Noctua: A secure messaging platform

Lucas Givens

November 27, 2022

1 Intro

Since the advent of the internet, the issue of privacy has become ever more prevalent. Large companies use the low technological literacy of their customers to their advantage by convincing them to opt into giving up their personal data. Google, amazon, apple, Facebook, and even some big box retailers like Walmart collect vast amounts of data that they sell off or abuse, Bruce Schneier discusses in his book *Data and Goliath* [14]. This abuse comes in many forms but one of the most common is using your data to target you with ads or media to increase your use of their services [14]. Multiple types of technologies have risen to help bring some control back to the average citizen such as those centered around decentralization. The Onion Routing project (TOR), a internet networking system, uses algorithms and information hiding techniques to deliver information to websites across the internet without them knowing where the data came from [6]. This prevents websites from tracking data on users by hiding their true IP address. Other technologies such as blockchains like Ethereum have taken advantage of cryptographic technologies to validate transactions anonymously [18]. This allows for users to pay for items or make exchanges of data securely without the personal information of either party being known, skipping the bank all together. Both of theses systems work using a similar principle called decentralization, which allows for large computer networks to work securely with each other to accomplish a goal. These examples are also open source as well, allowing anyone to connect and run a computer to help grow or use the network.

With Facebook, Google, Microsoft, Apple, and Amazon being responsible for almost all of the online communication channels we use today, it would be of financial interest of the companies to pull data from private chats. However these organizations currently make a stance against doing so but why should we rely solely on their word? There are already circumstances where they encroach on the boundary such as when the governments request information from these organizations such as Amazon who co-operated with police when asked for ring doorbell data [7]. They have the capability to access this data as well as the motive which is why Noctua seeks to solve this problem before it starts, implementing peer-to-peer messaging in a secure and decentralized way. By combining the TOR network and blockchain technologies like Ethereum, it can securely move data between devices on the internet without a centralized system in between. This peer to peer structure allows for you to be in complete control of your data, with no entity to step in the middle. Only people you know and have given permission to can message you using proven cryptographic techniques that encrypt your data with personal keys that you can change.

2 Literature Review

It is necessary to go over several primary technologies to adequately understand the systems discussed further on. These being Blockchains and TOR which will be used in the implementation of the proposed decentralized messaging application.

Blockchains are a subsection of cryptographic techniques that chain data together using hashes that prevent unauthorized changes being made to the chain [18]. In the context of this paper, blockchains are most often used to share information between large networks of anonymous computers; Allowing for a distributed database with high data integrity that only allows for the original owners of the data to change it [18]. This can simulate peer to peer data transfers but is often **too slow** for large amounts of data to be sent or sTORed; Limiting communication over blockchains to small amounts of data like text with latency to high for things like wifi calling or video chat.

TOR is a method not for storing information but for sending data anonymously on the internet. The TOR network takes data and wraps it in several layers of encryption, protecting your data from malicious attackers. The data is then sent to a network of computers that reroute your data around between TOR servers called routers [6]. No router on the network knows both the sender address and the destination address [6]. This makes TOR effective at hiding user information. The coming paragraphs, will go into detail on TOR and the block chain used.

Blockchains used for distributed computing are described in the Ethereum yellow paper[18] as a "simple application on a decentralized, but singleton, compute resource". The Ethereum blockchain takes this idea and uses it to create a generalized block chain that works as one distributed computational resource. Resources are managed via transactions that are signed off on using cryptographic techniques that verify their authenticity [18] . The method that the Ethereum network uses is known as a Blockchain [18]. The Ethereum blockchain is a ledger of information with each entry being tied to an account known as

a "wallet". Each time this ledger is updated all computers on the network verify the authenticity of the update with proof of stake algorithms. Computers are motivated to update the ledger via payments of a token called "gas" [18] and updates being known as transactions. When a transaction is made the user who initiated it will pay a small fee in gas depending on the size of the transaction and the load the network is under. The nature of the block chains cryptography allows for near absolute certainty of the validity of a transaction [18]. Another well-known blockchain is Bitcoin, which differs from Ethereum in that it can only sTORe information relating to how much bitcoin a wallet owns. Ethereum however, allows sTORage of many different types of information such as code, which the network is able to use to run decentralized apps [18]. This ability to sTORe predefined behavior that operate securely on a distributed network can be used to enforce a shared behavior between devices like unique usernames. The major downside of Ethereum however is its greatest strength. Due to the open nature of the ledger every computer on the network is able to see all the data in the ledger. Transactions can also be limited based on how much stress the network is currently under as well. This prevents Ethereum from being used to send messages secretly via the network, however in combination with TOR, this can be achieved.

TOR is a networking protocol that can anonymize internet traffic. It works via encrypting data which is then sent through the TOR network. The network is made up of many peer to peer connections that bounce your data around preventing any one server from knowing where your data is going and where it came from [6]. TOR is an open-source project that was originally created by the US navy to hide secret communications [6]. Since then, it has seen wide adoption by the greater public for accessing network resources anonymously. The core interest of this project in TOR is its ability to host resources anonymously. Using TOR nodes as rendezvous servers (Servers that allow clients behind network address translators to communicate) it is possible to host servers located behind firewalls [6].

This secure true peer to peer behavior would allow each user on a communication network to be their own message server. Combining the TOR and the Ethereum networks allows for decentralized anonymous network traffic to be directed by a decentralized authority that can verify users securely and anonymously, the implementation of which will be explained later on. Several other projects have attempted to create similarly secure apps which are discussed below.

D.I.M Or Decentralized Instant Messenger is a software that uses the GunJs framework to implement a distributed messaging app [16] and is one of the projects that implements a similar structure to Noctua. GunJs provides tools to create Distributed Data Bases (DDB) across many nodes [16]. This allows for network of self-hosted server nodes that sTORe user information. Client nodes on the network are able to read user information from the DDB which allows for logins verification from any device. The main benefit of this model is that it is easily used by users who just need to install the app or access the web portal. However, This structure lacks several features due to the inherited server node client - node model, it is not truly peer to peer even though it is distributed. This is because in a true peer to peer network node in the network act as both servers and clients [15] while in the GunJs network some nodes are dedicated servers. However this model has a single point of failure in communication besides access to the web portal or app, but does use encryption methods to maintain and ensure transactions on the DDB are legit. Making it secure and easy to use.

Muhammad S. A.[4] proposes a solution not for a chat app but an approach for patients to transfer data from smart sensors to their docTORs securely and without a third party. This works is similar in structure to the proposed solution of this paper. [4] focuses on two technologies that allow for a secure peer-to-peer architecture. [4] States that they use the Ethereum block chain for "a medium for negotiation and record keeping" and that TOR is used for "Delivering data from patients to doctors". This architecture is favorable for the medical setting due to its ability to keep data in the hands solely of the patient and the docTOR. [4] uses the blockchain for identity management and thus is able to securely transfer the data between docTOR and patient using tor rendezvous servers. This is true peer to peer communication and since its using TOR it has the added benefit of hiding user metadata such as origin IP.

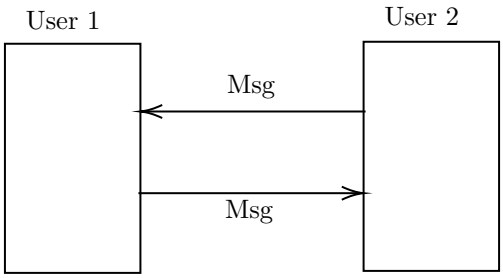
3 Design

The fundamental design approach of this system is use several services to add features. These services are as follows; the database, the message server, the Tor hidden service, and the Ethereum Name Service. The goal of the message server is to receive and send HTTP (Hyper Transfer Text Protocol) POST (Not an acronym) requests to other message servers on the network. HTTP POST requests allow for devices to send data to servers instead of retrieving it. Each device is used to act as its own message server and client. Using Tor hidden services to bypass the firewall on the local network allows for requests to be sent or received by other servers regardless of local network configurations [11]. To keep track of what address goes to what user an name service implemented on the Ethereum network is used to tie users to their TOR hidden service address. The database stores the information of the primary user, the user contacts, and the received messages. Each is designed to work independently of the system below it, allow for more flexibility while implementing the application. In the following sections, the specific design of each service will be reviewed.

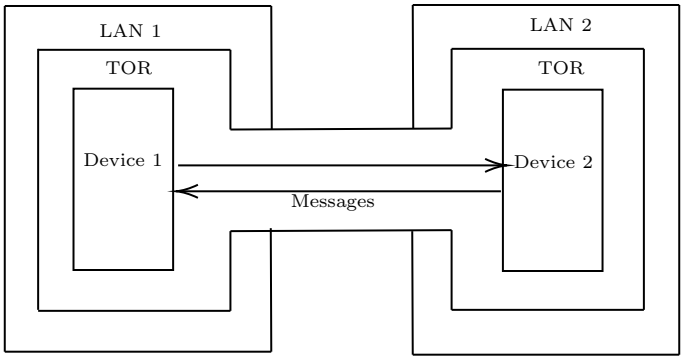
The Database contains several tables: primary user, contacts, conversations, and messages. Primary user is the user of the current instance of the application. Information regarding the address of their TOR hidden server and their username is stored in the primary user table. The contacts table is used to store the contacts of the primary user, their associated shared keys, and the last updated address of the contact. The conversations table stores the active conversations, associating the contacts with their conversation. Finally messages are stored in the message table which holds information on the

conversation, the message itself, and a time stamp of when the message was received or sent. The message server makes use of the database to store messages and look up conversations to validate received messages.

The Message Server is an HTTP server which receives and sends POST requests and uses SSL (Secure Socket Layer a encryption protocol) for further encryption of the data stream. Requests contain a JSON object with two fields: the name of the sender, and the message, both of which are encrypted by AES256 which uses block ciphers to symmetricly encrypt data. To decode or encode the data each user has a key that they first agreed upon when creating each other contacts. This key is hashed via SHA256 (Hashing algorithm) and is used as the key for the AES256 encode and decoding. The name of the sender is used to identify the conversation that the message belongs to. Using this,the message server then inserts the message into the database storing the associated conversation ID and a timestamp of when the message occurred. To send messages the server again uses POST requests to transmit the data in the same fashion it received it, retrieving destination address from the database. Once received the http server will return an OK as an acknowledgement Each user acts as both client and server which forms the peer to peer structure of the network.



TOR hidden services are used to facilitate connections to servers behind NATs without port forwarding in a method know as NAT hole punching [11]. NAT stands for natural address translation, and is used to hide network IP ranges behind a single network address. They are needed because IPV4 only has a set number of IPs that can be assigned in a network [9]. So by splitting larger networks up into their own isolated networks IPs can be reused. Many NATs and firewalls do not allow servers from outside the network to initiate connections inside the network [9]. For most messaging services like, iMessage, use centralized servers that devices can call out to retrieve data. Tor can be used to skip this intermediate step by acting as a rendezvous server [6]. Devices behind NATs can reach out to these servers which in turn are able to forward incoming connections to the device. This effectively is what as known as "NAT hole punching" and is what allows devices to act as their own message server. By making a TOR hidden service hosted on the device, incoming http POST requests can be received by Noctua message servers. The hidden service is separate from the message server and just transfers data between a listening port.



ENS(Ethereum Name service) will be used to keep track of usernames and ensure each name is unique. Using smart contracts to allow users to create a username and set an associated address securely. The ENS stores the last update address of the users Tor hidden service. Since each username is unique, user's are able to add contacts via usernames like they would with any other service. When a contact is added the application will attempt to initiate a transaction to find the user in the ENS. If found it will add its contact and address to the user's hidden service. To keep addresses fresh, whenever a new hidden service is created, a new transaction will be initiated to update the users hidden service address. If a user sends a message to a contact and their HTTP server does not respond another transaction will attempt to find the new address of the contact, or save the message to send again later. Due to the nature of the Ethereum network transactions are automatically authenticated by the network using the users wallet address [18], meaning only the user can update their own addresses.

4 Implementation

RoomDB is used to implement the database. RoomDB is an high level Kotlin library that provides abstraction for creating and managing SQlite databases. It is implemented through several layers: the tables, data access object, repository, and the

AppViewModel [2]. The tables are defined using special syntax that correlates to Sqlite table qualities like primary keys and columns. Tables are defined using Kotlin data classes using syntax like the following.

Listing 1: Example of table definition.

```
@Entity(tableName = "contacts_table")
data class Contacts (
    @PrimaryKey(autoGenerate = false)
    val user: String,
    val key: String,
    val address: String
    )
```

Four tables are needed for the basic functions of this application: the contact table, primary user table, conversation table, and the message table.

- The contact table has three columns: Contact name(primary key), key, and last updated address
- Primary user table has two columns: ID(primary key), hidden service address. Only one entry should ever been in this table.
- The conversation table has three columns: ID(Primary key), the contact name, and the conversation name
- The message table has five columns: ID(Primary key), the contact name, conversation name, message, and time stamp

Data Access Objects(DAO) are used to define query's, insertions, deletions, and other SQL activities. Due to the nature of IO like database access it is best to use co-routines to preform database access independently of the current thread. Co-routines are simple threads that allow for non-blocking code to be easily implemented [1]. This prevents the blocking of UI or of the message server when accessing the database. For the purpose of this application four insertion and four queries are necessary of the base functionality of the app.

Listing 2: Example of DAO definition.

```
@Dao
interface DAO {
    // Adds a contact to the contacts table, takes the contact data class from
    // the Table definition as input and returns nothing.
    // If there is a conflict like a non-unique
    // primary key the function will throw a fail exception
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun addContact(user: Contacts)

    // Adds a primary user to the primary user table.
    // This is where data like username and current hidden address are stored.
    // On conflict the insertion fails and thows a fail exception
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun addPrimaryUser(user: PrimaryUser)

    // Adds a contact to a coversation by
    // making an entry in the conversations table.
    // This is how contacts are grouped into conversations.
    // Setup in a way to allow for group
    // conversations in the future but currently this is
    // missing feature.
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun addToConversation(convo: Conversations)

    // Inserts a message into the message database.
    // Messages from all users are
    // stored here even messages from the primary user.
    // They are associated to conversations by the convo ID.
    @Insert(onConflict = OnConflictStrategy.ABORT)
    suspend fun addMsg(msg: Msgs)

    // Selects all msgs that match the conversation ID
    // and returns them ordered by their timestamp.
    // This is used in the UI to
    // show the message history of a coverstation.
    @Query("Select␣*␣FROM␣msg_table␣WHERE␣convo␣=␣:convo␣ORDER␣BY␣time_stamp␣ASC")
    fun getConvo(convo: Int): LiveData<List<Msgs>>

    // Gets all contacts and returns them as a livedata object.
    // Live data objects are update when new data
```

```
    // is observed and must be accessed using an observer
    @Query("Select * FROM contacts_table")
    fun getContacts(): LiveData<List<Contacts>>

    // Gets all contacts and returns them as a regular list
    // This is nessacry for when contacts must be
    // checked for validity an observer object would be uneccesary.
    @Query("Select * FROM contacts_table")
    fun instantGetContacts(): List<Contacts>

    // Gets Information about primary user
    @Query("Select * FROM primary_user_table")
    fun getPrimaryUserInfo(): LiveData<List<PrimaryUser>>

    // Lists all Conversations is used in UI
    // for listing all availble messages
    @Query("Select * FROM conversation_table")
    fun getConvos(): LiveData<List<Conversations>>

}
```

Accessing these functions can be done from the database repository which holds functions that abstract multiple data sources. It acts as an intermediate between the DAO and the database View Model. View models are used for when a database need to be interacted with by UI components. Functions here are defined as normal functions with the exception of using the "suspend" key word. The suspend key work tells the JVM that the function will be called inside of a co-routine and that it can be a blocking function. The repository uses functions declared in the DAO and returns their values. The syntax for declaring repository functions are as follows.

Listing 3: Example of repository definition.

```
class AppRepository(private val dao: DAO) {

    //Inits variables that can me found on start
    val getConvos: LiveData<List<Conversations>> = dao.getConvos()
    val getContacts: LiveData<List<Contacts>> = dao.getContacts()

    // returns all the messages that have the same
    // message ID orderd by time stamp
    fun getConvo(id: Int): LiveData<List<Msgs>>{
        return dao.getConvo(id)
    }

    // Gets all contacts as a list
    fun instantGetContacts(): List<Contacts>{
        return dao.instantGetContacts()
    }

    // Adds a contact to the contact table
    // Uses the suspend keyword to be used in a coroutines
    suspend fun addContact(user: Contacts){
        dao.addContact(user)
    }

    // Adds a primary user to the primary user table
    suspend fun addPrimaryUser(user: PrimaryUser){
        dao.addPrimaryUser(user)
    }

    // Adds a contact to a conversation table
    suspend fun addToConversation(convo: Conversations){
        dao.addToConversation(convo)
    }

    // Adds a message to message table
    suspend fun addMsg(msg: Msgs){
        dao.addMsg(msg)
    }


}
```

The view model uses calls functions from the repository but from inside of coroutines that allow for asynchronous access to the database. Co-routines are high level functions that Kotlin uses for easy multi-threading. This is necessary because

the viewmodel is used by UI componets to interact with the data base so IO calls should be none blocking to prevent a slow UI. The app view model used for this implementation is defined by the following code.

Listing 4: Example of Viewmodel definition.

```
// Takes application as context
class AppViewModel(application: Application): AndroidViewModel(application) {
    private val repository: AppRepository

    val getConvos: LiveData<List<Conversations>>
    val getContacts: LiveData<List<Contacts>>

    init {
        val appDao = AppDatabase.getDatabase(application).dao()
        repository = AppRepository(appDao)
        getConvos = repository.getConvos
        getContacts = repository.getContacts
    }

    fun addContact(user: Contacts){
        viewModelScope.launch(Dispatchers.IO){
            repository.addContact(user)

        }
    }

    fun addConvo(convo: Conversations){
        viewModelScope.launch(Dispatchers.IO){
            repository.addToConversation(convo)
        }
    }

    fun getMsgs(id: Int): LiveData<List<Msgs>>{
        return repository.getConvo(id)
    }

    fun instanctGetContacts(): List<Contacts>{
        return repository.instantGetContacts()
    }

}
```

These are the necessary database functions to implement the base functionality of the messaging app. If additional features where to be added it is likely there would need to be changes to the implementation. For example the current design is centered around single user conversations. If group conversation where to be implemented, there would need to be queries to gather all users in the shared conversation to a list. In the next section the implementation of the HTTP server and how it interacts with the database will be outlined and expanded upon.

The http server is implemented using a Kotlin library called "Ktor" [3], which is a light Kotlin library for creating http servers and clients. Ktor uses co-routines to allow for asynchronous receiving and sending of http requests. For encryption it can use SSL(Secure Sockets Layer) allowing for messages to be sent securely adding an extra layer to the AES256 Encryption that is done when messages are sent. SSL uses private and public certificates to provide asynchronous encryption between servers and clients. Ktor allows for all standard http messages, but the only one used for this implementation will be the post request. The two methods implemented to handle these requests are the "/txt" http route and the send() function. Below is the code for each function.

Listing 5: code for /txt.

```
data class Jsonmsg(
        convoEncrypt: String,
        msgEncrypt: String
)

routing{
        // sets the name of the route
        // accessed at http://IP_ADDRESS:PORT/txt
        post("/txt"){
                    lastmsg = call.receive<JsonMsg>() // the message most recently recieved
                    CoroutineScope(Dispatchers.IO).launch {
                        Log.v("TXT", "RECIEVED␣MESSAGE")
                        val contacts = repository.instantGetContacts() // gets an frozen instant of the contacts
                        // Goes through each contact and attempts to see if its key
                        // can decrypt the conversation name and checks
```

```
                    // to see if that contact exits. If it can and the convo exists
                    // then it decrpyts the message and exits.
                    for(contact in contacts){
                        val convo = decrypt(lastmsg.convoEncrypt, contact.key)
                        Log.v("DE", convo + "␣" + lastmsg.convoEncrypt)
                        if(repository.contactExists(convo)){
                            val msg = decrypt(lastmsg.msgEncrypt, contact.key)
                            repository.addMsg(Msgs(0,convo, convo, msg, LocalDateTime.now().toString()))
                        }
                    }
                }
                // return a OK status code to let the other side know the message was heard
                call.response.status(HttpStatusCode.OK)
}
```

Listing 6: code for send().

```
    suspend fun send(msg: String, convoName: String){
    // Get the primary variables to build the message
    //  - The contact for the address
    //  - the primary users name as this is used
    //    as the conversations name on the other users
    //    device
    //  - The AES encrypted message and convoname
    val contact: Contacts = repository.getContact(convoName)
    val primaryUser: PrimaryUser = repository.getPrimaryUserInfo()
    val convoEncrypt = encrypt(primaryUser.userName, contact.key)
    val msgEncrypt = encrypt(msg, contact.key)

    // TODO("Add check for if response was ok and move this section")
    repository.addMsg(Msgs(0,"internal", convoName, msg,LocalDateTime.now().toString()))

    // Build the message data class
    val Jmsg = JsonMsg(convoEncrypt,
        msgEncrypt)

    // Sends the message and saves the response
    val response = client.post {
        url(contact.address)
        contentType(ContentType.Application.Json)
        header(HttpHeaders.ContentType, Json)
        setBody(Jmsg)
    }

    Log.v("MSG", response.toString())

}
```

TOR, for this project, implement of the hidden service was done using kpm-tor [12] which a branch of the Tor Onion Proxy Library [17]. The goal of the kmp-tor is to ease the management of TOR proxies and allow each app to act as its own TOR proxy. This allows for TOR to be baked into an app instead of relying on third parties like Orbot to create a hidden service. The TOR proxy listens to the port that the HTTP server runs on and broadcasts it on the rendezvous server on another port. First, the configuration of the TOR server must be done by setting all the ports that it will forward. Then the hidden service is configured and its ports are directed to the ports of the http server. Below is an example of the configuration of the hidden service. The rest of the code can be found at the Github for this project in the /tor package [8].

Listing 7: Psuedocode for hidden service.

```
put(HiddenService()
                    .setPorts(ports = setOf(
                        // Use a unix domain socket to communicate via IPC instead of over TCP
                        HiddenService.UnixSocket(virtualPort = Port(80), targetUnixSocket = hsPath.builder {
                            addSegment(HiddenService.UnixSocket.DEFAULT_UNIX_SOCKET_NAME)
                        }),
                    ))
                    .setMaxStreams(maxStreams = HiddenService.MaxStreams(value = 2))
                    .setMaxStreamsCloseCircuit(value = TorF.True)
                    .set(FileSystemDir(path = hsPath))
                )

                put(HiddenService()
                    .setPorts(ports = setOf(
                        HiddenService.Ports(virtualPort = Port(80), targetPort = Port(5001))
```

```
                    ))
                    .set(FileSystemDir(path =
                        workDir.builder {
                            addSegment(HiddenService.DEFAULT_PARENT_DIR_NAME)
                            addSegment("test_service_2")
                        }
                    ))
                )
            }.build()
```

Implementation of the **ENS** is done using Solidity which is a langauge that supports compilation into byte-code that can run on the Ethereum Virtual Machine(EVM) [18]. The EVM is similar to the Java Virtual Machine in idea, as it allows for many machines with different architecture to run the same code. When publishing a smart contract to the Ethereum network, it is first compiled into byte-code and is then uploaded and given an address space in the Ethereum ledger [18]. Using this address and Web3j, a Java library that allows users to link their Ethereum wallet to android apps, the application can call the functions of the smart contract. This will allow users to manipulate their data on the Ethereum block chain and provide a way for users to query for other users addresses. To facilitate these interactions the smart contract consists of a struct, a dictionary object with String keys and values of instances of the struct, and four functions. Below is the pseudocode for creating the smart contract as well as comments explaining more on the fucntions.

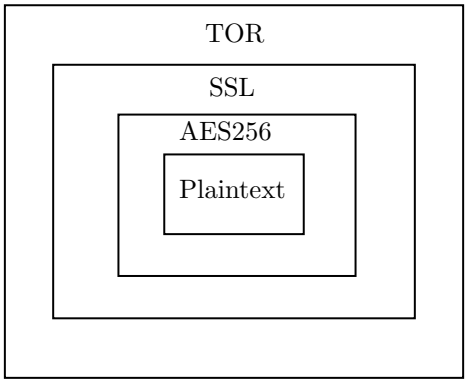
Listing 8: Psuedocode for ENS.

```
struct User {
        String username;
        String address;
        String owner; // Wallet address of the owner of the username
}

// Creates a dictionary object that contains keys
// of the string type that will be
// equal to the username of the user
// and values that will be equal
// to instances of the User struct
mapping(String => User) public ENS;

// Function to add a user to the ENS map
// First check if user exists if not
// add user
function addUser(username: String, address: String){
        if( EAS[username].exists ){
                return "Transaction failed username exists"
        }
        else {
                user = User(username, address, msg.sender)
        {
}

// Attempts to update the address of a user.
// Will not update unless username exists
// and username is owned by sender
function updateAddress(username: String, newAddress: String){
        if( !EAS[username].exists or msg.sender != EAS[username].owner){
                return "Transaction failed because username \
                doesn't exist or msg.sender is not owner"
        }
        else {
                EAS[username].address = newAddress
        }
}

// Looks up a user in the ENS and
// returns address of user
function lookupUser( username: String ){
        return EAS[username].address
}

// Deletes a user from the mapping
deleteUser(username: String){
        if( EAS[username].exists and msg.sender == EAS[username].owner){
                delete EAS[username]
        }
}
```

In conclusion the implementation of the core components of this system are outlined above. The database stores the data captured from the primary user and from incoming messages. While the HTTP server handles the incoming HTTP messages from other users and attempts to add them to a conversation if they are valid messages. The TOR hidden service allows the HTTP server to be accessed by users outside of the local network, and the ENS provides the application with a decentralized way for users to find each other.

5 Validation/Threat Models

Threats to this system can come from two main sources, internal threats that listen in from the device itself, or external decryption/deanonymization. Internal threats can occur if the device itself is compromised by malware which would allow the attackers to extract data before or after it is encrypted / decrypted. External threats happen once the data leaves the device and are less likely to occur due to layered encryption of the messages. Outgoing data is wrapped in several layers which serve to protect contents of the message. The first layer is AES256 is a blocker cipher algorithm which uses symmetric keys to decrypt and encrypt data. It is approved by NIST(National Institute of Standards and Technology) and the NSA (Nation Security Agency) for use in securing top-secret information [5]. Although there are theoretical attacks against AES, a security review [10] determined that "Therefore, at this moment, we believe that the AES still remains practically secure.". Unless the attacker knows the AES key it is highly unlikely they will be able to extract the data. The next layer is SSL which is a network layer protocol that encrypts TCP/IP data so that content is not available to anyone but the recipient. It also uses a combination of hashing algorithms and AES encryption to secure data [13]. SSL is important for securing the data sent over the http stream due to the fact that once data leaves the TOR exit node the last layer(The TOR layer) is no longer present. Which leads us to the final layer which is the TOR layer which also uses the AES standard. TOR, although secure data wise can suffer from attacks know as deanonymization. This is where the anonymity of the TOR data stream is uncovered and bad acters are able to determine the source/ destination address. Bad actors accomplish this by using and hosting exit nodes the keep track of data streams. TOR attempts to mitigate this by having the community keep a close eye on nodes in the network to ensure they are secure. However deanonymization is not the same as decrypting the actual data so even if the host address is found that does not mean the data has been decrypted. Internal threats are the most dangerous to the security of this application. If the android system is compromised then no amount of encryption can prevent eavesdropping. Thus it is vitally important that the user monitors their internal security closely.



6 Conclusion

In conclusion this paper focused on the implementation of a secure peer to peer chat app using known secure cryptographic techniques. A three layer approach was taken using SSL, TOR, and AES256 to secure messages sent between users. Using TOR hidden services, users are able to act as clients and servers fulfilling the peer to peer requirement [15]. This keeps message data out of the hands of large corporations, and instead in your pocket by storing the received messages locally.

Due to time constraints the ENS service has not been implemented as of yet but is in progress. Although increasing the usability of the app the ENS is not necessary for the secure sending of messages. In the future this feature and others are to be added to the app to increase its security and usability. These being group messaging, file sharing, database encryption and other smaller ease of life features. As of right now the app is not fully featured and shouldn't be used besides for testing.

References

[1] Concurrency and coroutines: Kotlin. `https://kotlinlang.org/docs/multiplatform-mobile-concurrency-and-coroutines.html#coroutines`.

[2] Save data in a local database using room android developers2022. https://developer.android.com/training/data-storage/room, 2022.

[3] Welcome: Ktor. https://ktor.io/docs/welcome.html, 2022.

[4] Muhammad Salek Ali, Massimo Vecchio, Guntur D. Putra, Salil S. Kanhere, and Fabio Antonelli. A decentralized peer-to-peer remote health monitoring system. *Sensors*, 20(6), 2020.

[5] Elaine Barker. Guideline for using cryptographic standards in the federal government: Cryptographic mechanisms, Mar 2020.

[6] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical Report ADA465464, NAVAL RESEARCH LAB WASHINGTON DC, 555 Overlook Ave SW, Washington, DC 20375, January 2004.

[7] Christopher Frascella. Amazon ringmaster of the surveillance circus. *Federal Communications Law Journal*, 73(3), May 2021.

[8] Lucas Givens. nocuta-messenger. https://github.com/Givlucas/noctua-messenger, Sep 2022.

[9] Arun Handa. Chapter 3 - basics of ip networks. In Arun Handa, editor, *System Engineering For IMS Networks*, pages 35–43. Newnes, Burlington, 2009.

[10] Herman Isa, Iskandar Bahari, Hasibah Sufian, and Muhammad Reza Z'aba. Aes: Current security and efficiency analysis of its alternatives. *2011 7th International Conference on Information Assurance and Security (IAS)*, 2011.

[11] Daniel Maier, Oliver Haase, Jürgen Wäsch, and Marcel Waldvogel. A comparative analysis of nat hole punching. *HTWG FORUM - Das Forschungsmagazin der HTWG Konstanz*, pages 40–48, 2011.

[12] Matthew Nelson. kmp-tor. https://github.com/05nelsonm/kmp-tor, Feb 2022.

[13] Vaclav Platenka, Antonin Mazalek, and Zuzana Vranova. Attacks on devices using ssl/tls. In *2021 International Conference on Military Technologies (ICMT)*, pages 1–6, 2021.

[14] Bruce Schneier. *Data and Goliath*. Wiley, 2016.

[15] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings First International Conference on Peer-to-Peer Computing*, pages 101–102, 2001.

[16] Sourabh. Inderprastha engineering college. *International Research Journal of Engineering and Technology*, May 2020. Accessed September 7, 2022.

[17] Thaliproject. Thaliproject/tor_onion_proxy_library: Provides a jar and an aar for embedding the tor onion proxy into a java or android program. https://github.com/thaliproject/Tor_Onion_Proxy_Library, Jun 2014.

[18] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger.