CPSC 452- 01 Final Project – Secure Chat

HavenChat

Professor:
Dr. Mikhail Gofman

Submitted by:
Josh Ibad & Winnie Pan
CWID: 888880036 & 888997434

Department of Computer Science and Engineering
California State University, Fullerton
Spring 2022

Table of Contents

Abstract

HavenChat is a Web Application that provides a Secure Live Chatting service. Users may register and befriend others with whom they may then chat with, through a custom message exchange protocol that provides Confidentiality, Authentication, Digital Signature & Integrity. Specifically, the chat is encrypted symmetrically using 256-bit AES in CBC mode, with the chat session key being distributed to each user using asymmetric encryption via RSA. Authentication is provided through the login system, with passwords hashed using BCrypt, and with a server signed JWT token using HS512. Finally, Digital Signature & Integrity is provided using the user's choice of either RSA or DSA signing, both of which are supported by the SHA256 hashing algorithm, with signature verification taking place on the client-side, to verify that the right person sent the message unmodified. The RSA/DSA public key for signatures are stored by the server for the duration of the session, and are distributed to all participants of the chat, acting as a Public Key Authority. All symmetric and asymmetric keys are temporary to a single chat session, which are cleared when all participants have left.

1.0 Introduction

In this project, we implemented a system which enables groups of users to chat securely. All users need to first register with the chat server, after which they may then log in. The user is provided general security and Authenticity using server signed JWT tokens storing the user's identity. After being authenticated, the server will change their status as "online". The user may then send a friend request to other registered users, as well as accept or deny pending friend requests. With their new-found friends, they will then see which of their friends are online in the Live Chat section, who they can invite as a participant in an upcoming chat. Users may not invite offline users into a chat, although the chat may have more than two participants. The user also selects a digital signature scheme, between RSA & DSA. Once the chat has been initiated, the server first generates a new symmetric key for the new chat session. All participants are then sent a notification about the chat invite, which will display as a pop-up which they can accept or deny. Participants of the chat are notified of a user's acceptance or rejection.

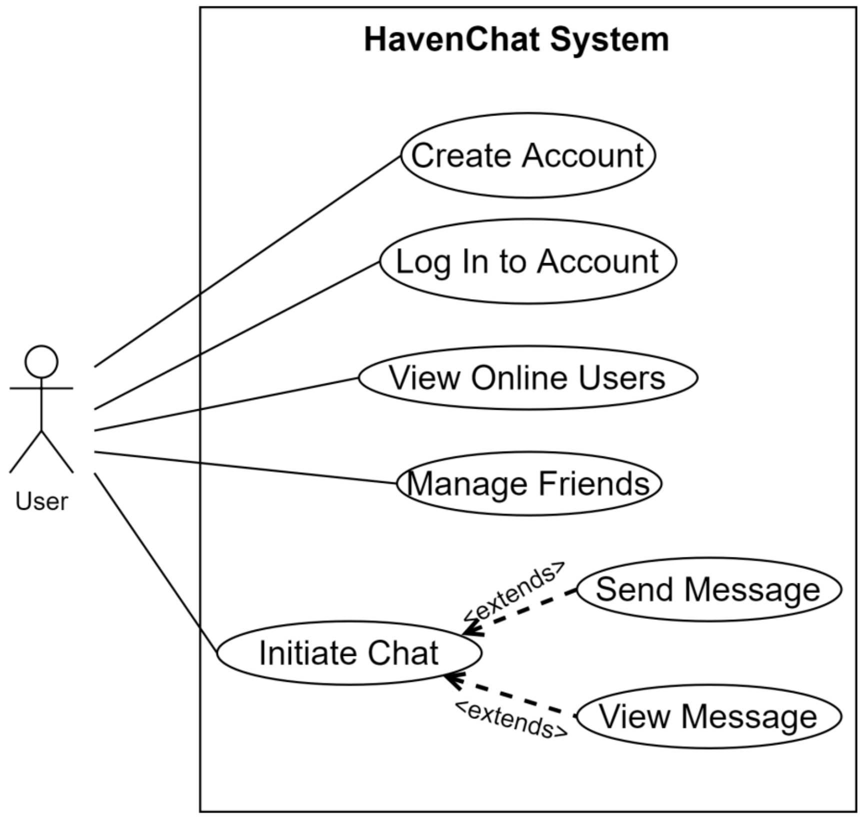
To achieve secure key distribution for the ensured Confidentiality, the server encrypts the symmetric key using each user's public key, sent to them. After the encrypted symmetric key has been distributed to all users, the users decrypt the symmetric key using their private keys, and the chat session may begin. All messages exchanged during the chat are encrypted using the symmetric key provided by the server and delivered to all other active participants of the chat, in doing so, the service provides Confidentiality. Users may also choose to leave the conversation, upon which the other users would be notified that they have left. If the user disconnects from the chat server entirely, their status changes to "offline", and he will no longer be available for chat invitations in other user's views.

The service also provides Digital Signature and Integrity, through the digital signature scheme selected by the initiating user. Upon starting the chat or accepting a chat request, a user generates a public key for signature verification, generating a DSA key if the DSA method was selected, or re-using the RSA key if RSA is selected. These public keys are distributed to all participants of the chat. Every message sent is signed using the corresponding private key, and the signature is sent with the message. This signature is then verified by each participant using the received public keys. The verification status of each message signature is displayed for each message.

2.0 Design

2.1 Use Case Diagram

Our application is designed with the base use cases focused on account creation and usage, connecting with others using a friend system, and using the chatting feature to send and view messages.



2.2 List of Use Case

2.2.1 Create Account

Use Case: Create Account

UC-ID: #001

Description: Visiting User can create a new account in order to use HavenChat

2.2.2 Log In to Account

Use Case: Log In to Account

UC-ID: #002

Description: Allow User to log into their account.

2.2.3 View Online Users

Use Case: View Online Users

UC-ID: #003

Description: User can view other users and see whether they are online or offline at the current moment.

2.2.4 Manage Friends

Use Case: Manage Friends

UC-ID: #004

Description: User can view their current friends and can send friend requests to other users. User can also accept or deny friend requests, as well as unfriend any of their current friends.

2.2.5 Initiate Chat

Use Case: Initiate Chat

UC-ID: #005

Description: User can select any of their online friends as participants of the chat and can select between DSA and RSA as a Digital Signature scheme. The User can then initiate such a chat with the given parameters, and the server must take care of generating a symmetric key, and **securely distributing this key** to the proper participants.

2.2.6 Send Messages

Use Case: Send Messages

UC-ID: #006

Description: User can send messages to the current chat session's participants. The service must encrypt these messages using the chat session's secret key, and must sign it using the appropriate selected algorithm, and send the message and signature (as well as IV due to CBC mode), to all active chat participants.

2.2.7 View Messages

Use Case: View Messages

UC-ID: #007

Description: User can view the messages sent to them in the current chat session. The service must decrypt the messages using the chat session's secret key, and must verify the message's data and origin integrity by verifying the signature, using the appropriate user's public key.

2.3 Components & Architecture

To accomplish these use cases, we designed a full stack Web Application, with users using their browsers to run the Front-end components, and a server that runs the Back-End components as well as the Database.

2.3.1 Front-end – SPA, w/ View-ViewModel Pattern

The Front-End uses a Single-Page Application design, with client-side rendering for a smooth user experience. The Front-End repeatedly makes queries to the Back-end. The data retrieves constitutes the ViewModel along with any light-weight logic for processing this data. The data is then rendered into the View.

2.3.2 Back-end – REST API w/ Model-Controller Pattern

The Back-End uses a series of Controllers that make up the REST API endpoints. These controllers receive requests from the clients, preprocess the parameters and use JWT tokens to retain user identity, then pass requests down to Models, which performs most of the heavy-weight logic. The Models also interfaces with the Database for persistence.

3.0 Security Protocols

For the secure chat, security protocols and a series of cryptographic algorithms had to be used to maintain Confidentiality, Authentication, Digital Signatures, and Integrity. In the following section, we will discuss the protocols and algorithms used for HavenChat. First, we will discuss the standard cryptographic protocols used for the core functionality of the chatting app. We will then discuss the other cryptographic protocols used that support the rest of the app in terms of authentication and HTTPS support. Finally, we will discuss the custom application-level protocol used by HavenChat service to coordinate the initiation of chats and messages exchange.

3.1 Cryptographic Algorithms Used for Chatting

3.1.1 Symmetric Message Encryption – AES 256 in CBC Mode

The core cryptographic module that supports the Confidentiality of messages exchanged through HavenChat, is the AES algorithm. We use the 256-bit version of AES configured to run in CBC mode. Although most messages are likely to remain short, we use CBC mode still to ensure that patterns in messages are hidden for the potential of future features to send larger media files. We find that the added complexity of CBC is manageable, and the performance impact negligible.

The symmetric AES 256 key is generated by the server upon initiation of a chat session. This key is stored for the duration of the chat session, for future participants to access, however only participants and invitants are given access to session's key. These keys are securely distributed (discussed in 3.1.2 Secure Key Distibution – RSA), to each participant and each participant stores these keys to encrypt and decrypt every message.

For each message sent, an initialization vector (IV), is randomly generated (later discussed in 4.0 Implementation, to be cryptographically sound by the library used). The IV is then used along with the session key to encrypt the message. The IV is then sent along with the message to the other participants. The other participants can then decrypt the message using both the IV and the session key, to obtain the plaintext message.

3.1.2 Secure Key Distribution – RSA

To ensure and support the Confidentiality provided by the symmetric message encryption above, the Confidentiality of the symmetric key must also be secured. The symmetric key must be distributed by the server to each participant without revealing the key to outside parties, even if they intercept it. To accommodate secure key distribution, HavenChat uses public key encryption – namely, we use the RSA algorithm.

Upon starting the chat, the initiator will generate an RSA public key pair. The public key is sent with the chat request to the server. Once the server generates the symmetric session key, the server encrypts the session key using the initiator's public key, and sends it back to the initiator user. The user can then decrypt this message using their private key, and securely obtains the session key.

For the other participants of the chat, they generate an RSA public key pair, and send their public key along with their chat acceptance to the server. As it did for the initiator, the server can then verify if the participant is invited to the chat session accepted, retrieves the session's symmetric key, encrypts it with the participants public key, and sends it back to the accepting participant. The participant can then decrypt the session key with their private key, and securely obtains the session key themselves.

3.1.3 Digital Signature Scheme 1 – RSA

To provide Digital Signature (and consequently Integrity, to be discussed in 3.1.5 Digital Signature Hash Digest – SHA-256), two Digital Signature schemes are offered – RSA and DSA. We will first discuss the RSA scheme.

Upon initiation of the chat, the chat initiator selects between the two digital signature scheme. If RSA is selected, then the same RSA public key used for secure key distribution is re-used for Digital Signatures. The public key for Digital Signature (which we will refer to as the signature key), is stored by server as associated to the chat session and the participant. For the invited participants, once they are informed that the initiator has selected the RSA signing scheme, they will also submit their RSA public key as a signature key. Upon acceptance, the server will also store their signature keys.

Upon acceptance, the acceptor will receive all active participant's IDs and signature keys. The acceptor's ID and signature keys are also broadcasted to all active participants for them to store.

Now, for every message generated, it is hashed using a hashing algorithm, and the hash digest is signed using the participant's private signature key. The signature is then appended to the message and sent to all participants. Once received a recipient and then decrypt the message as discussed above, hash the plaintext message, then use this hash digest, the appended signature, and the sender's public signature key, to verify the sender's identity and the message's integrity.

3.1.4 Digital Signature Scheme 2– DSA

If DSA is selected, then upon chat initiation, the chat initiator will generate a DSA signature key pair and send the public key to the server. The server stores this signature key. Each invited participant then sees that DSA was selected as the signature key and also generates their DSA signature key pair and sends the public key along with their acceptance. The same procedure discussed above is then used for the distribution of the DSA public key to each participant, and to sign and verify each message using DSA.

3.1.5 Digital Signature Hash Digest – SHA-256

To support the RSA and DSA Digital Signature schemes, the algorithms needed a hash digest in order to sign and verify. The hashing algorithm used to produce these hash digests is the SHA-256 algorithm. As the Digital Signature is dependent on the hash digest, which is dependent on the message contents, verification of a message's digital signature not only verifies origin integrity, but also message Integrity.

3.2 Other Cryptographic Algorithms Used

3.2.1 Password Hashing – BCrpyt

Authenticating into HavenChat involves the usage of credentials comprised of a username-password pair. It is crucial then to secure user passwords. For the secure storage of passwords, the BCrypt algorithm is used to hash passwords before storing it in the database. We use a cost of 16 for BCrypt as the default of 10 is deemed insecure for the current time, but any higher may be too high for a demo app running on a small Linux box. For true production builds, we recommend using a cost of <Current Year> - 2010. The higher costs are to prevent the feasibility of brute-forcing of passwords.

A further security measure that should be placed for true production builds would be password expiration features, however this is outside of the scope of the base message exchange system of the project.

3.2.2 JWT Token Server-Signing – HS512 (HMAC using SHA-512)

To accommodate Authentication, once a user is authenticated with the server, the server generates a JWT Token, signed by the server. The token is signed by the server using HS512, or HMAC using SHA-512. The token merely holds the user's ID and no excessively sensitive user information. This token is sent back to the client, and the client will store this as a cookie to be sent with any coming requests, both API requests and WebSocket requests.

For each API and WebSocket request that requires a user to be authenticated, the server will take the JWT Token in the requestor's cookie. The token is then verified by the server, to see that it is the one that the server has signed. Once verified, the user will have verified the user's identity.

The cookie is stored as an HTTP-only cookie, to prevent third parties from obtaining it programmatically. On production environments, the usage of SSL due to HTTPS (discussed in the next section, 3.2.3 SSL for HTTPS support), is also entrusted to prevent third parties from intercepting a client's request and obtaining their token for masquerade attacks.

3.2.3 SSL for HTTPS support

For the production web app, we run all web traffic through HTTPS, with a CA signed certificate to cleanly support HTTPS. We then let the client browser and the https library do the heavy lifting in handling encryption over SSL. This encryption adds an additional layer of cryptographic security in the production environment for API calls as well as the WebSocket connection over which message exchange takes place. The greatest protection provided by this is over the password sent for authenticating into the web app, as the password must be sent to the server, and the server is the one responsible for hashing the password.

3.3 Chatting Protocol

3.3.1 Initiating a Chat Session

1. Initiator REQUEST to initiate chat:

An initiator user must initiate a chat session using the following application-level protocol: The initiator sees which of their friends are online. Of these online friends, the initiator selects which participants they want to chat with. The initiator will also select which digital signature scheme will be used by the chat – RSA or DSA.

The initiator then generates an RSA public key pair for secure key distribution. Depending on the digital signature scheme selected, if RSA is used, then the RSA public key doubles as the signature public key. If DSA is selected, an additional DSA public key pair is generated, and the DSA public key is sent as a signature key.

The initiator then sends a REQUEST message over a WebSocket to the server, in the following scheme:

```
Let UID = User ID, SID = Session ID
PU/PR = RSA Public/Private Key,
SPU/SPR = RSA/DSA public/private key pair
signType = 0 for RSA, 1 for DSA
```

```
                         REQUEST:
        {sid: 0, uid: UID1, pubKey: PU_1, participants:
     [UID_2, ...UID_N], signType: signType, signKey: SPU_1}
```

2. Server processes REQUEST to initiate chat:

The server then sees the REQUEST message w/ sid=0 and recognizes the user1 wants to initiate a new chat. The server will then generate a new symmetric chat session key, which will be a 256-bit AES key. A new chat session will be created with a new **SID,** the requested signature scheme and the newly generated symmetric key **K**.

The initiator will then be marked as an active participant, and the invited participants as potential participants. The participants are stored as follows:

sid	userId	signKey	active
SID	UID_1	SPU_1	1
SID	UID_2	NULL	0
...	...	...	...
SID	UID_N	NULL	0

2a. Server responds to Initiator's REQUEST:

The server responds to the Initiator first, sending an ACCEPT messsage with the newly created SID and with the symmetric session key encrypted with the Initiator's RSA public key. The response is formatted as follows

```
ACCEPT: {sid: SID, uid: UID_1, secret: K' = E(K, PU_1)}
```

The Initiator sees the uid matching there's and recognizes that their request has been accepted. They then store the SID of the chat session, and decrypts the secret to obtain the session key K as follows:

```
K = D(K', PR_1)
K = D(E(K, PU_1), PR_1)
K = K
```

The symmetric key K is then stored for the upcoming chat session, and the initiator waits until the invited participants join. The app will take them to the chat screen to wait for notification of participant acceptance or rejection.

3. Server REQUESTS Participants to join session:

The server then takes the participant list UID_2 to UID_N, and invites them all to the chat by sending the REQUEST message

```
REQUEST: {sid: SID, uid: UID1, signType: signType}
```

Invited participant X will see the request to join chat session SID, sent to them by user UID1, with the given digital signature type. The user is then notified by HavenChat with a pop-up with the option to reject or deny. If a denied, a CLOSE message is sent, relayed by the server to active participants, and they will be notified in the chat that User X has rejected the chat request.

Otherwise, if accepted, User X generates an RSA Public key pair, as well as a signing public key according to the signType selected. If RSA was used, then the RSA key is re-used, otherwise a DSA public key pair is generated. An ACCEPT message is then sent in the following format:

```
ACCEPT: {sid: SID, uid: UID2, pubKey: PU_X, signKey: SPU_X}
```

<u>4. Server processes Participant ACCEPT</u>:

The server then receives the ACCEPT message from User X, sees the request to accept chat session SID, and verifies that UID_X is a participant of Session SID. Once verified, the server updates User X's entry in the Participants table as active, and stores their signature public key.

sid	userId	signKey	active
...	...	...	...
SID	UID_X	SPU_X	1
...	...	...	...

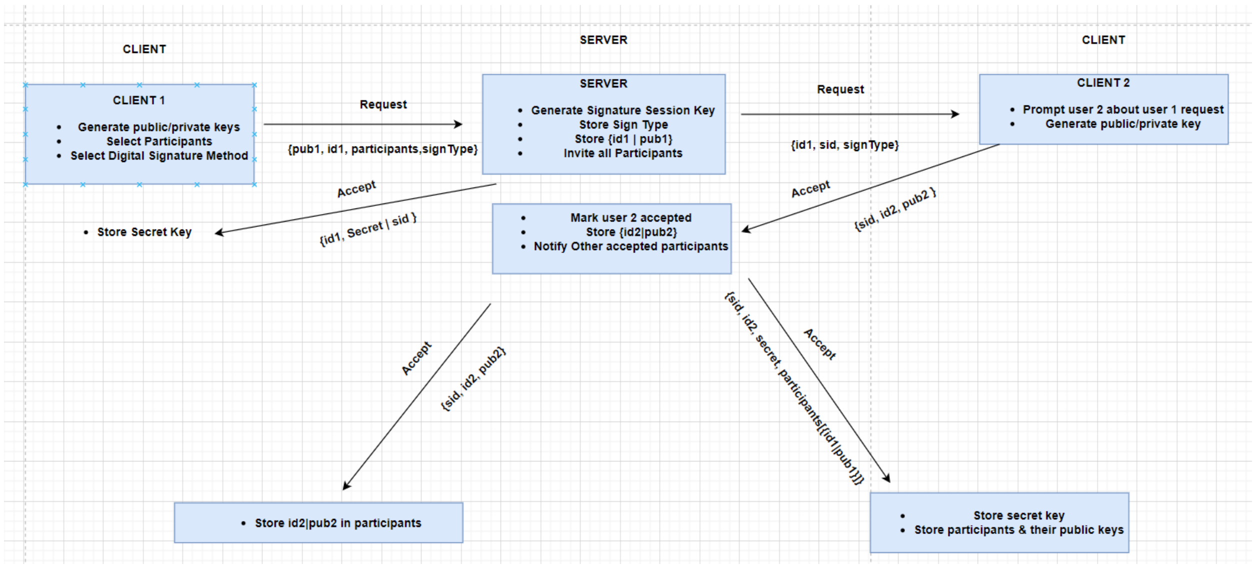
The session also retrieves Session SID's secret key, encrypts it with X's RSA public key, and sends back an ACCEPT message to User X, along with a list of currently active participants and their signature public keys.

```
ACCEPT:
{sid: SID, uid: UID_X, secret: K' = E(K, PU_X), participants: [
{uid: UID1, signKey: SPU_1}, …, {uid: UIDX, signKey: SPU_X}
]}
```

User X receives this message and treats it just as the Initiator does in section 2a, with one modification. This time, User X initializes their participant list with the array of active participants given, successfully catching them up with participants who joined before them.

<u>4a. Server propagates ACCEPT message to other users</u>:

Finally, the ACCEPT message sent by User X to the server is propagated to all active participants of the chat. These participants will then receive the ACCEPT request, recognize that the uid is different from theirs, and will append the user's info into their participant list. The participant will also be notified that User X has accepted the chat request.


CLIENT
SERVER
CLIENT
CLIENT 1
Generate public/private keys
Select Participants
Select Digital Signature Method
Request
{pub1, id1, participants,signType}
SERVER
Generate Signature Session Key
Store Sign Type
Store {id1 | pub1}
Invite all Participants
Request
{id1, sid, signType}
CLIENT 2
Prompt user 2 about user 1 request
Generate public/private key
Accept
{id1, Secret | sid }
Store Secret Key
Mark user 2 accepted
Store {id2|pub2}
Notify Other accepted participants
Accept
{sid, id2, pub2 }
Accept
{sid, id2, pub2}
Store id2|pub2 in participants
{sid, id2, secret, participants[{id1|pub1}]}
Accept
Store secret key
Store participants & their public keys

3.3.2 Sending and Receiving Messages

<u>1. Sender SENDs message</u>:

First, the sending user S types up a message they want to send, msg, and initiates message sending. The service will then generate an initialization vector, IV, retrieve the stored session key K, and encrypt the message as follows:

```
msg’ = E(msg, K, IV)
```

The sender will then hash their message, and create a digital signature of the message using their signature private key and the agreed upon digital signature scheme, as follows.

```
Let H(x) be the hashing function, ds be the digital signature
ds = E(H(msg), SPR_S)
```

The sender then sends these items in the following format.

```
SEND: {sid: SID, uid: UID_S, msg: msg’, iv: IV, sign: ds}
```

2. Server relays SEND message:

Once a server receives the SEND message above, it performs a look-up on the participants table and searches for participants of chat session SID, that are marked active. It then relays the SEND message to these participants.

3. Recipient receives SEND message:

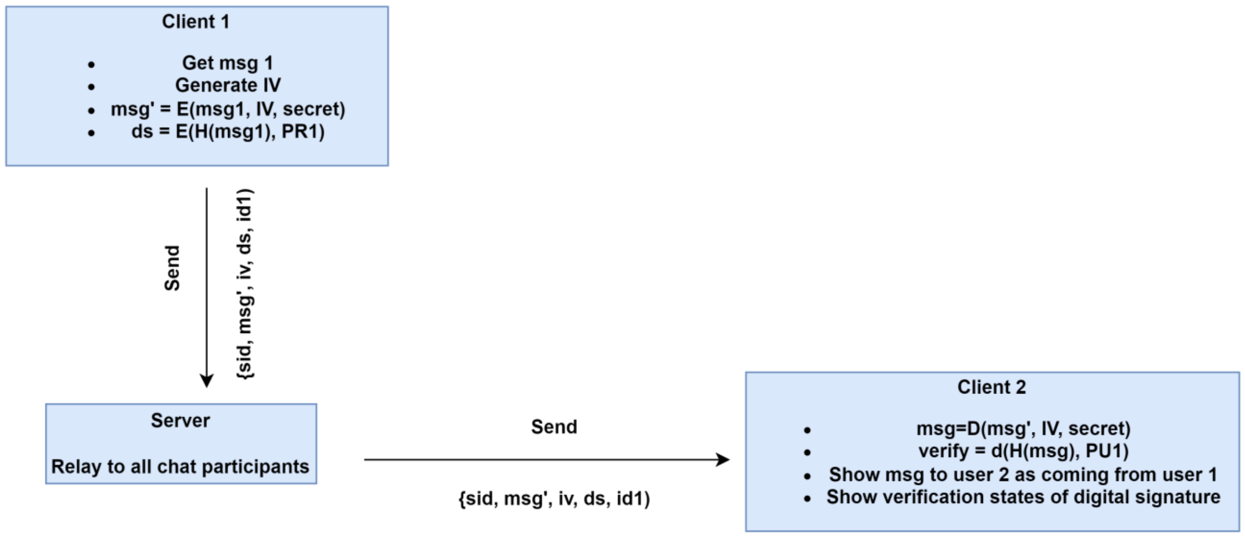
Finally, the recipient receives User S’s SEND message. The recipient will first takes the IV and encrypted message msg’, and uses the session key K to decrypt the message as follows:

```
msg = D(msg’, K, IV)
```

The recipient then uses the plaintext message to create a hash digest as the sender did. The recipient then verifies the digital signature against this hash digest, as well as the sender’s signing public key as follows:

```
Verified = verify(ds, H(msg), SPU_S)
```

Finally, the message is displayed to the recipients as coming from user S, and the verification status of the digital signature is displayed as well.



4.0 Implementation

4.1 Tech Stack

4.1.1 Front-End

The client-side component, also known as the front-end, uses **React** and is written in **NodeJS**. React is used in a View-ViewModel pattern, with React handling the rendering of the Views, while having some logic in the client-side app to interface with the server or do its own heavy-weight cryptographic tasks.

4.1.2 Back-End

The server-side component, also known as the back-end, uses the **Express** framework and is also written in **NodeJS**. Express served the API endpoints through a series of controllers. The controllers then relay client requests to the Models, which carried the bulk of the heavy-weight logic as well as interfaced with the Database.

4.1.3 Database

MySQL was used as the relational database of choice. Various tables were used to store persistent information such as user information and credentials, friendship relations, chat sessions, and chat participants.

4.1.4 Platform

The application was developed primarily developed for use in Linux. To host the production build, a small Linode box is used, proxied through other applications maintained by the owner of the box. The Linode Box runs Debian 11, while our local development environments ran Ubuntu 20.04.3 and Windows 10.

4.2 Libraries

4.2.1 Cryptographic Libraries

Node-forge is the primary cryptographic library used for symmetric and asymmetric encryption, as well as RSA digital signing and SHA256 hashing. For password hashing in BCrypt, the **bcrypt** library is used. For signing JWT tokens, the **jsonwebtoken** library is used along with its provided HS512 signature scheme.

For DSA, we had to develop our own implementation, with the help of **JSBN** for support of big integer arithmetic, as well as some prime helper functions from **node-forge.**

4.2.2 Supporting Libraries

The backbone of the chat service is the WebSockets through which the chatting system communicates. The client-side application uses the standard **WebSocket** library, while the backend used the **ws** library to host the WebSocket server. As mentioned before **React** and **Express** were used to host the front-end and the back-end API respectively. We also used the **mysql** library to allow the backend to interface with the database.

5.0 Conclusion

HavenChat is a Web Application that provides Confidentiality, Authentication, Digital Signature, and Integrity in its Secure Live Chatting service. Users authenticate using a bcrypt-hashed password system, and once authenticated, uses a JWT token system secured by HS512 and the usage of HTTP-only cookies. Users may then use a friend system and invite participants to the chat, and may begin chatting with any accepting participants. To support the chatting service, a custom application protocol comprised of REQUEST, ACCEPT, CLOSE, and SEND messages are used over WebSockets. The app-level protocol is then comprised of a series of procedures that use AES-CBC-256 for symmetric message encryption, RSA for secure key distribution, and RSA/DSA for digital signature (and SHA-256 for hashing). On the production site, as the WebSockets and the API endpoints are hosted over HTTPS, SSL adds further security to all web traffic involved in HavenChat