Messaging Encryption With Ephemeral Limits

Emiliano Huerta
ehuerta@macalester.edu

Ned Mayo
emayo@macalester.edu

Jiaying Wu
jwu4@macalester.edu

James Yang
cyang4@macalester.edu

1 Introduction

1.1 Motivation

To be alive in the 21st century is to be online. Having a smartphone is a requirement to participate in society and with that comes digital messaging. The World Economic Forum estimated in 2017 that 16 million texts are sent every second in our world [20]. These messages hold our most sensitive and private information such as social security numbers, embarrassing or compromising photos, and confidential conversations. A simple heuristic to protect all of this information would be to make it available to as few entities as possible.

This is the promise of End-To-End Encryption (E2EE), where within messaging only the sender and recipient have access to the information sent. E2EE protects "sensitive subjects such as business documents, financial details, legal proceedings, medical conditions or personal conversations [8]." An E2EE system also eliminates the possibility of information being leaked and or being seen by someone who it was not intended to because they have no access to the private and public key to decrypt the message. Although E2EE has shown many privacy and security benefits, compared with traditional communication protocols, many general users find it difficult to identify and use E2EE tools correctly [10]. Thus, it is crucial to not only bring end-to-end encrypted technologies to the public but to be transparent and intentional on what it means for the user.

1.2 Project Overview

What we aim to do is not to reinvent encryption but to bring end-to-end encryption to the web application space where it is far less robust in comparison to mobile development and (2) inform the users of the importance of these technologies being employed through the user interface (UI) and experience. Our application will be a web-based instant messenger that allows for private and group messaging. Notably, a few chat applications have already incorporated E2EE such as Telegram, WhatsApp, Signal and many more. Their success suggests a growing priority to communicate securely; however, you must (1) enter their environment to receive that encryption and (2) use their separate mobile app to communicate. Our implementation would allow users to quickly hop from an unencrypted platform into our encrypted messaging rooms for a short while.

We will analyze the existing landscape of encryption technologies and implementations to inform our own. To achieve this encryption, we will use existing algorithms to generate secure keys and hashes. Furthermore, it is important to not require extra effort out of a user to encrypt their messages. Therefore, a goal of ours will be to reduce user burden and to handle as many processes such as encryption and decryption in the background of the application. Managing this and clearly communicating these

technologies will be a balance we must evaluate. Although here the user would not be responsible for managing encryption and decryption; the process will be handled in the background to avoid unnecessary burden on the user. As for the ephemeral component of the messages, we would like to explore methods to temporally limit messages to prevent the accumulation of sensitive information, even if it may be encrypted.

2 Related Work

This section discusses the current landscape of secure messaging protocols used in messaging encryption. We will pay significant attention to Signal Protocol, specifically the techniques and algorithms it uses to facilitate E2EE.

2.1 Signal Protocol

Signal Protocol is the most used chat protocol across instant messaging services. Unlike email protocols, chat protocols historically have emphasized synchronous messaging, meaning users would need to be active in a chat for it to communicate. While the line between email protocols and chat protocols has blurred, some modern chat protocols overwhelmingly use asynchronous messaging, meaning messages can be sent and received whenever. Before diving into the Signal Protocol, we will first detail the development of chat protocols.

In 2004, XMPP (eXtensible Message and Presence Protocol) was made the IETF (Internet Engineering Task Force) standard for chat. It aimed to provide a technology for the asynchronous, end-to-end exchange of structured data by means of direct, persistent XML streams among a distributed network of globally addressable, presence-aware clients and servers [15]. Its extension, the OTR (Off-the-Record), also provides more security in messaging encryptions than PGP (Pretty good Protocol) which was created in 1991 to add end-to-end encryption to emails [6]. It offers deniable authentication for users and does not have long-term public keys that can be compromised [6]. While the OTR is fairly secure in encrypted messages, it only supports messages for chats between two people.

Fortunately, in 2013, the Signal Protocol developed 2013 by Open Whisper Systems solved that problem. The Signal Protocol provides end-to-end encryption for group chats. Based on the OTR, the Signal Protocol also has the property of ephemeral key exchange [6]. In addition to that, it also offers "support for asynchronous messaging and group messaging, going above and beyond OTR by allowing clients to be offline [6]".

Today, most instant messaging services, such as WhatsApp, Signal, and Facebook Messenger, adapt the Signal Protocols to secure messages. Specifically, the Signal Protocol performs good messaging security as it (1) has Post-Compromise Security such as forward secrecy that secures past messages even if the public keys are compromised, (2) outputs keys indistinguishable from random values, which in this case makes sure the public keys are extremely difficult to guess, and (3) shows plausible deniability so that you can never know that a party sent a message to the other party [18].

2.2 Public Key Cryptography Technique

Public key cryptography (PKC) is the encrypting approach used in the Signal Protocol. PKC, conceptually, can be understood as having a public key that is known to everyone and a private key that is only known to the recipients. The public key is used to encrypt messages and the private key is used to decrypt the encrypted messages [1]. PKC is so powerful because (1) it increases the security by preventing messages from being seen by a third party because the private key does not need to be shared and (2) it is able to achieve non-repudiable cryptography service [1].

Today, there are four algorithms are commonly used to implement Public Key Cryptography: Rivest-Shamir-Adleman (RSA), Elliptic Curve Digital Signature Algorithm (ECDSA),

Digital Signature Algorithm (DSA), and Diffie-Hellman Key Agreement Protocol [14]. By comparing them, we can learn where and when they are used. RSA is able to support digital signatures and data encryption, and it is by far the most widely used among the 4. Compared to RSA, ECDSA is able to produce shorter keys of comparable strength than the longer ones RSA produces; DSA only supports digital signatures [14].

2.3 Algorithm

Although there are many strong algorithms used to achieve PKC, we focus on the ones that are more relevant to our work.

Extended Triple Diffie-Hellman (X3DH) As the standard PCK technique is used in the X3DH, it is designed for asynchronous settings where one user is offline but has published information to a server, and another user wants to use that information to send encrypted information to the offline user. This process also establishes a shared secret key to be used for future information shared between the users [12]. In a well-developed secure system, we need "Identity Keys to help with identifying where the message came from; signed Keys to verify that only the user can control his/her respective identity key, and one-time prekeys to make sure that no one can replay attack the user by sending the whole conversation again later [9]".

Double Ratchet Mechanism The Double Ratchet Algorithm is used to exchange encrypted messages between two parties based on a shared secret key. The parties use some protocol, such as X3DH, to agree on a shared secret key and then use the Double Ratchet to deliver and receive messages [13]. The main reason why a Double Ratchet is necessary for messaging encryption is because we want to update the keys for every message regularly [9]. Since asynchronous messaging is very common today, it is easy for a message to sit in the server when the recipient end is offline. Without updating the messages regularly, we are exposing those encrypted messages to great risks. The Double Ratchet Algorithm can achieve key-update because of its property of the KDF chain. The KDF function is a cryptographic function that intakes a secret and random KDF key and then returns output data. It is used when some outputs from KDF are used as an output key and some are used to replace the KDF keys [13]. Those replacements of the KDF keys can be used with another input.

Double Ratchet can be seen as the combination of Symmetric-key Ratchet and Diffie-Hellman Ratchet. Symmetric-key Ratchet calculates the next chain and message keys from a chain key. While DH Ratchet updates the chain keys based on the DH result to prevent attackers from stealing the chain keys and decrypting all future messages [13]. To simplify the procedure of the Double Ratchet, we can understand that the algorithm uses Symmetric-key Ratchet to derive the message key when a message is sent or received. When a new ratchet public key is received, it uses DH Ratchet before Symmetric-key Ratchet to replace the chain key [16].

2.4 Current Implementation

In this section, we discuss modern secure chat applications like Signal, WhatsApp, Telegram, and a chat application developed on Android for research purposes.

Signal uses the Signal Protocol (Axolotl Protocol), which is also used by a number of messaging applications like WhatsApp, Google Allo and Facebook Messenger [2]. The most outstanding property provided by Signal Protocol is Post-Compromise Security (PCS).

Telegram, unlike WhatsApp, iMessage and other popular messaging applications that implement Signal Protocol, uses a self-designed protocol, MTProto. MTProto provides client-to-server encryption in cloud chats and end-to-end encryption in private chats between two devices [10]. Because it is a relatively new encryption protocol, not a lot of investigation on its security

and reliability has been conducted. Lee et. al has explored two vulnerabilities of MTProto in Telegram 2.7.0. The two main vulnerabilities are vulnerabilities in semantic security (IND-CCA) and integrity of ciphertext (INT-CTXT). Because of those weaknesses, there are two known vulnerability attacks against Telegram: Padding Length Extension and Last Block Substitution.

3 Methodology

Our project was developed off of Yaokun Lin's Realtime Chat Application [7]. React.js, Express.js, Node.js and Socket.io underlay Lin's project. To encrypt and decrypt messages sent to and received from the server, we implemented the Web Cryptography API. We also made improvements to the UI to further enhance the user experience and security of a web-based messaging application.

3.1 Encryption & Decryption

In this section, we will describe what web encryption and decryption method we have chosen, why we used it, and how we implemented it.

3.1.1 Web Cryptography API

The Web Cryptography API is a low-level JavaScript API that allows its developers to interact with cryptographic key material. It offers web applications operations such as hashing, signature generation and verification, and encryption and decryption without requiring the raw key material. Those cryptographic transformations are managed by the SubtleCrypto interface, which defines a number of methods for performing common cryptographic operations [19].

We chose Web Cryptography API for both the security and compatibility it brings to our app, but it must be noted that it is the low-level API that World Wide Web Consortium (W3C) recommends for increasing the security of web applications. Since our chat application is composed in React.js, which is a JavaScript Framework, the Web Cryptography API naturally fits in our web application.

Being a low-level API, the Web Cryptography API is not easy to implement. Yet by researching a number of different web encryption and decryption APIs, we found out that they were all built on the Web Cryptography API. Even though those high-level APIs could potentially be easier to use, they lack flexibility and security because they simplify the process of generating the keys and other security factors such as the unique initialization vector (IV) and the algorithms used.

Therefore, taking into account the factors of security and compatibility with our application, we chose the Web Cryptography API as our encryption and decryption tool.

3.1.2 AES-GCM (AES in Galois/-Counter Mode)

Another developer choice we had to make was the algorithm we wanted to use from the Web Cryptography API for our key generation, encryption, and decryption. The Web Cryptography API provides a set of algorithms to choose for cryptographic operations. Specifically, it provides RSASSA-PKCS1-v1_5, RSASSA-PSS, RSAES-OAEP, ECDSA, ECDH, AES-CTR, AES-GCM, SHA, HKDF, and a number of other algorithms. Before justifying our choice of AES-GCM, we would like to introduce this algorithm.

AES-GCM is the abbreviation for Advanced Encryption Standard in Galois/Counter Mode. AES is a specification of electronic data encryption, which was established by the U.S. National Institute of Standards and Technology (NIST) in 2001 [3] and adopted by the U.S. government since then. Not only approved by the federal government, AES has also been approved by the U.S. Secretary of Commerce and the U.S. National Security Agency (NSA). Technically, AES is a symmetric-key algorithm which means that we are using the same key for both encryption and decryption. GCM is the mode of operation

of the AES [5], which offers security because (1) it generates a unique initialization vector and key for encryption and decryption and (2) it ensures the key is freshly generated [11].

Due to its security and reputation, AES-GCM is ideal for our web application and we use Web Cryptography API to implement it.

3.1.3 Implementation

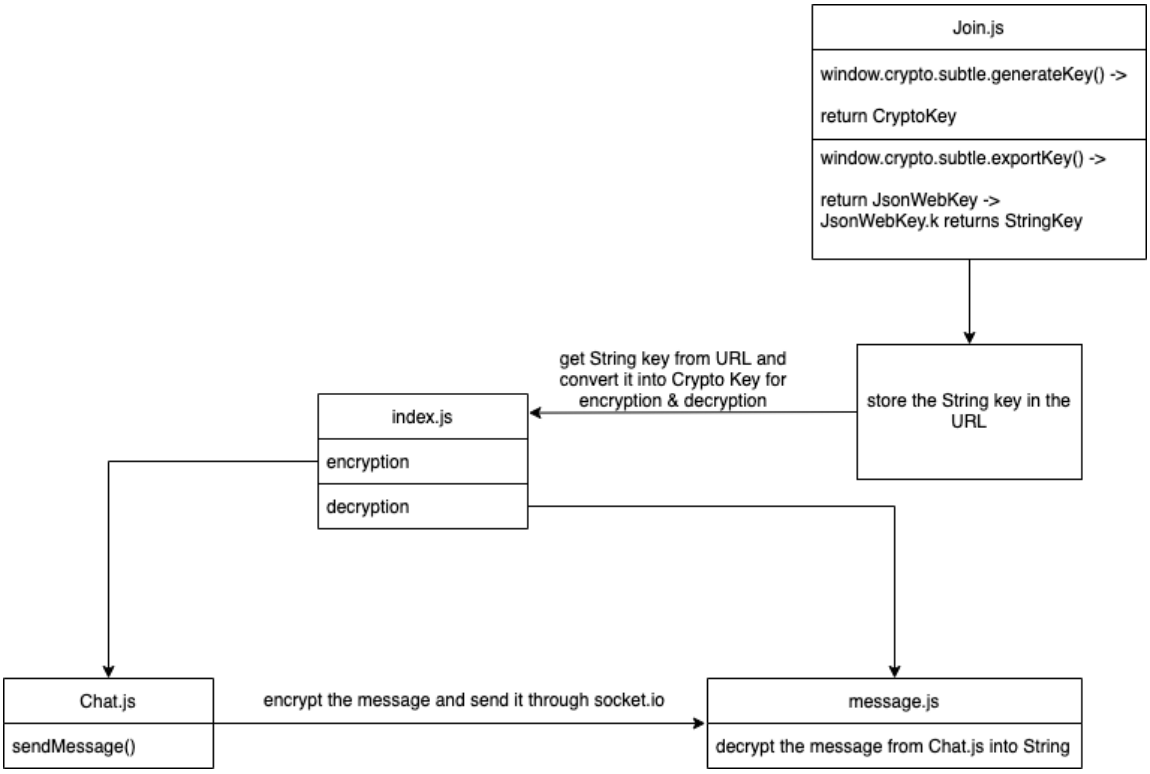

Figure 1: Flow of encryption and decryption in the code

We placed the encryption, decryption, and getCryptoKey methods into a file called index.js. Our encryption method requires a string text and a string key and returns an array of an array buffer and an IV. The encryption method intakes a string text and converts it into Uint8Array-typed data. With the help of getCryptoKey(StringKey), we are also able to convert the string key into a crypto key. The decryption method requires an array buffer message, a string key, and the IV. Thus, before using the data sent from Chat.js, we need to convert the data into an array buffer message and a Uint8Array IV.

Our encryption and decryption take place between Chat.js where messages from a client are sent to the socket server and message.js where messages are received from the socket server. We call the encryption method from index.js in Chat.js and convert the string text into an array of an array buffer (this is our encrypted message) and initialization vector (IV) — we bundle them into an array because the IV should be unique for each encryption. Because the object sent to the server will be converted into String automatically, to make sure the array buffer and IV can

be successfully transmitted to the message.js, we convert them first into a string and send it to the server.

```
    const encryptionUnit8ArrayInString = `${new Uint8Array(
      encryptionPackage[0]
    )}`;
    const ivInString = `${encryptionPackage[1]}`;
    const StringToSend = encryptionUnit8ArrayInString + "iv:" + ivInString;
    console.log(
      "test encryption array buffer and iv in string:" + StringToSend
    );
    socket.emit("sendMessage", StringToSend, () => setMessage(""));
  }
```

Right after that in the message.js, on the client end, a text of type String is received. According to the Web Cryptography API, the content to decrypt should be an array buffer and the IV should be a Uint8Array, before we do the decryption, we need to convert them into those types first. After converting, since the key is in the URL, we can simply decrypt the encrypted message and display it on the interface.

3.2 Software Implementation

We implemented our encryption and decryption algorithms entirely in JavaScript using the Node.js Framework, which provided us with a Cryptographic library to implement AES-GSM encryption. Our UI, composed in React.js, allows us to construct functional and reusable components being able to pass our encrypted and decrypted data directly to these components to be displayed to our users.

Our project has two different aspects that come together to make our application function: the client-side, and the server-side. The client-side is the UI that enables user interaction. Within the server-side, we utilize Express.js and Socket.io to construct and manage our server alongside the information being passed to it from the UI. Using Express.js we are able to deploy a server to handle the necessary data that is being passed from the client-side to the server-side such as the data associated with encryption and decryption. To introduce a messaging aspect to our project we implemented Socket.io, a JavaScript library for real-time web applications, which by definition enables low-latency, bidirectional and event-based communication between a client and a server built on top of the WebSocket protocol [17]. We decided to use Socket.io over directly using WebSocket due to the functionality that Socket.io provides such as broadcasting to multiple sockets, in this case users, and managing the handshake necessary for the client to communicate with the server. On the server, there are certain events that it is listening for such as: connect, join, send message, and disconnect. Each of these events corresponds to a process that may occur on the client-side, where a user can connect to the server, join a chatroom, send a message, and disconnect all of which must be handled on the server-side. Once the server is deployed, it awaits the necessary information from the endpoint that the UI is being hosted on to conduct the necessary processes required. For testing purposes, our server was run locally, however, we have deployed via Heroku. The addition of Heroku as a party privy to the messaging may seem to jeaprodize the security of our app, but the messages heroku passes will be hashed indistinguishable from random values.

On the client side, our application consists of three pages, a home screen that prompts the user to enter a name before chatting, a join screen that invited users will be sent to, and a chat screen that allows the users to send and receive messages. Figure 2 displays the home

screen UI and Figure 3 displays messaging interface. It must be noted that after 10 minutes the socket will disconnect and the user will be redirected to the home screen. After every user is removed from the room, Socket.io destroys the data locally associated with users, message data, and the unique room id that was created during those 10 minutes. The data that passes through heroku never was stored, so there is no added risk to the server.

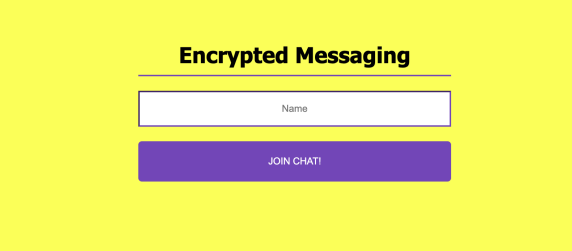

Figure 2: Home page

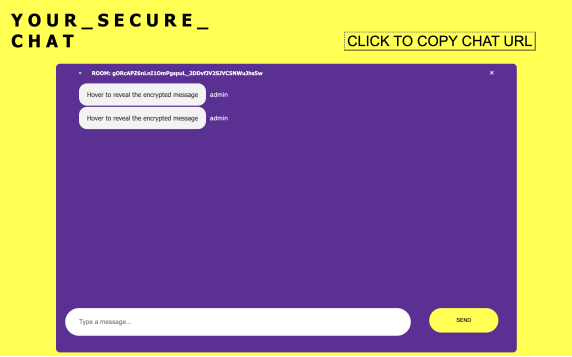

Figure 3: Messaing page

3.3 User Experience

As the realm of security and cryptography is very field specific, the user experience put-forth attempts to ameliorate the disconnection between users and their depth of understanding in managing their own security. For simplicity, users are not required to maintain their own encryption keys nor required to know when to use said keys in the process of encrypting and decrypting information. Rather, we manage every process on the backend, server-side, to ensure that users can use our application without hiccup.

A prevalent issue in applications that promote security, such as our own, is simplifying the required processes to encrypt and decrypt information. Complex and non-intuitive applications may deter users from utilizing the benefits that said application provides. As developers, our users are not at fault for poor design, rather we must prioritize clarity and simplicity of our application to ensure any individual is able to navigate, understand, and effectively communicate through our application.

3.4 User Interface

To construct a UI that is both intuitive and efficient, we constructed a chat feature similar to that of Telegram, Signal, and WhatsApp. The chat feature design is typically standardized across all message based communication, thus modeling our chat feature to what is standard benefits the user in the manner of familiarity. The pre-existing standard of chat UIs aids in promoting an intuitive user flow and effective user queues.

3.4.1 Intuitive User Flow

As users first encounter our application, they must enter their desired name before being placed in a chat room. Once a user is in a chat room, they are presented with a messaging UI, as shown in Figure 3, in which the user can copy their room link to send to others they would like to join as well as send encrypted messages to the chatroom. The user flow from joining to sending messages is simple in the necessary interactions required by the user, where the user is only prompted to enter a name, and later enter the information they would like to send. By minimizing the effort required by users to begin sending encrypted messages, a user does not necessarily need to be aware of how encryption and decryption take place, rather they are able to focus on

communicating securely with the intended recipients that they desire.

3.4.2 Effective User Queues

In conjunction with the intuitive user flow, providing users with concise on screen information further allows them to utilize our application for the intended purpose. As shown in Figure 3, the UI attempts to provide simple yet effective user queues for what must be done while not overwhelming the user with unnecessary or poor information. This appears in our buttons, form placeholder input, and direct components within our UI. For a user to be able to share their unique link with others, users can follow the on screen instructions to copy the link, then send it to the individuals they would like to chat with. To construct an efficient application, developers must consider the perspective of their users and in what ways a user may navigate throughout the application.

3.5 Limitations

Overall, we were limited in what we could achieve with the remaining semester left. We outline here what we failed to accomplish, but with time we hopefully will implement our original goals with the additional insights we've learned through this process.

The major limitation of our encryption and decryption methods was that we used one public crypto key rather than the private and public pair of crypto keys. Even though messaging is significantly more secure than without our level of encryption, we have not yet achieved E2E encryption. The encryption we use is symmetric encryption which is faster because only one key is involved, but it is less secure compared to asymmetric encryption that uses two keys.

Although we made attempts to construct a minimal UI to simplify the user flow process, we were constrained to implementing the most basic chat functions and not allowing everything a modern chat application like WhatsApp can do. Given the time constraint of this project, attempts were made to provide desktop and mobile support, but with more screens to optimize for, more bugs will arise. We have taken a continuous development approach in order to continuously refractor and improve all necessary components in order to be as smooth as possible when switching from desktop to mobile.

Another aspect that would require further investigation is the implementation of ephemeral limits we originally aimed to develop. Our current infrastructure times out users at 10 minutes, but for better integration of time limits we would time out a chat session after a given amount of time. In addition to the already provided encryption, this would function to temporally limit the availability of sensitive messages.

4 Results

In this section, we reflect on the project goals we have achieved as well as analyze the areas in which we could improve moving forward.

4.1 Application and User Experience

Our project is a secure web chat application that provides users with secured chat functionality. The whole application features a simplistic and straightforward interface. Our secured chat application displays three important pieces of information: 1) user name and room key, 2) chat page, 3) link to share with others.

When a user first visits our site, they will be directed to the home screen, and here they are able to provide a name. Once the name value is authenticated, users will be directed to the chat screen with its unique key id.

To accomplish the goal of engaging users with the security of the chat and the idea of encryption, we developed two main things. Firstly, our application includes a user engaged hover over to check each text message process to emphasize the chat message has been encrypted and decrypted, without any data being stored on the

server. When users receive a message from others, they will receive the prompt saying "Hover to reveal the encrypted message," and once they hover over the text, the decrypted message will be shown just as the second text here.

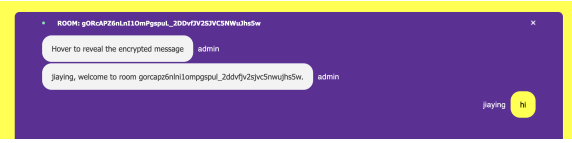

Figure 4: User flow

Our application also includes a function that makes the chat disconnect after 10 mins and redirects them back to the home screen, a similar idea to WhatsApp, to allow users fully engage with our secured chat experience. Users will receive a text from the admin saying "Your Session Will Expire In 10 Minutes" and once the session time is out, users will be redirected to the home page.

Our application also provides "CLICK TO COPY CHAT URL" to allow users to easily copy the current chat room URL to a clipboard and share it with people they intended to invite to the chat. After receiving the chat URL, the new user will be able to see the home screen and repeat the same process to join the chat.

4.2 Reflection

By the end of the project, we successfully bring end-to-end encryption to the web application space where it is far less robust than it is in mobile development. While most of our initial goals and minimal viable product for this application are met, there are many aspects of the application that can be improved on. Our current implementation of Socket.io limits us from implementing the ephemeral limits we originally aimed to develop. We provided an alternative solution by timing out the current user's session. More functionality, such as time being displayed on screen for better user experience, and tailoring our set timeout directly to the room session is needed. More time with the project will allow us to refine and implement all the functionality to allow a user to best understand the encryption processes and manage the participants and ephemerality of the chat session to their liking.

We also want to highlight the difficulty of adding end-to-end encryption and decryption difficulties to a web app using Web Cryptography API and the AES-GCM algorithm. While it does simplify the key generation process a lot, implementing end-to-end encryption in the given time frame eluded us. We remain committed to accomplishing this goal because, as we have described, it offers a significantly higher level of security than public key encryption.

Additionally, because we choose to prioritize encryption and decryption functionalities, to allow us practice writing the AES-GCM algorithm, our web application is a minimal UI and does not implement the most ideal chat applications' features. However, we have achieved our initial goal of creating a secure chat application and have an in-depth understanding of how end-to-end encryption works in real-world applications.

5 Discussion

5.1 Conclusions

During the research phase of this project, it became apparent that there is no "standard" way to implement end-to-end encryption and decryption. Different encryption algorithms prioritize different aspects and use different public and private keys. Most well-known encrypted chats or web applications use different encryption methods, in different languages too. It is interesting to note that in our research of existing implementations of end-to-end web apps we found that many of them were misleading their users about being end-to-end. For example, Excalidraw, an app that claims to be a "Virtual whiteboard for sketching hand-drawn like diagrams. Collaborative and end-to-end encrypted." used only one public key. While learning from these mislabeled technologies misled us as well at the start, this

phenomenon speaks to (1) the difficulty of implementing E2EE on a web framework and (2) a belief that E2EE is an important thing for users even if they are misled that it is implemented. We hope not only to be able to fully implement E2EE but to have an app that makes users aware of how it works. A more informed user base could develop a healthy skepticism regarding security especially as it is clearly being used to generate buzz. In various parts of this work, even we have basically referred to our app as a secure instant messenger. One expert in a study by De Luca et al. equated this term to "snake oil [4]". We and the entire field need to work to ensure our technologies fairly portray their risks and best educate average people about how to understand them.

While developing this web chat application, we also sought to include some educational aspects in it to inform the users of the importance of these technologies being employed through the UI and experience. We would like to have our application serve as a tool to emphasize the importance of security and bring the basic concepts of encryption to the general public with an easy-to-navigate UI. Therefore, in the UI, we include two functionalities, including hover over to decrypt the text, and session timeout, to engage users to participate in the encryption process. Clearly, further development of these ideas and more is needed to best communicate these concepts to a user. We know this limited amount of user testing we've done. Further user testing and development can allow us to shape the most informative and useful experience for a user. Furthermore, we must engage more with the existing cybersecurity research regarding mental models of encryption and security to help us discover the best methods to communicate with general users.

5.2 Future Work

Future work on our application will include refactoring and improving the chat feature, the appearance of UI components, implementation of full E2EE, and extensive user testing. Although the application we have constructed is in many ways what we sought to accomplish, adding further functionality to the chat feature such as a new user flow for direct messaging, displaying the time left on a session, and implementing non-text encryption. We think that improving on the existing functionality would provide users with a more robust and useful experience. Although we utilized direct user testing to improve our UI and user experience, conducting more extensive research on the needs and desires of the users would optimize our application for a more robust audience, allowing any user to simply join and communicate without being deterred due to complex and non-intuitive features.

Future research and extensive user testing would greatly benefit how our interactive encryption and decryption functionalities have influenced users' choices. Further user-testing, in the general public that is beyond people with computer science or cyber security background, would also help us understand what types of interactive encryption and decryption functionalities users feel comfortable using while having fun and learning more about privacy and security.

6 Acknowledgements

Special thanks to Abby Marsh, our cybersecurity course professor, and the other members of Macalester's MSCS department for providing feedback on this work.

References

[1] R Abobeah, M Ezz, and H Harb. "Public-key cryptography techniques evaluation". In: *International Journal of Computer Networks and Applications* 2.2 (2015), pp. 2–15.

[2] Katriel Cohn-Gordon et al. "On Ends-to-Ends Encryption". In: (2017).

[3] Information Technology Laboratory Computer Security Division. *Algorithm registration - Computer Security Objects Register: CSRC*. URL: https://csrc.nist.gov/projects/computer-security-objects-register/algorithm-registration#AES.

[4] Alexander De Luca et al. "Expert and {Non-Expert} Attitudes towards (Secure) Instant Messaging". In: *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. 2016, pp. 147–157.

[5] Morris J Dworkin. *Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac*. National Institute of Standards & Technology, 2007.

[6] Ksenia Ermoshina, Francesca Musiani, and Harry Halpin. "End-to-End Encrypted Messaging Protocols: An Overview". In: *International Conference on Internet Science*. Springer. 2016, pp. 244–254.

[7] Adrian Hajdin. *project_chat_application*. 2019. URL: https://github.com/adrianhajdin/project_chat_application.

[8] IBM. *What is End-to-End Encryption?* URL: https://www.ibm.com/topics/end-to-end-encryption (visited on Nov 4, 2022).

[9] Vertika Jain. *Developing a Real-Time Secure Chat Application Like WhatsApp amp; Signal with end-to-end encryption*. URL: https://www.qed42.com/insights/coe/javascript/developing-real-time-secure-chat-application-whatsapp-signal-end-end.

[10] Jeeun Lee et al. "Security Analysis of End-to-End Encryption in Telegram". In: *Simposio en Criptografıa Seguridad Informática, Naha, Japón. Disponible en https://bit. ly/36aX3TK*. 2017.

[11] Stefan Lemsitzer et al. "Multi-gigabit GCM-AES architecture optimized for FPGAs". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2007, pp. 227–238.

[12] Moxie Marlinspike and Trevor Perrin. "The X3DH Key Agreement Protocol". In: *Open Whisper Systems* 283 (2016).

[13] Trevor Perrin and Moxie Marlinspike. "The Double Ratchet Algorithm". In: *GitHub wiki* (2016).

[14] *Public Key Algorithms*. Mar. 2021. URL: https://www.ibm.com/docs/en/zos/2.3.0?topic=hierarchies-public-key-algorithms.

[15] Peter Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core*. Tech. rep. 2011.

[16] Hengki Tamando Sihotang et al. "Design and Implementation of Rivest Shamir Adleman's (RSA) Cryptography Algorithm in Text File Data Security". In: *Journal of Physics: Conference Series*. Vol. 1641. 1. IOP Publishing. 2020, p. 012042.

[17] Socket.IO. *Introduction*. Nov. 2022. URL: https://socket.io/docs/v4/.

[18] Ferran Van der Have et al. "The X3DH Protocol: A Proof of Security". In: (2022).

[19] *Web cryptography API*. URL: https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126/.

[20] *What happens in an Internet minute in 2017?* URL: https://www.weforum.org/agenda/2017/08/what-happens-in-an-internet-minute-in-2017.