

**Design and Development of a Secure End to End Encrypted Messaging Application**



Nikesh Uprety (220026, 12981377)

BSc. (Hons.) Ethical Hacking and Cybersecurity, Softwarica college of IT and E-commerce, Coventry University

ST6051CEM Practical Cryptography

Mr. Arbind Shakya

July\*\*\*\*\*\*\*\*\*\*, 2025

# Introduction

The increasing sophistication of cyber threats and the growing reliance on digital communication platforms have created an urgent need for robust, cryptographically secure messaging systems. Modern communication applications must provide not only confidentiality but also integrity, authenticity, and non-repudiation for sensitive information exchange. This technical report documents the comprehensive design, development, and security analysis of a secure end-to-end encrypted messaging application that leverages proven cryptographic techniques to address contemporary security challenges.

The application implements a **hybrid encryption architecture** combining AES-256 symmetric encryption with RSA-2048 asymmetric cryptography, complemented by digital signatures and secure key management protocols. This approach ensures both computational efficiency and cryptographic strength while maintaining usability for end users.

## Aim

To design and develop a cryptographically secure messaging application that provides end-to-end encryption, message integrity verification, and secure user authentication while maintaining practical performance characteristics for real-world deployment.

## Objectives

1. **Implement hybrid encryption** using AES-256 for message encryption and RSA-2048 for key exchange.
2. **Establish secure authentication** using bcrypt password hashing and session management.
3. **Ensure message integrity** through digital signature implementation.
4. **Provide secure transport** using HTTPS with proper security headers.
5. **Mitigate common web vulnerabilities** including XSS, CSRF, and injection attacks.

# Overview of Secure Messaging in the Modern Era

Contemporary secure messaging applications must address multiple threat vectors while maintaining user experience and performance. The proliferation of surveillance technologies and data breaches has heightened the importance of **end-to-end encryption** (E2EE), where messages are encrypted on the sender's device and remain encrypted until decrypted by the intended recipient. This approach ensures that even service providers cannot access message contents, providing a crucial layer of privacy protection.

The evolution of messaging security has been driven by several factors: increasing regulatory requirements (GDPR, HIPAA), growing awareness of privacy rights, and the need for secure business communications. Modern implementations must balance security with usability, ensuring that cryptographic complexity remains transparent to end users while providing robust protection against sophisticated attacks.

# Problem and Proposed Solution

## Identified Security Gaps in Existing Systems

Analysis of current messaging platforms reveals several critical vulnerabilities:

1. **Inadequate Key Management**: Many systems lack proper key lifecycle management, leading to long-term key exposure risks.
2. **Weak Authentication Mechanisms**: Reliance on simple password-based authentication without proper hashing or multi-factor authentication.
3. **Transport Layer Vulnerabilities**: Insufficient HTTPS implementation and missing security headers.
4. **Metadata Leakage**: Exposure of communication patterns and user relationships even when message content is encrypted.
5. **Lack of Perfect Forward Secrecy**: Compromise of long-term keys can expose historical communications.

## Threat Models Considered

The application addresses several critical threat scenarios:

**Man-in-the-Middle (MITM) Attacks**: Mitigated through certificate pinning, HTTPS enforcement, and public key verification mechanisms.

**Replay Attacks**: Prevented using timestamp validation, nonce implementation, and session-based authentication tokens.

**Phishing Attacks**: Addressed through Content Security Policy (CSP) implementation, secure cookie configuration, and user education mechanisms.

**Database Compromise**: Mitigated through bcrypt password hashing, encrypted key storage, and minimal data retention policies.

**Cross-Site Scripting (XSS)**: Prevented through input sanitization, output encoding, and strict CSP implementation.

**Cross-Site Request Forgery (CSRF)**: Mitigated using synchronizer tokens and SameSite cookie attributes.

## Proposed Cryptography-Driven Solution

The application implements a multi-layered security architecture combining:

1. **Hybrid Encryption**: AES-256-CBC for message encryption with RSA-2048-OAEP for key exchange.
2. **Digital Signatures**: RSA-2048 signatures for message authenticity and non-repudiation.
3. **Secure Key Management**: Automated key generation, secure storage, and lifecycle management.
4. **Transport Security**: HTTPS with Flask-Talisman security headers and mkcert for development.
5. **Authentication**: bcrypt-hashed passwords with secure session management.

# Scope and Use Case

## Application Domain

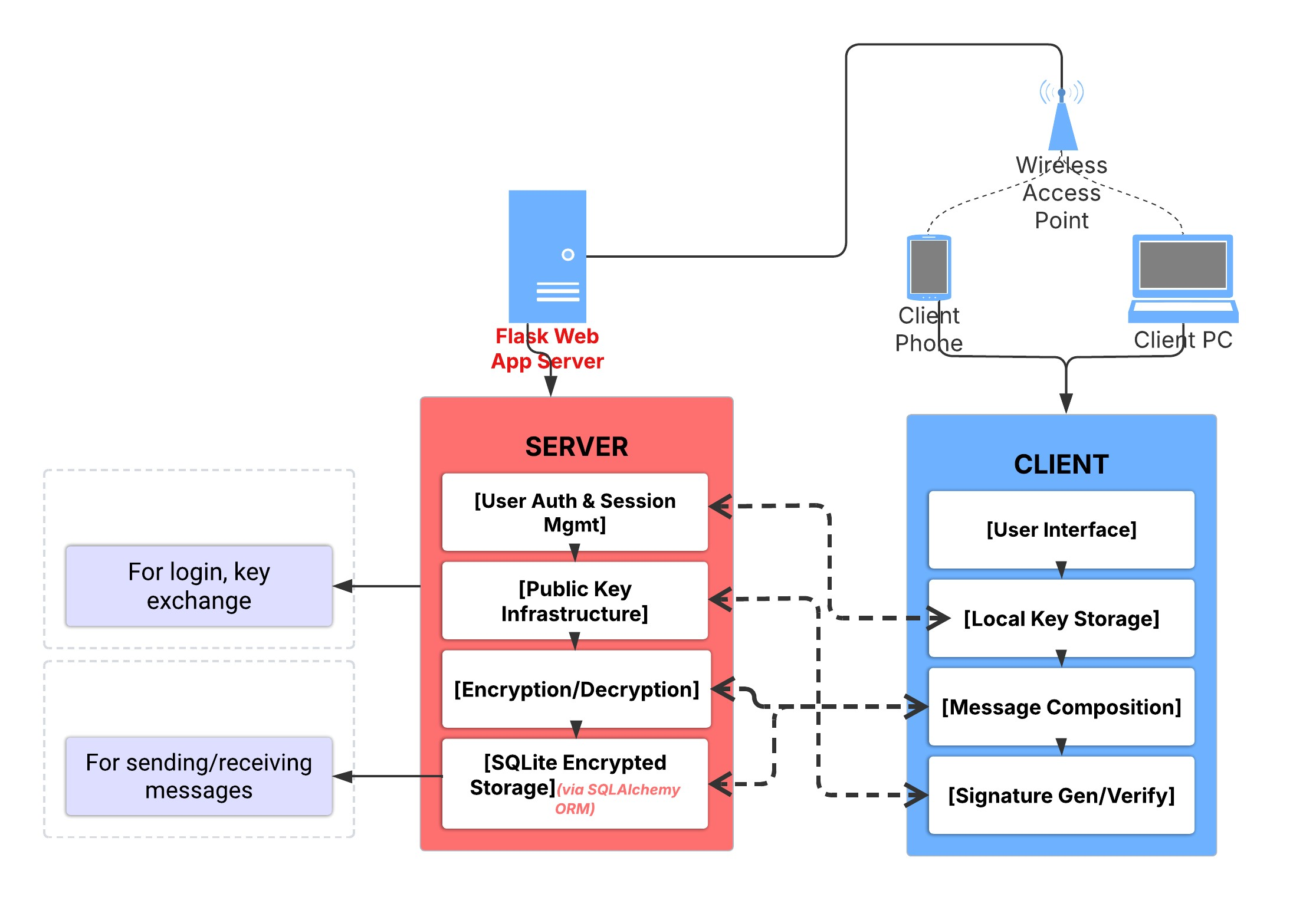
The application targets private messaging and secure file sharing scenarios where confidentiality, integrity, and authenticity are paramount. Primary use cases include:

* Healthcare Communications: HIPAA-compliant messaging for medical professionals.
* Legal Communications: Attorney-client privileged communications.
* Business Intelligence: Secure exchange of sensitive corporate information.
* Government Communications: Classified or sensitive government communications.
* Personal Privacy: Individual users requiring enhanced privacy protection.

## Real-world Scenarios

1. Medical Consultation: A physician securely sharing patient information with a specialist.
2. Legal Discovery: Attorneys exchanging confidential case documents.
3. Corporate Mergers: Executives discussing sensitive acquisition details.
4. Journalism: Secure communication between journalists and sources.
5. Personal Communications: Individuals requiring privacy from surveillance.

# System Architecture



*Fig: System Architecture Diagram*

**Client-Server Model**

The application employs a client-server architecture with the following components:

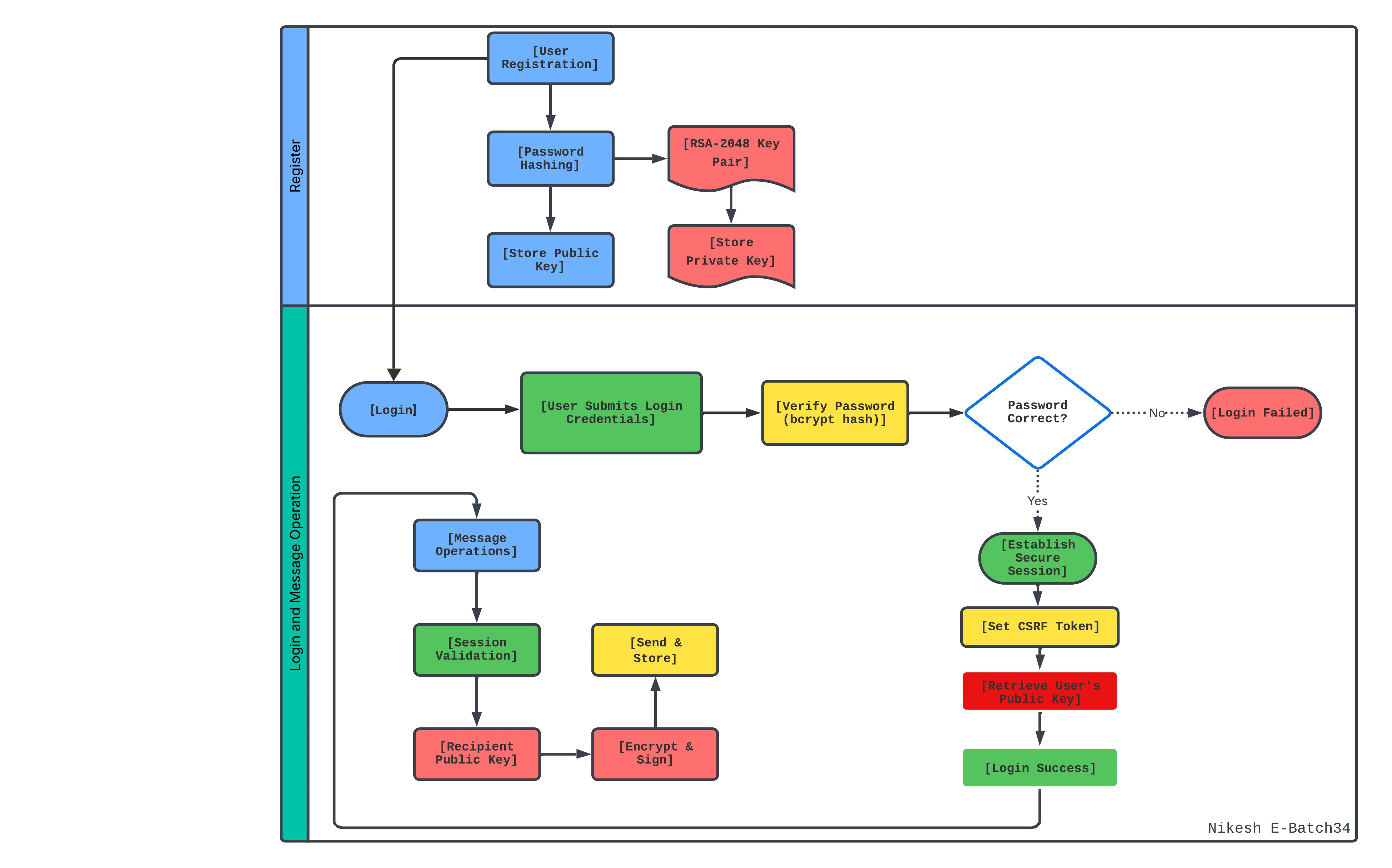
**Client Components**:

* Web-based user interface with JavaScript encryption/decryption
* Local key storage and management
* Message composition and display
* Signature generation and verification

**Server Components**:

* Flask-based web application server
* SQLite database for encrypted message storage
* User authentication and session management
* Public key infrastructure management

## Authentication and Authorization Flow



*Fig: Authentication and Authorization Flowchart Diagram*

**User Registration and Login**: Users register with passwords hashed using bcrypt (salt rounds ≥ 12) and generate an RSA-2048 key pair. The public key is stored on the server, while the private key is securely kept on the client device. During login, credentials are verified against the bcrypt hash, a secure session with CSRF protection is established, and the public key is retrieved for message operations.

**Message Operations**: The sender’s session is validated for authentication. The recipient’s public key is retrieved, and the message is encrypted and signed. The encrypted message is then securely transmitted and stored.

* **Message Sending Flow**:

User → Compose Message → Generate AES Key → Encrypt Message →   
Sign Message → Encrypt AES Key with Recipient's RSA Public Key →   
Transmit Encrypted Package → Store in Database

* **Message Receiving Flow**:

Database → Retrieve Encrypted Package → Decrypt AES Key with   
Private RSA Key → Decrypt Message → Verify Signature →   
Display Message to User

## Encryption and Decryption Mechanism

1. **AES-256 Symmetric Encryption for Messages**

The application implements AES-256 in CBC mode for message encryption, providing:

* Key Length: 256 bits (32 bytes) for maximum security.
* Block Size: 128 bits (16 bytes) as per AES standard.
* Initialization Vector: 128-bit random IV for each message.
* Padding: PKCS7 padding for variable-length messages.

**Encryption Process**:

The application employs AES-256 in CBC mode for secure message encryption, using a 256-bit (32-byte) key to ensure robust security and a 128-bit (16-byte) block size as defined by the AES standard. Each message is encrypted with a unique 128-bit random initialization vector (IV) to enhance security by preventing predictable patterns. PKCS7 padding is applied to accommodate messages of varying lengths, ensuring proper encryption and decryption.

**Performance Characteristics**: AES-256 provides excellent performance with hardware acceleration, typically achieving 15-20 times faster encryption compared to software-only implementations. Modern processors with AES-NI instructions can encrypt data at rates exceeding 1 GB/s.

1. **RSA 2048-bit Key Pair Generation for Each User**

Each user receives a unique RSA-2048 key pair providing:

* Key Size: 2048 bits minimum for current security standards.
* Security Level: Equivalent to 112-bit symmetric key strength
* Algorithm: RSA with OAEP padding for encryption, PSS for signatures.

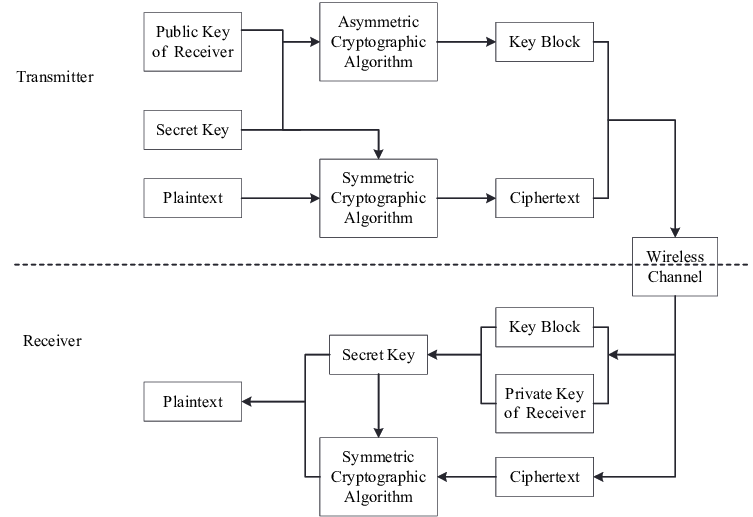
**Key Generation Process**

The key generation process involves creating two large prime numbers, p and q, and computing the modulus n by multiplying them. A public exponent e, typically set to 65537, is chosen, and the private exponent d is calculated as the modular inverse of e with respect to φ(n). The public key, consisting of (n, e), is stored on the server, while the private key, (n, d), is securely stored on the client device.

Security Considerations: RSA-2048 provides adequate security against current cryptanalytic attacks and is expected to remain secure until at least 2030 according to NIST recommendations.

## Hybrid Encryption Workflow

The application implements a hybrid encryption scheme combining the efficiency of symmetric encryption with the security of asymmetric cryptography:



*Fig: Flow chart of hybrid cryptosystem*

The message encryption process begins by generating a unique AES-256 key for each message, which is used to encrypt the message content in AES-256-CBC mode. The AES key is then encrypted using the recipient’s RSA-2048 public key with OAEP padding, ensuring that only the intended recipient can access the key. This hybrid encryption approach combines the efficiency of symmetric AES encryption for the message body with the security of asymmetric RSA encryption for key protection.

For **key exchange**, the encrypted AES key is securely distributed, leveraging RSA to ensure that only the recipient with the corresponding private key can decrypt it. Each message employs a distinct AES key, aligning with the concept of perfect forward secrecy to enhance security. By using AES for bulk data encryption and limiting RSA to encrypting the small 32-byte AES key, this hybrid system optimizes performance while maintaining robust security.

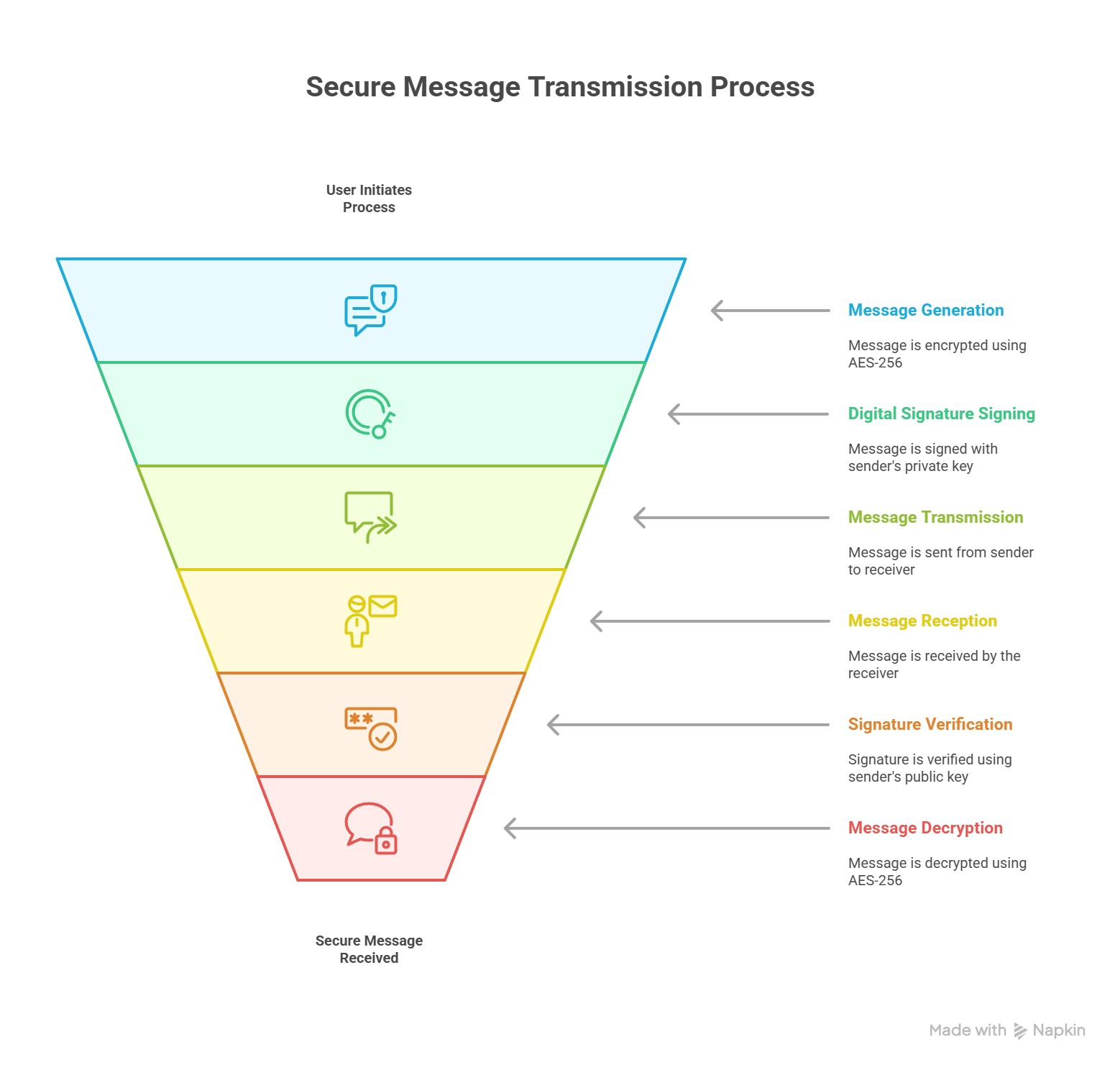
# Functionality

The application is designed with a comprehensive set of features to ensure both usability and robust security. Each functionality is carefully implemented to address specific user needs while providing strong protection against modern threats. The table below summarizes the core and advanced features, along with their security benefits.

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Functionality** | **Description** | **Security Benefit** |
| 1 | End-to-End Encryption | Encrypts messages from sender to recipient | Confidentiality, Integrity |
| 2 | Digital Signatures | Signs messages/files with sender’s private key | Authenticity, Non-repudiation |
| 3 | Secure Authentication | Bcrypt-hashed passwords, optional MFA | Access Control, Brute-force Resistence. |
| 4 | Password Policy Enforcement | Enforces minimum length (≥8), requires numbers, uppercase, and special characters | Stronger Passwords, Automated Attack Prevention, Reduced Credential Risk |
| 5 | Secure File Sharing | Encrypted file transfer and signature validation | Data Protection, Malware Defense |
| 6 | Chat Initialization & Approval | Users must send and accept chat requests before messaging begins | User Consent, Spam Prevention, Enhanced Privacy |
| 7 | Asynchronous AJAX Messaging | Handles message sending and receiving via AJAX without page reloads | Improved User Experience, CSRF/XSS Mitigation, Reduced Attack Surface |

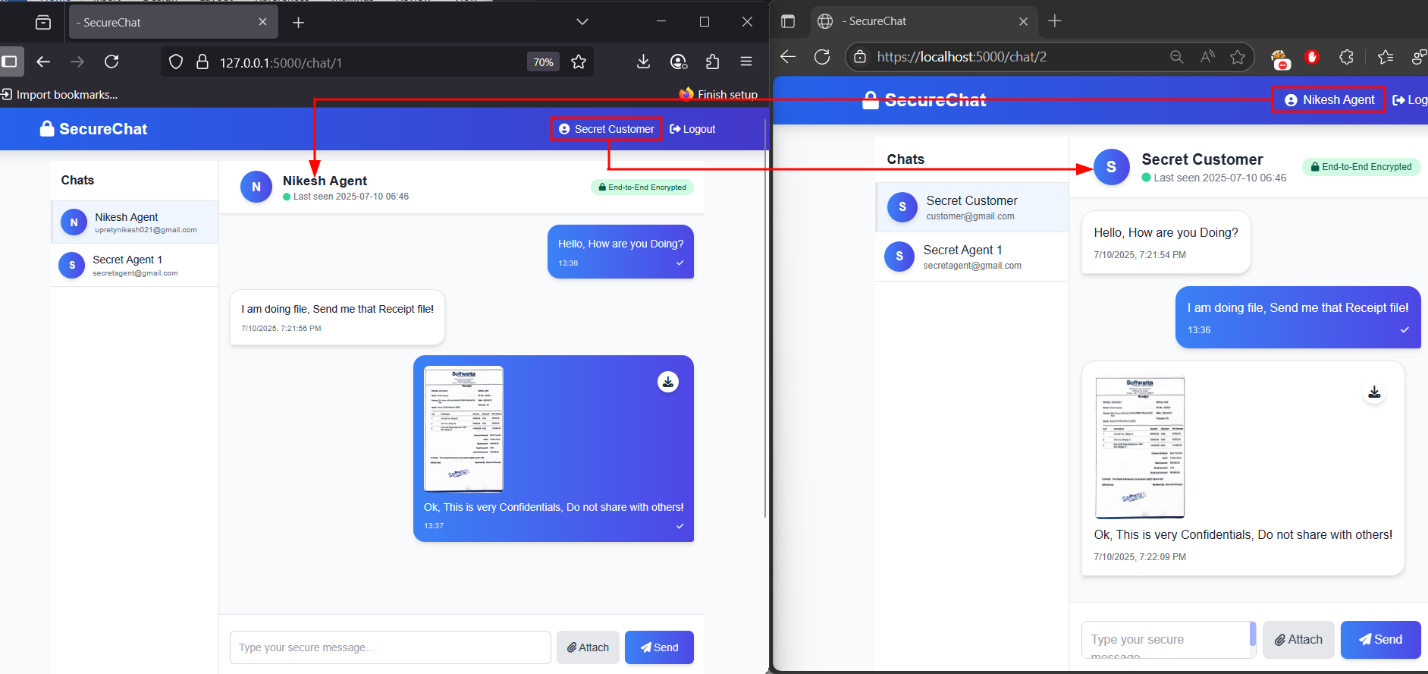
## Individual Features

1. End-to-End Encryption and Digital Signatures:

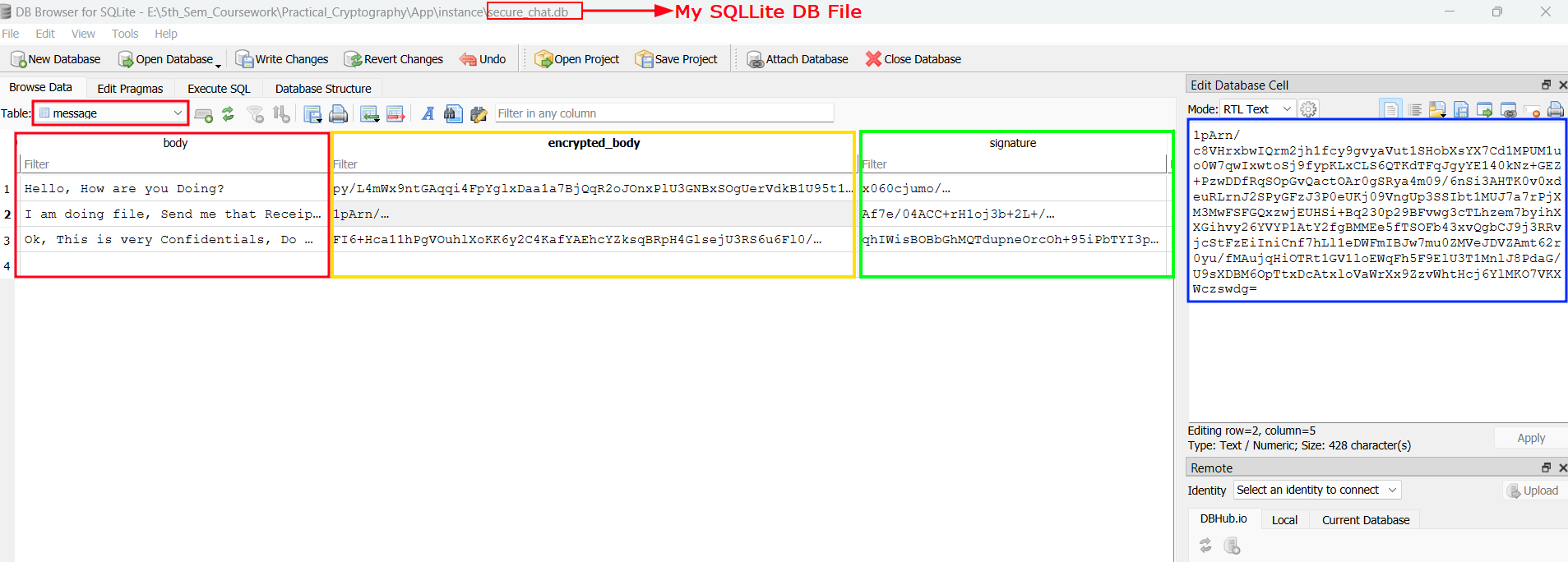


*Fig: Diagram illustrating the end-to-end encryption and digital signature verification process.*

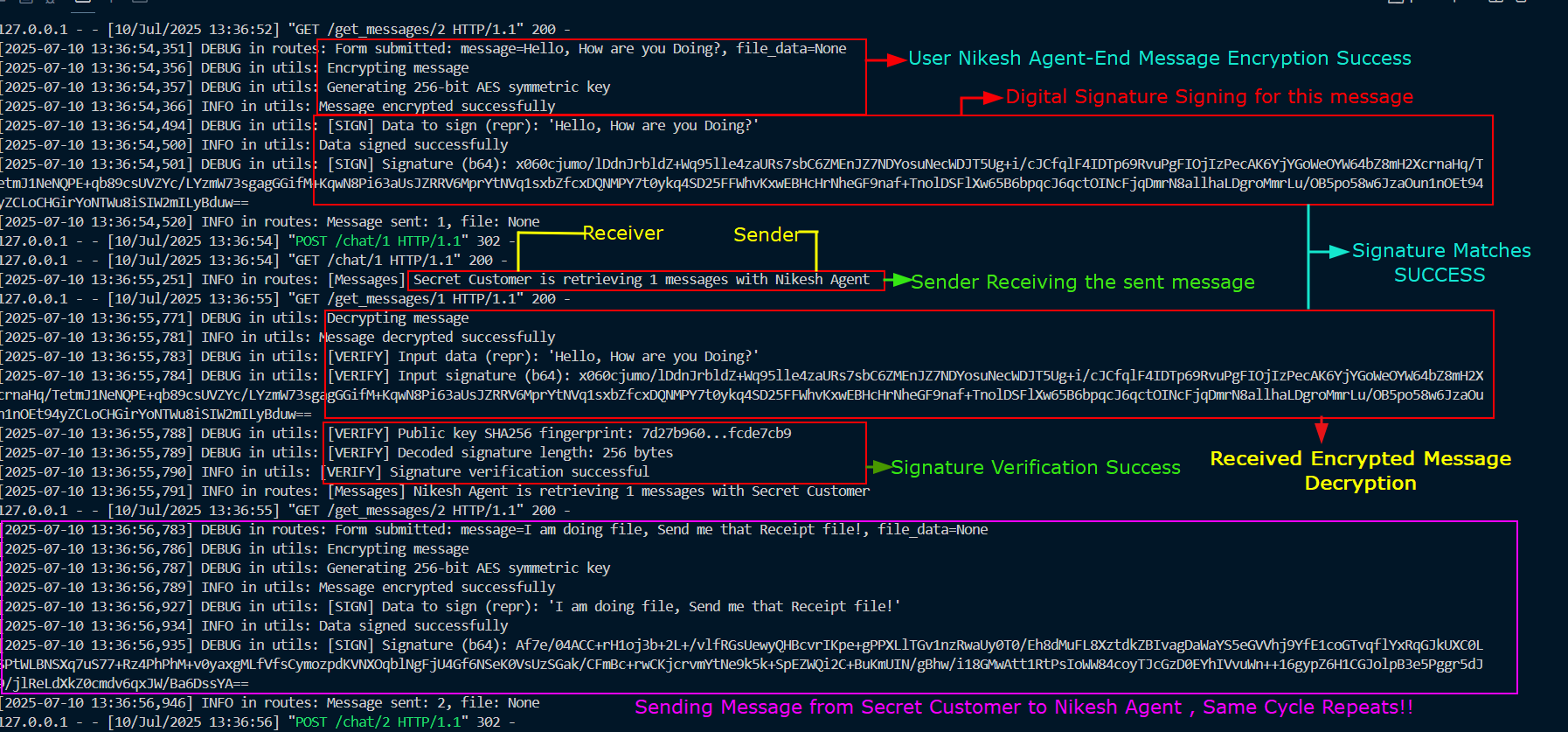
Secure end-to-end encryption is a critical feature in modern messaging applications. It ensures that only the intended sender and recipient can access the message content, protecting user privacy and data integrity.



*Fig: Message Sending and Receiving Process.*

**

*Fig: Visualizing Message Table with each message encryption and signature.*



*Fig: Server Log of actual End-End Encryption and Signature Verification process.*

In my secure messaging app, all messages are protected using robust end-to-end encryption. Each message is encrypted with a unique AES-256 key, which is itself securely transmitted using RSA-2048 encryption. Digital signatures are applied to every message to verify the sender’s identity and detect any tampering. To further safeguard privacy, only essential metadata is stored, minimizing the risk of information leaks. The app uses a new encryption key for every message, ensuring that even if one key is compromised, previous messages remain secure (forward secrecy). Private keys are stored safely on the user’s device, while public keys are managed by the server for secure exchanges. This system guarantees that only the sender and recipient can read the messages, verifies authenticity, and maintains message integrity throughout the communication process.

1. Secure File Upload with Signature Validation

Ensuring secure file uploads is crucial for protecting user data and preventing unauthorized access. My application uses strong encryption and signature checks to keep shared files safe.

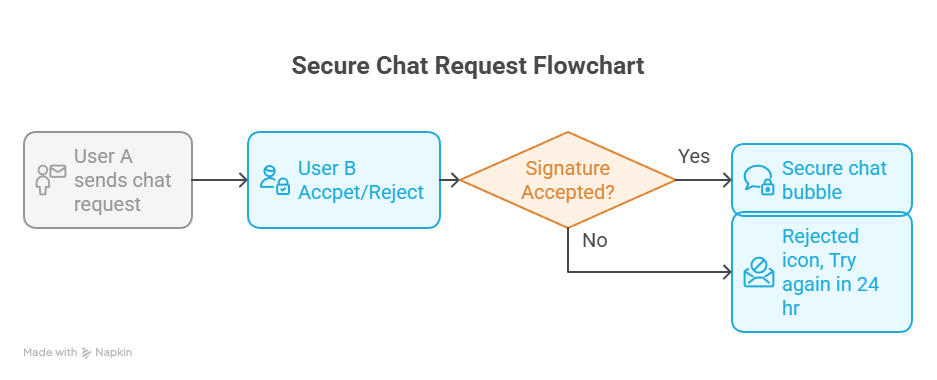
The file upload feature encrypts large files using AES-256 in streaming mode, and it verifies file types by checking their headers. Only approved file extensions are allowed, with strong error handling in place. Each file’s integrity and authenticity are confirmed through digital signatures and hash-based checks. Public key cryptography verifies the sender, while any tampered files are detected and rejected. This approach guarantees that only secure and verified files are shared within the app.

1. AJAX-based Asynchronous Communication

Asynchronous AJAX messaging allows users to send and receive messages instantly without reloading the page. This improves the overall user experience while maintaining strong security measures.

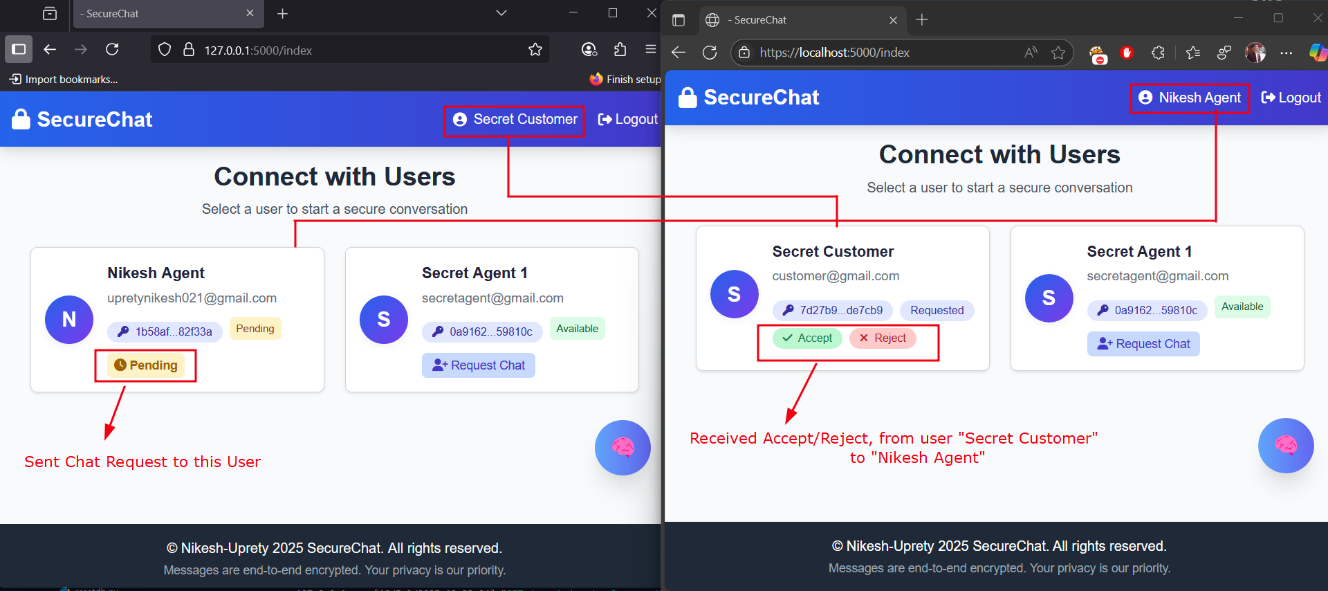
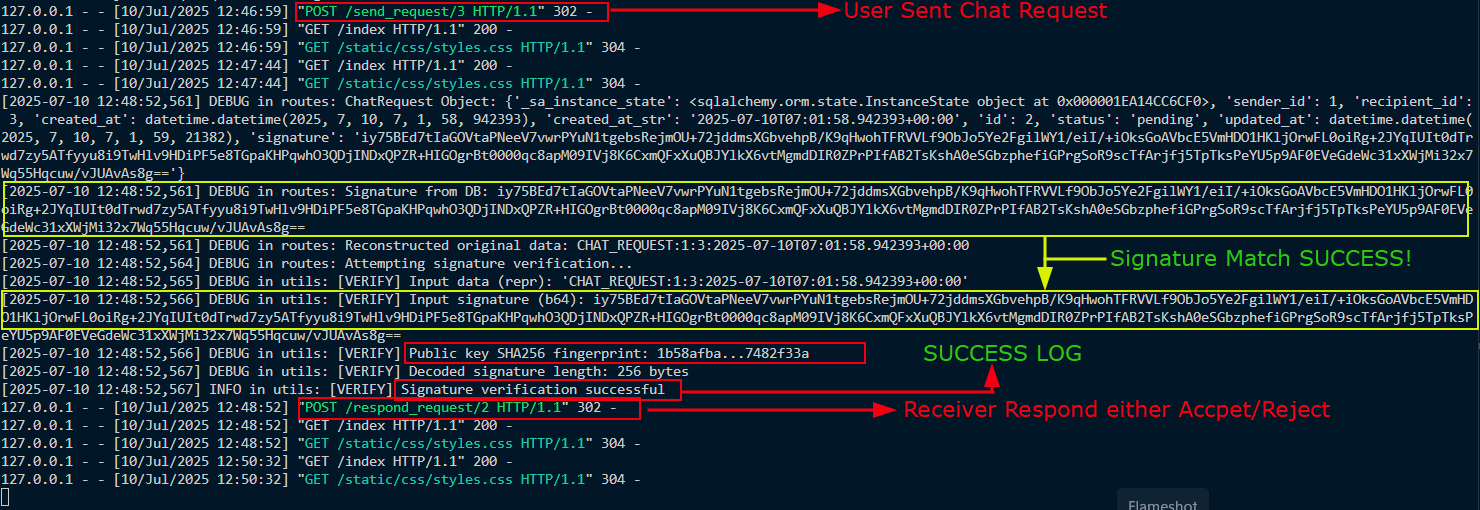
The application uses secure AJAX to handle messages smoothly and dynamically. Every AJAX request includes CSRF token validation, input sanitization, and strict content-type checks, along with rate limiting to prevent misuse. To guard against XSS attacks, the system enforces content security policies, validates all inputs, and avoids unsafe functions like innerHTML and eval(). Secure methods are used for parsing and handling JSON data. These steps ensure both a seamless user experience and robust protection against common web threats.

1. Chat Initialization & Approval



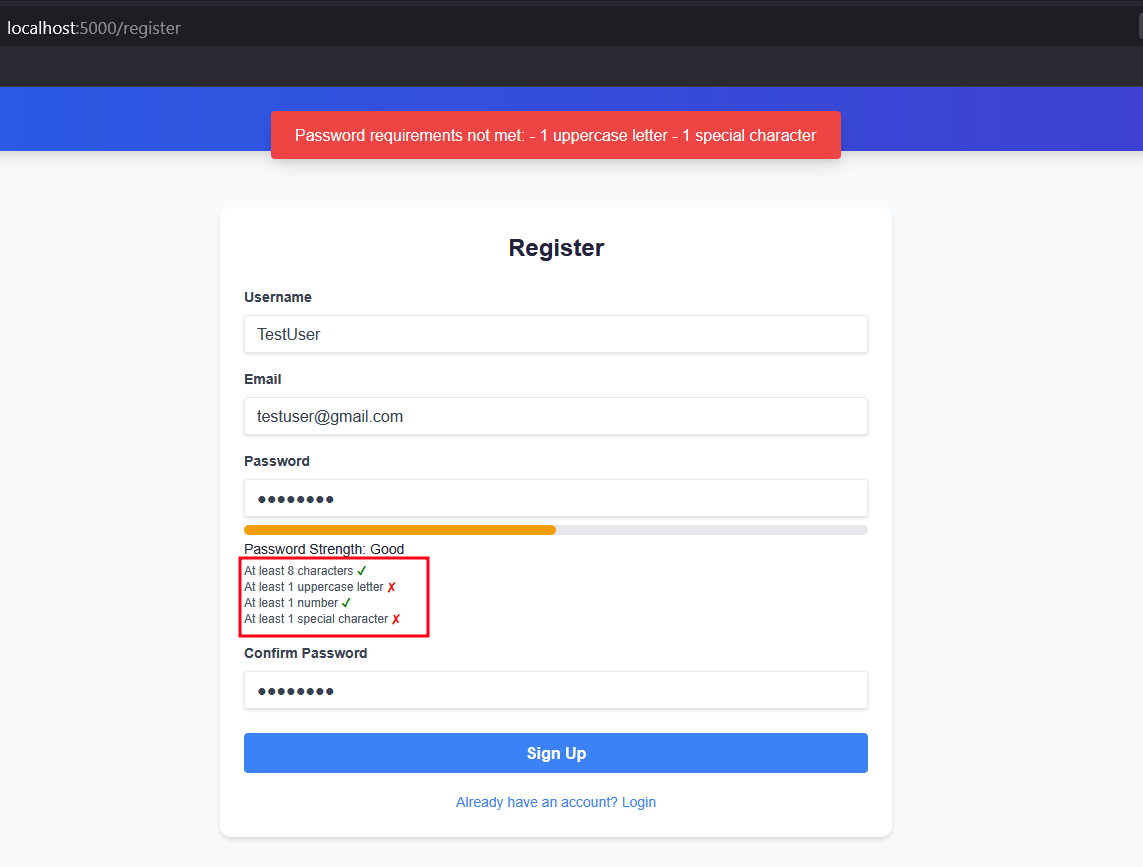
*Fig: Flowchart of Chat Initialization & Approval between User A & User B.*

The "Chat Initialization & Approval" feature ensures secure communication by requiring mutual consent via chat requests, validated through digital signature verification using Public Key Infrastructure (PKI). This enhances user control, privacy, and security while preventing unauthorized interactions.

The application requires users to send chat requests, which must be accepted or rejected by the recipient before messaging can begin. Digital signatures, verified using PKI and user public keys, ensure the authenticity of each request. This consent-driven process prevents spam, safeguards privacy, and ensures only trusted, verified connections can communicate, maintaining a secure and controlled environment.

1. Password Policy Enforcement



# Development Methodology

The project followed an **Agile development approach** structured over multiple sprints. Each sprint focused on a specific aspect of the application’s design, implementation, and security, following iterative development and continuous feedback principles.

The methodology emphasized regular testing, early integration of security features, and user-centric enhancements. Here's a breakdown of the sprint-wise implementation:

**Sprint 1: Foundation** (Weeks 1-2)

* Basic Flask application setup.
* User registration and authentication.
* Database schema design using SQLAlchemy.
* Initial security requirements gathering and threat identification

**Sprint 2: Cryptographic Core** (Weeks 3-4)

* RSA key pair generation.
* AES encryption implementation for secure message content.
* Digital signature functionality to ensure message integrity and authenticity.
* Hybrid encryption workflow combining RSA and AES for secure transmission.

**Sprint 3:** **Secure File Sharing + Signatures** (Week 4)

* File Uploading Core Function implementation.
* File Upload Sanitization with robust error handling.
* Digital file signature verification.
* Verified and Uploaded file download feature.

**Sprint 4: Security Hardening** (Weeks 5-6)

* CSRF protection implementation.
* Security headers configuration.
* Input validation and sanitization to prevent XSS and injection attacks.
* HTTPS setup with mkcert.

**Sprint 5: User Interface** (Weeks 7-8)

* Message composition and viewing interface.
* Encrypted message display.
* AI Chat Assistant with Gemini API.
* UI/UX enhancements for seamless user experience and error feedback

**Sprint 6: Testing and Deployment** (Weeks 9-10)

* Security testing and penetration testing.
* Performance optimization.
* Documentation completion.
* Deployment preparation (Dockerization, Gunicorn + Nginx setup).

# Cryptographic Evaluation and Justification

## Why AES-256 + RSA-2048?

## Deprecation of Outdated Standards

## Balancing Speed and Security

# Performance and Stress Testing

## Encryption/Decryption Latency

## Message Delivery Benchmarks

## Scalability Considerations

# Risk Management

## Mitigation Strategy Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Threat** | **Likelihood** | **Impact** | **Mitigation** | **Residual Risk** |
| Password Brute Force | Medium | High | Bcrypt + Rate Limiting | Low |
| MITM Attack | Low | High | HTTPS + Certificate Pinning | Very Low |
| Key Compromise | Low | Critical | Key Rotation + HSM | Low |
| Database Breach | Medium | High | Encryption at Rest | Medium |
| XSS Attack | Medium | Medium | CSP + Input Validation | Low |
| CSRF Attack | Medium | Medium | Synchronizer Tokens | Very Low |
| Replay Attack | Low | Medium | Timestamps + Nonces | Very Low |
| Phishing | High | Medium | User Education + Security Indicators | Medium |

# Limitations and Future Improvements

## What Could Be Done Better

Perfect Forward Secrecy Enhancement:

* Current: Unique AES keys per message provide some forward secrecy
* Improvement: Implement Diffie-Hellman key exchange for each session
* Benefit: Complete forward secrecy even if long-term RSA keys are compromised

WebAuthn Integration:

* Current: Password-based authentication with bcrypt
* Improvement: Add WebAuthn support for passwordless authentication
* Benefit: Stronger authentication resistance to phishing and credential theft

Hardware Security Module (HSM) Integration:

* Current: Software-based key storage and operations
* Improvement: HSM integration for production deployments
* Benefit: Hardware-level key protection and regulatory compliance

## Scalability and Mobile Platform Support

Horizontal Scaling Improvements:

* Microservices Architecture: Decompose monolithic application into specialized services
* Load Balancing: Implement sophisticated load balancing with session affinity
* Database Sharding: Distribute user data across multiple database instances
* Caching Layer: Implement Redis/Memcached for session and key caching

Mobile Platform Enhancements:

* Progressive Web App (PWA): Offline functionality and native app experience
* Mobile-Optimized Cryptography: Optimize for mobile CPU and battery constraints
* Push Notifications: Secure push notifications for real-time messaging
* Biometric Authentication: Integration with mobile biometric systems

## Logging, Monitoring, and Auditing Considerations

Security Logging:

* Authentication Events: Login attempts, failures, and suspicious activities
* Cryptographic Operations: Key generation, encryption/decryption events
* Access Control: Authorization decisions and policy violations
* System Events: Configuration changes and administrative actions

Monitoring and Alerting:

* Performance Metrics: Encryption/decryption latency and throughput
* Security Metrics: Failed authentication attempts and anomalous patterns
* System Health: Resource utilization and error rates
* Compliance Monitoring: Regulatory compliance and audit trail integrity

Audit Trail Requirements:

* Tamper-Evident Logging: Cryptographically signed log entries
* Long-Term Retention: Secure archival of audit logs
* Compliance Reporting: Automated generation of compliance reports
* Forensic Analysis: Tools for security incident investigation

# Conclusion