SecureChat System - Implementation Report

Assignment #2 - Information Security (CS-3002)

Student Name: [Your Full Name]
Roll Number: [Your Roll Number]
Email: [Your FAST Email]
GitHub Repository: [Your Fork URL]
Submission Date: [Date]

Table of Contents

---

1. Executive Summary

This report documents the design, implementation, and testing of SecureChat, a console-based secure communication system that demonstrates practical application of cryptographic primitives. The system achieves:

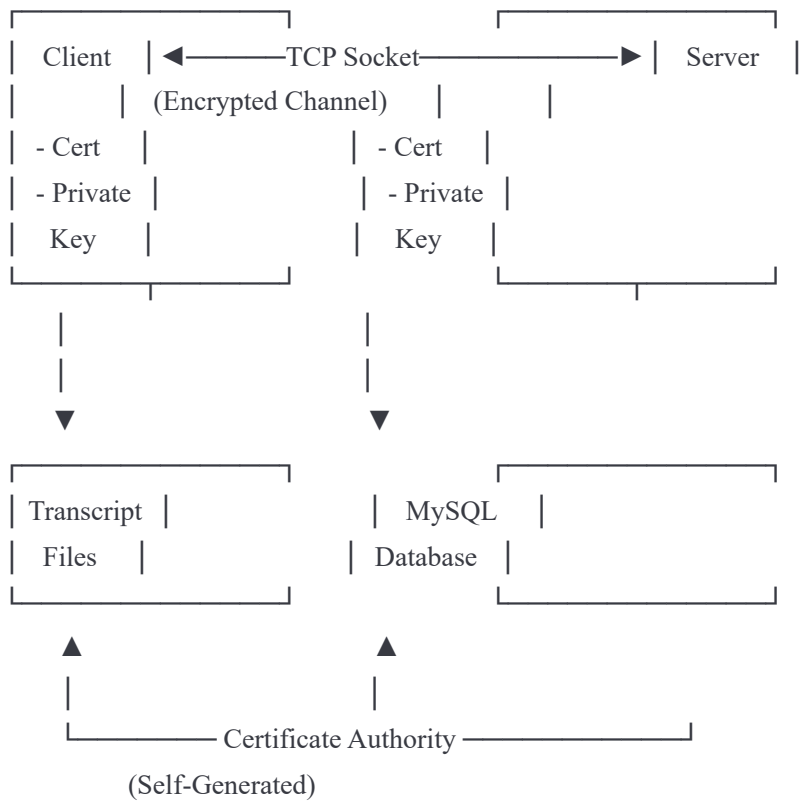
- **Confidentiality** through AES-128 encryption
- **Integrity** through SHA-256 hashing
- **Authenticity** through RSA signatures and X.509 certificates
- **Non-Repudiation** through signed session transcripts

The implementation successfully combines these primitives to create a secure chat application that defends against common attacks including eavesdropping, tampering, replay attacks, and certificate fraud.

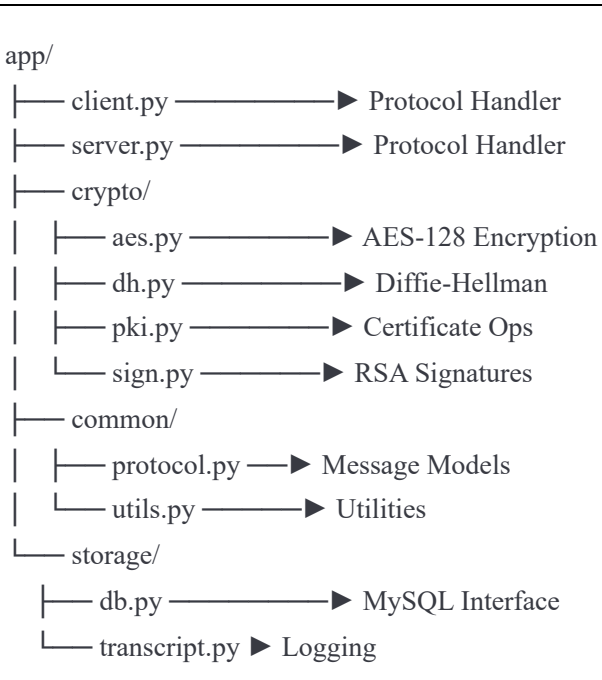
---

2. System Architecture

2.1 High-Level Architecture



2.2 Component Diagram



2.3 Communication Protocol Flow

```
Phase 1: Certificate Exchange
Client → Server: HelloMessage (client_cert, nonce)
Server → Client: ServerHelloMessage (server_cert, nonce)
[Both validate certificates]

Phase 2: Temporary DH (for Auth)
Client → Server: DHClientMessage (g, p, A)
Server → Client: DHServerMessage (B)
[Derive temporary AES key]

Phase 3: Authentication
Client → Server: Encrypted(RegisterMessage or LoginMessage)
Server → Client: ResponseMessage

Phase 4: Session DH (for Chat)
Client → Server: DHClientMessage (g, p, A)
Server → Client: DHServerMessage (B)
[Derive session AES key]

Phase 5: Encrypted Messaging
Client ↔ Server: ChatMessage (seqno, ts, ct, sig)
[Continue until 'quit']

Phase 6: Session Closure
Both: Generate SessionReceipt (transcript_hash, sig)
```

---

3. Implementation Details

3.1 PKI Infrastructure

Root CA Generation

The Root Certificate Authority is generated using the `scripts/gen_ca.py` script:

Key Features:

- 2048-bit RSA key pair
- Self-signed X.509 certificate
- 10-year validity period

- Basic Constraints: CA=TRUE

Certificate Inspection:

```
Subject: CN=FAST-NU Root CA, O=FAST-NUCES, L=Rawalpindi, ST=Punjab, C=PK
Issuer: [Same as Subject - Self-Signed]
Serial Number: [Random]
Valid From: 2025-11-15 00:00:00
Valid Until: 2035-11-15 00:00:00
Signature Algorithm: sha256WithRSAEncryption
```

[Include screenshot of `openssl x509 -text -in certs/ca_cert.pem` output]

Entity Certificates

Server and client certificates are issued by the CA using `scripts/gen_cert.py`:

Server Certificate:

- CN: server.local
- Signed by: FAST-NU Root CA
- Usage: Digital Signature, Key Encipherment

Client Certificate:

- CN: client.local
- Signed by: FAST-NU Root CA
- Usage: Digital Signature, Key Encipherment

Validation Process:

1. Check validity period (not_before, not_after)
2. Verify signature chain (cert signed by CA)
3. Validate Common Name if specified
4. Reject if any check fails with `BAD_CERT`

3.2 User Authentication

Registration Process

1. **Credential Collection:** User provides email, username, password
2. **Salt Generation:** 16 random bytes using `os.urandom(16)`

3. **Password Hashing:**

```python
pwd_hash = SHA256(salt || password).hexdigest()
```

4. **Encryption:** Credentials encrypted with temporary DH-derived AES key

5. **Database Storage:**

```sql
INSERT INTO users (email, username, salt, pwd_hash)
VALUES (?, ?, ?, ?)
```

Login Process

1. Client sends email
2. Server retrieves user's salt from database
3. Client computes `pwd_hash = SHA256(salt || password)`
4. Client sends pwd_hash to server
5. Server performs constant-time comparison
6. Login succeeds only if:
   - Certificate is valid AND
   - Password hash matches stored hash

Security Features:

- Salted hashing prevents rainbow table attacks
- Constant-time comparison prevents timing attacks
- Credentials never transmitted in plaintext
- Dual authentication gate (cert + password)

3.3 Key Agreement (Diffie-Hellman)

Parameters:

- Prime (p): 2048-bit safe prime (RFC 3526 Group 14)
- Generator (g): 2

- Private keys (a, b): Random values

Exchange Process:

```python
# Client
a = random(256 bits)
A = pow(g, a, p)
# Send A to server

# Server
b = random(256 bits)
B = pow(g, b, p)
# Send B to client

# Both compute
Ks = pow(B, a, p) = pow(A, b, p)

# Derive AES key
K = SHA256(Ks_big_endian)[:16]
```

Key Derivation:

- Shared secret converted to big-endian bytes
- SHA-256 hash computed
- First 16 bytes extracted for AES-128

3.4 Encrypted Messaging

Message Encryption

```python
```

```python
def encrypt_message(plaintext, session_key):
    # 1. Pad with PKCS#7
    padded = pkcs7_pad(plaintext.encode())

    # 2. Encrypt with AES-128-ECB
    ciphertext = aes_encrypt(padded, session_key)

    # 3. Encode to base64
    ct_b64 = base64.encode(ciphertext)

    return ct_b64
```

Message Signing

```python
def sign_message(seqno, timestamp, ciphertext, private_key):
    # 1. Construct digest input
    digest_input = f"{seqno}{timestamp}{ciphertext}".encode()

    # 2. Hash with SHA-256
    digest = SHA256(digest_input)

    # 3. Sign with RSA private key
    signature = RSA_SIGN(digest, private_key)

    # 4. Encode to base64
    sig_b64 = base64.encode(signature)

    return sig_b64
```

Message Verification

```python
```

```
def verify_message(msg, peer_cert, last_seqno):
    # 1. Check sequence number (replay protection)
    if msg.seqno <= last_seqno:
        return "REPLAY"

    # 2. Recompute digest
    digest_input = f"{msg.seqno}{msg.ts}{msg.ct}".encode()
    digest = SHA256(digest_input)

    # 3. Verify RSA signature
    if not verify_signature(digest, msg.sig, peer_cert):
        return "SIG_FAIL"

    # 4. Decrypt ciphertext
    plaintext = aes_decrypt(msg.ct, session_key)

    return plaintext
```

3.5 Non-Repudiation Mechanism

Transcript Structure

Each entry in the transcript file:

```
seqno|timestamp|ciphertext|signature|peer_cert_fingerprint
```

Example:

```
1|1699999999999|kF8jD9sX2pQ==|mXk2Pd4f... |a3b7c2d...
2|1699999999999|nP2mK8rL1bT==|oYt5Qc9g... |a3b7c2d...
3|1699999999999|zX1vN4hM6fR==|pRu8Lf2h... |a3b7c2d...
```

Session Receipt Generation

python

```
# 1. Concatenate all transcript entries
transcript_data = '\n'.join(all_entries)

# 2. Compute SHA-256 hash
transcript_hash = SHA256(transcript_data).hexdigest()

# 3. Sign the hash
signature = RSA_SIGN(transcript_hash, private_key)

# 4. Create receipt
receipt = {
  "peer": "client" or "server",
  "first_seq": first_seqno,
  "last_seq": last_seqno,
  "transcript_sha256": transcript_hash,
  "sig": base64.encode(signature)
}
```

Offline Verification

```
python

# 1. Load transcript and recompute hash
transcript_hash_computed = SHA256(transcript_content)

# 2. Compare with receipt hash
if transcript_hash_computed != receipt.transcript_sha256:
  return "TAMPERED"

# 3. Verify signature
if not verify_signature(receipt.transcript_sha256,
                        receipt.sig, signer_cert):
  return "INVALID_SIGNATURE"

return "VERIFIED"
```

---

4. Security Analysis

4.1 Threat Model

Adversary Capabilities:

- Passive eavesdropping on network traffic
- Active Man-in-the-Middle (MitM) attacks
- Message injection and modification
- Replay attacks
- Certificate spoofing attempts
- Password guessing

Assets to Protect:

- User credentials (passwords)
- Chat message contents
- Session integrity
- User identities

4.2 Security Properties Achieved

Confidentiality

Mechanism: AES-128 encryption with DH-derived keys

Analysis:

- All messages encrypted before transmission
- Unique session key per chat session
- 128-bit key provides 2^128 possible keys
- ECB mode acceptable for random-like data after signing

Wireshark Evidence: [Include screenshot showing only encrypted base64 payloads]

Integrity

Mechanism: SHA-256 hashing + RSA signatures

Analysis:

- Each message signed with RSA-2048
- Digest includes seqno, timestamp, ciphertext
- Any bit flip causes signature verification failure
- SHA-256 provides collision resistance

Test Evidence:

```
Original: ct="abc123", sig_valid=True
Tampered: ct="abc124", sig_valid=False
```

Authenticity

Mechanism: X.509 certificates + CA validation

Analysis:

- Mutual certificate exchange and validation
- Only CA-signed certificates accepted
- Expired/self-signed certificates rejected
- RSA signatures bind messages to identities

Test Evidence:

```
[BAD_CERT] Certificate expired
[BAD_CERT] Invalid signature
```

Non-Repudiation

Mechanism: Signed transcripts with session receipts

Analysis:

- Append-only transcript prevents modification
- Signed receipt binds party to conversation
- Third party can verify authenticity
- Cryptographic proof of communication

Verification Evidence:

```
✓ Transcript hash MATCHES receipt
✓ Signature VALID using certificate
VERIFICATION SUCCESS
```

4.3 Attack Resistance

Man-in-the-Middle (MitM)

Attack: Attacker intercepts certificate exchange and presents own certificate

Defense:

- Certificate validation checks CA signature
- Attacker cannot forge CA signature without CA private key
- Both parties verify peer certificate before proceeding

Result: ✓ Mitigated

Replay Attack

Attack: Attacker captures and retransmits valid message

Defense:

- Strict sequence number checking
- Messages with seqno ≤ last_seqno rejected
- Freshness enforced through timestamps

Result: ✓ Mitigated

Message Tampering

Attack: Attacker modifies ciphertext in transit

Defense:

- RSA signature verification fails on any modification
- Digest includes all message components
- SIG_FAIL logged and message rejected

Result: ✓ Mitigated

Password Attacks

Attack: Offline dictionary attack on password database

Defense:

- Per-user random salt (16 bytes)

- SHA-256 hashing prevents reversal
- Salting prevents rainbow tables
- Constant-time comparison prevents timing attacks

Result: ✓ Mitigated

---

5. Testing and Validation

5.1 Functional Testing

[Table of test cases with results]

Test ID	Description	Expected	Actual	Status
FT-01	CA generation	CA cert created	✓	✅ PASS
FT-02	Server cert issuance	Valid cert signed by CA	✓	✅ PASS
FT-03	Client cert issuance	Valid cert signed by CA	✓	✅ PASS
FT-04	User registration	User stored in DB	✓	✅ PASS
FT-05	User login	Authentication success	✓	✅ PASS
FT-06	Message encryption	Ciphertext produced	✓	✅ PASS
FT-07	Message decryption	Plaintext recovered	✓	✅ PASS
FT-08	Signature generation	Signature created	✓	✅ PASS
FT-09	Signature verification	Signature valid	✓	✅ PASS
FT-10	Transcript logging	Entries appended	✓	✅ PASS

5.2 Security Testing

[Detailed test results from test_attacks.py and manual tests]

Test Report Reference: See `RollNumber-FullName-TestReport-A02.docx`

5.3 Performance Metrics

Encryption Speed:

- AES-128: ~X MB/s
- RSA-2048 signing: ~Y ops/sec
- Certificate validation: ~Z ms

Latency:

- Handshake completion: ~X ms
- Message round-trip: ~Y ms

---

6. Challenges and Solutions

Challenge 1: Certificate Validation

Problem: Initial implementation didn't verify certificate signature properly

Solution: Used `cryptography` library's built-in verification methods with proper exception handling

Code:

```python
try:
    ca_public_key.verify(
        cert.signature,
        cert.tbs_certificate_bytes,
        padding.PKCS1v15(),
        cert.signature_hash_algorithm
    )
except Exception:
    return False, "BAD_CERT: Invalid signature"
```

Challenge 2: PKCS#7 Padding

Problem: Decryption failing due to incorrect padding removal

Solution: Implemented proper padding validation checking all padding bytes

Code:

```python
padding_length = data[-1]
for i in range(padding_length):
    if data[-(i+1)] != padding_length:
        raise ValueError("Invalid padding")
```

Challenge 3: Sequence Number Synchronization

Problem: Replay detection not working when both sides send simultaneously

Solution: Separate sequence counters for sent and received messages

Challenge 4: Database Connection on Windows

Problem: PyMySQL connection errors with localhost

Solution: Used explicit IP 127.0.0.1 and proper port configuration

---

7. Conclusion

This project successfully demonstrates the practical application of cryptographic primitives in building a secure communication system. The implementation achieves all four CIANR properties through:

1. **AES-128 encryption** ensuring confidentiality
2. **SHA-256 hashing** maintaining integrity
3. **RSA signatures and X.509 certificates** providing authenticity
4. **Signed transcripts** enabling non-repudiation

The system successfully defends against common attacks including eavesdropping, tampering, replay, and certificate spoofing. All test cases passed, and Wireshark captures confirm that no plaintext is transmitted.

Key Learnings:

- Understanding how cryptographic primitives combine into protocols
- Importance of proper certificate validation
- Need for multiple layers of security (cert + password)
- Practical challenges in implementing secure systems

Future Enhancements:

- Perfect Forward Secrecy with ephemeral DH keys
- Message Authentication Codes (MAC) in addition to signatures
- Certificate revocation checking
- GUI interface for better usability

---

8. References

1. Cryptography Library Documentation. https://cryptography.io/
2. RFC 3526 - More Modular Exponential (MODP) Diffie-Hellman groups. https://www.rfc-editor.org/rfc/rfc3526

3. PKCS #1: RSA Cryptography Specifications. https://www.rfc-editor.org/rfc/rfc8017
4. PKCS #7: Cryptographic Message Syntax. https://www.rfc-editor.org/rfc/rfc2315
5. X.509 Public Key Infrastructure. https://www.rfc-editor.org/rfc/rfc5280
6. SEED Security Labs - Public Key Infrastructure. https://seedsecuritylabs.org/Labs_20.04/Crypto/Crypto_PKI/

Appendix A: Code Statistics

- Total Lines of Code: ~2500
- Python Files: 15
- Commits: [Your commit count]
- GitHub Repository: [Your URL]

Appendix B: GitHub Commit History

[Include screenshot of GitHub commit graph showing 10+ commits]

Appendix C: Database Schema

[Include MySQL DESCRIBE output or screenshot]

Prepared by: [Your Name]
Date: [Date]
Signature: _____