemeral key material.
- **Signature Verification** Before an ephemeral public key is accepted, it is signed using the sender's persistent Ed25519 private key and verified by the recipient using the known persistent public key. This prevents man-in-the-middle (MITM) attacks by ensuring that the ephemeral key truly belongs to the claimed identity.

4.2 Session Key Derivation

Once both parties have exchanged and authenticated ephemeral public keys, they perform an `X25519` elliptic-curve Diffie–Hellman (ECDH) computation to derive a shared secret.

- The shared secret is passed through an `HKDF` (HMAC-based Key Derivation Function) using `SHA-256` as the hash function.
- From this HKDF output, **two distinct symmetric keys** are derived: one for sending messages and one for receiving messages. This key separation prevents subtle cryptographic vulnerabilities that could arise if the same key were reused in both directions.
- The initiator and responder deterministically derive *mirrored keys* so that each one's encryption key corresponds to the other's decryption key.

This approach ensures that session keys are:

- Unique for each session.
- Unlinkable between sessions.
- Resistant to key compromise (forward secrecy).

4.3 Message Encryption

4.3.1 AES-GCM

The symmetric encryption layer uses AES-256 in **Galois/Counter Mode** (AES-GCM), a modern authenticated encryption with associated data (AEAD) scheme.

AES-GCM provides:

- **Confidentiality** — The plaintext message content is encrypted using a per-message nonce.
- **Integrity and Authenticity** — Each ciphertext includes a cryptographic authentication tag that is verified before decryption. If verification fails, the message is discarded.

GCM mode uses a combination of:

- A *nonce* — a unique value per message, preventing replay and ensuring that identical plaintexts encrypt to different ciphertexts.
- A *counter* — incremented for each message to derive new nonces in a predictable but non-repeating way.

This makes AES-GCM resistant to a variety of attacks, including:

- Ciphertext pattern analysis.
- Replay attacks.
- Bit-flipping attacks (due to authentication tag verification).

The algorithm is computationally efficient and hardware-accelerated on most modern processors, which is why it is also the default choice in protocols such as TLS 1.3 and IPsec.

In our application, AES-GCM operates entirely at the *application layer* above TLS, meaning that messages remain encrypted even if the transport layer is compromised.

- AES-GCM (with associated data) ensures confidentiality + integrity.
- Nonces are derived from a per-message counter to guarantee uniqueness.

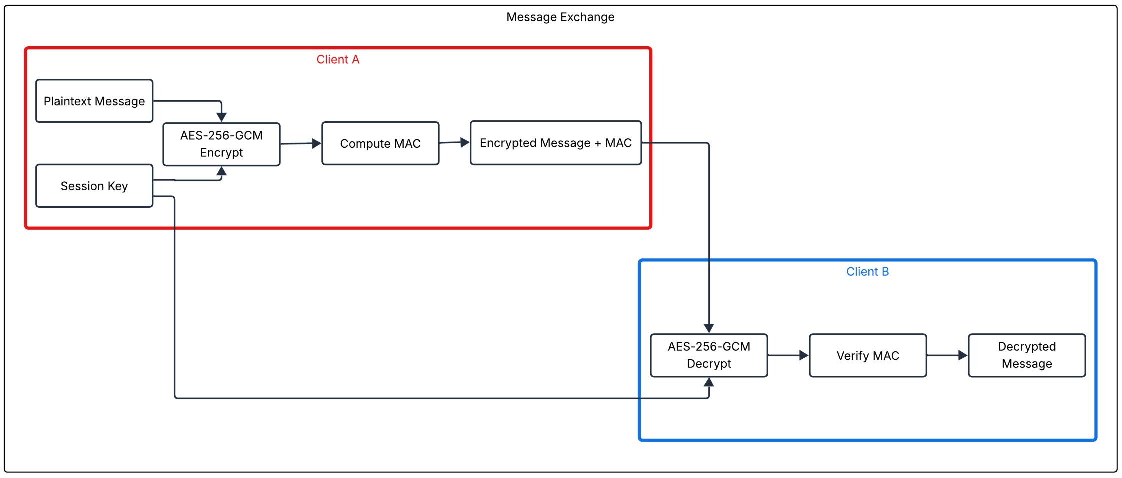


Figure 3: Message encryption workflow: plaintext is encrypted with AES-256-GCM using session keys, producing ciphertext with authentication tags that is sent over the TLS-protected WebSocket.

4.4 Broadcast Chat Functionality

Beyond one-to-one secure messaging, the system supports a **broadcast chat mode** in which multiple clients participate in a shared group conversation. This functionality builds on the same cryptographic foundation as the pairwise protocol, while introducing a group key management mechanism to ensure that only legitimate participants can access broadcast messages.

4.4.1 Joining a Broadcast

When a new client joins a broadcast session, it acts as the *committer*:

1. It requests the current membership list from the server.
2. It generates a fresh random **epoch secret** that defines the group's cryptographic state.
3. For each current member, it encrypts (wraps) the epoch secret with that member's public `X25519` key.
4. It sends the updated membership list and wrapped secrets to the server in a `group_commit` message.

Each member decrypts the wrapped secret using its private `X25519` key and updates its local state to the new epoch.

4.4.2 Sending Broadcast Messages

To send a broadcast message:

1. The sender derives a per-sender AES-256-GCM key from:

    - the epoch secret,
    - the group identifier,
    - the sender's username (to ensure key separation between members).

2. A message counter serves as the nonce, guaranteeing uniqueness across the sender's messages.

3. The ciphertext is transmitted as a `group_message` through the server, which relays it to all group members except the originator.

4.4.3 Receiving Broadcast Messages

Upon receiving a broadcast message:

1. The recipient derives the same per-sender AES-GCM key.

2. Using the sender's counter as the nonce, it decrypts the ciphertext.

3. Only members holding the current epoch secret are able to decrypt successfully.

4.4.4 Leaving a Broadcast

When a member leaves:

1. A new epoch secret is generated by the committer.

2. The secret is wrapped for the remaining members, excluding the leaver.

3. This ensures that past messages remain confidential (backward secrecy) and the departing member cannot read future traffic (forward secrecy).

4.4.5 Security Analysis

The broadcast protocol preserves the core guarantees of the one-to-one system:

- **Confidentiality**: Only members holding the current epoch secret can decrypt broadcast messages. The server never learns epoch secrets or derived keys.

- **Authentication and Integrity**: Messages are encrypted and authenticated with AES-256-GCM, and sender identity is tied to persistent Ed25519 keys.

- **Forward Secrecy**: Epoch rotation on membership changes prevents past or future compromise of group messages.

- **Resistance to Key/Nonce Collisions**: Per-sender derived keys and counters ensure uniqueness across group members, eliminating nonce reuse.

- **Minimal Trust in Server**: The server acts solely as a relay for opaque ciphertext and wrapped keys; compromise of the server reveals no plaintext.

This design resembles modern group messaging protocols such as MLS (Messaging Layer Security), though in a simplified form. It provides a balance between efficiency and strong security guarantees suitable for a relay-based broadcast architecture.

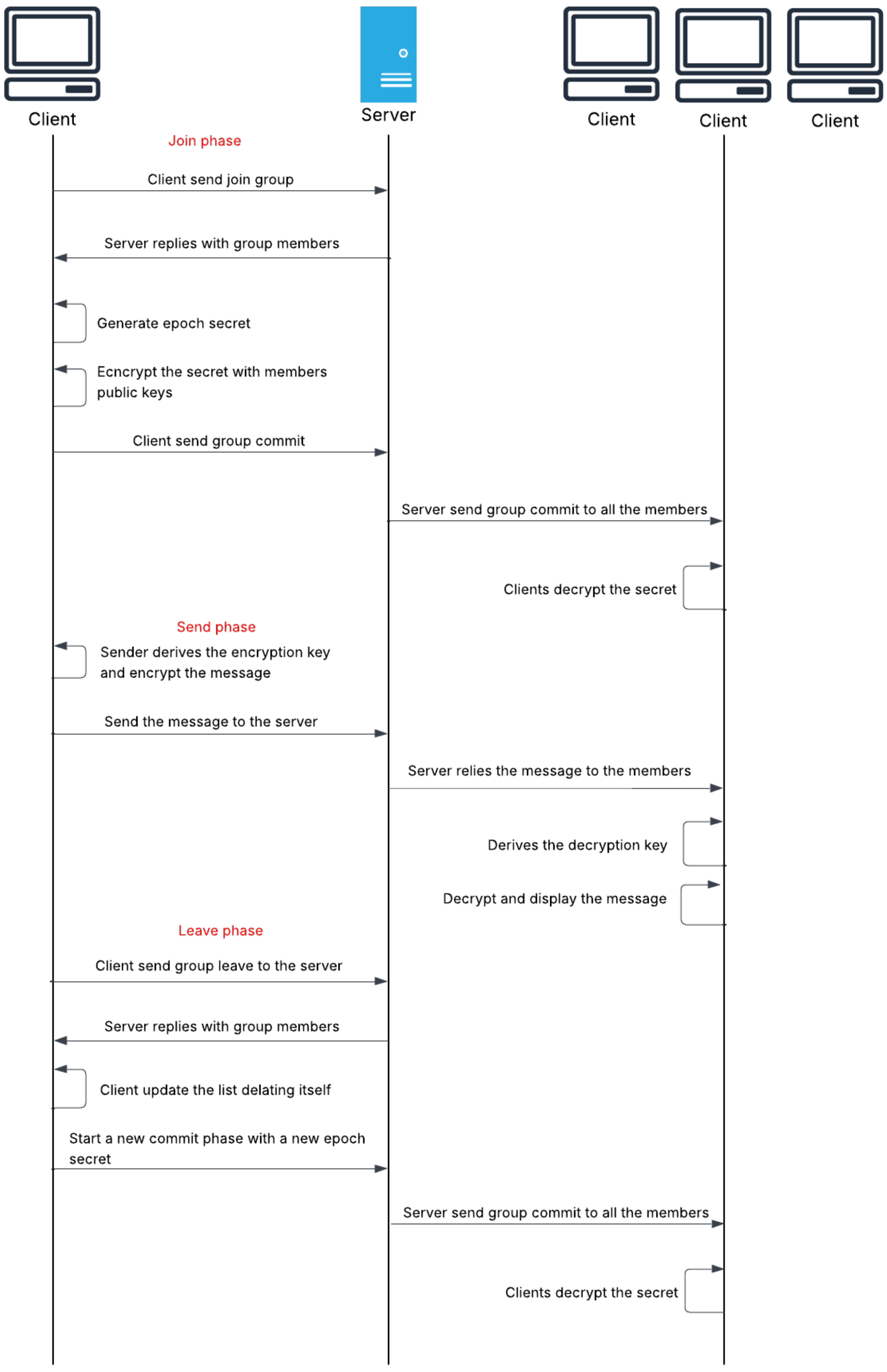

Client
Server
Client
Client
Client
Join phase
Client send join group
Server replies with group members
Generate epoch secret
Ecncrypt the secret with members public keys
Client send group commit
Server send group commit to all the members
Clients decrypt the secret
Send phase
Sender derives the encryption key and encrypt the message
Send the message to the server
Server relies the message to the members
Derives the decryption key
Decrypt and display the message
Leave phase
Client send group leave to the server
Server replies with group members
Client update the list delating itself
Start a new commit phase with a new epoch secret
Server send group commit to all the members
Clients decrypt the secret

Figure 4: Key exchange, sending, receiving, joining and leaving the broadcast chat mechanism

4.5 Libraries Used

We selected Python libraries that provide secure, well-maintained, and high-level abstractions for cryptographic and networking tasks:

- `cryptography` — Core cryptographic primitives: Ed25519 (signatures), X25519 (ephemeral key exchange), AES-GCM (authenticated encryption), and HKDF-SHA256 (key derivation). This library underpins the security of our protocol implementation.
- `websockets` — Enables asynchronous client–server communication over WebSockets, with built-in support for TLS encryption.
- `asyncio` — Python's event-driven concurrency framework, used to handle multiple clients, maintain responsive chat sessions, and manage asynchronous key exchanges.
- `tkinter` — Provides a simple graphical interface for interaction, user prompts, and session management.
- Standard library modules (`os`, `base64`, `hashlib`, `json`) — Support utilities for secure random generation, encoding/decoding keys and ciphertexts, hashing, and structured data exchange.

4.6 Code Structure

To maintain modularity and clarity, the project is organised into separate components:

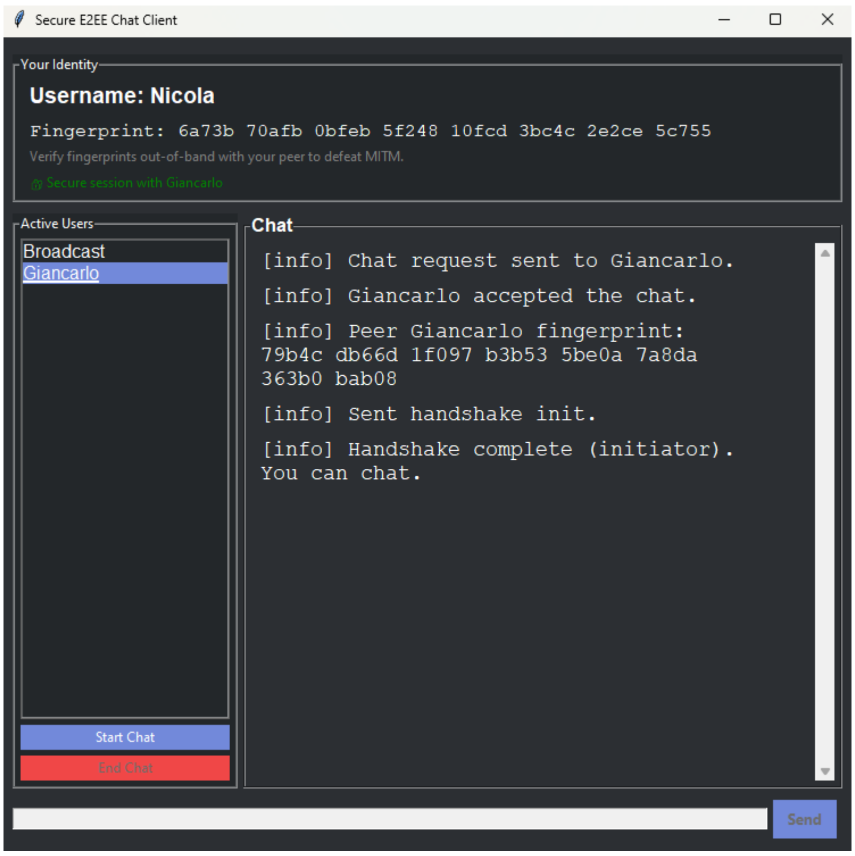
- `client.py` — Handles the user interface, handshake protocol, and message encryption/decryption.
- `server.py` — Implements the WebSocket relay server for user discovery and message routing.
- `common/crypto_utils.py` — Encapsulates all cryptographic operations, including key generation, signing, verification, ECDH, HKDF derivation, and AES-GCM encryption/decryption.

This separation of concerns simplifies debugging, testing, and future extensions to the system.

4.7 Interface

When the client is launched from the terminal, the secure messaging application opens and prompts the user to enter a username. Once the username is submitted, the main interface is displayed to the user.

The interface is divided into three main sections. The top section displays the user's identity, including the username and fingerprint, and the status of the currently selected chat. On the left side, the interface displays all the active clients and the broadcast chat, where a user can start or end client-to-client chats, or join and leave the broadcast chat by selecting the desired entry and using the corresponding buttons. Finally, the right section of the interface contains the selected chat window, where all incoming and outgoing messages are displayed. At the bottom of this section, a text input field allows users to write new messages and send them by clicking the send button.



Username: Nicola

```
Fingerprint: 6a73b 70afb 0bfeb 5f248 10fcd 3bc4c 2e2ce 5c755
```

Verify fingerprints out-of-band with your peer to defeat MITM.

Secure session with Giancarlo

```
[info] Chat request sent to Giancarlo.
[info] Giancarlo accepted the chat.
[info] Peer Giancarlo fingerprint:
79b4c db66d 1f097 b3b53 5be0a 7a8da
363b0 bab08
[info] Sent handshake init.
[info] Handshake complete (initiator).
You can chat.
```

Figure 5: Chat interface

5 Evaluation and Conclusions

In this chapter, we report the results obtained by analysing network traffic with Wireshark to demonstrate the security of the communications carried out with the application we developed.

We have identified the packets that correspond to the main functions of our application, including the handshake, starting a chat with another client, joining the broadcast chat, and finally sending messages in both chats. From the highlighted packets, it is not possible to retrieve meaningful plaintext about clients or payloads. To identify these packets we applied Wireshark filters based on packet length and by performing the operations at distinct instants, which allowed us to map observed records to application actions.

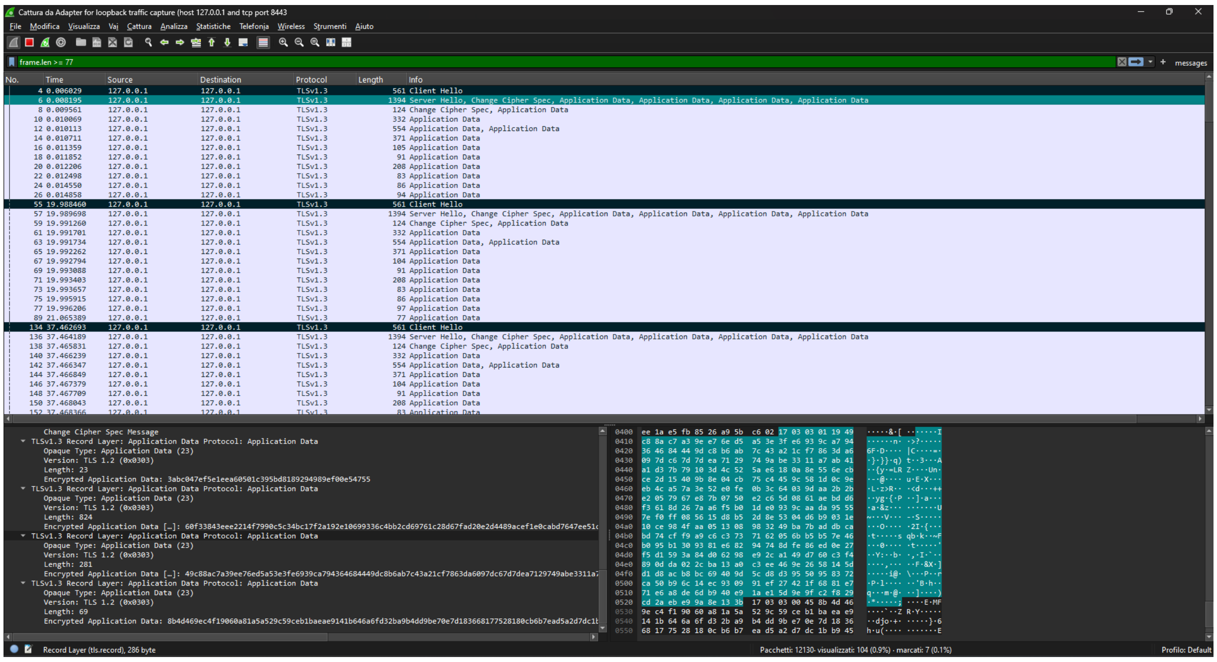
5.1 Initial Handshakes

To evaluate the security of the system, we conducted a packet capture while three clients were connected to the server.

This setup allowed testing both one-to-one secure chats and the group broadcast functionality, while simultaneously monitoring the exchanged traffic with Wireshark.

For clarity of presentation, we filtered out packets shorter than 77 bytes. These packets are still part of the TLS communication, but in practice they mostly correspond to acknowledgements, retransmissions, or very small encrypted TLS records that are not relevant for illustrating the main application-level operations.

Filtering them improves visualisation and focuses the analysis on packets that represent significant cryptographic events (handshakes, key updates, and application ciphertext carrying chat messages). In the raw capture the total packet count is higher because TCP and TLS produce auxiliary traffic to ensure reliable and secure delivery.


Cattura da Adapter for loopback traffic capture (host 127.0.0.1 and tcp port 8443
File Modifica Visualizza Vai Cattura Analizza Statistiche Telefonia Wireless Strumenti Aiuto
messages

```
frame.len >= 77

No.  Time        Source     Destination  Protocol  Length  Info
   4 0.006029    127.0.0.1  127.0.0.1    TLSv1.3    561 Client Hello
   6 0.008195    127.0.0.1  127.0.0.1    TLSv1.3   1394 Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
   8 0.009561    127.0.0.1  127.0.0.1    TLSv1.3    124 Change Cipher Spec, Application Data
  10 0.010069    127.0.0.1  127.0.0.1    TLSv1.3    332 Application Data
  12 0.010113    127.0.0.1  127.0.0.1    TLSv1.3    554 Application Data, Application Data
  14 0.010711    127.0.0.1  127.0.0.1    TLSv1.3    371 Application Data
  16 0.011359    127.0.0.1  127.0.0.1    TLSv1.3    105 Application Data
  18 0.011852    127.0.0.1  127.0.0.1    TLSv1.3     91 Application Data
  20 0.012206    127.0.0.1  127.0.0.1    TLSv1.3    208 Application Data
  22 0.012498    127.0.0.1  127.0.0.1    TLSv1.3     83 Application Data
  24 0.014550    127.0.0.1  127.0.0.1    TLSv1.3     86 Application Data
  26 0.014858    127.0.0.1  127.0.0.1    TLSv1.3     94 Application Data
  55 19.988460   127.0.0.1  127.0.0.1    TLSv1.3    561 Client Hello
  57 19.989698   127.0.0.1  127.0.0.1    TLSv1.3   1394 Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
  59 19.991260   127.0.0.1  127.0.0.1    TLSv1.3    124 Change Cipher Spec, Application Data
  61 19.991701   127.0.0.1  127.0.0.1    TLSv1.3    332 Application Data
  63 19.991734   127.0.0.1  127.0.0.1    TLSv1.3    554 Application Data, Application Data
  65 19.992262   127.0.0.1  127.0.0.1    TLSv1.3    371 Application Data
  67 19.992794   127.0.0.1  127.0.0.1    TLSv1.3    104 Application Data
  69 19.993088   127.0.0.1  127.0.0.1    TLSv1.3     91 Application Data
  71 19.993403   127.0.0.1  127.0.0.1    TLSv1.3    208 Application Data
  73 19.993657   127.0.0.1  127.0.0.1    TLSv1.3     83 Application Data
  75 19.995915   127.0.0.1  127.0.0.1    TLSv1.3     86 Application Data
  77 19.996206   127.0.0.1  127.0.0.1    TLSv1.3     97 Application Data
  89 21.065389   127.0.0.1  127.0.0.1    TLSv1.3     77 Application Data
 134 37.462693   127.0.0.1  127.0.0.1    TLSv1.3    561 Client Hello
 136 37.464189   127.0.0.1  127.0.0.1    TLSv1.3   1394 Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
 138 37.465831   127.0.0.1  127.0.0.1    TLSv1.3    124 Change Cipher Spec, Application Data
 140 37.466239   127.0.0.1  127.0.0.1    TLSv1.3    332 Application Data
 142 37.466347   127.0.0.1  127.0.0.1    TLSv1.3    554 Application Data, Application Data
 144 37.466849   127.0.0.1  127.0.0.1    TLSv1.3    371 Application Data
 146 37.467379   127.0.0.1  127.0.0.1    TLSv1.3    104 Application Data
 148 37.467709   127.0.0.1  127.0.0.1    TLSv1.3     91 Application Data
 150 37.468043   127.0.0.1  127.0.0.1    TLSv1.3    208 Application Data
 152 37.468366   127.0.0.1  127.0.0.1    TLSv1.3     83 Application Data
```

```
    Change Cipher Spec Message
▾ TLSv1.3 Record Layer: Application Data Protocol: Application Data
    Opaque Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 23
    Encrypted Application Data: 3abc047ef5e1eea60501c395bd8189294989ef00e54755
▾ TLSv1.3 Record Layer: Application Data Protocol: Application Data
    Opaque Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 824
    Encrypted Application Data […]: 60f33843eee2214f7990c5c34bc17f2a192e10699336c4bb2cd69761c28d67fad20e2d4489acef1e0cabd7647ee51c
▾ TLSv1.3 Record Layer: Application Data Protocol: Application Data
    Opaque Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 281
    Encrypted Application Data […]: 49c88ac7a39ee76ed5a53e3fe6939ca794364684449dc8b6ab7c43a21cf7863da6097dc67d7dea7129749abe3311a7
▾ TLSv1.3 Record Layer: Application Data Protocol: Application Data
    Opaque Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 69
    Encrypted Application Data: 8b4d469ec4f19060a81a5a529c59ceb1baeae9141b646a6fd32ba9b4dd9be70e7d183668177528180cb6b7ead5a2d7dc1b
```

```
0400  ee 1a e5 fb 85 26 a9 5b  c6 02 17 03 03 01 19 49   ·····&·[ ······I
0410  c8 8a c7 a3 9e e7 6e d5  a5 3e 3f e6 93 9c a7 94   ······n· ·>?·····
0420  36 46 84 44 9d c8 b6 ab  7c 43 a2 1c f7 86 3d a6   6F·D···· |C····=·
0430  09 7d c6 7d 7d ea 71 29  74 9a be 33 11 a7 ab 41   ·}·}}·q) t··3···A
0440  a1 d3 7b 79 10 3d 4c 52  5a e6 18 0a 8e 55 6e cb   ··{y·=LR Z····Un·
0450  ce 2d 15 40 9b 8e 04 cb  75 c4 45 9c 58 1d 0c 9e   ·-·@···· u·E·X···
0460  eb 4c a5 7a 3e 52 e0 fe  0b 3c 64 03 9d aa 2b 2b   ·L·z>R·· ·<d···++
0470  e2 05 79 67 e8 7b 07 50  e2 c6 5d 08 61 ae bd d6   ··yg·{·P ··]·a···
0480  f3 61 8d 26 7a a6 f5 b0  1d e0 93 9c aa da 95 55   ·a·&z··· ·······U
0490  7e f0 ff 08 56 15 d8 b5  2d 8e 53 04 d6 b9 03 1e   ~···V··· -·S·····
04a0  10 ce 98 4f aa 05 13 08  98 32 49 ba 7b ad db ca   ···O···· ·2I·{···
04b0  bd 74 cf f9 a9 c6 c3 73  71 62 05 6b b5 b5 7e 46   ·t·····s qb·k··~F
04c0  b0 95 b1 30 93 81 e6 82  94 74 8d fe 86 ed 0e 27   ···0···· ·t·····'
04d0  f5 d1 59 3a 84 d0 62 98  e9 2c a1 49 d7 60 c3 f4   ··Y:··b· ·,·I·`··
04e0  89 0d da 02 2c ba 13 a0  c3 ee 46 9e 26 58 14 5d   ····,··· ··F·&X·]
04f0  d1 d8 ac b8 bc 69 40 9d  5c d8 d3 95 50 95 83 72   ·····i@· \···P··r
0500  ca 50 b9 6c 14 ec 93 09  91 ef 27 42 1f 68 81 e7   ·P·l···· ··'B·h··
0510  71 e6 a8 de 6d b9 40 e9  1a e1 5d 9e 9f c2 f8 29   q···m·@· ··]····)
0520  cd 2a eb e9 9a 8e 13 3b  17 03 03 00 45 8b 4d 46   ·*·····; ····E·MF
0530  9e c4 f1 90 60 a8 1a 5a  52 9c 59 ce b1 ba ea e9   ····`··Z R·Y·····
0540  14 1b 64 6a 6f d3 2b a9  b4 dd 9b e7 0e 7d 18 36   ··djo·+· ·····}·6
0550  68 17 75 28 18 0c b6 b7  ea d5 a2 d7 dc 1b b9 45   h·u(···· ·······E
```

```
Record Layer (tls.record), 286 byte        Pacchetti: 12130· visualizzati: 104 (0.9%) · marcati: 7 (0.1%)        Profilo: Default
```

Figure 6: TLS handshake packets exchanged between clients and server

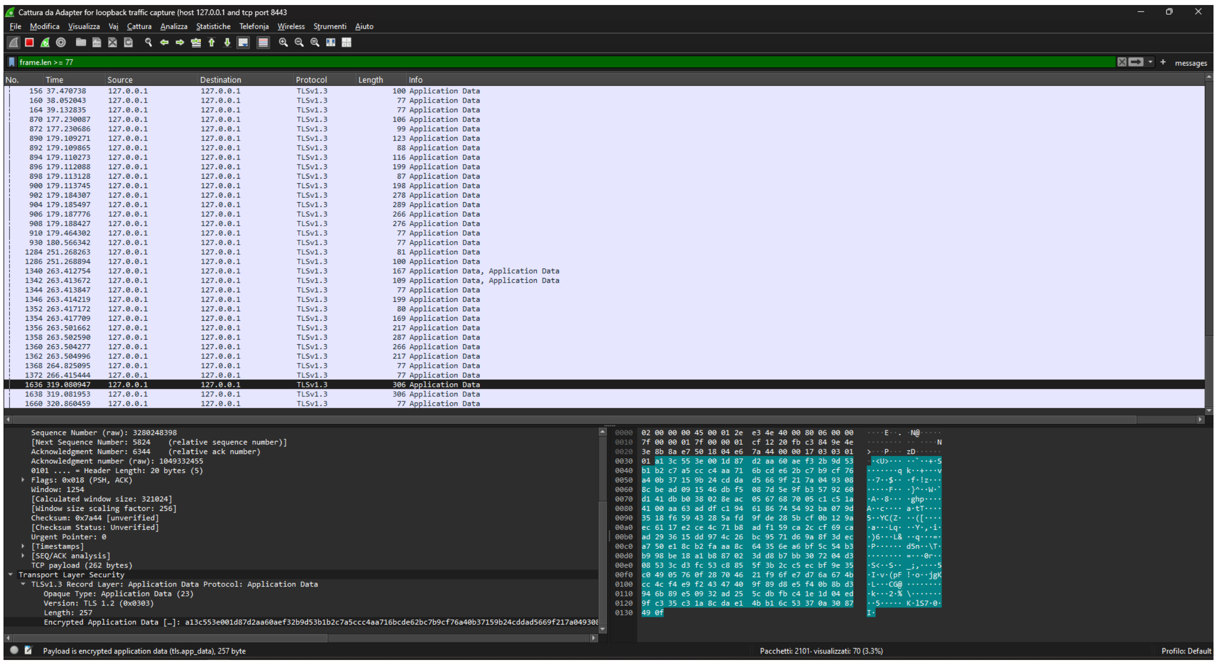
As shown in Figure 6, each client initiates communication with the server through a standard TLS handshake. Every session begins with a *ClientHello* message, followed by the server's response (*ServerHello*) and the subsequent key exchange. While the protocol-level items (message types, lengths, and timing) are visible to an observer, the negotiated cryptographic secrets and session keys remain opaque. This confirms that no attacker observing the traffic could reconstruct the keys or decrypt subsequent communication.

5.2 Private (client-to-client) messages

Figure 7 highlights the packets corresponding to sending a private message from one client to another. The first highlighted packet in the capture is the sender's TLS record directed to the server; the subsequent packet is the server routing the encrypted TLS record to the recipient.

By inspecting these records with Wireshark we can verify the following without exposing plaintext:

- The application-level payloads are carried inside TLS application records and are encrypted end-to-end with session keys negotiated inside TLS and with additional application-level encryption (X25519-derived ephemeral shared secret + AES-GCM) for the message bodies.
- Message integrity and authenticity are provided: AES-GCM provides authenticated encryption for ciphertexts and Ed25519 signatures are used at the application layer to bind sender identity to the message, preventing origin forgery.
- Replay protection is enforced: messages use unique nonces and associated data so that a recorder/replayer cannot successfully replay a previously captured ciphertext without detection.


Cattura da Adapter for loopback traffic capture (host 127.0.0.1 and tcp port 8443
File Modifica Visualizza Vai Cattura Analizza Statistiche Telefonia Wireless Strumenti Aiuto
frame.len >= 77
messages
Profilo: Default

```
No.   Time        Source       Destination  Protocol  Length Info
  156 37.470738   127.0.0.1    127.0.0.1    TLSv1.3      100 Application Data
  160 38.052043   127.0.0.1    127.0.0.1    TLSv1.3       77 Application Data
  164 39.132835   127.0.0.1    127.0.0.1    TLSv1.3       77 Application Data
  870 177.230087  127.0.0.1    127.0.0.1    TLSv1.3      106 Application Data
  872 177.230686  127.0.0.1    127.0.0.1    TLSv1.3       99 Application Data
  890 179.109271  127.0.0.1    127.0.0.1    TLSv1.3      123 Application Data
  892 179.109865  127.0.0.1    127.0.0.1    TLSv1.3       88 Application Data
  894 179.110273  127.0.0.1    127.0.0.1    TLSv1.3      116 Application Data
  896 179.112088  127.0.0.1    127.0.0.1    TLSv1.3      199 Application Data
  898 179.113128  127.0.0.1    127.0.0.1    TLSv1.3       87 Application Data
  900 179.113745  127.0.0.1    127.0.0.1    TLSv1.3      198 Application Data
  902 179.184307  127.0.0.1    127.0.0.1    TLSv1.3      278 Application Data
  904 179.185497  127.0.0.1    127.0.0.1    TLSv1.3      289 Application Data
  906 179.187776  127.0.0.1    127.0.0.1    TLSv1.3      266 Application Data
  908 179.188427  127.0.0.1    127.0.0.1    TLSv1.3      276 Application Data
  910 179.464302  127.0.0.1    127.0.0.1    TLSv1.3       77 Application Data
  930 180.566342  127.0.0.1    127.0.0.1    TLSv1.3       77 Application Data
 1284 251.268263  127.0.0.1    127.0.0.1    TLSv1.3       81 Application Data
 1286 251.268894  127.0.0.1    127.0.0.1    TLSv1.3      100 Application Data
 1340 263.412754  127.0.0.1    127.0.0.1    TLSv1.3      167 Application Data, Application Data
 1342 263.413672  127.0.0.1    127.0.0.1    TLSv1.3      109 Application Data, Application Data
 1344 263.413847  127.0.0.1    127.0.0.1    TLSv1.3       77 Application Data
 1346 263.414219  127.0.0.1    127.0.0.1    TLSv1.3      199 Application Data
 1352 263.417172  127.0.0.1    127.0.0.1    TLSv1.3       80 Application Data
 1354 263.417709  127.0.0.1    127.0.0.1    TLSv1.3      169 Application Data
 1356 263.501662  127.0.0.1    127.0.0.1    TLSv1.3      217 Application Data
 1358 263.502590  127.0.0.1    127.0.0.1    TLSv1.3      287 Application Data
 1360 263.504277  127.0.0.1    127.0.0.1    TLSv1.3      266 Application Data
 1362 263.504996  127.0.0.1    127.0.0.1    TLSv1.3      217 Application Data
 1368 264.825095  127.0.0.1    127.0.0.1    TLSv1.3       77 Application Data
 1372 266.415444  127.0.0.1    127.0.0.1    TLSv1.3       77 Application Data
 1636 319.080947  127.0.0.1    127.0.0.1    TLSv1.3      306 Application Data
 1638 319.081953  127.0.0.1    127.0.0.1    TLSv1.3      306 Application Data
 1660 320.860459  127.0.0.1    127.0.0.1    TLSv1.3       77 Application Data
```

```
    Sequence Number (raw): 3280248398
    [Next Sequence Number: 5824    (relative sequence number)]
    Acknowledgment Number: 6344    (relative ack number)
    Acknowledgment number (raw): 1049332455
    0101 .... = Header Length: 20 bytes (5)
  ▸ Flags: 0x018 (PSH, ACK)
    Window: 1254
    [Calculated window size: 321024]
    [Window size scaling factor: 256]
    Checksum: 0x7a44 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  ▸ [Timestamps]
  ▸ [SEQ/ACK analysis]
    TCP payload (262 bytes)
▾ Transport Layer Security
  ▾ TLSv1.3 Record Layer: Application Data Protocol: Application Data
      Opaque Type: Application Data (23)
      Version: TLS 1.2 (0x0303)
      Length: 257
      Encrypted Application Data […]: a13c553e001d87d2aa60aef32b9d53b1b2c7a5ccc4aa716bcde62bc7b9cf76a40b37159b24cddad5669f217a04930
```

```
0000  02 00 00 00 45 00 01 2e  e3 4e 40 00 80 06 00 00   ····E··. ·N@····
0010  7f 00 00 01 7f 00 00 01  cf 12 20 fb c3 84 9e 4e   ········ ·· ···N
0020  3e 8b 8a e7 50 18 04 e6  7a 44 00 00 17 03 03 01   >···P··· zD······
0030  01 a1 3c 55 3e 00 1d 87  d2 aa 60 ae f3 2b 9d 53   ··<U>··· ··`··+·S
0040  b1 b2 c7 a5 cc c4 aa 71  6b cd e6 2b c7 b9 cf 76   ·······q k··+···v
0050  a4 0b 37 15 9b 24 cd da  d5 66 9f 21 7a 04 93 08   ··7··$·· ·f·!z···
0060  8c be ad 09 15 46 db f5  08 7d 5e 9f b3 57 92 60   ·····F·· ·}^··W·`
0070  d1 41 db b0 38 02 8e ac  05 67 68 70 05 c1 c5 1a   ·A··8··· ·ghp····
0080  41 00 aa 63 ad df c1 94  61 86 74 54 92 ba 07 9d   A··c···· a·tT····
0090  35 18 f6 59 43 28 5a fd  9f de 28 5b cf 0b 12 9a   5··YC(Z· ··([····
00a0  ec 61 17 e2 ce 4c 71 b8  ad f1 59 ca 2c cf 69 ca   ·a···Lq· ··Y·,·i·
00b0  ad 29 36 15 dd 97 4c 26  bc 95 71 d6 9a 8f 3d ec   ·)6···L& ··q···=·
00c0  a7 50 e1 8c b2 fa aa 8c  64 35 6e a6 bf 5c 54 b3   ·P······ d5n··\T·
00d0  b9 98 be 18 a1 b8 87 02  3d d8 b7 bb 30 72 04 d3   ········ =···0r··
00e0  08 53 3c d3 fc 53 c8 85  5f 3b 2c c5 ec bf 9e 35   ·S<··S·· _;,····5
00f0  c0 49 05 76 0f 28 70 46  21 f9 6f e7 d7 6a 67 4b   ·I·v·(pF !·o··jgK
0100  cc 4c f4 e9 f2 43 47 40  9f 89 d8 e5 f4 0b 8b d3   ·L···CG@ ········
0110  94 6b 89 e5 09 32 ad 25  5c db fb c4 1e 1d 04 ed   ·k···2·% \·······
0120  9f c3 35 c3 1a 8c da e1  4b b1 6c 53 37 0a 30 87   ··5····· K·lS7·0·
0130  49 0f                                              I·
```

```
Payload is encrypted application data (tls.app_data), 257 byte      Pacchetti: 2101· visualizzati: 70 (3.3%)
```

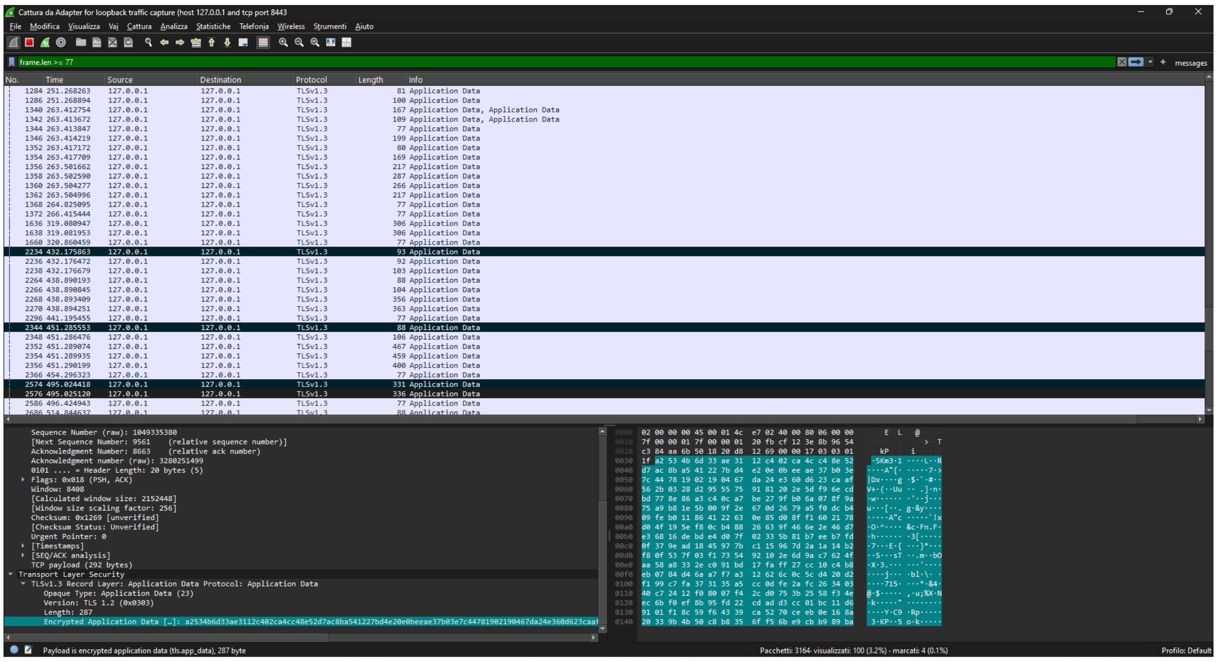
Figure 7: Packets observed when sending a private message (sender → server → recipient).

Because the ciphertexts, nonces, and associated authentication tags are all transmitted within encrypted TLS records, an external observer cannot retrieve message contents or replay them successfully without access to the appropriate keys.

5.2.1 Broadcast (group) messages and group key updates

Figure 8 shows the packets exchanged when a user sends a message in the broadcast chat. In the capture run there were three clients active, two of which were joined to the broadcast group. The server forwards the group ciphertext only to current group members; the packets illustrate that the server routes the encrypted application records to the relevant clients but does not expose plaintext content.

Importantly, the group mechanism implements per-epoch group keys with dynamic membership. When a member leaves the group, a new epoch key is distributed (encrypted for the remaining members) so that departed clients cannot decrypt future group messages. The capture shows key-update events (short encrypted records that differ from ordinary application ciphertexts) and subsequent messages protected under the new epoch key.


Cattura da Adapter for loopback traffic capture (host 127.0.0.1 and tcp port 8443
File Modifica Visualizza Vai Cattura Analizza Statistiche Telefonia Wireless Strumenti Aiuto
messages

```
frame.len >= 77

No.  Time        Source      Destination  Protocol  Length Info
1284 251.268263  127.0.0.1   127.0.0.1    TLSv1.3      81 Application Data
1286 251.268894  127.0.0.1   127.0.0.1    TLSv1.3     100 Application Data
1340 263.412754  127.0.0.1   127.0.0.1    TLSv1.3     167 Application Data, Application Data
1342 263.413672  127.0.0.1   127.0.0.1    TLSv1.3     109 Application Data, Application Data
1344 263.413847  127.0.0.1   127.0.0.1    TLSv1.3      77 Application Data
1346 263.414219  127.0.0.1   127.0.0.1    TLSv1.3     199 Application Data
1352 263.417172  127.0.0.1   127.0.0.1    TLSv1.3      80 Application Data
1354 263.417709  127.0.0.1   127.0.0.1    TLSv1.3     169 Application Data
1356 263.501662  127.0.0.1   127.0.0.1    TLSv1.3     217 Application Data
1358 263.502590  127.0.0.1   127.0.0.1    TLSv1.3     287 Application Data
1360 263.504277  127.0.0.1   127.0.0.1    TLSv1.3     266 Application Data
1362 263.504996  127.0.0.1   127.0.0.1    TLSv1.3     217 Application Data
1368 264.825095  127.0.0.1   127.0.0.1    TLSv1.3      77 Application Data
1372 266.415444  127.0.0.1   127.0.0.1    TLSv1.3      77 Application Data
1636 319.080947  127.0.0.1   127.0.0.1    TLSv1.3     306 Application Data
1638 319.081953  127.0.0.1   127.0.0.1    TLSv1.3     306 Application Data
1660 320.860459  127.0.0.1   127.0.0.1    TLSv1.3      77 Application Data
2234 432.175863  127.0.0.1   127.0.0.1    TLSv1.3      93 Application Data
2236 432.176472  127.0.0.1   127.0.0.1    TLSv1.3      92 Application Data
2238 432.176679  127.0.0.1   127.0.0.1    TLSv1.3     103 Application Data
2264 438.890193  127.0.0.1   127.0.0.1    TLSv1.3      88 Application Data
2266 438.890845  127.0.0.1   127.0.0.1    TLSv1.3     104 Application Data
2268 438.893409  127.0.0.1   127.0.0.1    TLSv1.3     356 Application Data
2270 438.894251  127.0.0.1   127.0.0.1    TLSv1.3     363 Application Data
2296 441.195455  127.0.0.1   127.0.0.1    TLSv1.3      77 Application Data
2344 451.285553  127.0.0.1   127.0.0.1    TLSv1.3      88 Application Data
2348 451.286476  127.0.0.1   127.0.0.1    TLSv1.3     106 Application Data
2352 451.289074  127.0.0.1   127.0.0.1    TLSv1.3     467 Application Data
2354 451.289935  127.0.0.1   127.0.0.1    TLSv1.3     459 Application Data
2356 451.290199  127.0.0.1   127.0.0.1    TLSv1.3     400 Application Data
2366 454.296323  127.0.0.1   127.0.0.1    TLSv1.3      77 Application Data
2574 495.024418  127.0.0.1   127.0.0.1    TLSv1.3     331 Application Data
2576 495.025120  127.0.0.1   127.0.0.1    TLSv1.3     336 Application Data
2586 496.424943  127.0.0.1   127.0.0.1    TLSv1.3      77 Application Data
2686 514.844637  127.0.0.1   127.0.0.1    TLSv1.3      88 Application Data
```

```
      Sequence Number (raw): 1049335380
      [Next Sequence Number: 9561    (relative sequence number)]
      Acknowledgment Number: 8663    (relative ack number)
      Acknowledgment number (raw): 3280251499
      0101 .... = Header Length: 20 bytes (5)
    > Flags: 0x018 (PSH, ACK)
      Window: 8408
      [Calculated window size: 2152448]
      [Window size scaling factor: 256]
      Checksum: 0x1269 [unverified]
      [Checksum Status: Unverified]
      Urgent Pointer: 0
    > [Timestamps]
    > [SEQ/ACK analysis]
      TCP payload (292 bytes)
  v Transport Layer Security
    v TLSv1.3 Record Layer: Application Data Protocol: Application Data
        Opaque Type: Application Data (23)
        Version: TLS 1.2 (0x0303)
        Length: 287
        Encrypted Application Data […]: a2534b6d33ae3112c402ca4cc48e52d7ac8ba541227bd4e20e0beeae37b03e7c44781902190467da24e360d623caa
```

```
0000  02 00 00 00 45 00 01 4c  e7 02 40 00 80 06 00 00   ····E·L··@·····
0010  7f 00 00 01 7f 00 00 01  20 fb cf 12 3e 8b 96 54   ········ ···>··T
0020  c3 84 aa 6b 50 18 20 d8  12 69 00 00 17 03 03 01   ···kP·· ·i······
0030  1f a2 53 4b 6d 33 ae 31  12 c4 02 ca 4c c4 8e 52   ··SKm3·1 ····L··R
0040  d7 ac 8b a5 41 22 7b d4  e2 0e 0b ee ae 37 b0 3e   ····A"{· ·····7·>
0050  7c 44 78 19 02 19 04 67  da 24 e3 60 d6 23 ca af   |Dx····g ·$·`·#··
0060  56 2b 03 28 d2 95 55 75  91 81 20 2e 5d f9 6e cd   V+·(··Uu ·· .]·n·
0070  bd 77 8e 86 a3 c4 0c a7  be 27 9f b0 6a 07 8f 9a   ·w······ ·'··j···
0080  75 a9 b8 1e 5b 00 9f 2e  67 0d 26 79 a5 f0 dc b4   u···[··. g·&y····
0090  09 fe b0 11 86 41 22 63  0e 85 d0 8f f1 60 21 78   ·····A"c ·····`!x
00a0  d0 4f 19 5e f8 0c b4 88  26 63 9f 46 6e 2e 46 d7   ·O·^···· &c·Fn.F·
00b0  e3 68 16 de bd e4 d0 7f  02 33 5b 81 b7 ee b7 fd   ·h······ ·3[·····
00c0  0f 37 9e ad 18 45 97 7b  c1 15 96 7d 2a 1a 14 b2   ·7···E·{ ···}*···
00d0  f8 0f 53 7f 03 f1 73 54  92 10 2e 6d 9a c7 62 4f   ··S···sT ··.m··bO
00e0  aa 58 a8 33 2e c0 91 bd  17 fa ff 27 cc 10 c4 b8   ·X·3.··· ···'····
00f0  eb 07 84 d4 6a a7 f7 a3  12 62 6c 0c 5c d4 20 d2   ····j··· ·bl·\· ·
0100  f1 99 c7 fa 37 31 35 a5  cc 0d fe 2a fc 26 34 03   ····715· ···*·&4·
0110  40 c7 24 12 f0 80 07 f4  2c d0 75 3b 25 58 f3 4e   @·$····· ,·u;%X·N
0120  ec 6b f0 ef 8b 95 fd 22  cd ad d3 cc 01 bc 11 d6   ·k·····" ········
0130  91 01 f1 8c 59 f6 43 39  ca 52 70 ce eb 0e 16 8a   ····Y·C9 ·Rp·····
0140  20 33 9b 4b 50 c8 b8 35  6f f5 6b e9 cb b9 89 ba    3·KP··5 o·k·····
```

```
Payload is encrypted application data (tls.app_data), 287 byte    Pacchetti: 3164· visualizzati: 100 (3.2%) · marcati: 4 (0.1%)    Profilo: Default
```

Figure 8: Encrypted broadcast messages forwarded by the server only to current group members; key-update events occur on membership changes.

5.3 What Wireshark reveals and what it does not

From packet inspection we can observe:

- Protocol-level metadata: TLS record boundaries, message types (e.g., ClientHello), packet lengths, source/destination IPs and ports, timing and packet order.
- The presence of application activity and roughly when key updates occur, since those events change record sizes and timing patterns.

What Wireshark (and a passive eavesdropper) cannot reveal:

- Application plaintext: all chat payloads remain encrypted and integrity-protected by AES-GCM and by TLS.
- Long-term private keys: Ed25519 private keys and X25519 ephemeral private keys are never transmitted on the wire.
- The exact session keys or HKDF outputs used to derive message keys.

5.4 Limitations

The tests above demonstrate the confidentiality and integrity properties of the implementation, but there are practical limitations to our current prototype and to this evaluation:

- Limited scalability: the prototype server and broadcast mechanism have not been stress-tested for large numbers of concurrent users or high message rates.
- No persistent message history or offline delivery: messages are only delivered to online recipients; persistent, encrypted storage is not implemented in the current prototype.
- Server-observable metadata: the server still learns routing metadata (which client sent a message and which client(s) received it) and can perform timing/traffic-analysis on communications.
- Use of self-signed TLS certificates: the server automatically generates a self-signed certificate at startup if none is present. While this guarantees that TLS is always available in a local development setting, it does not provide the authentication properties of a CA-issued certificate and will trigger warnings in standard clients.

5.5 Possible Improvements

We identify several practical enhancements and further tests that would strengthen the system:

- GUI improvements (bubble-style messages, user info, session status) for usability and clearer session indicators.
- Persistent, client-side encrypted chat history with forward secrecy for stored messages.
- Stronger forward secrecy across multiple sessions and a defined ratchet mechanism for long-lived conversations.
- Scalability and performance testing (load tests).
- Minimising server metadata leakage.
- Deployment with CA-signed TLS certificates to ensure strong server authentication and eliminate client-side trust warnings.

5.6 Conclusion

The application implements the core principles of secure messaging: identity verification (Ed25519), ephemeral session key negotiation (X25519), authenticated encryption (AES-GCM), and transport security (TLS). Our packet captures and Wireshark analysis confirm that message contents are not recoverable from passive network traces and that group key updates prevent departed members from decrypting subsequent group messages.

While this prototype is suitable for demonstration and teaching purposes, further work is required to harden it for production use: add persistent encrypted storage, perform formal verification and penetration testing, and reduce server-observable metadata. Nevertheless, the tests performed provide practical evidence that the implemented cryptographic primitives and protocols protect confidentiality and integrity as intended.