Sendr: Distributed and Private Instant Messaging

Anonymous Author(s)

ABSTRACT

In the age of the internet, many people have used apps like Discord, Slack, and WhatsApp to stay connected with their friends and family. At the same time, people have been growing more concerned than ever about their personal data being harvested by companies and sold off to the highest bidder. As a result, many chat applications have been implementing security features that protect data from the service provider, as well as from outside attackers. In this vein, this project proposed a distributed messaging system emphasizing privacy through end-to-end encryption (E2EE) of user messages and their metadata. Even metadata including timestamps and destinations can be used to mine a significant amount of information about users. The proposed system mitigated these common privacy concerns with contemporary applications by ensuring both the content of messages and the metadata (including timestamps and sender/receiver identities) are securely encrypted. This would allow users to communicate privately without revealing information.

This project created an application that showcased an implementation of a messaging platform utilizing E2EE of metadata for enhancing user privacy. Although the metadata encryption was not achieved, this modification was thoroughly researched and designed. The application employed the Redis publish/subscribe engine, a referentially and temporally decoupled publish/subscribe model to handle communication across distributed back ends which allow for scalability. Implementing a web front end made portability of the application to several platforms straightforward by allowing any device capable of running a modern web browser to access the application.

KEYWORDS

Private Information Retrieval, Data Privacy, Data Encryption, Data Confidentiality, Cryptographic Techniques, Messaging Applications, End-to-End Encryption, RSA, Public Key Encryption, Shared Key Encryption, Metadata Encryption, Metadata Obfuscation, Horizontally Scalable Systems, PIR Messaging Client, Distributed PIR, Privacy, Security, Distributed Systems, MQTT, Encrypted Data Recall, Performance and Security Trade-offs, Dynamic Load Balancing

1 OVERVIEW

Now that fast and accessible internet is available almost anywhere, private and secure Instant Messaging Systems have exploded in popularity [16]. These applications are relied on for their speed and reliability, often being used to send critical updates about important systems. Other users will feel free to send incredibly personal information, since they believe that their messages are private; only visible to them and their recipients. For most chat systems however, the server has complete and total access to every message, and consequently, all of their users' private information. Popular modern systems such as Whatsapp [7], Signal [6], and Telegram [9] utilize end-to-end encryption to hide conversations from the chat servers (of which clients do not have access) and potential attackers. This represents a significant step up in privacy from traditionally encrypted messaging services, since now the service provider cannot act on data contained within their clients messages.

All of these applications however, fail to encrypt the metadata of messages. This still represents a large concern for user privacy [21]. Metadata can be used to learn a lot about the conversation and build a profile around users [21]. Addressing this concern has been the primary motivation for this project. An application performing end-to-end encryption, including encryption of message metadata, would allow for completely anonymous communication between clients. Some attempts at totally anonymous systems, like private information retrieval (PIR) were attempted. However, they have proved difficult to scale. Tovey et al. [21] proposes a distributed PIR protocol to address this issue by off-loading work to the client.

The goal of this project is to develop an instant messaging system with private-metadata messaging, similar to that created by Tyagi et al. [22]. In order to achieve enhanced scalability, the application will be designed in a distributed manner, taking inspiration from Lu et al. [11] for architecture choices. The messaging functionality will be implemented using end-to-end encryption, meaning the server never knows message contents. The application itself will be hosted through a web-server, which will also be used as a reverse proxy. Traffic will be load balanced and routed through multiple back end servers. We will use a publish subscribe engine to process, replicate and retain messages, with an in-memory data store.

This report is organized as follows: Section 2 discusses related research and industry implementations in the topic of metadata end-to-end encryption. Section 3 details the design and implementation of the application, including the technology stack and tools used. Section 4 presents acquired results, and what was learned from the project. Sections 5 and 6 discuss legal and ethical considerations, respectively, in the context of the project. Finally, section 7 showcases the achieved results and highlights directions for future work.

2 RELATED WORK

This section reviews a number of papers that have presented security improvements to distributed messaging systems. Several solutions are already present in released applications and have been demonstrated to be scalable and feasible. Several other papers implement procedures to obscure and obfuscate metadata generated by sending and receiving messages, but these methods have not been implemented in a widely adopted messaging service. As such, there are significant doubts about the scalability of these systems.

2.1 Solutions with demonstrated scalability

Signal is one of the most widely used applications that features end-to-end encryption, and as such, the methods they use to encrypt and decrypt messages are extremely relevant to this project. All security on the application stems from a device-specific public and private key pair, called an identity key pair [5]. Users may load these keypairs onto different devices to access their communications from

both places. When creating a chat with another user, the public keys are exchanged unencrypted, allowing both parties to encrypt messages to each other. All of the information about each chat, including public keys, is saved on a user's device to ensure that no handshake procedure is needed on relaunch. For group chats or other sessions created with a shared key encryption system, a key ratcheting procedure is used to regularly refresh the key. This ensures that, even if a key is leaked, it will expire shortly after and not cause too much damage. This key ratcheting procedure is initialized based on secret information that was communicated at the start of the chat. This means that it would be difficult, if not impossible for malicious attackers to be able to guess any future keys based on a single leaked one. Signal encrypts user metadata sent to and from the server, but has no process in place to anonymize traffic or further protect user confidentiality. Overall, Signal has a fairly robust but simple end-to-end encryption scheme.

Another app that improved the privacy of their users is SimpleX [20]. Released in 2021, SimpleX has published a white paper that overviews their goals and solutions. They highlight that in widespread chat platforms, users lack privacy, suffer from spam, and overall, their data is not protected. They claim to solve these problems by having no identifiers associated with the user and also only storing data locally. Unlike other platforms, they have no phone numbers, email, usernames, or even random keys. They have no knowledge of the user, so they claim to not even know how many users they have. Malefactors and spammers cannot act, since the only way to be contacted is by sharing a one-time invitation. They achieve this by using unidirectional message queues, the analogy they make is "having a different email address or a phone number for each contact you have." Since SimpleX does not store data, and simply relays messages to other users, it is impossible to access it anywhere except your local device. They compare themselves to services using other protocols, specifically P2P, and highlight that SimpleX is more efficient, communication cannot be tied to SimpleX since it uses standard internet traffic, more secure (resistant to Man in the Middle Attacks, and the fact that it has no discovery mitigates many other attacks), and less likely to be attacked and taken down.

Message Queuing Telemetry Transport (MQTT), a communication protocol used for many Internet of Things applications, follows the publish/subscribe pattern and relies on a single, centralized broker, which is unsustainable for networks designed to serve 10^6 devices per square kilometer. Longo et. al. [10] implement a plugin that converts a single centralized broker into a system capable of being distributed. Routing is based on a spanning tree for each topic that is recreated every time a broker connects or disconnects, with all messages routed through the root broker. As all brokers are only on the routing tables of topics they are interested in, table size is significantly reduced and brokers are not relaying irrelevant information. Longo et. al. found this to reduce the end-to-end delay by greater than 50%, while traffic overhead increased dramatically relative to the number of topics.

The techniques and improvements described above have been shown to be feasible for large, scalable applications. In particular, end-to-end encryption has been shown to be robust and has been thoroughly tested. These methods seem to be practical, reasonable security improvements, and have been implemented in successful commercial messaging systems. In contrast, no metadata obfuscation methods have been implemented in a widely used application. This does not mean that they will never be useful, but simply that they are a little too cumbersome for practical application.

2.2 Metadata obfuscation procedures

The following papers proposed methods to obfuscate metadata generated by the act of sending or receiving a message. These procedures greatly increase the privacy of their users, but come at cost to scalability or latency.

Stadium, a novel messaging system proposed by Tyagi et al. [22], features an interesting methodology to protect the privacy of users. The goal is to ensure that malicious attackers cannot ascertain any information about the messages users are sending, such as the time they are being sent, who they are being sent to, or any other form of metadata. Stadium obfuscates this information by having every client constantly send out fake messages. These messages are then shuffled across all of the different servers. During this shuffling process, none of the metadata related to the messages is ever revealed. After being shuffled, the metadata of the messages is checked, and messages being sent in the same conversation are swapped. Due to the way this was implemented, it is a mathematical guarantee that each message will be swapped to go to its destination at some point during this process. After this, all of the messages are unshuffled and sent back to their original senders. Since some messages have been swapped, those senders would receive a new message instead of the one they originally sent. This procedure makes it extremely difficult for malicious actors to be able to tell where messages are being sent to, since it appears as though all messages are sent from the client back to themselves.

This procedure is extremely interesting, but likely very inefficient. In addition, this would require a significant amount of clients in order to ensure that there are always enough users online to anonymize the path of messages. Additionally, the amount of time needed to perform the procedure being measured in dozens of seconds ensures that this is not a feasible method of mass communication, especially since users can only communicate once per time the procedure is performed. This procedure may be feasible for a highly secure shopping app, where users might only want to make a request every few minutes, but certainly, this latency is far too high for a common messaging application. Removing the shuffling procedure, but keeping the message swapping may be a far more reasonable and efficient method. This would still ensure that malicious attackers cannot intercept messages and track them, and still add a high degree of anonymity to the service.

Private information retrieval, or PIR is a protocol that allows users to access information from a server without the server knowing what information was retrieved. This effectively keeps the metadata of the message secure from even the server, preventing the server from knowing virtually anything about the behavior of the clients. This methodology allows for improved confidentiality of users, but generally comes at the cost of bottlenecking performance on the server since every message must be treated as though it could be sent to every client. Tovey et al. [21] proposes a method to improve the scalability of this method, distributing some of the work to the clients. By using the clients to perform

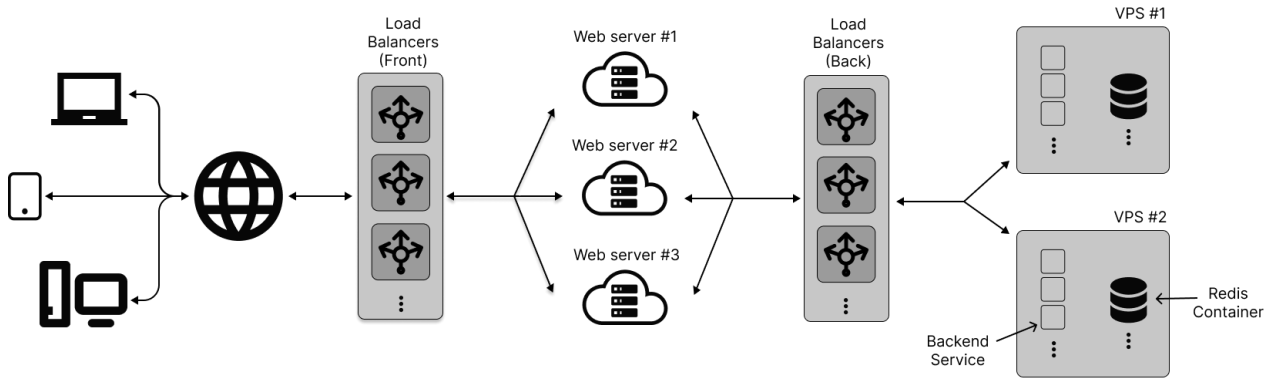

Figure 1: Proposed Deployment Architecture Diagram of *Sendr*. Front-facing load balancers handle load distribution for several web server instances and internal load balancers handle load distribution for an internal network of back end servers.

matrix multiplication, computation power scales with the number of clients. Since the computing power needed also scales with the number of clients, this effectively makes the computation occur in constant time, moving the bottleneck to the speed of communication between all of the clients and the server. While this still represents a significant amount of computation time spent and is reliant on all clients having somewhat powerful processors, the improved scalability makes PIR much more feasible for use than previous implementations. As is, PIR still imposes far too much overhead to be worth considering, but it is interesting to consider.

3 DESIGN AND IMPLEMENTATION

Our project is called *Sendr* - a web-based instant messaging platform utilizing end-to-end encryption of message content. We had also hoped to encrypt metadata, but were unable to implement this during the duration of the project.

The application structure consists of a front end and a back end. The *Sendr* back end can be scaled up to several identical processes across potentially multiple virtual private servers (VPS). This makes up the entire back end service. Redis [17] is used as a pub/sub engine for communication among the instances of the back end and for message retention and replication. Front end clients will be connected to the back end service through an Nginx [13] webserver instance which can be configured to handle load distribution among the several instances of the back end. For the purposes of this report, the application is hosted by a single Nginx webserver on a single machine. However, this is only a limitation of resources and not of the application; given sufficient access to resources, this application could be hosted across several machines and accessed through a load balancing service such as AWS Elastic Load Balancing [19] for scalability. Figure 1 shows a visualization of this proposed deployment architecture. The use of Docker Swarm would allow running multiple instances of the back end across different VPS.

The frontend of the application is developed using the React [12] web framework. It is accessible using any device with access to a modern browser (JavaScript is needed) and the internet. We originally intended for users to be able to send direct messages, create group chats, and view current and past chat history. Currently, they are only able to participate in group chats. All messages sent by the user are encrypted using RSA public key encryption. First, the text of the message is encrypted with the recipient's public key, so that the server cannot see it. Then, the metadata of the message is encrypted with the server's public key and prepended to the message. The server then decrypts this metadata and sends the encrypted message to the correct user. For group communications, a message is sent for each other user in the group chat, encrypted with their public key. Since we are using a pub-sub architecture, each user technically receives all of the message, but are only able to decrypt one. To share keys, every time a user joins a channel, they send their public key in a related channel. Additionally, they automatically check the message history for all previous keys to ensure that they can send messages to all other users in the chat. Users are not shown any messages that are encrypted with a public key other than their own. Unfortunately, this means that they cannot actually read any messages sent from before they joined the chat. We had an idea for users to be able to "reveal history" to new users, but were unable to implement it. This would have involved each user preserving a record of chat history and re-encrypting and sending it to new users.

The back end service is written in the Go programming language and uses the Centrifuge [8] web socket library for web socket communication. Using Redis as a pub/sub engine provides the several back end instances access to a distributed shared space for sending and receiving messages. This back end architecture is encapsulated as the back end service, which acts as a centralized black-box source that a frontend client can communicate with, even though the underlying services may be distributed across many machines. The back end service can thus be configured to use resources on-demand for high-traffic circumstances, that is, back end processes can be scaled up and down as needed. A single back end process primarily uses web socket communication to receive/send chat messages to the frontend clients and HTTP communication for sending and receiving API responses.

4 ANALYSIS

Working on this project provided a very clear glimpse into the real trade offs that must be made when adding security to distributed systems. For example, consider end-to-end encryption. End-to-end encryption acts to protect clients against possible information leaks on the server by keeping their data encrypted whenever the server handles it. The problem comes when deciding how to handle group chats. Using a public key encryption system as we are requires sending a message encrypted with each unique user's public key to the server, so that they can be distributed to recipients as necessary. This means that the amount of computational power and network power spent by a user now scales directly with the amount of people in the group chat. Normally, regardless of the number of members of the group chat, each user would send a message that would be decrypted and re-encrypted by the server. In this scenario, the computing power of the server, not the client, is the bottleneck. For web-based applications ostensibly meant to be used by people on their phones, this is obviously far superior.

Another method of encryption that could have been utilized is shared key encryption, which works when every client shares a single key used for encryption and decryption. Anyone who discovers the shared key can immediately decrypt all messages sent in that group chat. The initial creation of a group chat requires a secure channel to communicate a shared key across, so shared key encryption cannot be implemented in an application by itself. In essence, creating an end-to-end shared key encryption scheme first requires creating a channel with end-to-end public key encryption. Regardless, end-to-end encryption poses a significant restriction on the design and implementation of a project. By itself, encryption already imposes a significant performance overhead on an application. For a chat application, messages are sent infrequently enough that complete encryption is reasonable. Applications that are in constant communication with their servers may not be able to encrypt all information, though. Of possible concern are applications that record their users' habits and impressions. Either the constant stream of data generated is processed locally on the application, or it would be sent, possibly unencrypted, to a server.

Our theorized modification of hiding metadata would have imposed a very small overhead on the server. End-to-end encryption generally means that the server does not have to encrypt anything since the clients do that. The limitation of sending encrypted metadata does mean that they still need to actually perform encryption. This is not a big deal, since servers should generally be equipped to handle encryption tasks.

The scope of work for this project was far broader than what we initially anticipated. We had a number of goals in mind as improvements to make to a chat application, and we did not manage to complete all of them. While the end goals of end-to-end encryption and metadata encryption are reasonable, the amount of work that was needed to create a chat application that could show off these encryption methods was more than expected. Overestimation and overplanning are not inherently bad, but most of our time has been spent on basic infrastructure for a chat application. We also added a number of useful features for users to improve their experience, but these features were not necessary to demonstrate the main research goals of this project. This reduced the amount of time that we could spend investigating encryption or improving server scalability, which are the main areas of focus of our research. We probably did not need to include as many niceties, such as Google authentication or message history as we did, and likely should have chosen a single set of improvements to focus heavily on. Incorporating all of these distinct features into a single application would have taken significantly longer than the allotted time, even for experienced web developers.

That being said, implementing message history did allow us to investigate another interesting side effect of end-to-end encryption. Since the server only receives fully encrypted messages, it must store all messages encrypted. This obviously prevents moderation or the enforcement of a variety of communication laws, but it also affects users. If a user tries to load their message history, they must receive encrypted messages. This is not a problem if their public and private keys are the same. Unless the public and private keys are stored on the device, there is no way of reloading messages, since they would be randomly generated when the application is launched. Additionally, this prevents users from being able to reload message history on a different device unless they transport their key pair over. End-to-end encryption, which is meant to prevent the ways service providers can interact with consumer data also limits the ways in which users can interact with their data.

There are a number of opportunities for future work on this project. Most obviously, actually implementing metadata encryption would be extremely important. One actual improvement could be using shared key encryption for user-made group chats. By automatically creating and sharing a shared key over end-to-end encrypted channels, users could make large group chats that would not impose significant overhead on sending and receiving messages. Another improvement would be implementing the anonymity procedure in Stadium [22]. Finally, there is a significant opportunity for research into end-to-end encryption protocols that would actually enable users to view their message history on any device they choose to log into. Currently, this is one feature of most messaging applications and social media that cannot be replicated by services that use end-to-end encryption. If someone managed to enable this feature, end-to-end encrypted messaging may be more viable than traditional encryption methods.

5 LEGAL CONSIDERATIONS

Since our application features end-to-end encryption, it is subject to some legal precedence that traditionally encrypted apps are not. Some governments may require access to message logs and user information, or to ban specific users from the service. Failure to comply may result in country-wide bans [18], though countries such as the USA have legal protections for preventing access to private information [14]. Other countries are debating a ban on end-to-end encryption services entirely, saying that they prevent the government from being able to track illicit activities. There is a significant amount of legislation on the books that is related to private messaging services, since before encrypted messaging was even being widely used. Back in the day of telephones, the wiretap act [15] carved out the ability for officers of the law to "intercept wire, oral, or electronic communications," an action which would be criminal for the layman. It requires that law enforcement get

a warrant or otherwise obtain authorization before surveilling a person's communications, but it sets legal precedent for following laws. The most recent relevant law passed in the United States codifies this authority into a duty: under the Protect Our Children Act of 2008 (POCA), all electronic service providers, be they a website, messaging service, or other application, are required to report any sharing of child pornography, hold onto such images for evidence, and may be held liable for failing to do so [3]. Applications that have true end-to-end encryption do not allow servers to view the data, and therefore service providers cannot identify illicit materials unless reported by a user. Since the recipient of illicit materials are likely not going to report the material, many argue that apps with end-to-end encrypted messaging should be required to either allow some backdoor access or some other way to moderate content. As of right now, organizations like the FBI do not have access to all message information on encrypted messaging services.

As of 2021, the FBI can only lawfully access the date and time of registering, and the user's last access date for Signal [14]. While this information is certainly relevant and could help investigations, it can still be argued that it is in violation of POCA. I believe that a procedure could be implemented that would change the encryption of certain chats under investigation to be treated as group chats, and thus having a shared key which would be shared with government agencies. While this would not give access to channel history, and thus could not be used to find proof of previous criminal communications, it would allow for future acts to be documented and recorded as proof. For our application, since users register through Google, the only information we are capable of sharing is their Google account, as our authorization key does not allow us to access any other data. We or Google may be capable of sharing registration and sign-in dates or times, but neither of us would have access to any of the information sent on this platform.

In the US, a few pieces of legislation have been proposed that could disincentivize messaging platforms from using end-to-end encryption. One such example is the EARN IT Act of 2023 [4]. This bill is designed to hold service providers accountable for any content on their platforms that features the exploitation of children. Among the provisions in this bill, it mandates that companies are held accountable for failing to properly record and report instances of child exploitation on their platforms. Services that feature strict end-to-end encryption would have to put some measures in place to accept reports and verify their contents, since their servers, by design, do not know the contents of conversations. In fact, end-to-end encryption is specifically named in section 5.7 of the act as something that does not shield the company from liability. Under this act, the provider being unable to decrypt secure communications would not shield them from liability, if they had received any information informing them of relevant abusive content. This law would effectively prevent the development of end-to-end encrypted applications without moderation, as the developers could be prosecuted for developing an app that fails to adequately protect minors.

6 ETHICAL CONSIDERATIONS

The ACM Code of Ethics and Professional Conduct [2] provides a standard guideline addressing ethical concerns relating to the world of computing. This guideline provides the means for computing professionals to ensure ethical standards are being met and is the closest step towards enforcing these ethical standards. The implementation and integration of *Sendr* raises many of these concerns:

- **ACM 1.1 Contribute to society and human wellbeing**: This principle obligates computing professionals to ensure that the work they do benefits society and humanity. *Sendr*, as is aligned with our goals, would provide complete private communication between two or more parties. This can be utilized by people who may wish to disclose information without exposing their identity (e.g. whistle blowers or people living under oppressive governments). While privacy is considered the main goal for this application, and a beneficial feature to implement, it may invite misuse among actors who wish to confide in it for nefarious purposes (such as criminal activity). Without knowing the identity of the application users (should we proceed with a custom authentication method) then it would be extremely difficult for police and investigators to track down these individuals.
- **ACM 1.2 Avoid Harm**: Computing professionals should also be wary of harmful side effects caused by their work. *Sendr* is capable of concealing communication between users, which opens the possibility of harmful conversations. Without having any real way to identify hateful and harmful rhetoric in conversations, it may be necessary to include additional safety settings for users (such as the ability to block other users).
- **ACM 1.3 Be honest and trustworthy**: This principle requires that computing professionals be transparent and honest with the claims they make about their works. *Sendr* is claimed to be compliant with E2EE, so users should be able to validate this claim. One potential option is to make the client program open-source. This would allow individuals to control for themselves that the application uses state-of-the-art encryption methods on message data and metadata, which would forgo the need to trust in the application developers.
- **ACM 1.6 Respect privacy**: It is important for computing professionals to be aware of how the technologies they develop can be a threat to user privacy. It is very easy to collect user data and, in some instances, highly desirable to perform analyses on the data. *Sendr* is intended to be a private messenger application utilizing E2EE and if made open-source, users would have full autonomy over what data about them gets sent and how. The server itself may retain message logs for a short period of time, but given that these messages will be encrypted the actual content would be useless.
- **ACM 1.7 Honor confidentiality**: Computing professionals should respect confidential information about clients unless otherwise required to be exposed to law enforcement. *Sendr*, being a messaging app, will naturally collect incoming messages from users and will temporarily cache them server-side. However, given that the messages utilize E2EE, they are unlikely to be useful. Furthermore, messages will only be retained for a short period of time before being discarded, which would make uncovering past conversations impossible. Should the need arise, *Sendr* will need to provide

whatever data it has to law enforcement, which would include metadata for recent messages and user information (as it stands, authentication is handled via Google, so there is likely to be substantial information pertaining to a user, this is subject to change upon implementing a custom authentication method).

- **ACM 2.1 Strive to achieve high quality in both the processes and products of professional work**: As computer services become integrated into society, it is necessary to ensure that the quality of these works meets conventional security and reliability expectations. *Sendr* uses E2EE in order to provide user privacy, so it is crucial that modern encryption algorithms are used. Third-party libraries should be up-to-date and regularly updated to address discovered vulnerabilities. Regular security audits and consultation with security experts can be utilized to ensure the app remains secure.
- **ACM 2.3 Know and respect existing rules pertaining to professional work**: Computing professionals must understand and recognize and abide by any local, national, and international laws pertaining to their works. Depending on the kinds of regulations that are in place, it may not be legally admissible to provide access to *Sendr* in certain regions.

The IEEE Code of Ethics [1] defines additional professional guidelines in the computing community. As computing professionals we are expected to follow these as well:

- **1. to hold paramount the safety, health, and welfare of the public**: As mentioned previously, works involving sensitive user information should ensure modern cryptographic facilities are used to secure it. *Sendr* being a private and secure system should avoid third-party cryptographic libraries that are out-of-date and/or no longer being maintained.
- **5. to seek, accept, and offer honest criticism of technical work**: Should *Sendr* be distributed as an open-source project, it would be necessary to accept authentic changes made by third party individuals concerning vulnerabilities and bugs.

7 CONCLUSIONS

The project successfully implemented a distributed messaging system that emphasizes security by using techniques such as end-to-end encryption. Users can communicate securely with messages remaining private, decreasing the risk of being exposed to malefactors due to server breaches. While we would have liked to implement metadata encryption or obfuscation, this goal was unfortunately not achieved. The application is scalable and can be used concurrently by a large number of users without compromising security, or incurring significant slowdown. It is also resilient to failure, being able to provide a consistent experience where the user doesn't notice any unusual behavior.

In the future, the scalability of this system should be rigorously tested, it should also be updated as users discover issues. Additionally, instead of using Google or other third-party authentication methods, it may be desirable to implement a custom one which ensures user privacy. It would also be desirable to support shared key encryption to reduce overhead in group chat settings. Overall, we have created a functional chat application that incorporates end-to-end encryption techniques to create a functional and secure experience, with plenty of room for improvement.

REFERENCES

[1] 1984. IEEE code of ethics. *IEEE Transactions on Reliability* R-33, 1 (1984), 14–14. https://doi.org/10.1109/TR.1984.6448267
[2] ACM. 2018. ACM Code of Ethics and Professional Conduct. https://www.acm.org/code-of-ethics. Accessed: 2024-11-20.
[3] U.S. Congress. 2007. S.1738 - 110th Congress (2007-2008): Privacy and Civil Liberties Oversight Board Act of 2007. https://www.congress.gov/bill/110th-congress/senate-bill/1738 Accessed: 2024-10-28.
[4] U.S. Congress. 2023. S.1207 - 118th Congress (2023-2024): EARN IT Act of 2023. https://www.congress.gov/bill/118th-congress/senate-bill/1207 Accessed: 2024-12-2.
[5] Signal Foundation. 2013. Asynchronous Multi-Device Encryption. https://signal.org/docs/specifications/sesame/
[6] Signal Foundation. 2013. Signal: A Private Messenger. https://signal.org/
[7] WhatsApp Inc. 2009. WhatsApp Security. https://www.whatsapp.com/security/
[8] Centrifugal Labs. 2023. centrifuge. https://github.com/centrifugal/centrifuge.
[9] Telegram Messenger LLP. 2013. Encryption. https://core.telegram.org/api/end-to-end
[10] Edoardo Longo and Alessandro E.C. Redondi. 2023. Design and implementation of an advanced MQTT broker for distributed pub/sub scenarios. *Computer Networks* 224 (2023), 109601. https://doi.org/10.1016/j.comnet.2023.109601
[11] Shoulei Lu, Jun Ye, and Zheng Xu. 2023. Analysis of Instant Messaging Systems for Users Based on the Go Language. In *Innovative Computing Vol 2 - Emerging Topics in Future Internet*. Springer, Springer Nature Singapore, Singapore, 638–646.
[12] Meta. 2024. React. https://react.dev/
[13] Netcraft. 2024. Nginx. https://nginx.org/en/
[14] Federal Bureau of Investigation. 2021. FBI's Ability to Legally Access Secure Messaging App Content and Metadata. https://propertyofthepeople.org/document-detail/?doc-id=21114562
[15] Bureau of Justice Assistance. 1986. Privacy and Civil Liberties Authorities. https://bja.ojp.gov/program/it/privacy-civil-liberties/authorities/statutes/1285 Accessed: 2024-10-28.
[16] Media Engagement Project. 2023. Encrypted Messaging Applications and Political Messaging. https://mediaengagement.org/research/encrypted-messaging-applications-and-political-messaging/
[17] Redis. 2024. Redis. https://redis.io/
[18] Reuters. 2023. Brazil court suspends Telegram for not complying with order on neo-Nazi groups. https://www.reuters.com/technology/brazil-court-suspends-telegram-not-complying-with-order-neo-nazi-groups-2023-04-26/
[19] Amazon Web Services. 2024. Elastic Load Balancing. https://docs.aws.amazon.com/elasticloadbalancing/
[20] SimpleX Chat Team. 2023. SimpleX Messaging Protocol. https://github.com/simplex-chat/simplex-chat/blob/stable/docs/SIMPLEX.md
[21] Elkana Tovey, Jonathan Weiss, and Yossi Gilad. 2024. Distributed PIR: Scaling Private Messaging via the Users' Machines. Cryptology ePrint Archive, Paper 2024/978. https://eprint.iacr.org/2024/978.
[22] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. 2017. Stadium: A Distributed Metadata-Private Messaging System. In *Proceedings of the 26th Symposium on Operating Systems Principles*. Symposium on Operating System Principles, Association for Computing Machinery, Shanghai, China, 423–440.