National University of Computer and Emerging Sciences (FAST–NUCES)

Assignment #2

Information Security

Semester:	Fall 2025
Total Marks:	100 (+5 Bonus)
Submission Deadline:	As Per GCR

Repository / Code Skeleton:
https://github.com/maadilrehman/securechat-skeleton

Introduction

You will design and implement a practical cryptographic system using **the material covered in class**: **AES-128 (block cipher)**, **RSA with X.509 certificates**, **basic Diffie–Hellman (DH)**, and **SHA-256**. You will build a small **root CA(self learning)**, issue certificates, and implement registration, authentication, key agreement, and encrypted messaging to achieve **confidentiality, integrity, authenticity, and non-repudiation (CIANR)**.

You will implement a mini **Console Based Secure Chat System** (client–server) that demonstrates how real systems combine primitives to achieve **CIANR**.

Threat Model & Goals

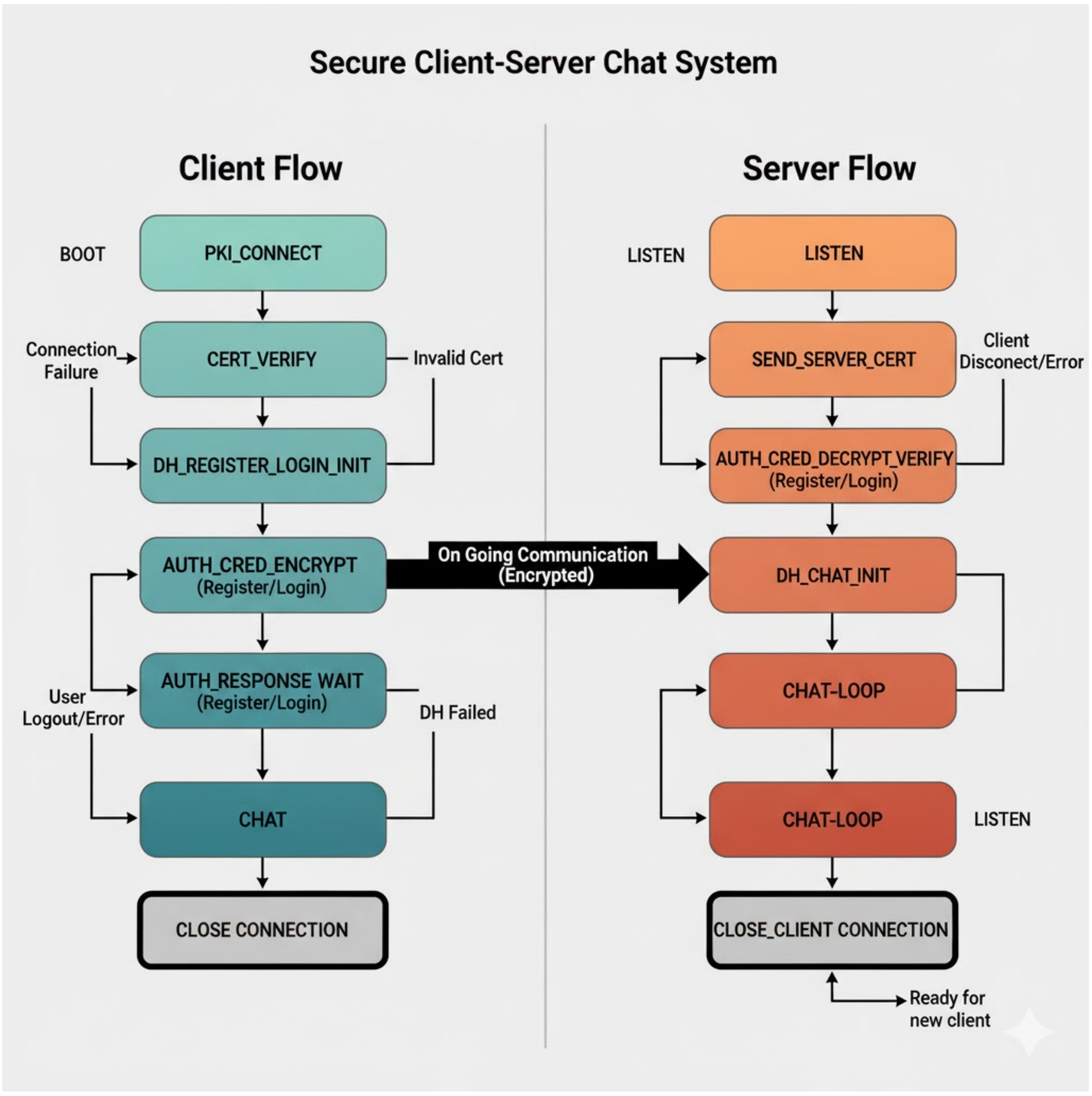
Adversary: passive eavesdropper, active MitM who can replay/modify/inject, untrusted client attempting login/password guessing.

Goals: confidentiality, peer authenticity, integrity (no undetected tampering/replay), certificate validation, no plaintext credential transit, **non-repudiation via per-message signatures and a signed session transcript**.

1 Secure Chat Protocol

This section details the comprehensive secure protocol implemented for the client-server chat system, designed to rigorously achieve **Confidentiality, Integrity, Authenticity, and Non-Repudiation (CIANR)**. It outlines the full communication lifecycle — from identity verification and session initialization to encrypted message exchange and verifiable evidence generation.
The protocol is divided into four major phases: Control Plane (Negotiation and Authentication), Key Agreement, Data Plan (On Going Communication), Tear down (Non-Repudiation)

Workflow: The following figure visualizes the basic flow of client and server interact during these phases, showing how cryptographic primitives interlock to ensure end-to-end CIANR.



1.1 Control Plane (Negotiation and Authentication)

This phase establishes **trust and identity** between the client and the server before any sensitive data exchange. Both parties present their X.509 certificates, issued by the same trusted CA, and perform mutual verification to confirm authenticity. Once identities are verified, registration or login messages are exchanged under an initial ephemeral AES key derived from a temporary DH exchange.

The control plane ensures that:

- Each participant is authenticated through their CA-issued certificate.
- No credentials are transmitted in plaintext.
- New users are securely registered, and returning users are authenticated through salted password hashes verified at the server side.

Control Plane Message Formats:

```
{ "type":"hello", "client_cert":"...PEM...", "nonce": base64 }
{ "type":"server_hello", "server_cert":"...PEM...", "nonce": base64 }
{ "type":"register", "email":"", "username":"", "pwd":
base64(sha256(salt||pwd)), "salt": base64 }
{ "type":"login", "email":"", "pwd": base64(sha256(salt||pwd)),
"nonce": base64 }
```

Explanation: The "hello" exchange carries the digital certificates and nonces (for freshness). The "register" and "login" messages encapsulate credentials encrypted under the negotiated AES key to prevent leakage of sensitive information on the wire. Mutual certificate validation and credential hashing form the foundation of authentication and trust.

1.2 Key Agreement (Post-Authentication)

Once authentication is successful, both sides proceed to derive a **session key** that will secure all future chat messages. This is done through a classical **Diffie–Hellman (DH)** key exchange, which provides a shared secret even over an untrusted network.

Purpose: To derive a common symmetric key K known only to the client and server, which will later serve as the AES-128 encryption key for the chat session.

Key Exchange Process:

- The client sends public DH parameters (p, g) and its public value $A = g^a \bmod p$.
- The server responds with its own public value $B = g^b \bmod p$.
- Both compute the shared secret $K_s = A^b \bmod p = B^a \bmod p$.
- The final AES key is derived as:

$$K = \text{Trunc}_{16}(\text{SHA256}(\text{big-endian}(K_s)))$$

- This 16-byte key (K) is used for AES-128 encryption and decryption during the data exchange phase.

Key Agreement Message Formats:

```
{ "type":"dh_client", "g": int, "p": int, "A": int }
{ "type":"dh_server", "B": int }
```

$K_s = A^b \bmod p = B^a \bmod p; \quad K = \mathrm{Trunc}_{16}(\mathrm{SHA256}(\text{big-endian}(K_s)))$.

Explanation: This phase guarantees **confidentiality** of the encryption key without ever transmitting it. Even if an adversary captures all DH messages, they cannot derive K without knowing the private exponents a or b. The derived key K now securely binds both participants into an encrypted session.

1.3 Data Plane (Encrypted Message Exchange)

After the session key is established, the client and server can securely exchange chat messages. The data plane applies multiple layers of protection—encryption for confidentiality, hashing and digital signatures for integrity and authenticity, and sequence numbers/timestamps for freshness and replay protection.

Per-Message Workflow:

1. The sender composes a plaintext message in the console.
2. The message is padded (PKCS#7) and encrypted using **AES-128** with the shared session key K.
3. A **SHA-256** digest is computed over the concatenation of `seqno`‖`timestamp`‖`ciphertext`.
4. This digest is then signed with the sender's RSA private key to create a digital signature.
5. The resulting structure ensures that even a single-bit change in ciphertext or metadata will invalidate the signature.

Data Plane Message Format:

```
{ "type":"msg", "seqno": n, "ts": unix_ms, "ct": base64,
   "sig": base64(RSA_SIGN( SHA256( seqno || ts || ct ) )) }
```

Explanation: Each field serves a specific role:

- `seqno` – ensures messages are processed in order and prevents replays.
- `ts` – a timestamp to reject stale messages.
- `ct` – ciphertext produced by AES-128 encryption.

- `sig` – RSA signature over the SHA-256 hash of metadata and ciphertext.

This structure collectively enforces:

- **Confidentiality** (AES encryption),
- **Integrity** (SHA-256 digest),
- **Authenticity** (RSA signature validation),
- **Freshness** (sequence + timestamp enforcement).

1.4 Non-Repudiation (Session Evidence)

The final stage ensures that neither participant can deny sending or receiving messages. Both client and server maintain an append-only transcript containing all message metadata:

```
seqno | ts | ct | sig | peer-cert-fingerprint
```

At session end, each side computes a **TranscriptHash**:

$$\text{TranscriptHash} = \text{SHA256}(\text{concatenation of transcript lines})$$

This hash is digitally signed with the sender's RSA private key, producing a verifiable **SessionReceipt** that serves as cryptographic proof of communication.

SessionReceipt Format:

```
{ "type":"receipt", "peer":"client|server", "first_seq":...,
"last_seq":...,
   "transcript_sha256":hex, "sig":base64(RSA_SIGN( transcript_sha256
)) }
```

Explanation: The SessionReceipt acts as a digital evidence artifact for non-repudiation. It allows any third party (e.g., auditor or TA) to:

- Recompute the transcript hash,
- Verify the RSA signature using the participant's certificate,
- Confirm the authenticity and completeness of the recorded session.

Together, these phases transform the simple act of chatting into a fully accountable, verifiable, and cryptographically secure communication process.

2 System Requirements & Implementation

All students must fork and work in their own GitHub copy of:

https://github.com/maadilrehman/securechat-skeleton

All development, commits, and pushes **must be made on GitHub**. Your fork will serve as both the submission and the audit trail for grading. Each feature i.e., CA generation, MySQL integration, Diffie–Hellman (DH) key exchange, AES encryption, and RSA signatures—should appear as a distinct, meaningful commit in your repository history.

The complete functional and technical requirements of the system are defined below.

2.1 PKI Setup and Certificate Validation

Before any secure communication begins, both the client and the server must prove their identities using certificates issued by a self-built Certificate Authority (CA). You will create this CA yourself and use it to issue signed certificates to the server and client.

Implementation Requirements:

- Implement two scripts:
  - `scripts/gen_ca.py`: creates a root CA by generating a private key and a self-signed certificate.
  - `scripts/gen_cert.py`: issues RSA X.509 certificates for both server and client, signed by the root CA.
- The root CA's private key and certificate should be stored locally in the `certs/` folder (never committed to GitHub).
- Each entity (client and server) must possess:
  - Its own RSA keypair (private + public key),
  - A signed certificate issued by your CA.
- During connection establishment:
  - The client sends its certificate to the server.
  - The server sends its certificate to the client.
  - Both verify the received certificate by checking:
    1. Signature chain validity (trusted CA),
    2. Expiry date and validity period,
    3. Common Name (CN) or hostname match.
- The server must **reject and log** any self-signed, expired, or untrusted certificate with a clear error (e.g., `BAD_CERT`).

- Your report must include certificate inspection results (either through a script or using `openssl x509 -text`) as evidence.

This stage ensures **authentication**—each party can verify the other's identity before any sensitive data (credentials or messages) is exchanged.

You may refer to the SEED Security Lab on Public Key Infrastructure for conceptual guidance, certificate hierarchy understanding and certification authority creation: Crypto_PKI Seedslab

2.2 Registration and Login

All user information—email, username, and password—must be handled confidentially and stored securely in a **MySQL database**, not in plaintext files. This database will only store user credentials; chat messages and transcripts must never be written to the database.

1. The client starts by connecting to the server and exchanging certificates. Both validate each other's certificates before proceeding.
2. A temporary **Diffie–Hellman (DH)** exchange is performed to generate a shared secret K_s.
3. The shared secret is converted into an **AES-128 encryption key** using:
$$K = \text{Trunc}_{16}(\text{SHA256}(\text{big-endian}(K_s)))$$
4. The client then encrypts the registration data (email, username, password) using AES-128 (block cipher + PKCS#7 padding) and sends it to the server.
5. The server decrypts this payload, verifies that the username or email is not already registered, and then:
   - Generates a 16-byte random **salt** for the user.
   - Computes the salted password hash as:
     ```
     pwd_hash = hex(SHA256(salt || password))
     ```
   - Stores this in a MySQL table as:
     ```
     users(email VARCHAR, username VARCHAR UNIQUE, salt VARBINARY(16),
                          pwd_hash CHAR(64))
     ```
6. During login, a new DH exchange and AES key are used. The client sends encrypted credentials, and the server recomputes the salted hash to verify them.
7. Login succeeds only if:
   - The client certificate is valid and trusted, and
   - The salted hash matches the stored `pwd_hash`.

This phase ensures that all user credentials remain confidential during transmission and that password integrity is maintained through salted hashing.

2.3 Session Key Establishment (Basic Diffie–Hellman)

After successful login, both client and server must establish a new **chat session key** for encryption.

- Use classical DH with public parameters (p, g) known to both.
- Each side chooses a private key (a or b) and computes $A = g^a \bmod p$, $B = g^b \bmod p$.
- The shared secret is derived as $K_s = B^a \bmod p = A^b \bmod p$.
- The final session key is obtained by truncating the SHA-256 hash of K_s to 16 bytes:

$$K = \text{Trunc}_{16}(\text{SHA256}(\text{big-endian}(K_s)))$$

- This key will be used as the AES-128 key for encrypting and decrypting chat messages in that session.

This ensures that every chat session has a unique, derived encryption key that cannot be reused or derived from prior sessions—providing confidentiality and forward separation.

2.4 Encrypted Chat and Message Integrity

Once a session key is established, all communication between the client and server must be encrypted and signed to guarantee privacy and message integrity.

Message Transmission Process:

1. The sender reads plaintext input from the console.
2. The plaintext is padded using PKCS#7 and encrypted using AES-128 (block cipher) with the session key K.
3. The sender computes:

$$h = \text{SHA256}(\texttt{seqno || timestamp || ciphertext})$$

4. The hash h is signed with the sender's RSA private key: `sig = RSA_SIGN(h)`.
5. The message is sent as JSON:

```
{ "type":"msg", "seqno":n, "ts":unix_ms, "ct":base64,
  "sig":base64(RSA_SIGN(SHA256(seqno||ts||ct))) }
```

6. Upon receiving, the recipient:
   - Checks that `seqno` is strictly increasing (replay protection).
   - Verifies the signature using the sender's certificate and recomputed hash.
   - Decrypts the ciphertext using AES-128 and removes PKCS#7 padding.

Purpose: This guarantees:

- **Confidentiality** only encrypted ciphertext is transmitted.
- **Integrity** any bit change causes hash mismatch and signature failure.
- **Authenticity** only the legitimate sender with the private key can generate valid signatures.
- **Freshness** replayed messages are rejected using sequence numbers.

2.5 Non-Repudiation and Session Closure

At the end of the chat, both participants must create a verifiable proof of the conversation to prevent denial of communication.

Process:

1. Each side maintains a local append-only transcript file containing: `seqno | timestamp | ciphertext | sig | peer-cert-fingerprint`.
2. Compute a transcript hash:

$$\text{TranscriptHash} = \text{SHA256}(\text{concatenation of all log lines})$$

3. Each side signs the transcript hash to generate a **SessionReceipt**:

```
{ "type":"receipt", "peer":"client|server", "first_seq":...,
"last_seq":...,
   "transcript_sha256":hex, "sig":base64(RSA_SIGN(transcript_sha256))
}
```

4. The SessionReceipt is exchanged or stored locally.
5. Offline verification must confirm that any transcript modification invalidates the receipt signature.

This phase provides **non-repudiation**, ensuring that neither side can later deny participation since every exchanged message and final transcript hash is digitally signed.

3 Testing & Evidence

- **Wireshark**: show encrypted payloads (no plaintext). Add display filter(s) used.
- **Invalid certificate test**: forged/self-signed/expired certs → `BAD_CERT`.
- **Tampering test**: flip a bit in `ct` → recomputed digest/signature fails → `SIG_FAIL`.
- **Replay test**: resend old `seqno` → `REPLAY`.

- **Non-repudiation:** export transcript & **SessionReceipt**; show offline verification:
  1. Verify each message: recompute SHA-256 digest; verify RSA signature.
  2. Verify receipt: verify RSA signature over `TranscriptHash`.
  3. Show that any edit breaks verification.

4 Submission Instructions

1. **Fork** securechat-skeleton and do all work on your fork. At least **10 meaningful commits** showing progress.
2. Submit on GCR:
   - Downloaded ZIP of your GitHub repo.
   - Mysql schema dump and sample records.
   - GitHub readme.md file, clearly stating, execution steps, configurations required(if any), sample input/output formats and also link to your github repo.
   - **Report**: `RollNumber-FullName-Report-A02.docx`.
   - **Test Report**: `RollNumber-FullName-TestReport-A02.docx`.

5 Grading Rubric

Objective	Excellent (10–8)	Good (7–4)	Bad (3–0)	Weight
GitHub Workflow	• Fork accessible; ≥10 clear commits. • Sensible `README`; proper `.gitignore`; no secrets.	• 4–6 commits; basic `README`; minor hygiene issues.	• ≤3 commits or end-dump; repo missing/inaccessible secrets committed.	**20%**
PKI Setup & Certificates	• Root CA works; server & client certs issued. • Mutual verification; expiry/hostname checks. • Invalid/self-signed/expired certs rejected.	• CA/certs created; only one-way checks or missing expiry/hostname.	• Accepts self-signed; no verification; crashes or leaks internals.	**20%**

Objective	Excellent (10–8)	Good (7–4)	Bad (3–0)	Weight
Registration & Login Security	• Per-user random salt ≥16B; store `hex(sha256(salt||p` • Credentials sent only after cert checks and under encryption. • No plaintext passwords in files/logs; constant-time compare.	• Salted hashing but weak/reused salts; transit protected.	• Plain/unsalted storage; plaintext transit; logs contain passwords; login without valid cert.	**20%**
Encrypted Chat (AES-128 block only)	• DH after login; $K = \mathrm{Trunc}_{16}(\mathrm{SHA256}(K_s))$ • AES-128 used correctly with PKCS#7 padding. • Clean send/receive path; clear error handling.	• AES works but padding bugs or occasional crashes.	• Wrong key derivation; accepts corrupted data; frequent crashes.	**20%**
Integrity, Authenticity & Non-Repudiation	• For each message: compute h = SHA256(`seqno`||`ts`||`ct` and **RSA-sign** h; verify every message by recomputing h. • Strict replay defense on `seqno`. • Append-only transcript; **Session-Receipt** (signed transcript hash) produced and exported. • Offline verification documented (message digests + sigs; receipt over transcript hash).	• Sign/verify present but missing field in digest or weak replay handling. • Transcript saved but receipt or verification steps incomplete.	• No signatures; accepts modified/replayed messages. • No transcript/receipt; NR not demonstrable.	**10%**

Objective	Excellent (10–8)	Good (7–4)	Bad (3–0)	Weight
Testing & Evidence	• PCAP/screens show encrypted payloads; filters included. • Invalid/expired cert rejection; tamper + replay tests shown. • Steps reproducible by TA.	• Most tests shown; short notes; partially reproducible.	• Little/no evidence; not reproducible.	**10%**

6 Implementation Notes

- You are **not required** to implement the internal mathematics of AES, RSA, Diffie–Hellman or Hash function. Use reliable, standard Python libraries.
- The objective is to integrate these primitives into a correct **application-layer** protocol that achieves **Confidentiality, Integrity, Authenticity, and Non-Repudiation (CIANR)** as specified.
- **Do not use TLS/SSL** or any secure-channel abstraction (e.g., `ssl`, OpenSSL socket wrappers, HTTPS, wss). All certificate exchange/validation, key agreement, encryption, signatures, replay defense, transcripts, and the final SessionReceipt must be implemented at the application layer.
- Your grade focuses on correct protocol logic, state handling, and evidence (Wireshark/PCAPs), not cryptographic algorithm reimplementation.
- Cite any external snippets you adapt in code comments and your report.

7 Academic Integrity & Reproducibility

- **Do not** share keys/salts; **do not** commit secrets. Provide a `.env.example`.
- Scripts must be reproducible (CA, certs, start server/client, replicate screenshots).
- Cite any external snippets you adapt.

Outcome Alignment: Assignment Outcomes (AOs) → CLOs → SA

Assignment Outcome (AO)	Measured by (in this assignment)	CLO Ref.	SA Ref.
AO1: PKI engineering and certificate validation	CA + cert scripts; mutual verification; chain/expiry checks; rejection of invalid certs.	CLO 1	SA3, SA4, SA5
AO2: Secure credential handling	Salted SHA-256 storage; no plaintext credentials; dual gate: valid cert *and* correct creds.	CLO 1	SA1, SA3, SA5
AO3: Authenticated key agreement & symmetric protection	DH; AES-128 key from SHA-256 truncation; correct padding; no modes.	CLO 1, CLO 3	SA3, SA4, SA5
AO4: Message integrity, authenticity, and non-repudiation	Per-message RSA signatures over SHA-256 digests; replay protection via `seqno`; append-only transcript; signed SessionReceipt; offline verifiability.	CLO 1, CLO 3	SA2, SA3, SA5
AO5: Evidence-driven validation	Wireshark/PCAPs; invalid/expired cert tests; tamper/replay demos; clear documentation.	CLO 3, CLO 4	SA5, SA7
AO6: Reproducible engineering practice	GitHub workflow (10+ commits), README, reproducible scripts, no secrets in VCS.	CLO 2, CLO 4	SA6, SA7, SA8, SA9

Course Learning Outcomes (CLOs)

Code	Statement
CLO 1	Explain key concepts of information security such as design principles, cryptography, risk management.
CLO 2	Discuss legal, ethical, and professional issues in information security.
CLO 3	Apply various security and risk management tools for achieving information security and privacy.
CLO 4	Identify appropriate techniques to tackle and solve real-life problems in information security.

Seoul Accord Graduate Attributes (SA)

Code	**Attribute**
SA1	Problem Analysis
SA2	Investigation
SA3	Design / Development of Solutions
SA4	Modern Tool Usage (incl. computing foundations)
SA5	Use of Appropriate Tools and Techniques (implementation, experimentation)
SA6	Individual and Team Work
SA7	Communication
SA8	Professionalism / Ethics
SA9	Life-long Learning