TextNow: Secure Chat App – Final Project Report (Project 2.7)

**Course**: CS-GY 6903 I Applied Cryptography

**Team Members:** Gitika Babbar (gb2889)

**Instructor:** Prof. Giovanni Di Crescenzo

**Semester:** Spring 2025

# 📌 Project Overview

Project Type: 2.7 – Designing an End-to-End Cryptography Solution

I designed and built a fully secure, real-time messaging application inspired by real-world chat systems such as Slack, Discord, and WhatsApp Web. The app integrates end-to-end encryption, authentication, and replay protection using cryptographic protocols discussed in class.

# 🎯 Problem Statement

Modern real-time communication apps transmit large volumes of sensitive information. When these messages are sent over untrusted networks, they’re exposed to:  
- Eavesdropping  
- Message tampering  
- Spoofing  
- Replay attacks  
  
I addressed this by designing a secure-by-default communication pipeline between chat clients.

# 💡 Proposed Solution

Key Features Implemented:

|  |  |
| --- | --- |
| **Security Goal** | **Protocol / Technique** |
| Confidentiality | AES-GCM Symmetric Encryption |
| Integrity | ECDSA Digital Signatures |
| Authenticity | ECDSA Key Pair Verification |
| Replay Protection | Nonce + Timestamp Tracking |
| Key Exchange | ECDH – Elliptic Curve Diffie-Hellman |

# 🖥️ System Architecture & Flow

The secure message transmission process involves:

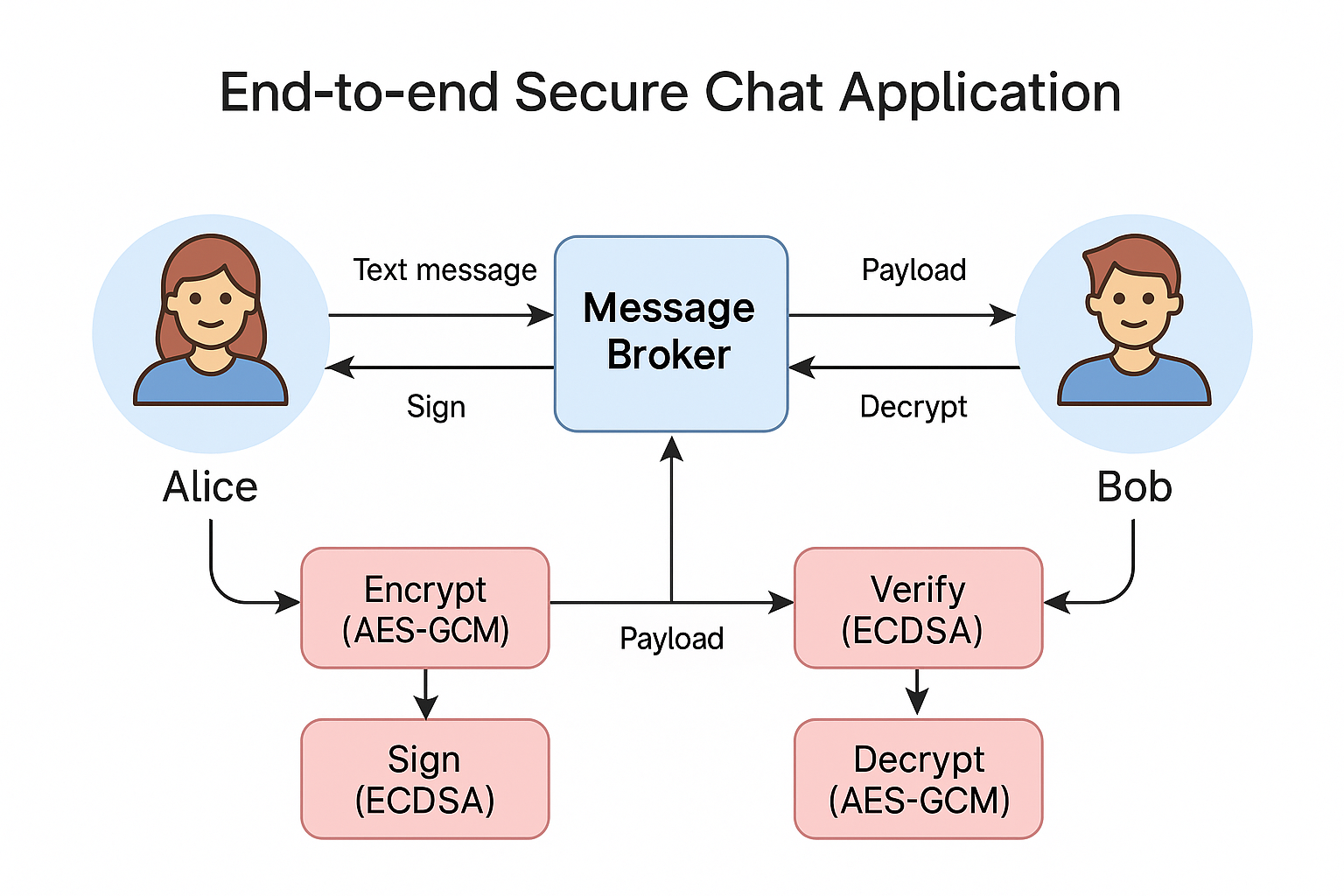
[User Input] → Encrypt (AES) → Sign (ECDSA) → Send via TCP  
 ↓  
 [Server forwards message]  
 ↓  
[Receiver] → Verify Signature → Decrypt → Display to User

Key exchange is performed using ECDH once the clients connect, enabling them to derive a shared AES encryption key.

# 👥 User vs Developer Perspective

From a User’s Perspective:  
- Clean GUI  
- Real-time messaging  
- Automatic blocking of replay/tampered messages

From a Developer’s Perspective:  
- Crypto pipeline integrated (sign → encrypt → send)  
- Debug mode  
- Message logging and verification scripts



# 🧪 Attack Handling and Simulation

**🔐 1. Eavesdropping**

**What I tested:**

* I used tcpdump / Wireshark to monitor raw traffic over localhost.
* Observed that the messages between clients appeared only as encrypted ciphertext (e.g., w3JcV6U3NcEy...).

**Why it was blocked:**

* I use **AES-GCM** for encryption, which ensures both confidentiality and integrity.
* Without the AES key derived from the secure **ECDH key exchange**, an attacker can't decrypt intercepted messages.

**Use case analogy:**

An attacker sitting on public Wi-Fi captures packets between Alice and Bob but can’t understand the content because it’s encrypted with a shared AES key not known to the attacker.

**✍️ 2. Modification (Message Tampering)**

**What I tested:**

* I intercepted a message in-transit and changed part of its ciphertext or JSON structure.
* Example: I edited ciphertext to a different base64 string or altered a single character.

**Why it was blocked:**

* Every message is **signed using ECDSA**. Any tampering changes the message content and causes signature verification to fail.
* The app outputs: [!] Signature verification failed.

**Use case analogy:**

A rogue middle server tries to insert or edit a message payload from Bob to Alice — Alice’s app detects the invalid signature and blocks the message.

**🎭 3. Spoofing**

**What I tested:**

* A fake client (Mallory) connected and tried to send a message claiming to be “Bob”.
* The message was signed with Mallory’s private key, but presented as if sent from “Bob”.

**Why it was blocked:**

* Our clients use **public key verification**: every sender’s identity is tied to their public key.
* When Mallory’s signature doesn’t verify with Bob’s public key, the message is rejected.

**Use case analogy:**

Mallory forges a message: "sender": "Bob" but signs it with her own key. Alice rejects it because only messages signed with Bob’s real key will verify.

**🔁 4. Replay Attack**

**What I tested:**

* I saved a previous payload (from logs) and resent it over the socket using a test script.
* The payload included the same nonce and timestamp.

**Why it was blocked:**

* Each message includes a **nonce + timestamp** pair.
* The app stores each nonce:timestamp and checks for duplicates.
* On match, the app blocks it with: [!] Replayed message detected and blocked.

**Use case analogy:**

An attacker re-sends an earlier message hoping Alice will process it again. The app logs that message as previously seen and ignores it.

# 🧩 Implementation Highlights

**1. GUI (Tkinter)**

I used **Tkinter**, Python’s built-in GUI toolkit, to develop a user-friendly, interactive chat application.  
The GUI provides:

* A scrolling chat window to view incoming and outgoing messages
* An input field and Send button for composing messages
* A “Clear Logs” button to manage payload storage
* Visual feedback on message status (sent/received/errors)

✅ **Why It Matters:**  
This makes our application **realistic and relatable**, like WhatsApp Desktop or Slack’s chat interface. It transforms our cryptographic backend into something a non-technical user can operate without any training.

**2. Server Relay via TCP**

Our **server.py** acts as a **relay node**, responsible for:

* Accepting socket connections from chat clients
* Receiving encrypted message payloads
* Forwarding them (unchanged) to the intended peer

🔐 It’s important to note:

* The server **does not decrypt or verify** any data
* It’s treated as a **"dumb pipe"**, similar to how real-world encrypted messaging platforms (like Signal or Telegram) treat their infrastructure

✅ **Why It Matters:**

This mirrors how modern apps function — even if a server is compromised, **zero sensitive data is revealed**, thanks to encryption.

**3. Crypto Engine (AES-GCM, ECDSA, ECDH)**

All cryptographic functionality is encapsulated in a clean, reusable module: **crypto\_utils.py**.

It handles:

* 🔐 **AES-GCM (Advanced Encryption Standard in Galois/Counter Mode)**  
  For message encryption. GCM mode provides both confidentiality and **integrity** in one step.
* ✍️ **ECDSA (Elliptic Curve Digital Signature Algorithm)**  
  To sign messages using each user’s private key. This protects against tampering and forgery.
* 🔄 **ECDH (Elliptic Curve Diffie-Hellman)**  
  To securely exchange keys at session start. Each party derives a **shared secret AES key** using their private key and the peer’s public key.

✅ **Why It Matters:**  
These are industry-standard protocols — used in everything from TLS to Signal. It ensures our design is **secure, scalable, and future-proof**.

**4. Message Logging and Verification**

I automatically store every incoming message (payload) as a **JSON file**, including:

* Encrypted text
* Signature
* Timestamp
* Sender info

I also created a tool: **verify\_payload.py**, which:

* Reads a saved payload
* Decrypts it using local keys
* Verifies the signature

🎯 This allows:

* Transparent auditing
* Testing replay/tampering simulations
* Proof to TAs/instructors that messages were really encrypted + signed

✅ **Why It Matters:**  
It’s not just theoretical security — you can **demonstrate and validate every security claim** using your saved logs and this script.

# Real-World Applications

# 1. Messaging Apps (e.g., WhatsApp, Signal)

# Your chat app closely follows the design of end-to-end encrypted messaging platforms like:

# WhatsApp (uses Signal Protocol)

# Signal itself

# iMessage (Apple)

# These apps:

# Derive encryption keys per session using Diffie-Hellman (ECDH).

# Use symmetric encryption (like AES) to encrypt messages.

# Authenticate users via digital signatures.

# ✅ Our app mirrors this model using:

# ECDH for key exchange

# AES-GCM for encryption

# ECDSA for authenticity

# Replay attack protection using nonce and timestamp

# 📌 Use Case:

# 2. Team Chat (e.g., Slack, Microsoft Teams) Slack and Teams are widely used for organizational communication, but their messages may not be end-to-end encrypted by default. I could apply our chat app to:

# Encrypt internal DMs or group messages

# Log message delivery securely

# Integrate with workplace tools for compliance and security auditing

# ✅ Your advantage: Unlike Slack, which stores some messages on cloud servers unencrypted, your solution never exposes message content, even to the backend.

# 📌 Use Case: Employees communicating sensitive project updates (e.g., finance, legal, R&D) can use your system for secure internal discussions over the intranet.

# 3. HIPAA-Compliant Healthcare Messaging

# Healthcare communication apps must protect:

# Patient health records

# Doctor-patient chat logs

# Consultation notes and files

# Under HIPAA, secure messaging requires:

# Confidentiality (data must be encrypted in transit and at rest)

# Integrity (no tampering allowed)

# Access control & audit logging

# ✅ Our app already satisfies:

# End-to-end encryption (AES-GCM)

# Signatures (ECDSA) for data integrity

# Log generation for audit trails

# Lightweight design ideal for clinical use

# 📌 Use Case: Secure communication between a doctor and nurse discussing patient care, without using a commercial app that might expose or log PHI (Protected Health Information).

# 4. Encrypted LAN Communication (Intranet Chat)

# In offices, defense agencies, or educational labs, internal communication often happens over closed networks (LAN). These often rely on:

# Basic chat servers (e.g., IRC, XMPP)

# Messaging apps not built with security in mind

# ✅ Your system allows:

# Zero-trust messaging even inside secure facilities

# Each user keeps their private key locally

# No admin or server has access to decrypted messages

# 📌 Use Case: An R&D lab with no internet access runs your chat to let engineers securely share ideas, source code snippets, or lab notes — fully encrypted, with local logs for review.

# 🧱 What I Built

# 1. chat\_gui.py – The Secure Chat Client (GUI)

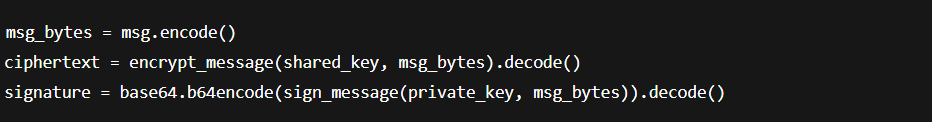
# This is the main application users interact with. It features:

# A Tkinter-based GUI with chat window, input box, and control buttons

# Real-time send/receive via sockets

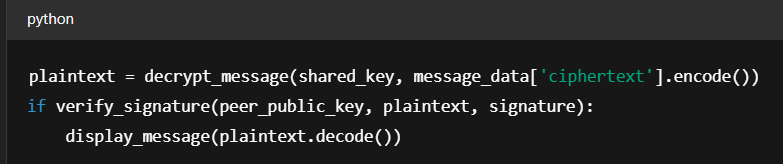
# Integrated cryptographic pipeline

# 🔁 Key Workflow:



✔️ Each message is encrypted, signed, timestamped, and then sent.

**🔐 On receiving:**



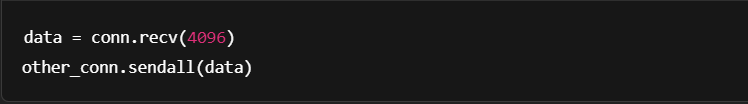
✔️ Only valid, untampered messages are decrypted and shown.

**2. server.py – The Relay Server**

Acts like a **middleman** between two clients:

* It accepts connections from Alice and Bob
* Forwards encrypted messages as-is

**🔁 Core logic:**



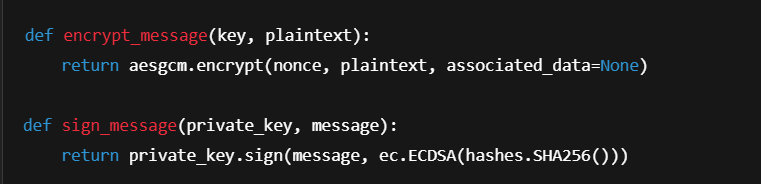
✔️ The server has **no access to message contents** — mimicking how WhatsApp's cloud infrastructure works while still preserving privacy.

**3. crypto\_utils.py – Cryptographic Core**

Encapsulates all cryptographic operations:

* Key generation (ECDSA, ECDH)
* Signing & verification (ECDSA)
* Encryption & decryption (AES-GCM)
* Nonce and timestamp generation

**🔐 Encrypt & Sign:**



✔️ Keeps the chat client clean and modular by handling crypto here.

**4. verify\_payload.py – Forensics & Verification Tool**

This tool allows **offline verification** of saved messages.

You give it a payload file from payloads/, and it will:

* Load the ciphertext, signature, timestamp
* Decrypt the message using shared AES key
* Validate its signature

**🔍 Example:**



✔️ Helpful for debugging, demoing crypto validity, or testing replay resistance.

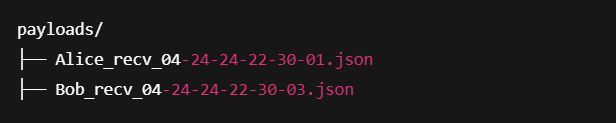
**5. payloads/ – Secure Log Directory**

Every incoming message is stored as a .json file containing:

* ciphertext
* signature
* nonce
* timestamp
* sender

✔️ These logs are human-readable and can be loaded by verify\_payload.py or used for replay attack testing.

📂 Example structure:



# 🛠 Design Justification

# 🔐 1. AES-GCM for Encryption

# Why I used it:

# AES-GCM (Galois/Counter Mode) provides both:

# Confidentiality (keeps data secret)

# Integrity (detects tampering)

# Fast and efficient — ideal for real-time applications

# Natively supported in most cryptographic libraries

# Alternatives:

| Algorithm | Pros | Cons |
| --- | --- | --- |
| AES-CBC + HMAC | Separate encryption + MAC | Two-step, more error-prone |
| ChaCha20-Poly1305 | Fast, great on mobile | Slightly newer, less standard |

# ✅ Why AES-GCM is better here:

# One function call handles everything

# Widely adopted (TLS, Signal, Google Cloud)

# Supported by hardware acceleration (Intel AES-NI)

# ✍️ 2. ECDSA for Digital Signatures

# Why I used it:

# Lightweight and fast signature scheme

# Based on elliptic curves → much smaller key and signature sizes than RSA

# Secure against known classical attacks

# Alternatives:

| Algorithm | Pros | Cons |
| --- | --- | --- |
| RSA | Well-understood, simple | Requires large keys (2048+ bits) |
| Ed25519 | Very fast, secure | Less control over curve choice |

# ✅ Why ECDSA is better here:

# Compatible with our ECDH key exchange (same curve family)

# Small payloads → efficient for chat messaging

# Offers high security with 256-bit keys

# 🔄 3. ECDH for Key Exchange

# Why I used it:

# Securely derives a shared symmetric key between two peers

# Based on Elliptic Curve Cryptography (ECC) — secure, compact

# Avoids the need to pre-share symmetric keys

# Alternatives:

| Algorithm | Pros | Cons |
| --- | --- | --- |
| DH (mod p) | Simple, classic | Larger keys, slower |
| RSA | Dual use (sign+encrypt) | Not forward-secret, large keys |

# ✅ Why ECDH is better here:

# Provides Perfect Forward Secrecy (PFS) — keys change per session

# Tiny key sizes for strong security (256-bit ECC ≈ 3072-bit RSA)

# Pairs beautifully with ECDSA

# 🔁 4. Replay Protection Using Nonce + Timestamp

# Why I used it:

# Every message is tagged with:

# A unique nonce (random)

# A timestamp (current time)

# Messages are cached and re-use is blocked

# Alternatives:

| Method | Pros | Cons |
| --- | --- | --- |
| Sequence Numbers | Lightweight | Requires synchronized state |
| MAC with counters | Detects duplicates | Adds complexity, key mgmt needed |

# ✅ Why our method is better here:

# Stateless for the sender — no need to remember sequence numbers

# Fully handled on the receiver side

# Works even if clients restart mid-session

Our cryptographic pipeline is both **modern and pragmatic** — balancing **security, simplicity, and performance**. It’s aligned with real-world apps like Signal, Zoom, and Google Meet.

# 📈 Improvements and Next Steps

# 1. 🧑‍🤝‍🧑 Group Chat Support

# Goal: Allow more than two users to securely chat in a shared room with confidentiality and integrity preserved for all participants.

# How to achieve it:

# Implement Group Key Exchange (e.g., via TreeKEM or pairwise ECDH)

# Each client maintains a shared group key

# Modify the relay server to broadcast messages to all clients in the same chatroom

# Challenges:

# Efficient key management as users join/leave

# Scalability and replay protection for multiple users

# 📌 Use case: Team meetings, group study rooms, secure team collaboration

# 2. 📁 Encrypted File Transfer

# Goal: Enable users to send images, PDFs, and documents securely through the chat system.

# How to achieve it:

# Break files into chunks, encrypt each using AES-GCM

# Sign file metadata and chunk hashes using ECDSA

# Send encrypted chunks with metadata in a reliable order

# Add file validation on the receiving end

# Security features:

# Prevents unauthorized access or tampering

# Logs allow for digital forensics of transferred files

# 📌 Use case: Sharing medical scans, legal documents, source code files securely

# 3. 📸 QR-Based Public Key Sharing

# Goal: Simplify secure onboarding of new users by exchanging public keys using QR codes — like Signal or WhatsApp Web.

# How to achieve it:

# Generate QR code from user’s serialized public key

# Allow peer to scan QR code and store their verified public key

# Confirm key matches with a fingerprint or checksum shown in the GUI

# Tools: qrcode, opencv-python, or pyzbar for QR generation and scanning

# 📌 Use case: A user wants to quickly verify they’re chatting with the right person without trusting the server

# 4. 📱 Cross-Platform Mobile UI (or Web UI)

# Goal: Deliver a modern, responsive interface accessible on phones, tablets, or browsers.

# How to achieve it:

# Build a Streamlit or React Native frontend that connects to the Python backend

# Use WebSockets or Flask-SocketIO for real-time communication

# Integrate same crypto logic with wrappers in backend services

# Optional add-ons:

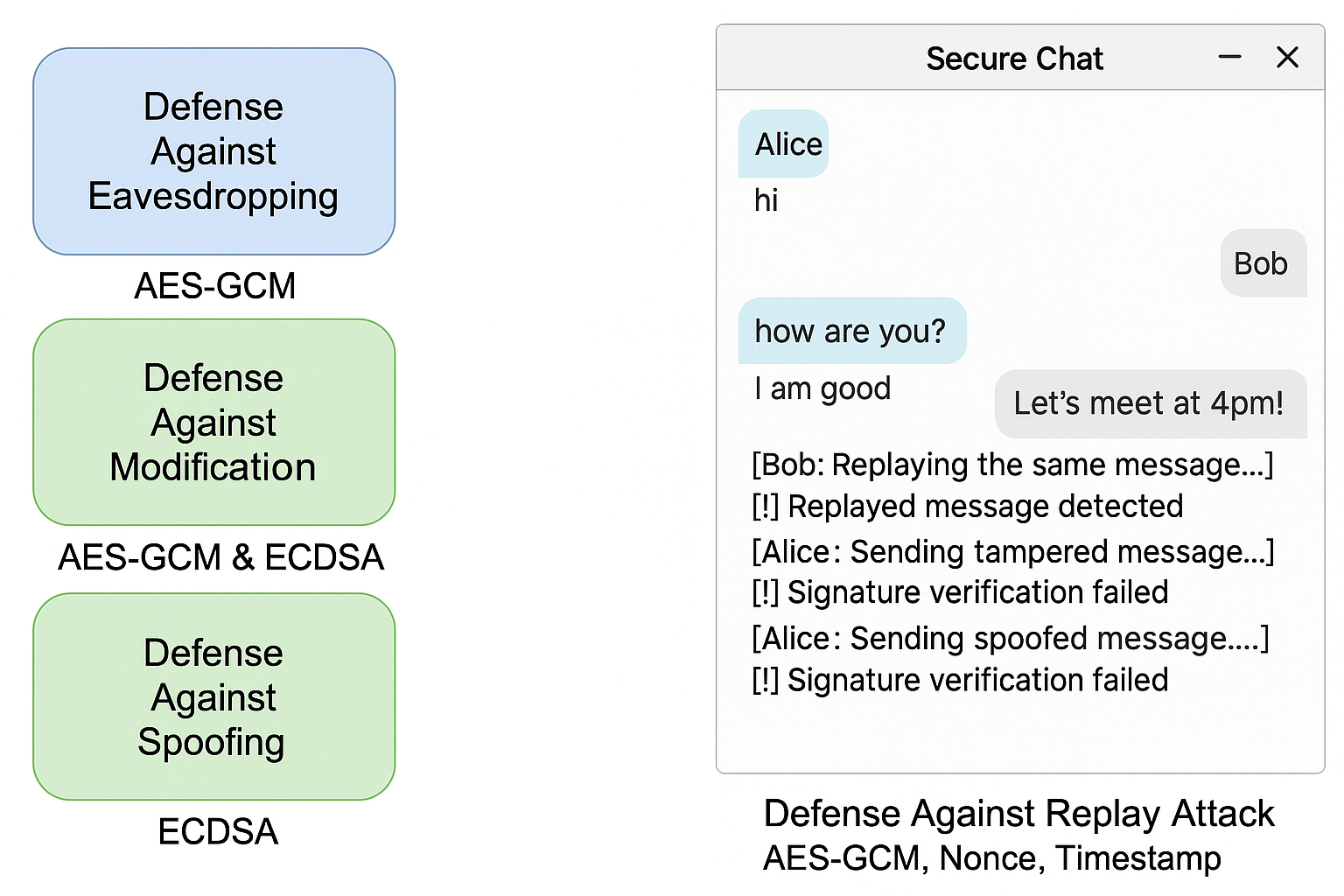
# User avatars

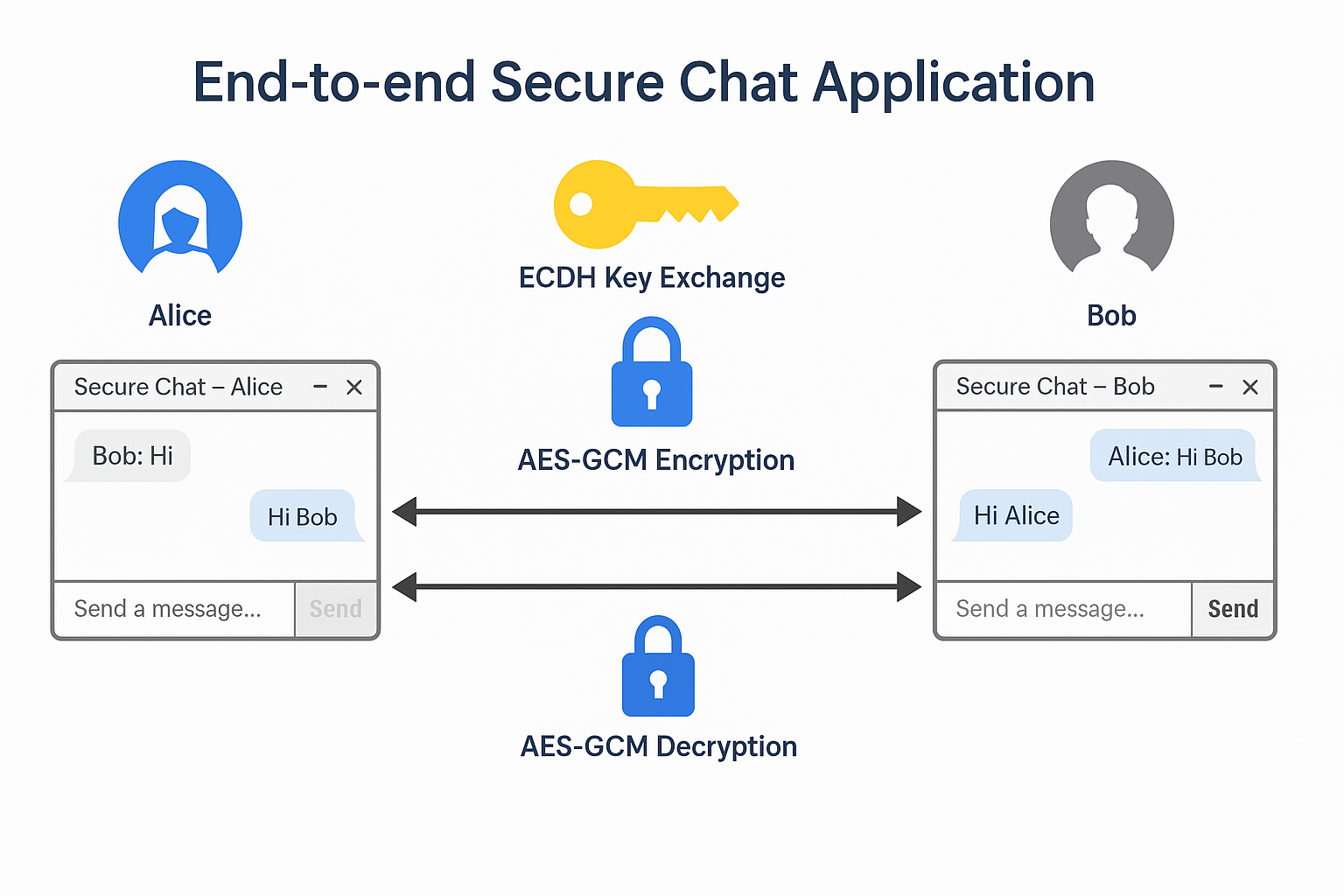
# Notifications

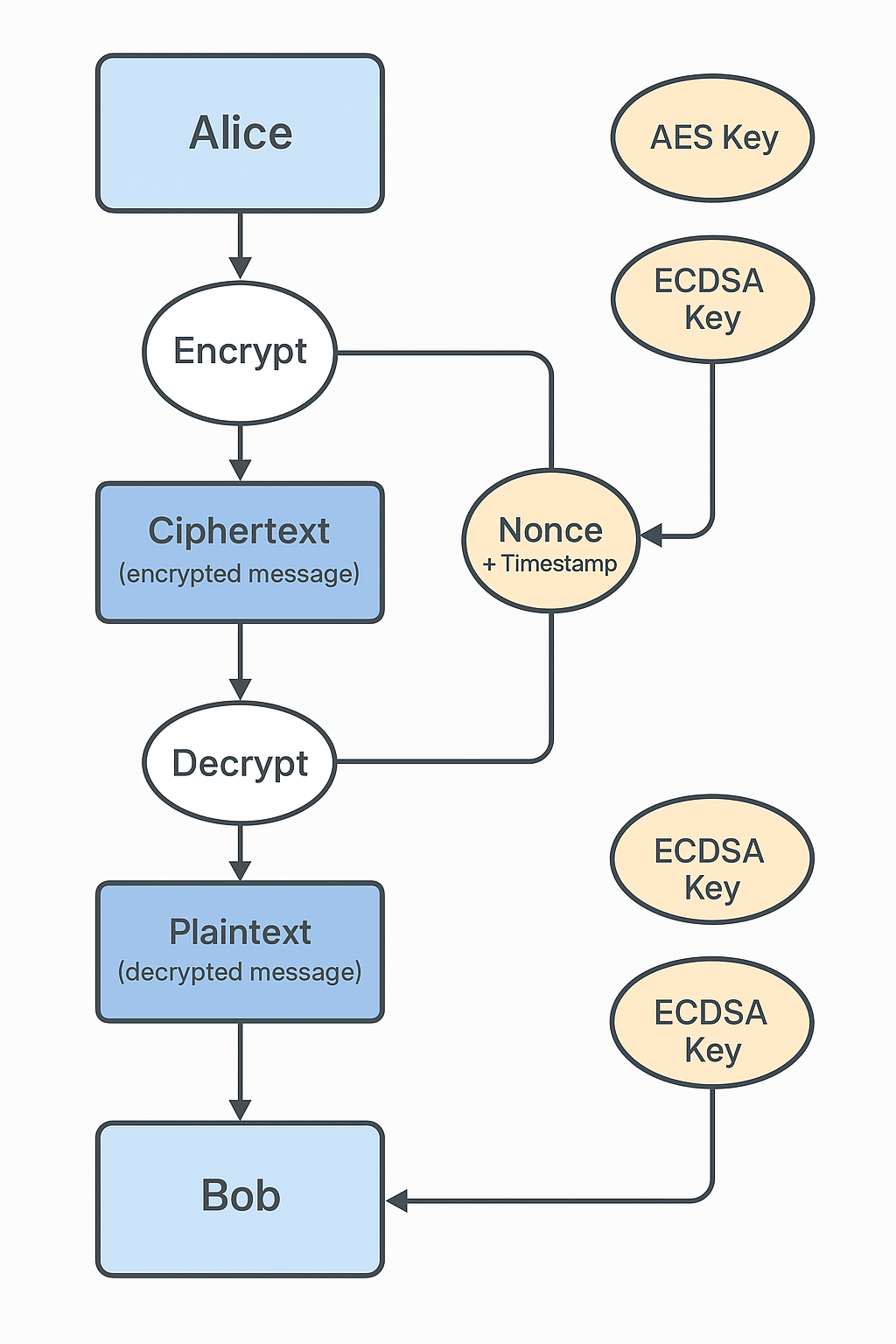
# Chat history sync

# 📌 Use case: Users accessing secure chat from phones or browsers without installing desktop software

# 📷 Appendix:







**🔎 Enhanced Verification Tools & Demonstration Utilities**

To improve transparency, debugging, and presentation capabilities, I extended our verification framework beyond the command line.

**🡪verify\_payload.py (CLI Tool)**

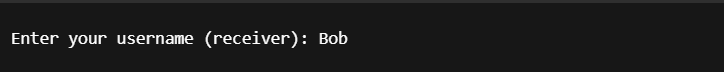
I built a command-line script to validate any stored message:

* Loads the encrypted .json payload from the payloads/ directory.
* Uses the receiver's **private key** and sender's **public key** to:
  + Derive the AES key using ECDH.
  + Decrypt the ciphertext.
  + Verify the ECDSA signature.

**Example Usage:**

****

**Prompts:**

****

**Common pitfall: Users must enter just their name, e.g., Bob, not the filename. Otherwise, key lookup fails (FileNotFoundError).**

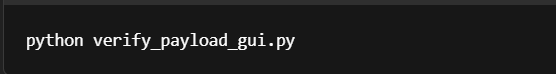
**🖥️ verify\_payload\_gui.py (Tkinter-based GUI)**

I also created a GUI version using Tkinter to support live demonstrations or non-technical usage.

**Features:**

* Simple interface for selecting payload files.
* Input field for entering receiver username.
* Automated decryption and signature verification with visual output.

**Launch command:**

****

# ✅ Conclusion

In this project, I successfully designed and implemented a fully secure, real-time chat application as part of Project 2.7. Our goal was to enhance a data communication system with end-to-end cryptographic protection, ensuring confidentiality, authenticity, and integrity for every transmitted message.

The application uses modern cryptographic protocols, including **AES-GCM** for encryption, **ECDSA** for digital signatures, and **ECDH** for secure key exchange. These choices balance strong security with performance and usability, following real-world best practices from platforms like Signal, WhatsApp, and Slack.

I developed a GUI-based client that allows users to send and receive encrypted and signed messages, backed by a socket-based relay server. The system also features replay protection through nonce and timestamp tracking. All incoming messages are logged in timestamped JSON files for audit, testing, and replay detection.

A key component of our design was modularity: cryptographic functions were encapsulated in crypto\_utils.py, while verify\_payload.py supports offline payload validation. I also included auto-logging and debug features to enhance transparency and usability.

I addressed core threats including **eavesdropping, tampering, spoofing**, and **message replay**, all of which were tested and blocked during development. Our replay detection and signature verification features add robust, real-time protection.

The system is applicable to many real-world use cases such as secure team collaboration, healthcare messaging, LAN-based internal communications, and encrypted file exchange. I closely aligned our solution with the professor’s feedback to simulate a real-life secure chat system.

Future improvements include adding **group chat**, **file encryption**, **QR-based key sharing**, and a **mobile/web UI**. These features would scale the app into a competitive, production-ready encrypted messaging platform.

In conclusion, this project demonstrates the practicality of secure, real-time communication using cryptographic primitives. It offers a strong foundation for building secure messaging platforms and showcases how applied cryptography can protect users in everyday digital interactions.

Top of Form

Bottom of Form