CS 6349.001 – Network Security

Fall 2025 Programming Project

Project Report

On

"SECURE RELAY-BASED CHAT SYSTEM"

Project Report By

Chinmayi C. Ramakrishna (ccr240000)

Mahima Bhushan (mxb240023)

Date: 9th December 2025

Master of Science in Computer Science

Erik Jonsson School of Engineering & Computer Science

The University of Texas at Dallas

Course Instructor: ***Kamil Sarac***

Project Instructor: ***Sai Tharun Reddy Mulka***

ABSTRACT

The Secure Relay-Based Chat System developed in this project enables two clients to communicate securely and authentically through an untrusted relay server. The relay is limited to forwarding messages, it cannot decrypt, modify, or impersonate either client. The system achieves confidentiality using a hash-based XOR stream cipher, integrity through HMAC-SHA256, forward secrecy via an ephemeral Diffie–Hellman key exchange, and authentication using RSA digital signatures.

During registration, each client provides its public key and a self-signature to the relay, ensuring identity binding. Mutual authentication is performed through a nonce-exchange protocol, preventing impersonation attacks. Once authenticated, clients establish a shared session key using Diffie–Hellman, this key is never revealed to the relay. All subsequent communication is encrypted using the derived session key, with sequence numbers and timestamps incorporated to defend against replay attacks.

The system fulfills essential security goals such as confidentiality, integrity, mutual authentication, replay resistance, and forward secrecy. Implemented without SSL or advanced cryptographic libraries, the design relies solely on Python socket programming, RSA signatures, SHA-256 hashing, XOR-based encryption, and HMAC. The resulting protocol provides a lightweight yet robust framework for secure end-to-end communication over an untrusted intermediary.

Table of Contents

Table of Figures

1.System Overview

The Secure Relay-Based Chat System is composed of three primary entities: Client A, Client B, and a central Relay Server. Clients do not communicate directly, instead, all messages pass through the Relay. The Relay is considered *untrusted* with respect to message confidentiality, but it is *trusted* for correct message forwarding.

1.1 Entities

1. **Client A (Alice)** – client 1 wants to securely chat.
2. **Client B (Bob)** – client 2 is the peer client/receiver.
3. **Relay Server (R)** – routes messages but is *not trusted* with plaintext.
4. **Adversary (Trudy)** – can eavesdrop, inject, replay, or impersonate.

1.2 System Architecture Diagram

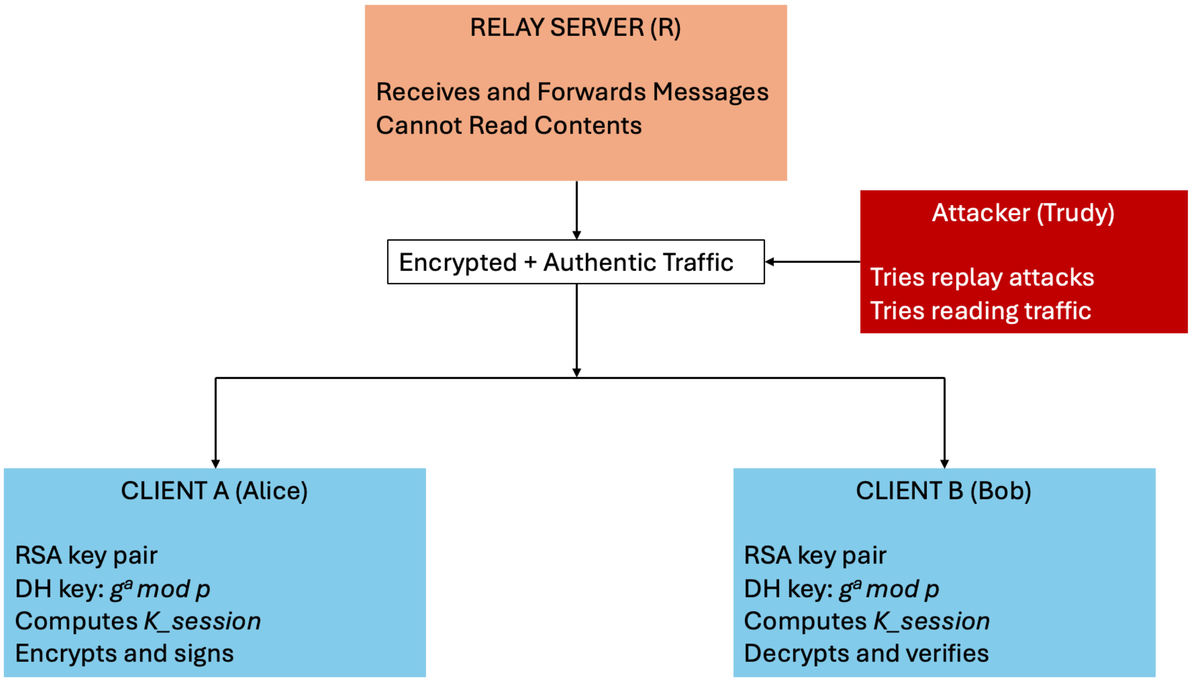


Figure 1: Client-Relay-Client Secure Chat System

1.3 Client A (Alice) & Client B (Bob)

Each client represents an end user participating in secure communication.

Each client:

- Possesses a long-term RSA key pair
- Knows the Relay's public key and other clients' public keys
- Registers with the Relay using RSA signatures
- Authenticates the Relay using a nonce-challenge protocol
- Initiates or responds to Diffie–Hellman (DH) key exchange
- Computes a shared session key unknown to the Relay
- Sends encrypted + authenticated messages
- Verifies integrity, freshness, and authenticity of incoming messages

Clients operate symmetrically; either client may initiate a session.

1.4 Communication Model

All communication follows the pattern:

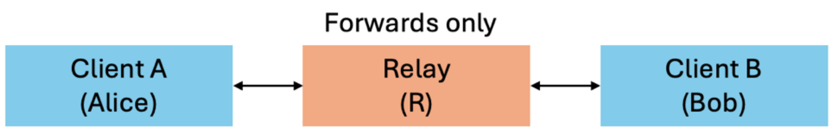


Figure 2: Communication Model

As you can see in Figure 2, there is no direct channel between clients.

All messages that travel through the Relay are:

- **Encrypted** using a keystream derived from SHA-256
- **Authenticated** using HMAC-SHA256
- **Protected against replay** using sequence numbers and timestamps

1.5 Threat Model

The adversary (Trudy) is powerful and may:

- Eavesdrop on all channels
- Replay captured messages
- Inject forged packets
- Attempt impersonation of Relay or clients
- Compromise the Relay (metadata exposure but not plaintext)

Despite these capabilities, the system ensures:

- **Confidentiality** via hashed XOR encryption
- **Integrity & authenticity** via HMAC
- **Forward secrecy** via ephemeral Diffie–Hellman
- **Replay protection** using sequence numbers + timestamps
- **Mutual authentication** using RSA signatures

2.Protocol Design

The Secure Relay-Based Chat System is divided into four major protocol phases:

1. Registration
2. Mutual Authentication
3. Session Setup & Diffie–Hellman Key Exchange
4. Secure Messaging
5. Session Termination

Each phase ensures a different part of the security model: identity verification, confidentiality, integrity, replay protection, and forward secrecy.

3.1 Phase 1 - Registration

The Relay learns each client's identity and public key. The client proves ownership of its RSA key pair.

Message Flow

Client A → Relay:

{ client_id, pub_key_A, Sig_A(priv_key_A, client_id || pub_key_A) }

Relay → Client A:

{status = success | failure }

Table 1: Registration Phase

Field	**Purpose**
client_id	Unique identifier for the client
pub_key_A	Long-term RSA public key of Alice
Priv_key_A	Long -term RSA private key of Alice
Sig_A()	RSA-PSS signature proving A owns private key

Security Guarantees

- Prevents impersonation (adversary cannot fake the signature).
- Relay stores verified public keys.
- Establishes trust anchors used later during authentication.

3.2 Phase 2 - Mutual Authentication

Clients confirm they are talking to the real Relay. Relay confirms the client is legitimate. This protects against man-in-the-middle and relay spoofing attacks.

Message Flow

1. Client challenges Relay

Client A → Relay:

{ type = auth_challenge, nonce_C }

2. Relay proves its identity

Relay → Client A:

{ server_nonce, Sig_R(nonce_C || server_nonce), relay_public_key }

Client A verifies:

Verify_R(Sig_R(nonce_C || server_nonce)) == True

If this fails → Relay is fake → Abort.

3. Client proves identity

Client A → Relay:

{ Sig_A(server_nonce) }

Relay verifies using A’s public key (from registration).

4. Relay confirms authentication

Relay → Client A:

{ status = success }

Security Guarantees

Table 2: Attack Protections

Attack Prevented	**How**
Relay impersonation	Relay proves identity using RSA signature
Client impersonation	Client proves identity using RSA signature
Replay of old challenges	Nonces ensure freshness

3.3 Phase 3 - Session Setup & Diffie–Hellman Key Exchange

Clients A and B derive a shared session key without the Relay learning it.

This provides:

- Forward secrecy
- Confidentiality even if the relay is malicious

Parameter Setup (Public)

- Prime modulus p
- Generator g

DH values

A generates private a, public g^a mod p

B generates private b, public g^b mod p

Message Flow (Relay just forwards)

1. A sends g^a to B

Client A → Relay → Client B:

{ type = dh_init, g_a }

B receives g^a and computes:

K_session = (g^a)^b mod p

Then B sends back g^b.

2. B sends g^b to A

Client B → Relay → Client A:

{ type = dh_response, g_b }

A computes:

K_session = (g^b)^a mod p

Both A and B share the same session key and relay knows nothing about it.

3.4 Phase 4 - Secure Messaging

Provides:

- **Confidentiality** (XOR encryption with SHA-derived keystream)
- **Integrity** (HMAC)
- **Replay protection** (timestamps + sequence numbers)
- **Sender verification**

Message Format

```
{
  type: "send", sender_id,  receiver_id, session_id, ciphertext, seq_num, timestamp,
  tag   = HMAC( ciphertext || seq_num || timestamp || sender_id || receiver_id )
}
```

Encryption

Keystream is generated using:

keystream = SHA-1 / SHA-256 (session_key || seq_num || timestamp || counter)

plaintext XOR keystream → ciphertext

SHA-1 is used for keystreams for better performance, output size and compatibitlity.

Integrity + Authentication

HMAC-SHA256 is used to ensure:

- Message was not modified

- Message is from the authenticated sender
- Recipient identity is bound inside the tag

Replay Protection

If:

seq_num <= last_seq_num_from_sender

Message is rejected as replayed.

3.5 Phase 5 - Session Termination

Either client may end the session.

Client A → Relay → Client B: { type = end_session, session_id }

Relay forwards the termination notice.

Both sides delete:

- Session key
- DH secrets
- Sequence numbers

This prevents later misuse.

4. Message Construction

All communication between Clients and the Relay uses newline-delimited JSON messages.
Every message belongs to one of four categories:

1. **Registration Messages**
2. **Authentication Messages**
3. **Session Setup (Diffie–Hellman) Messages**
4. **Secure Messaging**
5. **Session Termination**

Below, each message type is documented with:

- **Message fields**
- **Purpose of each field**
- **Security properties enforced**
- **How the receiver validates it**

4.1 Registration Messages

Message Sent from Client → Relay

```
{
  "type": "register",
  "client_id": "Alice",
  "pub_key": "<Alice RSA public key>",
  "signature": "<Sig_A(client_id || pub_key)>"
}
```

Field Explanation

Table 3: Relay Validation

Field	Purpose
client_id	Unique identity of client
pub_key	Long-term RSA public key
signature	Proves client owns corresponding private key

Relay Validation

- Reconstructs client_id || pub_key
- Verifies rsa_verify(pub_key, data, signature)
- Stores client’s public key

Security Achieved

- Prevents impersonation
- Registers trusted public keys
- Forms the basis for authentication

4.2 Mutual Authentication Messages

A. Client → Relay: Authentication Challenge

```
{
  "type": "auth_challenge",
  "client_id": "Alice",
  "nonce": "<random_nonce_C>"
}
```

Purpose: Client asks the Relay to prove identity.

B. Relay → Client: Challenge Response

```
{
  "type": "auth_challenge_response",
  "server_nonce": "<nonce_R>",
  "signature": "<Sig_R(nonce_C || nonce_R)>",
  "pub_key": "<Relay public key>"
}
```

Relay validates client, and client verifies Relay signature.

C. Client → Relay: Authentication Response

```
{
  "type": "auth_response",
  "client_id": "Alice",
  "signature": "<Sig_A(server_nonce)>"
}
```

Relay verifies client identity using stored public key.

D. Relay → Client: Authentication Confirmation

```
{
  "status": "success"
}
```

Security Achieved

- Mutual authentication
- Prevents relay impersonation
- Prevents client spoofing
- Nonces ensure freshness (no replay)

4.3 Diffie–Hellman Session Setup Messages

A. A → Relay → B: DH Init

```
{
  "type": "dh_init",
  "sender_id": "Alice",
  "peer_id": "Bob",
  "g_a": "<Alice public DH value>"
}
```

Relay forwards this to Bob without modification.

B. B → Relay → A: DH Response

```
{
  "type": "dh_response",
  "sender_id": "Bob",
  "peer_id": "Alice",
  "g_b": "<Bob public DH value>"
}
```

Relay forwards this to Alice.

Security Achieved

- Relay does NOT know a, b, or session key
- DH ensures forward secrecy
- Shared key is never transmitted

4.4 Secure Messaging Format

Once both sides compute K_session, all messages use this structure.

Message Sent from Client → Relay → Peer

```
{
  "type": "send",
  "sender_id": "Alice",
  "receiver_id": "Bob",
  "session_id": "Alice-Bob-1733331",
  "ciphertext": "<encrypted bytes hex>",
  "seq_num": N,
  "timestamp": T,
  "tag": "<HMAC(K_session, ciphertext || seq_num || timestamp || sender_id || receiver_id)>"
}
```

Field Explanation

Table 4: Integrity Check

Field	Purpose
ciphertext	XOR-encrypted plaintext
seq_num	Monotonic counter → prevents replay
timestamp	Ensures freshness, prevents delayed replay
tag	HMAC authentication + integrity
session_id	Identify the specific chat session

Receiver Validation Steps

1. Replay Protection

Check:

seq_num > last_seq_num_from_sender

If not → Reject message.

2. Integrity Check

Recompute HMAC:

HMAC(K_session, ciphertext || seq_num || timestamp || sender_id || receiver_id)

Compare with received tag.

If mismatch → Reject (tampered message).

3. Decryption

plaintext = XOR(ciphertext, keystream)

Where keystream = SHA-based KDF using:

- session_key

- seq_num
- timestamp

Security Achieved

- Confidentiality (Relay cannot decrypt)
- Authentication (tag binds sender identity)
- Integrity (Relay cannot modify)
- Replay protection
- Forward secrecy

4.5 Session Termination Message

```
{
  "type": "end_session",
  "session_id": "...",
  "sender_id": "Alice",
  "peer_id": "Bob"
}
```

Clients delete:

- DH private values
- Session key
- Sequence numbers

Security Achieved

- Prevents further use of key
- Denies any resumed replay attacks

5. Security Analysis

The Secure Relay-Based Chat System achieves the major security goals required in the project which is authentication, confidentiality, integrity, replay protection, and forward secrecy. This section analyzes each mechanism in the protocol and shows how it defends against adversaries such as Trudy, who may intercept, replay, alter, or inject messages.

5.1 Authentication

A. Authenticating the Relay

Clients verify the Relay Server using:

- A nonce nonce_C sent by the client
- A Relay response containing:

Sig_R(nonce_C || nonce_R)

This proves:

- Only the real Relay, with access to the Relay's private RSA key, could have generated the signature.
- Prevents relay impersonation attacks (fake server pretending to be relay).

B. Authenticating Clients

Each client signs the Relay's nonce:

Sig_Client(nonce_R)

Relay verifies it using the client's registered public key.

This ensures:

- Only legitimate clients can authenticate
- Prevents spoofing and unauthorized access

Security Guarantees:

- Mutual authentication
- Prevents impersonation of clients and relay
- Protects against MITM attempts

5.2 Confidentiality

The Relay never sees plaintext. Once session keys are established through Diffie–Hellman, clients encrypt all messages using:

ciphertext = plaintext XOR keystream

keystream = SHA-based derived stream from session_key

Relay only forwards:

- ciphertext
- metadata
- authentication tags

Relay cannot reverse SHA-derived keystream, so it cannot decrypt messages.

Security Guarantees:

- Relay cannot read messages
- Eavesdroppers learn nothing
- Confidentiality even if the Relay is malicious

5.3 Integrity and Authenticity of Messages

Each encrypted message includes an HMAC:

```
tag = HMAC(K_session,
    ciphertext || seq_num || timestamp || sender_id || receiver_id)
```

Receiver recomputes HMAC and compares it.

If HMAC mismatches:

- Message is discarded
- Attack is detected immediately

This ensures:

- Messages cannot be altered by Relay or attacker
- Attacker cannot inject fake messages (no access to session key)

Security Guarantees:

- Message integrity
- Sender authentication
- Tamper detection

5.4 Replay Protection

The system uses two replay defense layers:

A. Sequence Numbers

Each sender maintains a monotonically increasing seq_num.

If a message arrives with:

seq_num <= last_seq_num_from_sender

It is rejected as a replay attempt.

B. Timestamps

Used in keystream derivation and HMAC input.

If attacker replays an old ciphertext:

- Timestamp mismatch results in invalid HMAC
- Keystream will be different → decryption fails

Security Guarantees:

- Prevents delayed message replay
- Prevents duplicate messages
- Detects reordered messages

5.5 Forward Secrecy

Your design uses ephemeral Diffie–Hellman, meaning:

- Each session uses fresh DH private exponents (a and b)
- Long-term RSA keys are not used to derive session keys
- Compromise of RSA keys does not reveal old session keys

Even if Trudy later obtains:

- Relay private key
- Alice/Bob private RSA key

She still cannot reconstruct past session keys, because:

DH shared key = (g^b)^a mod p

Requires knowledge of a or b — never transmitted.

Security Guarantees:

- Past conversations remain safe
- Long-term key compromise does not affect past chats

5.6 Resistance to Major Attacks

Table 5: Types of attacks protected

Attack Type	**How System Defends**
MITM attack on Relay	Authentication prevents relay spoofing
Message modification	HMAC detects any tampering
Replay attack	Sequence numbers + timestamps
Fake message injection	No access to session key → HMAC fails
Eavesdropping	XOR encryption with SHA-derived keystream
Relay compromise	Relay still cannot decrypt messages
Key compromise	DH provides forward secrecy
Identity spoofing	RSA signatures verify sender authenticity

5.7 Threat Model Summary

Trudy may:

- Intercept all traffic
- Modify packets
- Replay old messages
- Impersonate clients
- Hijack relay communication

The protocol ensures she **cannot**:

- Decrypt messages
- Generate valid session keys
- Forge signatures
- Replay packets
- Modify ciphertext undetected
- Insert fake messages

6. Threat Model

The system assumes a powerful adversary, commonly referred to as Trudy, who can interfere with communication between Clients and the Relay. The following capabilities and threats are considered:

6.1 Adversary Capabilities

1. Eavesdropping

Trudy can read any message transmitted over the network, including:

- Client - Relay messages
- Relay - Client messages

Mitigation: All message contents are encrypted with a session key derived from Diffie–Hellman and never visible to the Relay or attacker.

2. Message Modification

Trudy may alter ciphertext or message headers while they are in transit.

Mitigation: HMAC-SHA256 ensures any modification is detected immediately by the receiver.

3. Replay Attacks

Trudy can capture old, encrypted packets and re-send them to confuse clients.

Mitigation: Sequence numbers and timestamps prevent acceptance of outdated messages.

4. Message Injection

Trudy may attempt to create fake messages and send them as if they came from a client.

Mitigation: Attack fails because Trudy does not know the session key and cannot generate a valid HMAC.

5. Impersonation

Trudy may try to pretend to be:

- A client
- The Relay

Mitigation: RSA-based mutual authentication ensures identity validation in both directions.

6. Relay Compromise

If the Relay is malicious or compromised:

- It can see metadata (sender, receiver, timestamps)
- It cannot decrypt messages
- It cannot forge valid messages
- It cannot modify messages without detection

Forward secrecy ensures that compromising the Relay at any point does **not** reveal old communication.

6.2 Security Guarantees Under Threat Model

Even with all the above adversary powers, the system guarantees:

- **Confidentiality:** Relay and attackers cannot learn plaintext.
- **Integrity:** Any tampering is detected via HMAC.
- **Authentication:** Only legitimate clients and the real relay can participate.
- **Replay Protection:** Sequence numbers and timestamps prevent reuse of old packets.
- **Forward Secrecy:** Past messages remain secure even if long-term keys are leaked.

7. Implementation Plan

This section summarizes how the protocol was implemented and maps each protocol phase to the actual components in the codebase. The goal is to show that every phase of the design is supported by working code.

7.1 Technologies Used

- **Programming Language:** Python 3
- **Networking:** TCP sockets (socket module)
- **Concurrency:** threading for handling multiple clients
- **Cryptography:**
  - RSA key generation, signing, verification
  - SHA-256, HMAC-SHA256
  - Custom XOR-based encryption using SHA-derived keystream
  - Custom Diffie–Hellman implementation
- **Message Format:** JSON over TCP with newline-delimited framing

7.2 Code Structure

Table 6: Code structure

File	**Responsibility**
relay.py	Untrusted relay server that forwards messages, performs registration and authentication.
client.py	Handles registration, authentication, DH key exchange, encryption, HMAC, messaging.
crypto_utils.py	All cryptographic primitives (RSA, DH, hashing, keystream derivation, XOR encryption).

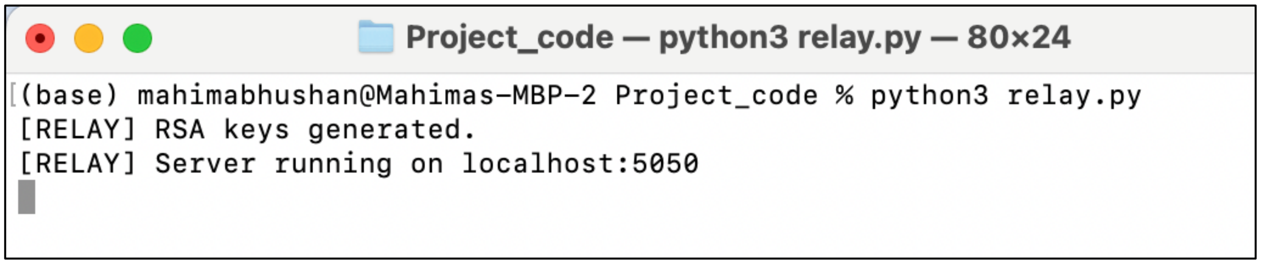
8. Execution & Example Message Flow

This section demonstrates the full execution of our secure relay-based chat system using two clients: Alice and Bob. Screenshots are included to show the behavior of the Relay and clients during registration, authentication, key exchange, and secure messaging.

8.1 Starting the Relay Server

Command:

>>python3 relay.py



```
Project_code — python3 relay.py — 80×24
(base) mahimabhushan@Mahimas-MBP-2 Project_code % python3 relay.py
[RELAY] RSA keys generated.
[RELAY] Server running on localhost:5050
```

Figure 3: Keys generation

This indicates the server is operational and waiting for client connections.

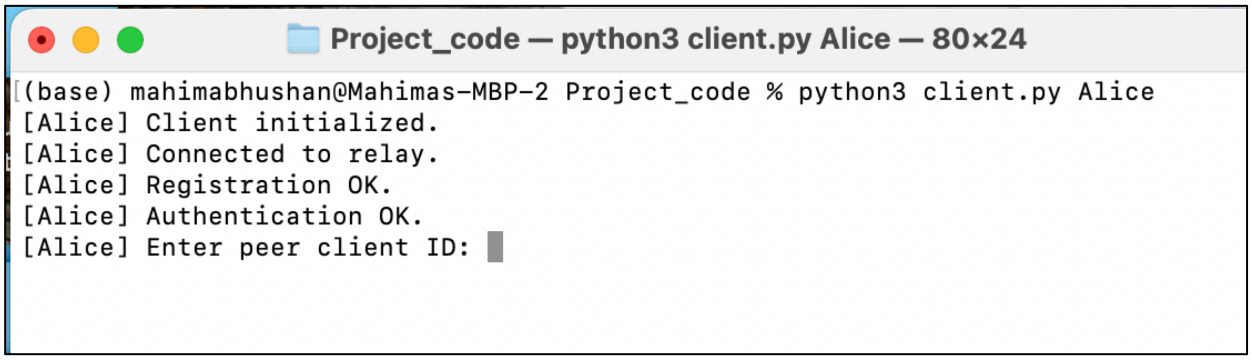
8.2 Starting Client Alice

>>python3 client.py Alice

Alice performs:

1. Registration with Relay
2. RSA-based authentication
3. Waits for user to enter peer ID

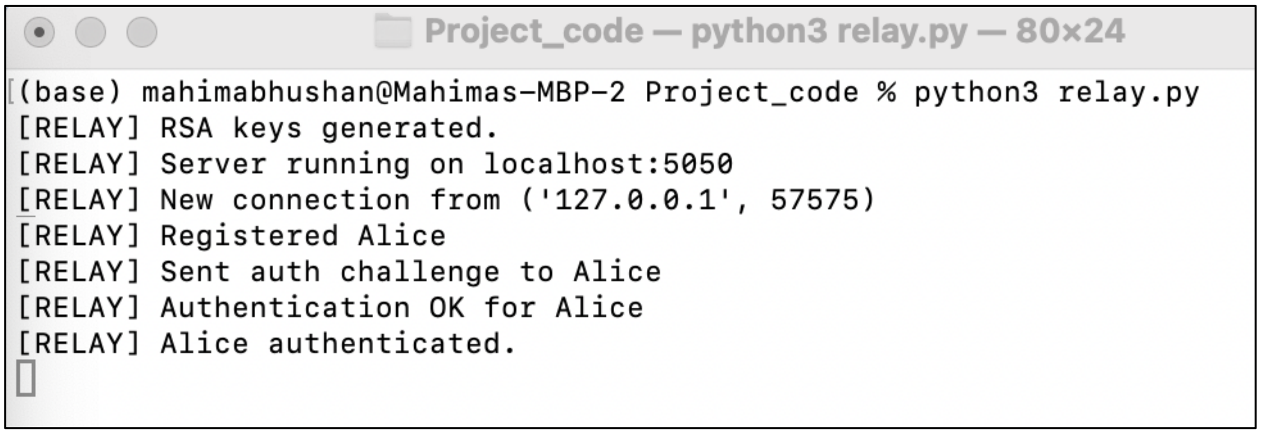
Alice Terminal:


Project_code — python3 client.py Alice — 80×24

```
(base) mahimabhushan@Mahimas-MBP-2 Project_code % python3 client.py Alice
[Alice] Client initialized.
[Alice] Connected to relay.
[Alice] Registration OK.
[Alice] Authentication OK.
[Alice] Enter peer client ID:
```

Figure 4: Registration and Authentication

Relay Terminal:

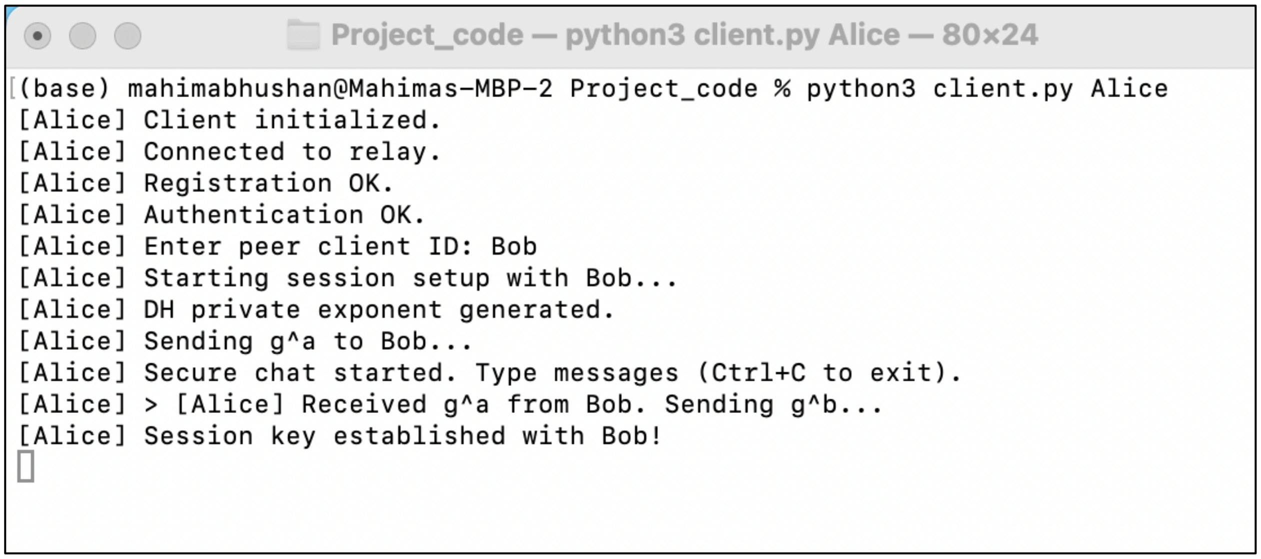

Project_code — python3 relay.py — 80×24

```
(base) mahimabhushan@Mahimas-MBP-2 Project_code % python3 relay.py
[RELAY] RSA keys generated.
[RELAY] Server running on localhost:5050
[RELAY] New connection from ('127.0.0.1', 57575)
[RELAY] Registered Alice
[RELAY] Sent auth challenge to Alice
[RELAY] Authentication OK for Alice
[RELAY] Alice authenticated.
```

Figure 5: Authentication Complete

Alice now enters:

>> Enter peer client ID: Bob



```
[(base) mahimabhushan@Mahimas-MBP-2 Project_code % python3 client.py Alice
[Alice] Client initialized.
[Alice] Connected to relay.
[Alice] Registration OK.
[Alice] Authentication OK.
[Alice] Enter peer client ID: Bob
[Alice] Starting session setup with Bob...
[Alice] DH private exponent generated.
[Alice] Sending g^a to Bob...
[Alice] Secure chat started. Type messages (Ctrl+C to exit).
[Alice] > [Alice] Received g^a from Bob. Sending g^b...
[Alice] Session key established with Bob!
```

Figure 6: Session Key Established

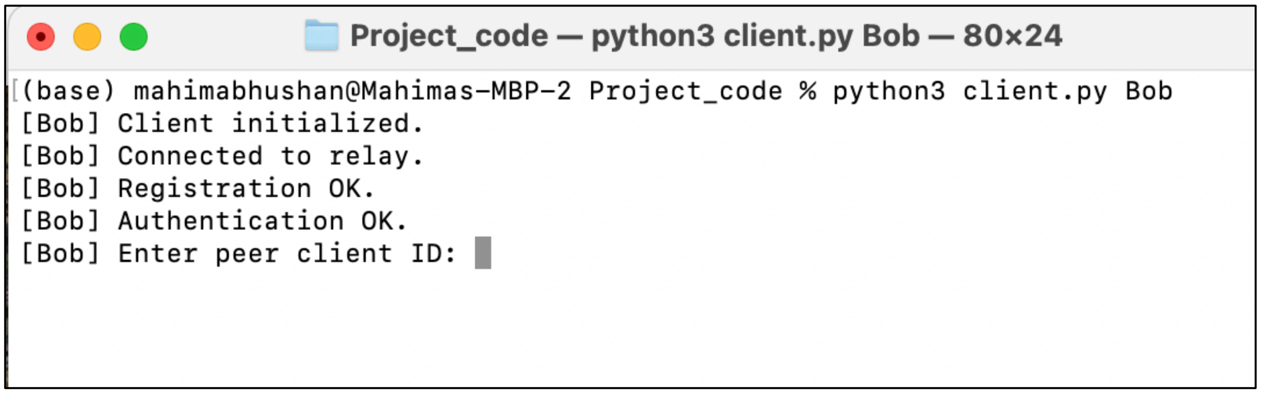
8.3 Starting Client Bob

>>python3 client.py Bob

Bob also performs:

1. Registration
2. Authentication
3. Receives DH initiation from Alice

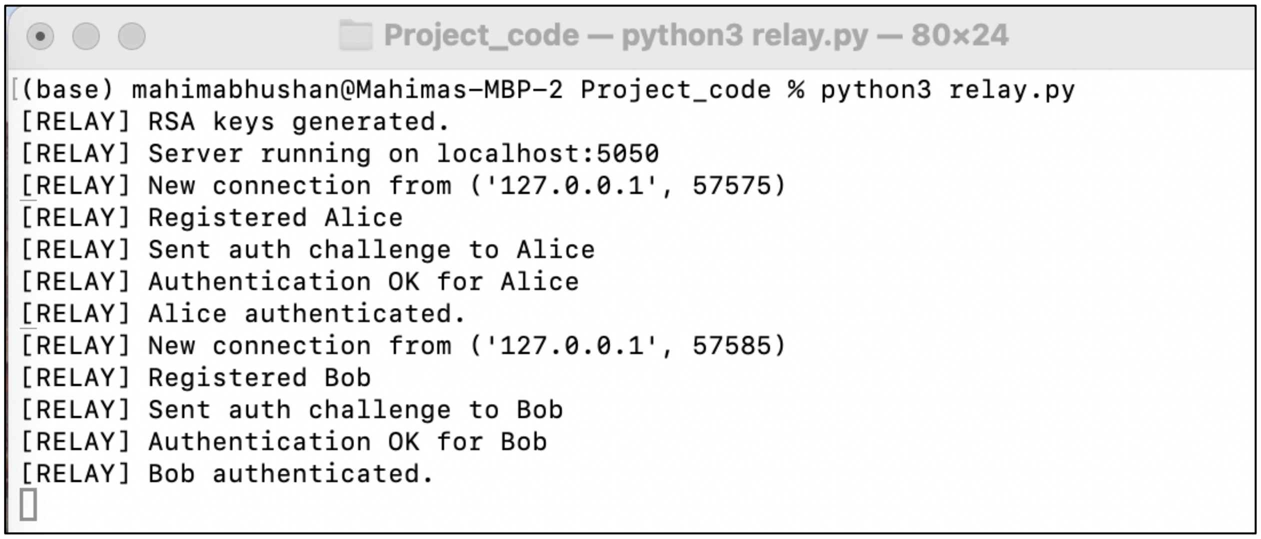
Bob Terminal:



```
[(base) mahimabhushan@Mahimas-MBP-2 Project_code % python3 client.py Bob
[Bob] Client initialized.
[Bob] Connected to relay.
[Bob] Registration OK.
[Bob] Authentication OK.
[Bob] Enter peer client ID: 
```

Figure 7: Bob connecting to Alice

Relay Terminal:



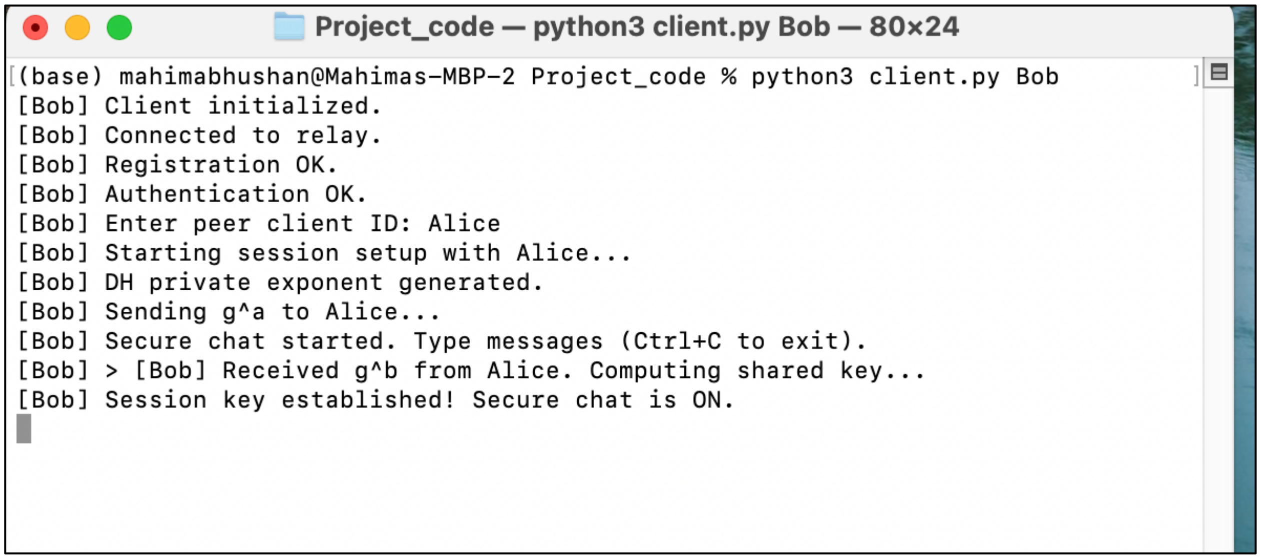
```
(base) mahimabhushan@Mahimas-MBP-2 Project_code % python3 relay.py
[RELAY] RSA keys generated.
[RELAY] Server running on localhost:5050
[RELAY] New connection from ('127.0.0.1', 57575)
[RELAY] Registered Alice
[RELAY] Sent auth challenge to Alice
[RELAY] Authentication OK for Alice
[RELAY] Alice authenticated.
[RELAY] New connection from ('127.0.0.1', 57585)
[RELAY] Registered Bob
[RELAY] Sent auth challenge to Bob
[RELAY] Authentication OK for Bob
[RELAY] Bob authenticated.
```

Figure 8: Bob and Alice Authentication

Bob now types:

>>Enter peer client ID: Alice

This allows Bob to initiate or respond to session setup with Alice.

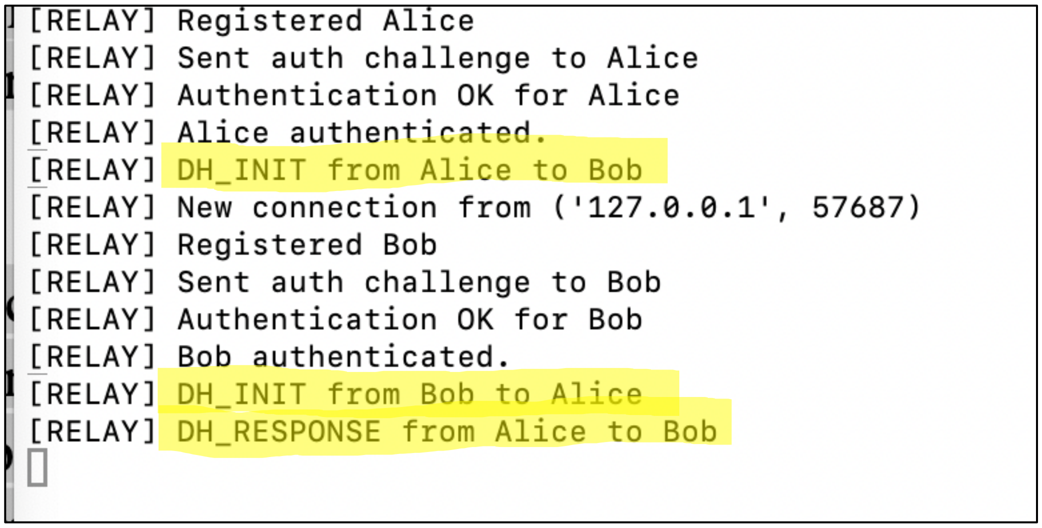


```
(base) mahimabhushan@Mahimas-MBP-2 Project_code % python3 client.py Bob
[Bob] Client initialized.
[Bob] Connected to relay.
[Bob] Registration OK.
[Bob] Authentication OK.
[Bob] Enter peer client ID: Alice
[Bob] Starting session setup with Alice...
[Bob] DH private exponent generated.
[Bob] Sending g^a to Alice...
[Bob] Secure chat started. Type messages (Ctrl+C to exit).
[Bob] > [Bob] Received g^b from Alice. Computing shared key...
[Bob] Session key established! Secure chat is ON.
```

Figure 9: Bob session key established

8.4 Relay DH Forwarding

Relay prints messages like:



```
[RELAY] Registered Alice
[RELAY] Sent auth challenge to Alice
[RELAY] Authentication OK for Alice
[RELAY] Alice authenticated.
[RELAY] DH_INIT from Alice to Bob
[RELAY] New connection from ('127.0.0.1', 57687)
[RELAY] Registered Bob
[RELAY] Sent auth challenge to Bob
[RELAY] Authentication OK for Bob
[RELAY] Bob authenticated.
[RELAY] DH_INIT from Bob to Alice
[RELAY] DH_RESPONSE from Alice to Bob
```

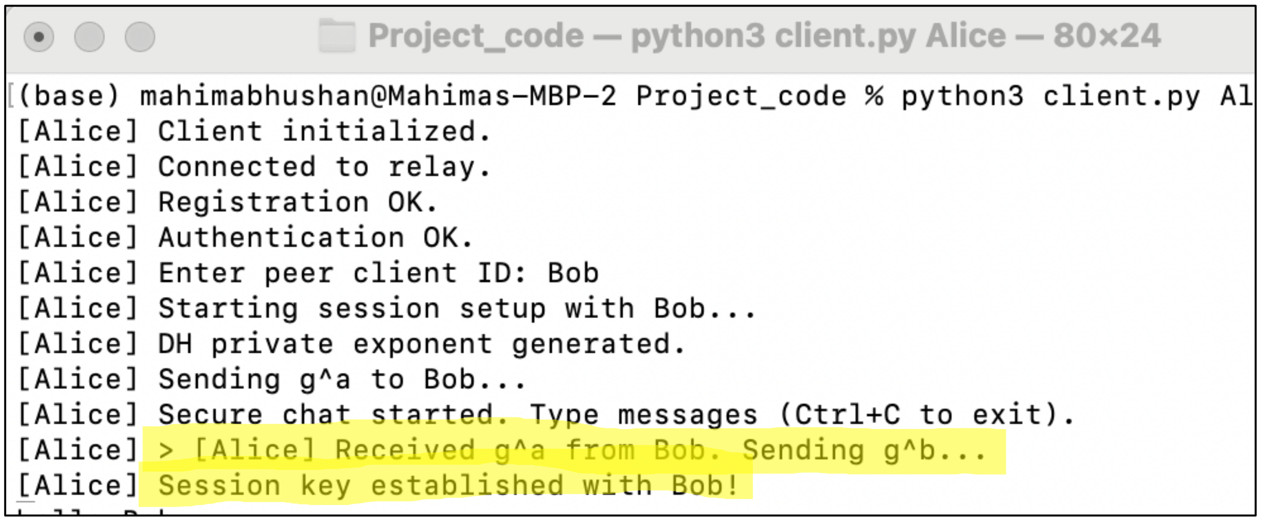
Figure 10: DH keys session establishment Relay

- Alice sent **g^a** to Bob
- Bob sent **g^a** to Alice (both started simultaneously which is normal)
- Alice sent **g^b** back

Relay forwards the DH messages but cannot decrypt anything**.**

8.5 Session Key Established

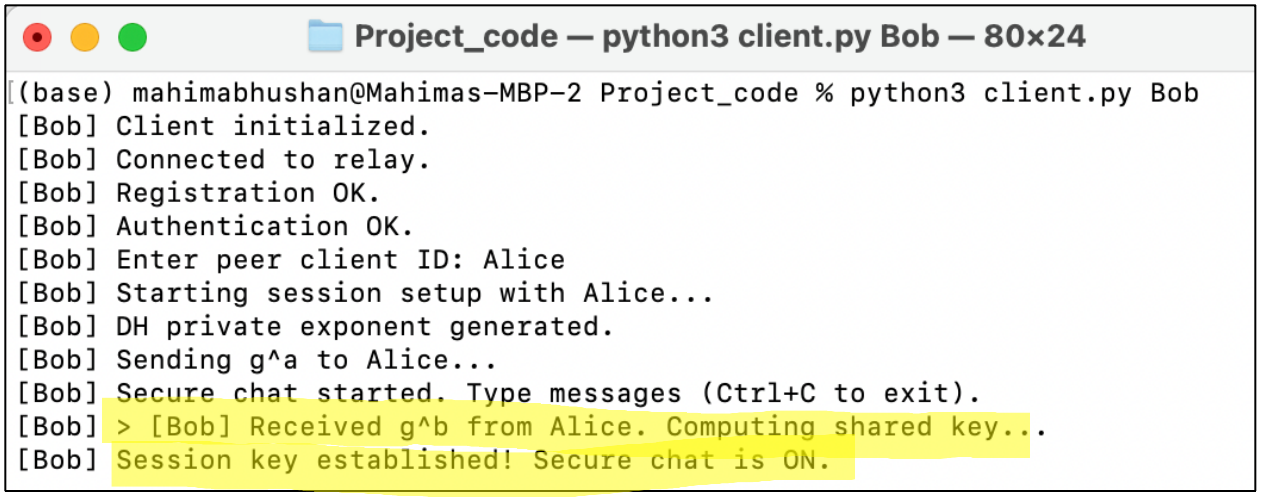
Alice Terminal:



```
(base) mahimabhushan@Mahimas-MBP-2 Project_code % python3 client.py Al
[Alice] Client initialized.
[Alice] Connected to relay.
[Alice] Registration OK.
[Alice] Authentication OK.
[Alice] Enter peer client ID: Bob
[Alice] Starting session setup with Bob...
[Alice] DH private exponent generated.
[Alice] Sending g^a to Bob...
[Alice] Secure chat started. Type messages (Ctrl+C to exit).
[Alice] > [Alice] Received g^a from Bob. Sending g^b...
[Alice] Session key established with Bob!
```

Figure 11: DH keys session establishment Alice

Bob Terminal:



```
(base) mahimabhushan@Mahimas-MBP-2 Project_code % python3 client.py Bob
[Bob] Client initialized.
[Bob] Connected to relay.
[Bob] Registration OK.
[Bob] Authentication OK.
[Bob] Enter peer client ID: Alice
[Bob] Starting session setup with Alice...
[Bob] DH private exponent generated.
[Bob] Sending g^a to Alice...
[Bob] Secure chat started. Type messages (Ctrl+C to exit).
[Bob] > [Bob] Received g^b from Alice. Computing shared key...
[Bob] Session key established! Secure chat is ON.
```

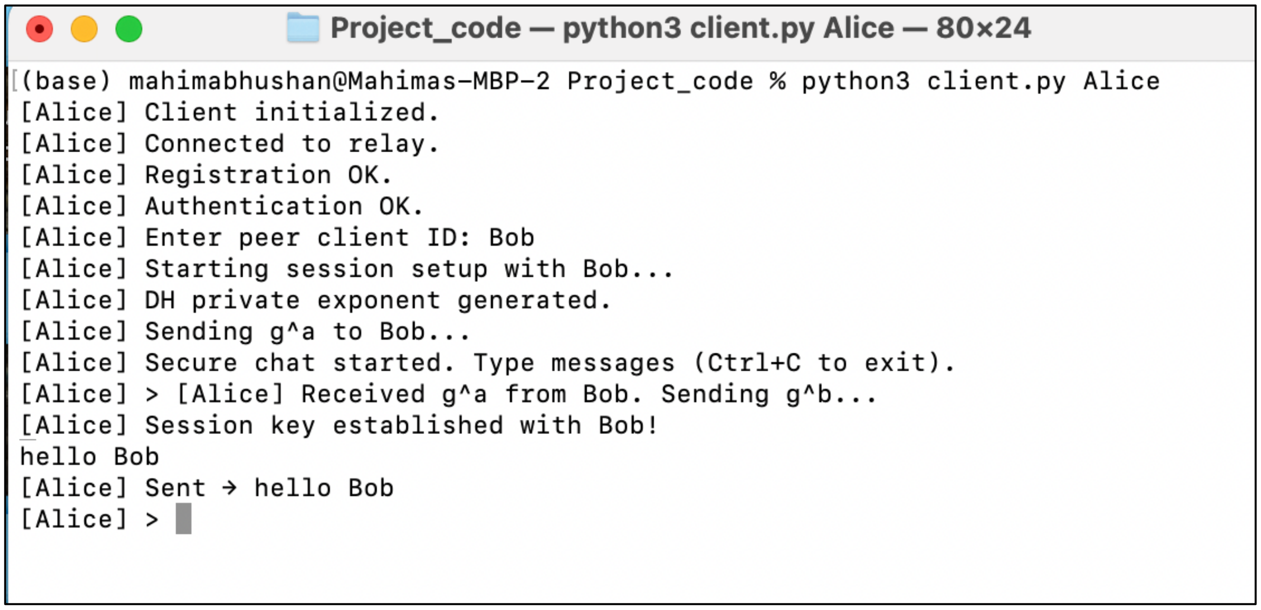
Figure 12: DH keys session establishment bob

Both clients now share the same secret session key.

8.6 Secure Messaging

Alice sends message:

[Alice] > hello Bob

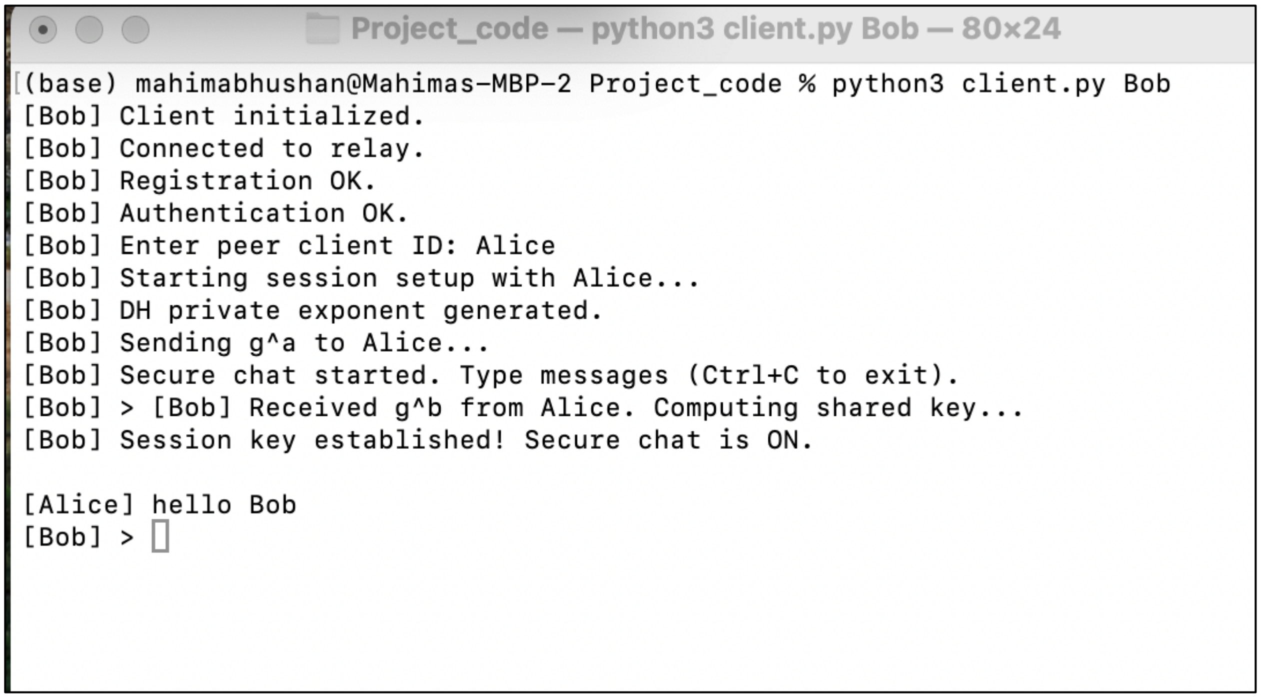


```
(base) mahimabhushan@Mahimas-MBP-2 Project_code % python3 client.py Alice
[Alice] Client initialized.
[Alice] Connected to relay.
[Alice] Registration OK.
[Alice] Authentication OK.
[Alice] Enter peer client ID: Bob
[Alice] Starting session setup with Bob...
[Alice] DH private exponent generated.
[Alice] Sending g^a to Bob...
[Alice] Secure chat started. Type messages (Ctrl+C to exit).
[Alice] > [Alice] Received g^a from Bob. Sending g^b...
[Alice] Session key established with Bob!
hello Bob
[Alice] Sent → hello Bob
[Alice] >
```

Figure 13: Conversation Alice to Bob

Bob sees:

[Alice] hello Bob

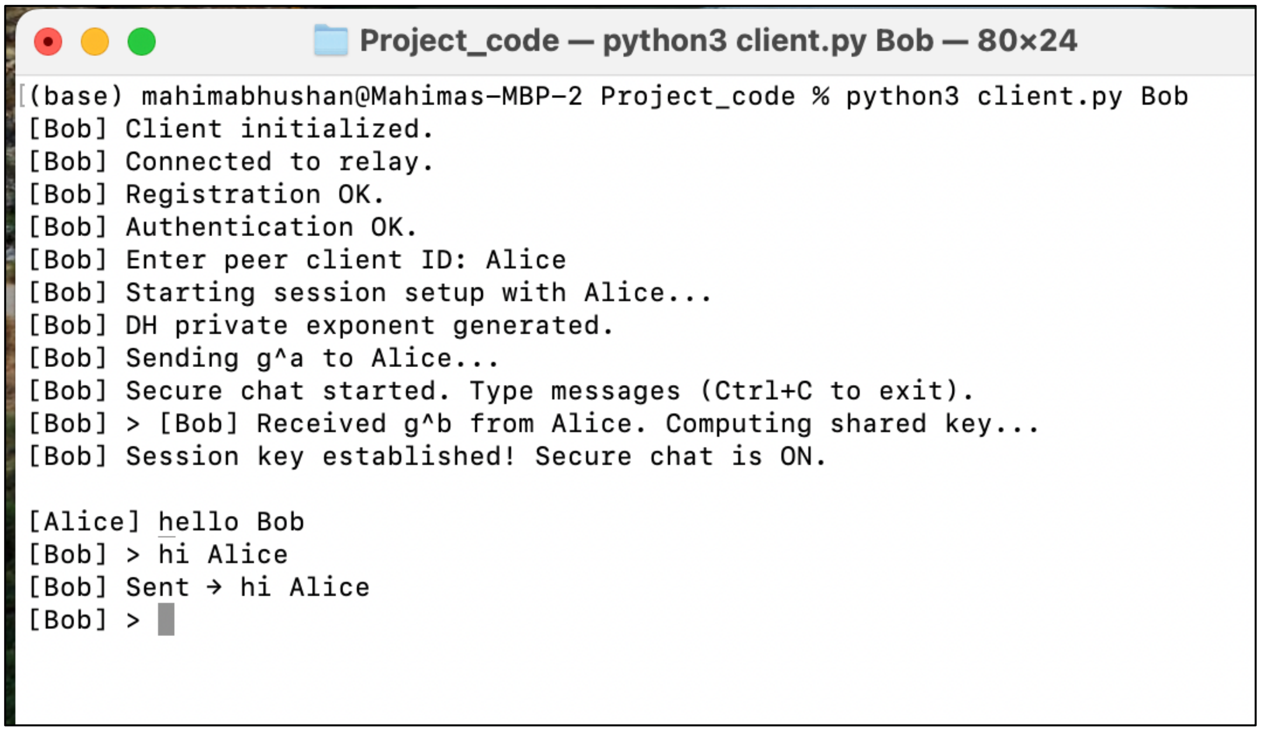


```
(base) mahimabhushan@Mahimas-MBP-2 Project_code % python3 client.py Bob
[Bob] Client initialized.
[Bob] Connected to relay.
[Bob] Registration OK.
[Bob] Authentication OK.
[Bob] Enter peer client ID: Alice
[Bob] Starting session setup with Alice...
[Bob] DH private exponent generated.
[Bob] Sending g^a to Alice...
[Bob] Secure chat started. Type messages (Ctrl+C to exit).
[Bob] > [Bob] Received g^b from Alice. Computing shared key...
[Bob] Session key established! Secure chat is ON.

[Alice] hello Bob
[Bob] >
```

Bob replies:

[Bob] > hi Alice

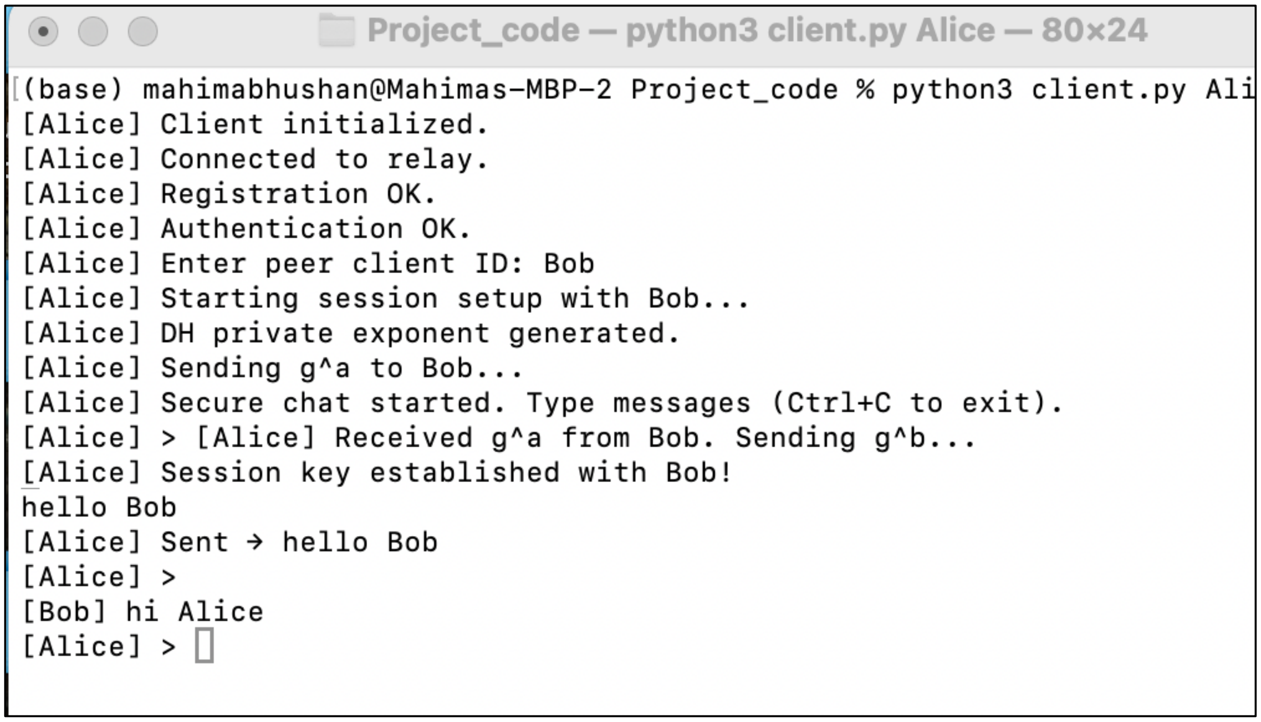


```
(base) mahimabhushan@Mahimas-MBP-2 Project_code % python3 client.py Bob
[Bob] Client initialized.
[Bob] Connected to relay.
[Bob] Registration OK.
[Bob] Authentication OK.
[Bob] Enter peer client ID: Alice
[Bob] Starting session setup with Alice...
[Bob] DH private exponent generated.
[Bob] Sending g^a to Alice...
[Bob] Secure chat started. Type messages (Ctrl+C to exit).
[Bob] > [Bob] Received g^b from Alice. Computing shared key...
[Bob] Session key established! Secure chat is ON.

[Alice] hello Bob
[Bob] > hi Alice
[Bob] Sent → hi Alice
[Bob] >
```

Alice sees:

[Bob] hi Alice



```
(base) mahimabhushan@Mahimas-MBP-2 Project_code % python3 client.py Ali
[Alice] Client initialized.
[Alice] Connected to relay.
[Alice] Registration OK.
[Alice] Authentication OK.
[Alice] Enter peer client ID: Bob
[Alice] Starting session setup with Bob...
[Alice] DH private exponent generated.
[Alice] Sending g^a to Bob...
[Alice] Secure chat started. Type messages (Ctrl+C to exit).
[Alice] > [Alice] Received g^a from Bob. Sending g^b...
[Alice] Session key established with Bob!
hello Bob
[Alice] Sent → hello Bob
[Alice] >
[Bob] hi Alice
[Alice] >
```

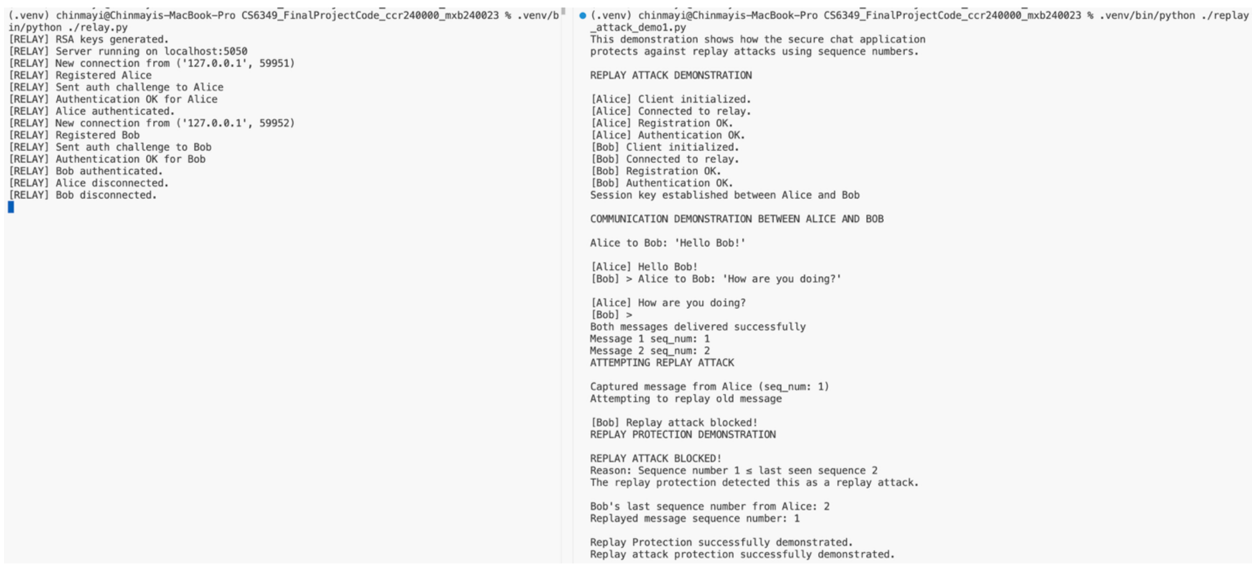
All messages are:

- XOR-encrypted using SHA-256–derived keystream
- Authenticated using HMAC
- Protected using sequence numbers + timestamps

Relay only forwards ciphertext.

8.7 Replay Attack Protection in Action

Attacker fakes Bob's identity and tries to replay a message.
Our Secure Chat System blocks such replayed messages.



```
(.venv) chinmayi@Chinmayis-MacBook-Pro CS6349_FinalProjectCode_ccr240000_mxb240023 % .venv/b
in/python ./relay.py
[RELAY] RSA keys generated.
[RELAY] Server running on localhost:5050
[RELAY] New connection from ('127.0.0.1', 59951)
[RELAY] Registered Alice
[RELAY] Sent auth challenge to Alice
[RELAY] Authentication OK for Alice
[RELAY] Alice authenticated.
[RELAY] New connection from ('127.0.0.1', 59952)
[RELAY] Registered Bob
[RELAY] Sent auth challenge to Bob
[RELAY] Authentication OK for Bob
[RELAY] Bob authenticated.
[RELAY] Alice disconnected.
[RELAY] Bob disconnected.
```

```
(.venv) chinmayi@Chinmayis-MacBook-Pro CS6349_FinalProjectCode_ccr240000_mxb240023 % .venv/bin/python ./replay
_attack_demo1.py
This demonstration shows how the secure chat application
protects against replay attacks using sequence numbers.

REPLAY ATTACK DEMONSTRATION

[Alice] Client initialized.
[Alice] Connected to relay.
[Alice] Registration OK.
[Alice] Authentication OK.
[Bob] Client initialized.
[Bob] Connected to relay.
[Bob] Registration OK.
[Bob] Authentication OK.
Session key established between Alice and Bob

COMMUNICATION DEMONSTRATION BETWEEN ALICE AND BOB

Alice to Bob: 'Hello Bob!'

[Alice] Hello Bob!
[Bob] > Alice to Bob: 'How are you doing?'

[Alice] How are you doing?
[Bob] >
Both messages delivered successfully
Message 1 seq_num: 1
Message 2 seq_num: 2
ATTEMPTING REPLAY ATTACK

Captured message from Alice (seq_num: 1)
Attempting to replay old message

[Bob] Replay attack blocked!
REPLAY PROTECTION DEMONSTRATION

REPLAY ATTACK BLOCKED!
Reason: Sequence number 1 ≤ last seen sequence 2
The replay protection detected this as a replay attack.

Bob's last sequence number from Alice: 2
Replayed message sequence number: 1

Replay Protection successfully demonstrated.
Replay attack protection successfully demonstrated.
```

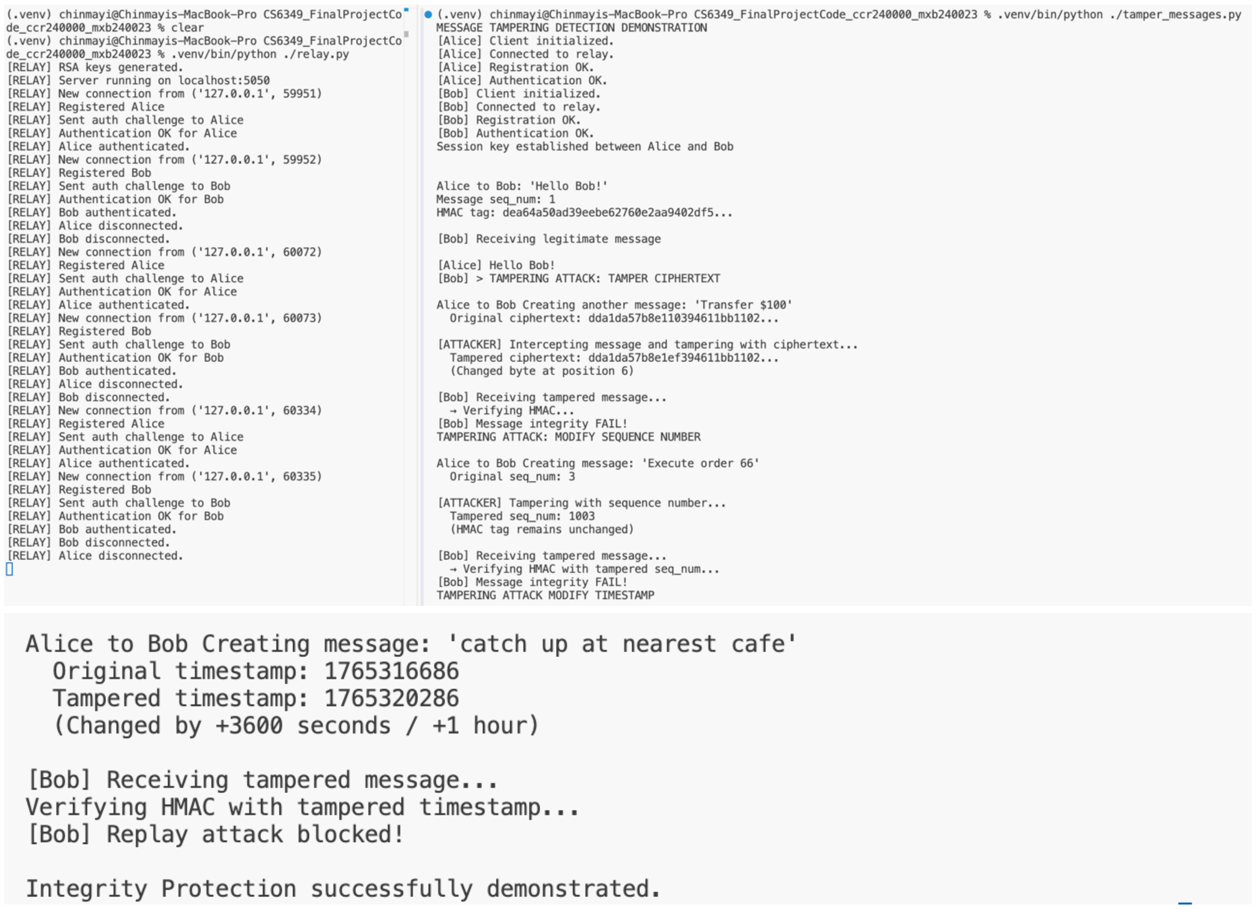
Figure 14: Replay Attack Protection in Action

8.8 Tampering Message

Message is tampered in three ways:

1. Tampering Ciphertext
2. Tampering Sequence Number
3. Tampering Timestamp

Our Secure Chat System catches all the three tampering.



```
(.venv) chinmayi@Chinmayis-MacBook-Pro CS6349_FinalProjectCo
de_ccr240000_mxb240023 % clear
(.venv) chinmayi@Chinmayis-MacBook-Pro CS6349_FinalProjectCo
de_ccr240000_mxb240023 % .venv/bin/python ./relay.py
[RELAY] RSA keys generated.
[RELAY] Server running on localhost:5050
[RELAY] New connection from ('127.0.0.1', 59951)
[RELAY] Registered Alice
[RELAY] Sent auth challenge to Alice
[RELAY] Authentication OK for Alice
[RELAY] Alice authenticated.
[RELAY] New connection from ('127.0.0.1', 59952)
[RELAY] Registered Bob
[RELAY] Sent auth challenge to Bob
[RELAY] Authentication OK for Bob
[RELAY] Bob authenticated.
[RELAY] Alice disconnected.
[RELAY] Bob disconnected.
[RELAY] New connection from ('127.0.0.1', 60072)
[RELAY] Registered Alice
[RELAY] Sent auth challenge to Alice
[RELAY] Authentication OK for Alice
[RELAY] Alice authenticated.
[RELAY] New connection from ('127.0.0.1', 60073)
[RELAY] Registered Bob
[RELAY] Sent auth challenge to Bob
[RELAY] Authentication OK for Bob
[RELAY] Bob authenticated.
[RELAY] Alice disconnected.
[RELAY] Bob disconnected.
[RELAY] New connection from ('127.0.0.1', 60334)
[RELAY] Registered Alice
[RELAY] Sent auth challenge to Alice
[RELAY] Authentication OK for Alice
[RELAY] Alice authenticated.
[RELAY] New connection from ('127.0.0.1', 60335)
[RELAY] Registered Bob
[RELAY] Sent auth challenge to Bob
[RELAY] Authentication OK for Bob
[RELAY] Bob authenticated.
[RELAY] Bob disconnected.
[RELAY] Alice disconnected.
```

```
(.venv) chinmayi@Chinmayis-MacBook-Pro CS6349_FinalProjectCode_ccr240000_mxb240023 % .venv/bin/python ./tamper_messages.py
MESSAGE TAMPERING DETECTION DEMONSTRATION
[Alice] Client initialized.
[Alice] Connected to relay.
[Alice] Registration OK.
[Alice] Authentication OK.
[Bob] Client initialized.
[Bob] Connected to relay.
[Bob] Registration OK.
[Bob] Authentication OK.
Session key established between Alice and Bob


Alice to Bob: 'Hello Bob!'
Message seq_num: 1
HMAC tag: dea64a50ad39eebe62760e2aa9402df5...

[Bob] Receiving legitimate message

[Alice] Hello Bob!
[Bob] > TAMPERING ATTACK: TAMPER CIPHERTEXT

Alice to Bob Creating another message: 'Transfer $100'
  Original ciphertext: dda1da57b8e110394611bb1102...

[ATTACKER] Intercepting message and tampering with ciphertext...
  Tampered ciphertext: dda1da57b8e1ef394611bb1102...
  (Changed byte at position 6)

[Bob] Receiving tampered message...
  → Verifying HMAC...
[Bob] Message integrity FAIL!
TAMPERING ATTACK: MODIFY SEQUENCE NUMBER

Alice to Bob Creating message: 'Execute order 66'
  Original seq_num: 3

[ATTACKER] Tampering with sequence number...
  Tampered seq_num: 1003
  (HMAC tag remains unchanged)

[Bob] Receiving tampered message...
  → Verifying HMAC with tampered seq_num...
[Bob] Message integrity FAIL!
TAMPERING ATTACK MODIFY TIMESTAMP
```

```
Alice to Bob Creating message: 'catch up at nearest cafe'
  Original timestamp: 1765316686
  Tampered timestamp: 1765320286
  (Changed by +3600 seconds / +1 hour)

[Bob] Receiving tampered message...
Verifying HMAC with tampered timestamp...
[Bob] Replay attack blocked!

Integrity Protection successfully demonstrated.
```

Figure 15: Integrity Protection

9. Conclusion

This project successfully demonstrates the design and implementation of a secure, end-to-end encrypted chat system operating through an untrusted relay. Using only basic cryptographic primitives such as RSA signatures, SHA-256 hashing, HMAC, XOR encryption, and Diffie–Hellman, the system achieves confidentiality, integrity, authentication, replay protection, and forward secrecy. We use simple socket programming in Python to accomplish all the five steps: Connection, Registration, Authentication of Clients, Secure Messaging and Session Termination.

The relay plays no role in key generation or message decryption, it merely forwards ciphertext, ensuring zero-knowledge of session keys. Through mutual authentication and ephemeral DH exchange, two clients establish a shared secret key that even a compromised relay cannot derive. Subsequent messages are encrypted and authenticated using HMAC, sequence numbers, and timestamps, providing strong protection against modification, replay, and impersonation attacks.

Overall, the system provides a lightweight yet robust model for secure communication, implemented entirely with Python sockets and custom cryptographic logic. It not only fulfills all project requirements but also illustrates practical challenges in building secure communication protocols from scratch.

10. Contributions

Mahima Bhushan (mxb240023):

- Implemented client-side protocol logic
  (registration, authentication, DH key exchange, secure messaging).
- Integrated cryptographic utilities and keystream-based encryption.
- Prepared execution testing, screenshots, and documentation sections.
- Responsible for debugging and verifying end-to-end message flow.
- Tampering Messages in Action

Chinmayi C. Ramakrishna (ccr240000):

- Implemented relay server logic
  (registration verification, forwarding, authentication handling).
- Developed the initial project architecture and message formats.
- Wrote crypto utility file structure and RSA/DH helper logic.
- Assisted with protocol design and documentation review.
- Replay Protection in Action

Both collaborators:

- Jointly designed the security protocol.
- Conducted testing.
- Prepared the final report and analysis.