Saba Secrets

Jemin Park, Alice Yu, Tyler Yang, Laura Wu

I. Motivation and Project Goals

Many of the most popular end-to-end encrypted (E2EE) messaging services, such as Signal, Whatsapp, and iMessage, are currently implemented through native applications on mobile and desktop devices, but not on web-based applications. Upon further research, we discovered that, unlike with native applications, there exists "no way to effectively version, sign, and securely distribute a web page" - reducing the security of such an application to that of the SSL/TLS connection to the server. This would mean that web-based implementations of E2EE messaging are vulnerable to compromised servers and state-level attacks such as government subpoenas. Compared to mobile and desktop clients, which are downloaded only once and have their integrities verified via digital signature, web applications are in constant communication with servers for the source code (which may change in an instant without notifying the client), which makes integrity verification more impractical [1].

Despite these glaring issues, we believe E2EE messaging, even via the web, is still useful in guaranteeing a minimum level of privacy for end users. In particular, E2EE would protect users from data exploitation by companies with malicious intentions. In countries with emerging economies, where many have access to the internet but do not own smartphones, web-based E2EE would provide accessibility for encrypted channels of communication for anyone with an internet-accessing device. However, we found many other popular applications used for messaging do not have E2EE, including Twitter, Instagram, Snapchat, and WeChat. Our group raised the following questions: Why is E2EE messaging not implemented (and enabled) by default? Is it due to technical difficulty or cost?

Therefore, we intend to explore these questions by building our own web-based E2EE messaging platform using the Signal protocol, as Whatsapp has done. Through this, we will provide basic support for end-to-end encryption using the secret key/public key (SK/PK) scheme that will allow users to encrypt using the other user's PK, hiding content from the server - which will only host ciphertexts passed between users. We will also provide out-of-band authentication, which will allow users to verify the public key of the other user in their own external channels to ensure the other person is who they claim to be.

The goal of this project will be to build a functional web-based end-to-end encrypted platform using popular technologies and tools to allow internet users to have secure means of communication. We intended to have this project give the users ability to interact in one-on-one and group settings via texts, images, and files, while providing the following additional security features: one-time messaging, sealed sender, and ephemeral message storage. With extensibility as the highest priority, the core focus of this project was to deliver strong security features while not sacrificing any functional requirements that a user might expect of a modern messaging application.

II. Related Works

Many of the most popular messaging services have billions of users globally, with over 5 billion unique people using mobile devices globally, with 3 billion total messaging app users. We will explore the implementation and security features of some of the most popular ones, including those that provide E2EE optionally, such as Facebook Messenger and Telegram.

A. Signal

Signal developed by Open Whisper System, the creator behind Signal protocol, is by far one of the most security-forward messaging services in the market. Signal relies on Signal Protocol, an open-source protocol, for its end-to-end encryption. The signal protocol uses a combination of different encryption algorithms to provide secure communication. The Sesame enables end-to-end encryption in asynchronous or multi-device settings and Double Ratchet Algorithm and Advanced Diffie Hellman Algorithm allow message encryption, by providing forward secrecy and generating new encryption keys for each message [2].

One unique feature Signal has is its "Sealed Sender" feature. Signal's Sealed Sender feature is a privacy-enhancing feature that helps to protect the metadata of a message. Metadata includes information such as the sender, recipient, and timestamp of a message that can reveal a lot about a user's communication patterns and relationships, even if the content of the message itself is encrypted. Signal has also attempted to protect user information by enhancing its contact discovery feature [2]. Signal's contact discovery feature uses hashed phone numbers to find Signal users in a user's contacts, which ensures that the user's information is not being stored on its server or other user's device. By not only taking a strong stance in end-to-end message encryption and pushing for the protection of user privacy Signal is now recognized as one of the most secure encryption protocols available.

B. WhatsApp

WhatsApp's end-to-end encryption relies on Signal Protocol and is used when one chat with another person using WhatsApp Messenger, ensuring security with every message. WhatsApp features "Verify Security Code", which is optional for end-to-end encrypted chats, and are only used to confirm

that the messages and calls that are sent are end-to-end encrypted. Each chat contains its own security code that is used to verify whether calls and messages one send to another are end-to-end encrypted. Along with verifying whether the chat is end-to-end encrypted, this feature also verifies that the linked devices between the user and the contact lists are up to date [3]. To see whether the messages are encrypted with the contact list, we can click the name of the user, go to "Encryption" and see the QR code or a 60-digit number. If you scan each other's QR code or if the 60-digit number matches, it means that the chats have not been intercepted [3]. Overall, WhatsApp's use of the Signal Protocol and the "Verify Security Code" feature ensure that messages and calls are end-to-end encrypted and that users can confirm the security of their conversations. The unique security code for each chat and the verification of linked devices further enhance the platform's security. By providing a secure and private messaging experience, WhatsApp continues to be a popular choice for users worldwide.

C. Facebook Messenger

Facebook Messenger currently has an optional "secret chat" feature in which E2EE is provided. While they have recently shared plans to turn E2EE by default in its platform (even conducting A/B tests for some users), they have claimed implementing basic security is difficult due to the size of its platforms. Ironically, this transition was done by Whatsapp (another Meta-owned platform) years before on a similar scale. While looking into Facebook Messenger, we took particular interest in their implementation of message franking. Facebook recently implemented message franking in their messaging platform in 2017, to allow for authenticated reporting of messages by users in an end-to-end encrypted environment. They introduced new primitives to make their system more efficient and outline specific implementation details that make the system more efficient for Facebook as a whole [4]. Notably, they use a private key associated with Facebook to sign the signed message, so Facebook doesn't have to store the data on their server. But the basic design should be relatively simple to emulate to implement message franking.

D. iMessage

The biggest messaging platform currently, iMessage differs quite significantly from our aforementioned applications, taking its encryption to effectively use the hardware as a native platform on devices. Using Apple hardware, the service generates keys and connects to iCloud to encrypt and decrypt messages, such that all the devices that are logged into the same account can view all their messages. This differs wildly from Signal's multi-device support, as the encrypted messages there are kept on the server until they are seen by recipients - they reportedly hold a queue of a thousand of the most recent undelivered messages, and deletes messages that are older than 60 days [5]. In iMessage's approach, they encrypt every message several times for every device linked to the participants iCloud accounts - however, they do not encrypt text messages sent outside of Apple devices [5]. The hardware requirement naturally limits the true security potential of this service, which further emphasizes the need for fully internet-accessible platforms on the web.

III. Platform selection

Like with WhatsApp, our approach towards encrypted messaging will be facilitated by the Signal Protocol while using a popular tech stack by modern web developers: React, TailwindCSS, Next.js, TypeScript, tRPC, PostgreSQL. In this section, we will go over the background of key components used in building our application, as well as any supplemental decisions that came along with it.

A. Messaging

The following section outlines the technology stack for our implementation of messaging as a minimum viable product (MVP). In the spirit of this project's motivation, we focused on using modern technologies that are either 1) often used in production at large-cap firms (e.g., Meta or Amazon) or 2) heavily used in new tech startups or projects.

- *React*: One of the most popular open-source front-end frameworks, developed by a team at Meta, this framework uses a component-based system to allow efficient, DRY code across a web application, converting from JSX-formatted code to native HTML and vanilla JavaScript. It uses a virtual DOM to make reactive updates to enhance user experience.
- *Next.js*: Next.js is a popular web-based meta framework built on top of React, allowing for efficient delivery of fast and scalable web applications. It provides a comprehensive set of tools and features out of the box, such as server-side rendering, automatic code splitting, static site generation, and optimized performance. Importantly, this allows us to better separate client and server-side code, leaving minimal, if any, access to database clients, user data, and environment variables exposed to end users.
- *TypeScript*: Our programming language of choice, TypeScript is a statically-typed superset of JavaScript that provides optional type annotations and other features, such as interfaces and classes, to enable more robust and maintainable code for both front-end and back-end development.
- *tRPC*: We chose tRPC for building lightweight, type-safe APIs for modern Remote Procedure Calls (RPCs) in TypeScript and Node.js. The additional type-safety would enable rapid agility when building backend applications intricately connected to client-side reactivity, such as a chat application.
- *PostgreSQL*: PostgreSQL is a powerful open-source relational database management system (RDBMS) that is often used for large-scale and complex applications due to its reliability, performance, and flexibility. We chose PostgreSQL due to its simpler design and performance at a lower scale, though we have considered NoSQL systems such as Cassandra and highly scalable variations

such as CockroachDB. Note that all message content stored here will be encrypted as ciphertext, which would not be accessible to anyone but the respective users with their secret keys.
- *Amazon Simple Storage Service*: Commonly known as S3, we chose a highly scalable and durable object storage service offered by Amazon Web Services, designed to store and retrieve any amount of data from anywhere on the web. We will use this popular AWS service to store and retrieve items like encrypted images and files where necessary.
- *Prisma*: A TypeScript-based Object Relational Mapping (ORM), Prisma allows us to define a type-safe data model to improve developer experience while having a nice safeguard against SQL injection attacks through prepared statements [6].

B. Security

While we will be using Signal's Typescript library for their protocol (referred to as libsignal, found here: https://github.com/privacyresearchgroup/libsignal-protocol-typescript), the following section outlines the security protocols and tools we are depending on for privacy:

- *Hypertext Transfer Protocol Secure*: HTTPS provides security by encrypting data in transit between a client and a server, preventing interception and tampering of sensitive information such as login credentials and credit card details, and verifying the identity of the server to ensure that the communication is not intercepted by a third party.
- *Signal Protocol*: The Signal Protocol uses the Double Ratchet algorithm for its key management, which provides perfect forward secrecy and encryption for each message exchanged between two parties, ensuring that even if a shared secret is compromised, only a small amount of data is revealed [14]. The protocol also utilizes pre-keys, a shared secret, and hash functions for additional security and authentication. In the protocol, pre-keys are used as part of the key agreement process to provide forward secrecy and allow for new session keys to be established between the sender and recipient for each message. The sender generates a set of pre-keys, which are then uploaded to the server. When a recipient initiates a session, they request a pre-key bundle from the server, which includes a pre-key signed by the sender's identity key. The recipient then uses the pre-key to derive a new session key and sends it to the sender, along with a message encrypted with the session key. The sender uses the session key to decrypt the message and can then establish a new session key for future messages. By using pre-keys in this way, the Signal Protocol ensures that even if long-term keys are compromised, past messages remain secure, and future messages are protected with new session keys. We have taken strong consideration for this when designing our data model.
- *Argon2 password hashing*: The winner of the Password Hashing Competition in 2015, this password hashing algorithm is designed to provide strong resistance against password cracking attacks, such as dictionary attacks, brute-force attacks, and time-memory trade-off attacks [7], [8]. It is also considered to be the current state-of-the-art in password hashing, recommended by the Open Worldwide Application Security Project (OWASP) [9]. As a standard feature among modern hashing algorithms, Argon2id adds a unique, randomly generated string (also known as a "salt") to the password, which makes cracking large numbers of hashes significantly harder, and protects against attacks using precomputation, as salts are unique per user and different salts will result in different hashes, even if the passwords are the same.
- *Browser local storage*: To avoid storing secret keys in a database, even if encrypted, we have considered using browser local storage to store and retrieve secret keys for encryption/decryption, as well as messages. The main benefit here is that this allows only the permitted browser to decrypt messages, preventing logins from any other location from accessing messages. This, however, does come with drawbacks, which will be discussed later.
- *JSON Web Token*: To maintain browser sessions, we will be using JSON Web Tokens (JWT) as a compact method of authentication. JWTs are signed payloads that contain additional information about a user's session, such as the ID and the expiration time. The primary benefit here is that they are easily transmitted between the client and server in HTTP headers and are also easily verified by the server without needing to perform additional database or API calls, which makes JWTs a stateless solution and a suitable replacement for server-side session storage. This method does also have drawbacks, however, which will be discussed later.
- *HTTP-only cookie*: JWTs are at risk for tampering, despite being digitally signed due to being held in local storage. This, however, can be mitigated by using a HTTP-only cookie, which is inaccessible to client-side scripts, hence a strong method of protection against cross-site scripting (XSS) attacks [10].
- *CSRF tokens*: Similar to XSS attacks, another method of attack against vulnerable websites, particularly for login information, is through cross-site request forgery (CSRF). These attacks work by tricking users into unintentionally sending authenticated requests to applications they're already logged into, allowing attackers to perform unauthorized actions on their behalf. Our method of prevention will be through CSRF tokens, which are unique secret values that are generated server side (typically by an auth service, which we will be hand-rolling) that client applications must include in their request when performing sensitive actions, such as submitting a form [11].

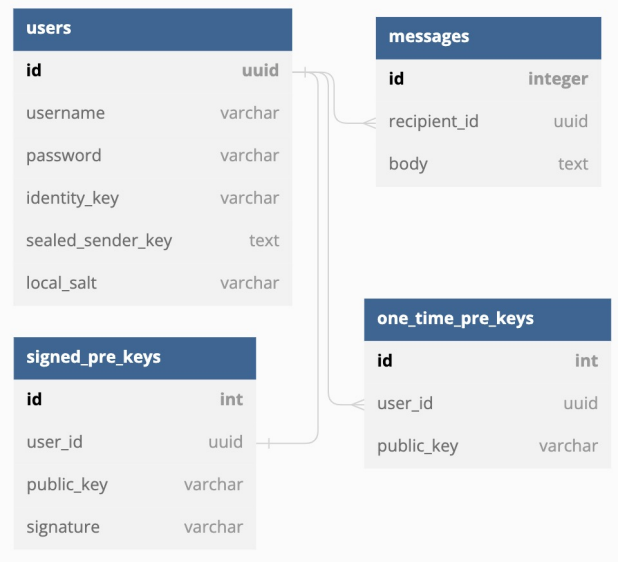

Fig. 1. Simplified Chat Schema

IV. Data Model

We will first begin by defining the data model, which can be seen in Fig. 1. As this is a web-based application and not limited to singular device, the server must store some level of data to authenticate and authorize and act as an intermediary entity to pass on messages. We will discuss the specifics of the model and its usage in the following subsections.

A. Users

We must store some limited data on users in the server to act as a trusted entity to validate users and enable authentication/authorization for logins and sending messages. The users table will thus contain a global identifier (UUID), a hashed username, hashed password (using argon2), public identity key, public sealed sender key, and a local salt used for encrypting local storage. The public identity key relates to message signatures as it relates to messaging, and the public sealed sender key is a separate key used to ”seal” messages upon send. Further discussions on these will come later.

While this does hold some concerns over privacy over global user identifiers, Signal and other E2EE messaging platforms also have some form of user data stored on the server. Unlike other platforms, however, we have decided to prioritize pseudo-anonymity. By not requiring personal identifiable information via emails, phone numbers, or even device fingerprinting, we can successfully prevent malicious actors with access to server data from identifying users.

We also introduce a hash for the username - as usernames are strongly linked across multiple platforms, we want to prevent identifiable information from being casually leaked to the server. The hash will need to be done over a smaller keyspace, as a larger keyspace (such as one obtained using SHA-2 algorithms) will not be sufficient in preventing identification through dictionary attacks. While we recognize the inherent weaknesses of a username hash, this prevents a casual attacker from being able to find a specific user given a username.

A salted hash or any form of encryption would prevent usability across browsers - a user would need to know both their username *and* salt/secret key. Using a key derivation function would also enlargen the keyspace, which is ultimately what we want to prevent. We are currently searching for an effective hashing algorithm that is small and not collision resistant.

B. Pre-Keys

We must also store public pre-key bundles on the server, as defined by the Signal Protocol (subsection III.B.2). Every user will have one signed pre-key, and up to a certain number of one time pre-keys. One-time pre-keys are used to start sessions with the associated user - a user would request a random pre-key from the server to then begin communication with a given contact, which starts the X3DH chain. The purpose of the signed pre-key is to act as a ”backup key”, in the event a one-time pre-key is not available to start a session.

It is worth noting that Signal rotates their signed pre-keys every week, and generates new one-time pre-keys up to 100 [12]. Our current implementation does not rotate signed pre-keys, and our one-time pre-key threshold is set to 25, due to the limited number of users using our platform.

C. Messages

Our messages table exists to ephemerally store unrecieved messages. This means that while the messages have not been fetched by the recipient, they will be temporarily stored on the server (Signal also does this), and immediately deleted once the fetch is successful. Note that only three fields exist here: the message id, the recipient id, and the message body. This is due to our implementation of sealed sender, in which we hide the sender and other metadata from the server. The message body will be the result of the following encryption:

```
msg_json = {content, type, timestamp, convo_id}

sig_cipher = signal_enc(msg_json, ratchet_key)

sig_envelope = {sig_cipher, sender_id, sig_type}

body = rsa_enc(sig_envelope, sender_sealed_pk)
```

This can then be decrypted using the recipient’s sealed private key, to get the sender id and signal whisper type, which can then be used to locally retrieve the ratcheting key and decrypt the signal encryption.

V. System Design

In this section, we will cover the system design of our application. This will be divided into 4 key sections to denote each of the major flows in the application: 1) authentication/authorization, 2) initial message fetch, 3) decryption and local storage, and 4) continuous real-time messaging.

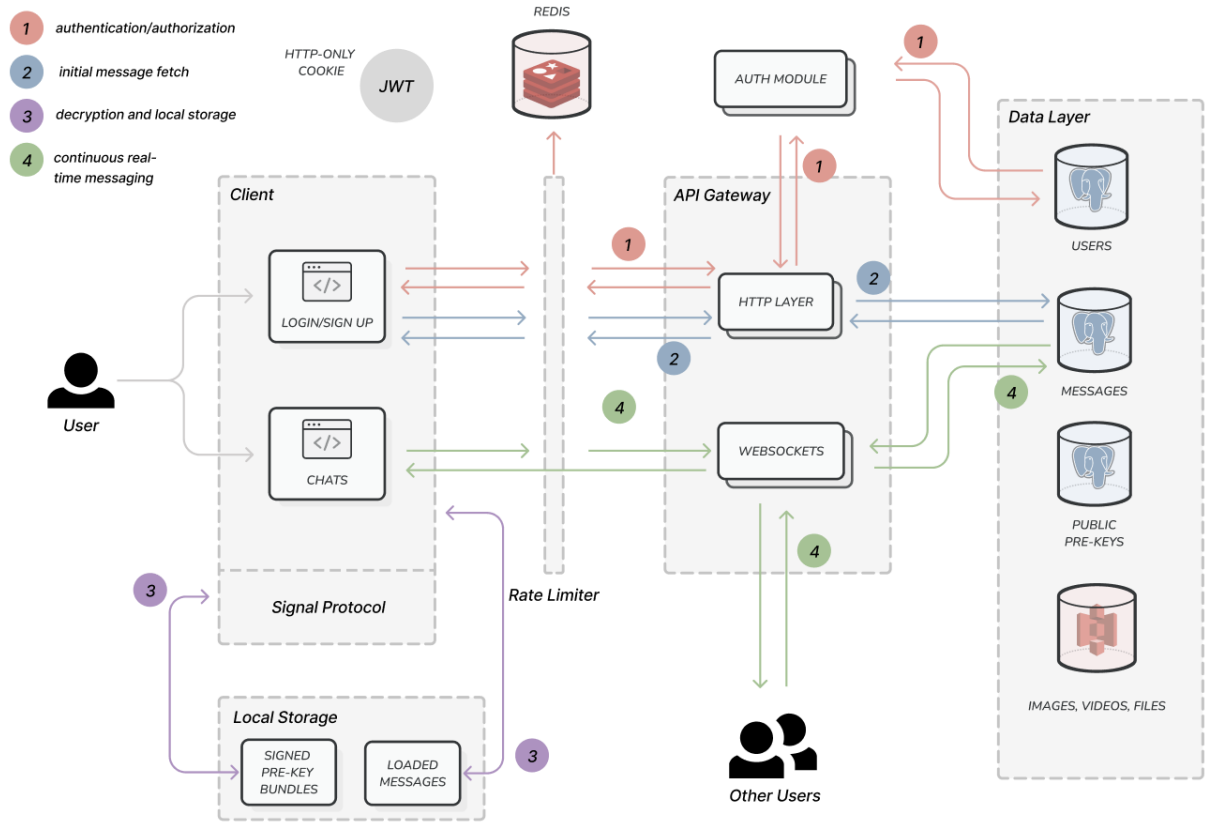

Fig. 2. System Architecture and Flow

A. Authentication/Authorization

We will be using a form extending on HTTP basic access authentication, where a user simply supplies a username/password combination to the server, and the server validates the user and grants access to protected resources. If the client attempts to access protected resources (the chats page) while unauthenticated, the server will respond with a 401 Unauthorized status code and redirect them to the login page. Here's how this works:

1) The client will be prompted to register their username and resource at the signup page. The username and password will individually be hashed using the Argon2id algorithm and sent to the server.
2) The auth module receives the request, and checks if the supplied username exists on the database. If the user exists, it will return an error and the user will be prompted to register another username. Otherwise, the new user record is created.
3) The auth module then sends the client a session as a cookie via encrypted JWT, which the client stores to help authenticate future requests.
4) For subsequent requests, the client sends the cookie along with the request headers. If the cookie has expired, they will be prompted to log in once again.
5) The auth module then validates the session for the server, and grants the client access to the protected resource. When the user logs out, or the JWT expires, the server will invalidate the session and the client deletes the session cookie.

B. Initial Message Fetch

Messages will be stored on the database with a one-to-many relationship with users, as can be seen in Fig. 2. The initial fetch will occur once users are authenticated in the first flow, which then pulls and decrypts the symmetrically encrypted local stored state for that user (which is encrypted on user id and password). After deserializing the decrypted state, encrypted messages will then be feetched where recipients IDs equal the current user's. The messages are then decrypted (twice, as defined by subsection IV.C.), and then sorted by conversation id and then asynchronously stored in local storage (encrypted) and the application's global state (unencrypted).

Once the messages have been successfully sorted, the application will immediately subscribe to the realtime server via the WebSockets channel unique to their id. More on this flow is elaborated in subsection V.D.

Note the biggest concern here with regards to performance is the decryption of a long message queue (i.e., when there are many messages that have not been received in a while). As the Signal Protocol decryption is slow, we expect it to take over a minute to decrypt 1000 messages.

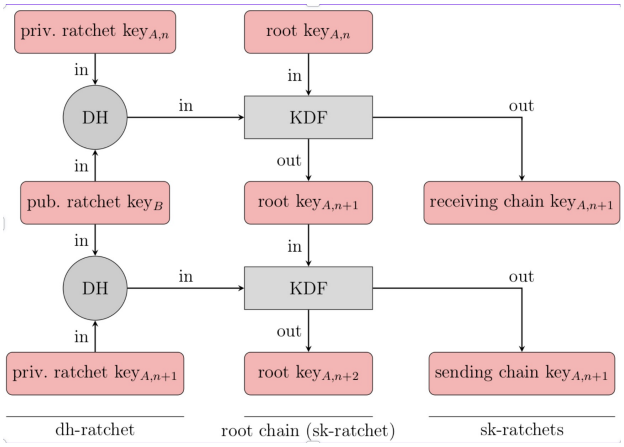

Fig. 3. Signal Protocol Triple Diffie Hellman Encryption

C. Decryption and Storage

One feature of the Signal Protocol is its use of the Double Ratchet algorithm. A core feature here is its use of KDF chains, which generates and uses a chain of keys for every message passed between uses [14]. While this does provide resilience and forward secrecy, it means we must have all existing messages and keys from the root chain to continue being able to decrypt/encrypt messages. In the case of native applications, they have access to device storage - in fact, Signal stores all message data on the device itself and only holds unreceived messages temporarily [2]. In our case, we would use local storage to first store all the keys - including the signed pre-key bundles needed to decrypt/encrypt messages, then use them to decrypt the incoming messages that are then stored in local storage. This would then be loaded onto the chat application's global state. All future messages and keys received would also be immediately stored into local storage.

Locally, messages will also hold the following information as JSON: `{{user_id+convo_id: [messages]}`, where `user_id+convo_id` is the user id appended to the conversation id. Note this serialization is slightly different from how messages are sent to the server as defined in IV.C.. The client will also group messages by channel before displaying them to the user, which is how we achieve separate chats between different users (and eventually groups).

Note all state stored in global storage is first serialized, then symmetrically encrypted using AES-GCM on the user id and password key derived from PBKDF2. While the AES encryption/decryption itself is extremely fast (being able to process GBs per second), serialization/deserialization itself can be slow (which must happen upon recieving every new message - the corresponding messages array is serialized and encrypted prior to storing in local storage. More on this to be discussed in section VI.

D. Realtime Messaging

To ensure the user recieves live updates on messages, we would have to make full use of WebSockets, specifically over the secure protocol (WSS), which protects this from MitM attacks. Once the user has recieved and decrypted their messages, we intended for the client to initiate continuous, full-duplex communication with the server. By using a publish-subscribe model, we can obtain the following flow:

1) The client uses the WSS module to subscribe to all updates to the specified conversation channels.
2) Any updates to the channel (i.e., messages "published" by other users) will be received in an encrypted format via the Signal Protocol and immediately decrypted and stored client side.
3) Any posts to the channel will be first encrypted by the new keys and sent over the channel, which the server also stores in the database, in the event the recieving user is not currently online.

However, there exists a fundamental issue: maintaining sync betweeen the websockets channel and the messages recieved by the server. As the X3DH protocol Signal employs is largely dependent on the *order* of the messages sent/received, we found that this method can *break* the encryption scheme, and thus the session between the users. When combined with the impermanence of the messages in the database, we find that this is a massive usability issue that cannot be simply addressed.

Here is the original flow: (1) sender sends message via the tRPC API (HTTP), (2) sender confirms success, (3) sender posts to realtime channel, (4) sender confirms success, (5) recipient receives message (5.1) if online, via the realtime channel, (5.2) otherwise, via initial message fetch, (6) recipient deletes message from DB.

Using this method, we face not only the issues with sync, we find that *anyone* would be able to "listen" in on the Websockets channel, as the Websocket client is on the client. To mitigate this, we would need to supplant the architecture with an additional server to handle authorized listeners on the websockets channels - essentially, the server would still act as a middle man, rather than letting clients message each other directly via the protocol. Unfortunately, this was a more complicated architecture to implement with the limited resources we had.

Our solution was then to implement realtime subscriptions to Postgres changes. We took Supabase's approach to establish a realtime server based in Exilir, in which users can subscribe to over a WSS channel. [15] This would then listen to changes by reading on the Write Ahead Log [16], which would enable much faster and resilient performance based on changes to the DB without waiting for the change itself due to the ACID compliance of databases. This would also enable us to double up on the authorization procedure using database policies (row level security), such that unauthorized users cannot read or subscribe to the changes in the database.

This flow would then look as follows: (1) sender sends message via tRPC (HTTP), (2) sender confirms success, (3) recipient receives message (3.1) if online, via subscription to db change, (3.2) otherwise, via initial message fetch, (4) recipient deletes message from DB.

Naturally, this has the drawback of not being able to send messages over WSS, which has us essentially mimicking server-sent events [17]. However, the added resilience here from data arbitrage makes this an effective solution for our needs.

VI. Additional Discussion

A. Password Authentication

Password authentication is a fundamental and widely used mechanism for authenticating users on many websites and applications. Via this method, users store their unique username and password combination to gain access to protected resources (in this case, messages and user data). It does, however, have many limitations, and has been strongly recommended against in many security-forward groups, including the NextAuth team [13]. Many modern approaches towards strengthening security in this area include using two-factor authentication (2FA), magic links, and passwordless open authorization (OAuth), all which provide additional security against attacks, including socially engineered attacks such as phishing.

Despite this, we chose to use a standard username/password authentication due to our prioritization of anonymity. Through other methods, we would have to obtain personal information like emails or phone numbers. While this can be mitigated via the same hashing/encryption protocol we explored for usernames, we wanted to minimize as much direct connection to users as we could.

B. Group Chats

We explored several options for doing group chats in our application. We initially attempted to fit an encryption method to our existing data model, which included conversations and participants as tables to enable one-to-many relationships between users and messages. However, this raised significant challenges in our research in finding methods of creating group keys via the Signal Protocol library, in order to do some form of Diffie-Hellman key exchange so every user in the pool would have the same symmetric key which could be used to encrypt messages. The alternative method was simply creating messages for each pair of users in the group chat and having users send N-1 messages to every other user. The advantage of Diffie-Hellman is that only one message would have to be sent [22].

Two other things we considered when making our decision was the complexity of adding and removing users from group chats and the security of the solutions. In the pairwise solution, it's as simple as altering the metadata a little and sending the message to one or fewer users. For Diffie-Hellman, you have to do another round of key exchanges which requires all users to be online or wait for all users to be online at some point. This greatly hinders the usability of group chats, especially in a limited-usage environment.

Furthermore, the pairwise solution was an extension of normal 1-1 messaging, with all of the group chat functionally being added on the client side. This means any security

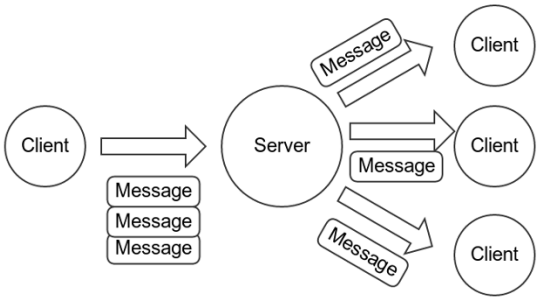

Fig. 4. Pairwise Group Message Distribution [19]

functionality we have one on one chats (forward security, sealed sender, etc.) is still valid. Another side benefit is that the server is unable to see the users in a group chat (since it would look like normal 1-1 communication). However, for multi-party Diffie-Hellman, any security benefits would have to be re-implemented as it is a functionality that is separate from the 1-1 messaging.

So, in the interests of simplicity and saving development time, we have decided to move forward with using pairwise to create group chats, revising our data model to our current implementation in Fig. 1.

C. Implementation Challenges

When first researching the Signal Protocol, we misinterpreted the usage of PublicPreKeys and IdentityKeys, thinking that PublicPreKeys were supposed to be used on a message-to-message encryption basis. We will simply refer to this as "per-message encryption". It turns out, one of the earliest versions of the protocol actually encrypted each message in this manner, but more mature versions use the 'double ratchet' protocol [18]. At a high level, the 'double ratchet' protocol uses each message to 'piggyback' extra data that allows users to establish a new message session key every back and forth, ensuring that keys stay fresh and can't compromise the entire conversation. Our misinterpretation of the protocol had a number of security and implementation ramifications.

Technically, this would provide each message with a stronger notion of security than the one provided by the original asynchronous protocol. In the double ratchet, every 'group' of messages sent by a user can be decrypted by compromising one key, whereas every message has a new key in this scheme. However, these keys are relatively expensive to generate and are consumed at a much greater rate. Messaging also becomes impossible once these keys are exhausted.

Since all PublicPreKeys could potentially be used to encrypt a message, the receiver must check the message against all their secret keys to decrypt the message. This increases the computational cost for little benefit.

The implementation for per-message encryption is actually simpler than the double ratchet, due to the fact every message from the user to the receiver can be treated exactly the same, and since the receiver does not have to worry about double

processing messages. However, we still have vulnerabilities in messages via timestamps and the message type.

D. Denial of Service (DOS)

DOS is when a user is unable to fully utilize the service because there is a malicious actor denying their usage. In our service, the primary ways of DOS are related to keys, or messaging.

PublicPreKeys in the "per-message encryption" are required to send a message. Since these can only be used once, a malicious actor can exhaust all of these keys, denying messages from being sent until the keys are refreshed when the user becomes active. In the double ratchet, this is less of an issue since once a conversation is started no further keys are required, but an attacker could still deny starting conversions by exhausting all keys.

Our current implementation of the double ratchet is fairly dependent on the order of messages received, and ensuring all and only messages from the sender are received. Since our "inboxes" are open for anyone to mutate, this would allow an attacker to insert a malicious message, breaking the session and forcing the users to start a new session. Combined with the previous key exhaustion technique, this an also allow an attacker to deny attacks to a user.

E. Local Storage Limitations

Currently, we're using local storage to store both keys and messages. However, web storage (except HTTP-only cookies) are vulnerable to XSS attacks. Any website could access and retrieve sensitive messages. Storing unencrypted information is the worst case scenario here, hence we introduce symmetric encryption on local storage. However this holds two main problems: (1) a malicious user could attack the recipient by scripting a deletion the local storage of the recipient, and (2) the performance degradation with the ever-increasing size of messages due to serialization. Local Storage has a hard limit of 5MB in Chrome, so eventually, we have to either prune old messages or store them in some cache, encrypted behind some user-chosen password or generated secret key.

VII. Future Work

We unfortunately were not able to accomplish some of the core features we set out to complete at the beginning of the semester. We are hoping to expand the work done here, and improve the core UI to ensure completeness of the work. A discussion on some areas of future work follows:

A. IndexedDB

A way to mitigate many of the issues raised by local web storage is by using IndexedDB - this would provide significant performance improvements due to not requiring serialization of JSON, and being able to selectively get messages locally by timestamp. We would still maintain encryption, and many IndexedDB wrappers like Dexie.js have open-source implementations which improve the DX of working with IndexedDB. This would also prevent RAM overload on the browser tab due to the amount of data from a large amount of messages. We can also introduce asynchronous (non-blocking) storage with this as well, improving UX.

B. Encrypted File Sharing

We would also be able to introduce encrypted file sharing using an S3 bucket and some form of byte-level AES encryption on the BLOB. The encrypting key can be randomly generated and sent within the message content along with the S3 bucket link, and the content type would then determine how to decrypt and handle/display the file.

C. Rate Limiting

Naturally, a big risk in our current application comes in the form of bot attacks and DOS. An attacker would be able to reasonably overload the server or future clients (due to the bottlenecks of decryption explained), thereby breaking the service. A rate limit would eliminate this risk entirely, which we were not able to fully implement in our current iteration. We would add several forms of rate limiting, including user-based, IP-based, and geographic-based rate limiting using Redis as the intermediary store. This has the added benefit of preventing cracking attempts on usernames.

D. Transfer Devices

Similar to rate limiting, we can utilize Redis and AES encryption to enable transfer of data between browsers. While we don't want to permanently store private user data on the server, we can temporarily store it when the user requests a transfer. The following method is heavily inspired by Andreas Thomas, an engineer at Upstash who developed github.com/chronark/envshare.

Flow: (1) Extract (encrypted) local storage associated with a user, (2) encrypt again using a randomly generated secret key, (3) post to Redis database along with userId, (4) generate link for data access that includes the secret key, (4.1) generate tinyurl from link?, (5) prompt user login, (6) upon login, prompt user to confirm device sync/transfer, (7) upon confirmation, load encrypted data and decrypt using the secret key, (8) store into local storage, (9) delete from Redis.

VIII. Current Work Done

We were able to achieve the core functionality of our application. Since the previous update, each person has completed the following:

- Alice, Laura: These two worked on the redesign and completion of the Figma prototypes (we changed direction entirely with the vision and scope after the past redesign). They also worked on researching additional methods with local storage and sealed sender encryptions. They provided significant contributions to the final presentation.
- Jemin, Tyler: These two worked on the restructuring of the application due to schema changes and modularization issues. They also implemented and tested the various encryption primitives and serialization/deserialization methods. They also added the realtime chat functionality

and local storage implementation, though with additional schema changes from adding sealed sender it ended up breaking the app.

Our work for the demo can be found here: github.com/Chickenislife6/saba-secrets/tree/chat-implementation. The complete project will eventually be found here: sabasecrets.dev

REFERENCES

[1] "Why is there no web client for signal," Stack Overflow. [Online]. Available: https://security.stackexchange.com/questions/238011/why-is-there-no-web-client-for-signal

[2] "Signal Messenger: Speak Freely," Signal Messenger, 2013. [Online]. Available: https://signal.org/docs/

[3] "About end-to-end encryption," WhatsApp. [Online]. Available: https://faq.whatsapp.com/

[4] "How does end-to-end encryption work?," Facebook. [Online]. Available: https://www.facebook.com/help/messenger-app/786613221989782

[5] C. Hoffman, "Apple's iMessage Is Secure … Unless You Have iCloud Enabled," How-To Geek, Jul. 30, 2021. [Online]. Available: https://www.howtogeek.com/710509/apples-imessage-is-secure...-unless-you-have-icloud-enabled/

[6] "Raw Database Access," Primsa.io, [Online]. Available: https://www.prisma.io/docs/concepts/components/prisma-client/raw-database-access [Accessed: 06-Apr-2023].

[7] "Password Hashing Competition," password-hashing.net, [Online]. Available: http://www.password-hashing.net/ [Accessed: 06-Apr-2023].

[8] "Argon2," Github.com, [Online]. Available: https://github.com/p-h-c/phc-winner-argon2 [Accessed: 06-Apr-2023].

[9] "Password Storage Cheat Sheet," OWASP Cheat Sheet Series, [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password-Storage-Cheat-Sheet.html [Accessed: 06-Apr-2023].

[10] "Using HTTP cookies," mdn web docs, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies [Accessed: 06-Apr-2023].

[11] "Cross-site request forgery (CSRF)," PortSwigger, [Online]. Available: https://portswigger.net/web-security/csrf [Accessed: 06-Apr-2023].

[12] "Signal Protocol - Better way to generate one-time pre keys (OTPK)," Stack Exchange, [Online]. Available: https://crypto.stackexchange.com/questions/71511/signal-protocol-better-way-to-generate-one-time-pre-keys-otpk [Accessed: 06-Apr-2023].

[13] "Credentials," NextAuth.js, [Online]. Available: https://next-auth.js.org/providers/credentials [Accessed: 06-Apr-2023].

[14] "The Double Ratchet Algorithm," Signal, [Online]. Available: https://signal.org/docs/specifications/doubleratchet/ [Accessed: 06-Apr-2023].

[15] "Supabase Realtime," Github, [Online]. Available: Shttps://github.com/supabase/realtime [Accessed: 06-Apr-2023].

[16] "Write-Ahead Logging (WAL)," PostgreSQL, [Online]. Available: https://www.postgresql.org/docs/current/wal-intro.html [Accessed: 06-Apr-2023].

[17] "Using server-sent events," MDN Web Docs, [Online]. Available:https://developer.mozilla.org/en-US/docs/Web/API/Server-sentevents/Usingserversentevents [Accessed: 06-Apr-2023].

[18] "Forward secrecy for asynchronous messages," Signal Messenger, [Online]. Available: https://signal.org/blog/asynchronous-security/ [Accessed: 06-Apr-2023].

[19] "Pairwise Groups," Signal, [Online]. Available:https://signal.org/blog/images/groups-pairwise.png [Accessed: 06-Apr-2023].

[20] M. A. H. B. Azhar and T. E. A. Barton, "Forensic Analysis of Secure Ephemeral Messaging Applications on Android Platforms".

[21] "Legal Messages Privacy," Apple Legal. [Online]. Available:https://www.apple.com/legal/privacy/data/en/messages/

[22] "T. Slide, "Challenges in E2E encrypted group messaging - tjerand silde." [Online]. Available:https://tjerandsilde.no/files/GroupMessagingReport.pdf. [Accessed: 06-Apr-2023].

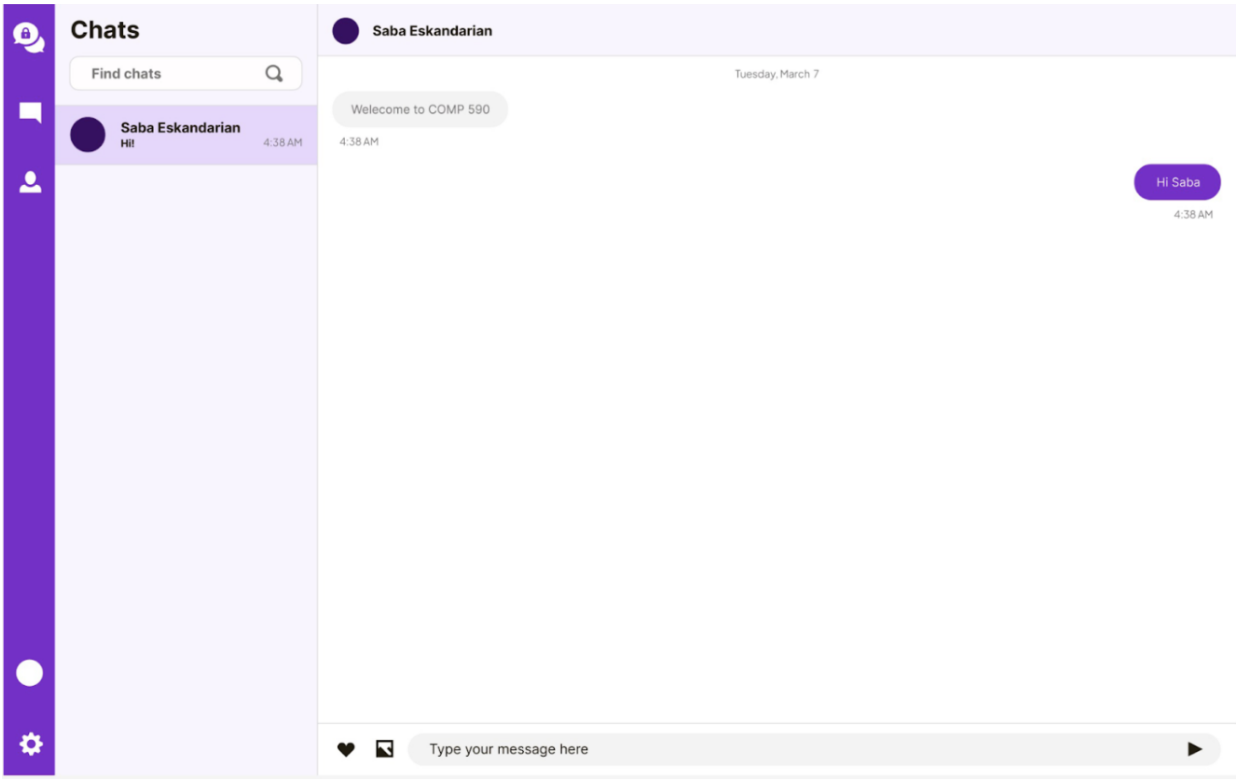

Fig. 5. Chat UI

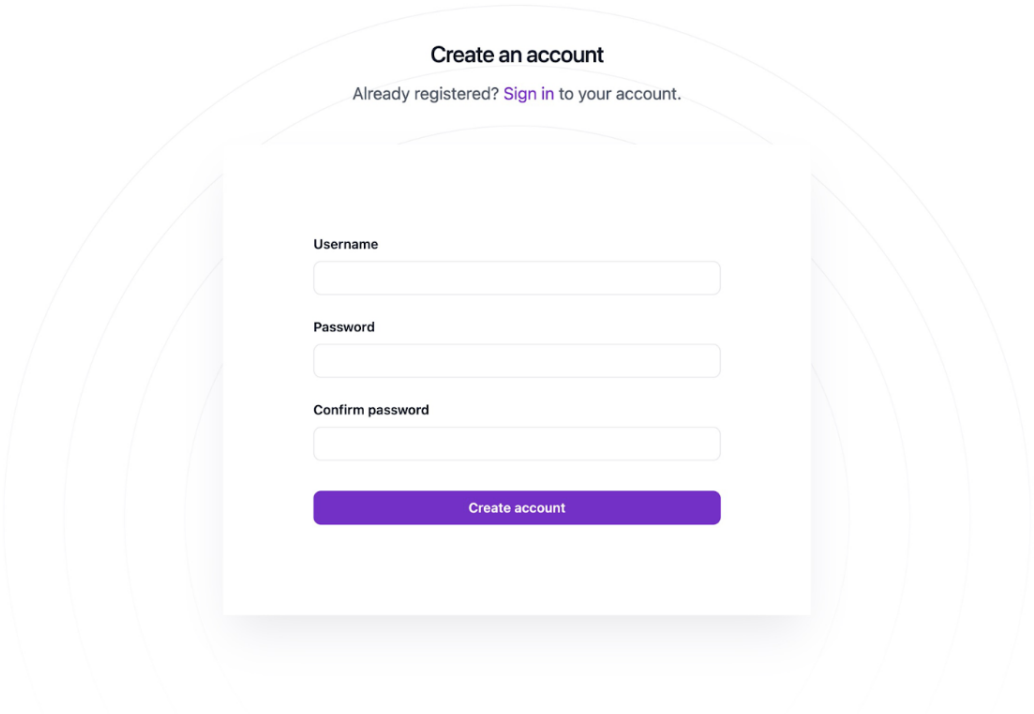

Fig. 6. Sign Up UI