EncChat: A Native Android Encrypted Chat System

Kevin Zhang
kevinzhang@wayne.edu/keevee810@gmail.com
Department of Computer Science,
Wayne State University
Detroit, MI

ABSTRACT

This paper presents EncChat, a native Android encrypted chat system that provides end-to-end encryption for chat messages. The program uses hybrid encryption, meaning symmetric session keys are used to encrypt messages and asymmetric keys are used to encrypt and send session keys over the network. Encrypted messages, as well as encrypted session keys, are stored on Cloud Firestore, while session keys and private asymmetric keys are stored locally on the Android device, in SharedPreferences. Ultimately, the system provides an end-to-end encrypted chat with full user transparency (the user does not have to manually manipulate keys).

KEYWORDS

chat, end-to-end encryption, Android

1 BACKGROUND

Text messaging through short message service (SMS) is one of the primary ways people communicate with each other over cellular devices. However, it has a couple of disadvantages. First, it is tied to phone plans, meaning SMS requires people to pay to use. Also, messages sent through SMS are generally not encrypted, and even if they are encrypted, they usually do not provide full end-to-end encryption. As a result, many people have moved to online messaging applications, which have a lower cost and usually offer encryption as well.

This paper presents EncChat, which is an online end-to-end encrypted chat messaging application for Android. All messages are stored on a central database, but sent and stored in an encrypted format. It uses a hybrid encryption scheme: symmetric session keys (AES 256-bit) are used to encrypt messages, and asymmetric keys (RSA 2048-bit) are used to send the session keys.

1.1 Goals

The primary goal of EncChat is to provide end-to-end encryption for chat messages. Therefore, any message sent over the network will be encrypted. If intercepted, attackers must derive the key. The message can only be viewed at both ends, which are the mobile devices for the two chatters. An important note is that EncChat does not provide encryption for chat metadata, such as message timestamps. Due to the program's structure, much of the chat metadata is needed to properly retrieve the encrypted chat messages from the database. While encrypting the metadata is beneficial, we decided that ensuring encryption of the message data was the most crucial aspect.

Furthermore, another goal of EncChat is to limit the chat messages attackers can read. If each chat had a single key to encrypt chat messages, attackers would have access to the entire chat history if they cracked the key. However, by using multiple keys for encryption, attackers would need to perform additional attacks to read messages. Thus, a goal of EncChat is to provide session keys for chats, meaning message encryption keys will switch to new ones after a certain duration of time.

In addition to encryption, one goal of EncChat is user transparency. In other words, EncChat should be secure while requiring as little manual interaction with security aspects as possible. In other words, users should not have to manually generate and select keys; all encryption, decryption, key generation, and key selection should be done automatically by the system without users needing to know any details of the implementation.

2 PROJECT DESIGN

Usage of EncChat consists of four major phases:

- Account Creation
- Chat Initiation
- Chat
- Challenge-Response Authentication (optional)

The design of each phase is discussed in the following subsections. In addition, the structure of the app's database, Cloud Firestore, is detailed.

2.1 Account Creation

To use EncChat, users must first register an account. Afterwards, they can freely log in using the registered account. A full flow of the user account creation, as well as the artifacts created, is shown in Figure 1.

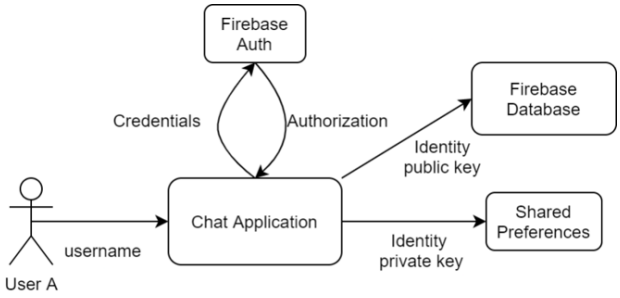

Figure 1: Account Creation

To register an account, a user must provide an email and a password. Those credentials are given to Firebase Authentication services [1], which authorize the user based on OAuth 2.0 and OpenID

Connect standards. In other words, users do not give up their passwords and use authorization tokens instead.

Once an account is generated, the user creates an 2048-bit RSA public-private key pair. This key pair is known as the identity key pair because it is a key that denotes a user's identity when encrypting. The public key must be known to anyone, so it is uploaded to the database. To save it properly, it is first Base64-encoded into a string and then uploaded to Cloud Firestore, under the user's database document. The private key should only be known locally, so it is Base64-encoded and saved into SharedPreferences, which works as a local app database. Essentially, EncChat has two databases: Cloud Firestore for information any user can retrieve, and SharedPreferences for information only the device owner can know.

One thing to note is that if a user switches their device, they lose access to their private key, since it was stored in SharedPreferences on the original device. If this occurs, the user is given a new public-private key pair. Changing devices is detected through device tokens, which are provided by Firebase Cloud Messaging.

2.2 Chat Initiation

To start a chat, a user can search for the username of another user. Figure 2 shows the flow for chat initiation. Suppose User A wants to chat with another user, *pixel@gmail.com.* The user types the string "pixel@gmail.com", selects "Confirm", and is presented with an alert displaying *pixel@gmail.com's* Base64-encoded public key. The displayed public key can be used to verify the identity of the other user. If User A wants to be sure that the other user is truly *pixel@gmail.com,* User A can view *pixel@gmail.com's* profile page (Figure 3) out of band, and check if the public keys match. After the user is confirmed, a new chat document is uploaded to the database.

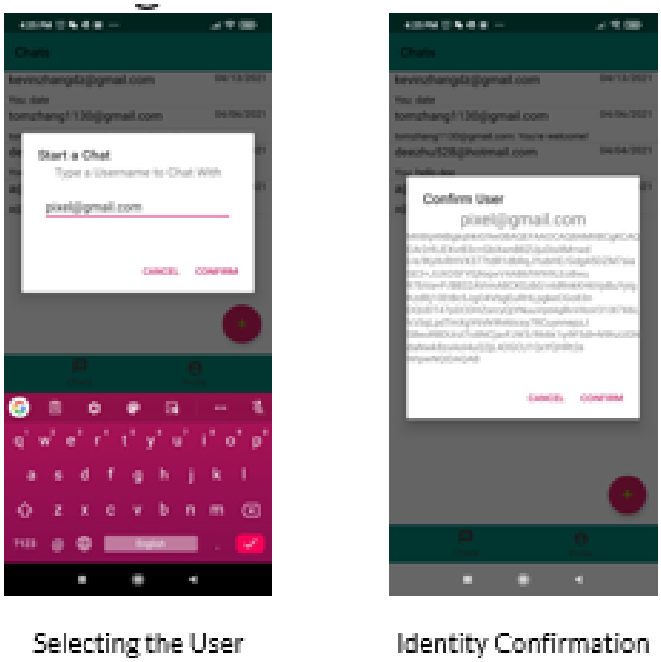

Figure 2: Chat Initiation

Figure 3: Profile Page

2.3 Chat

Once two users have initiated a chat, they can send encrypted chat messages to each other using hybrid encryption. To understand how the chat functions, the description will be divided into two sections: the encryption scheme and session key retrieval.

2.3.1 Encryption Scheme. Suppose User A wants to send a message to User B, as shown in Figure 4.

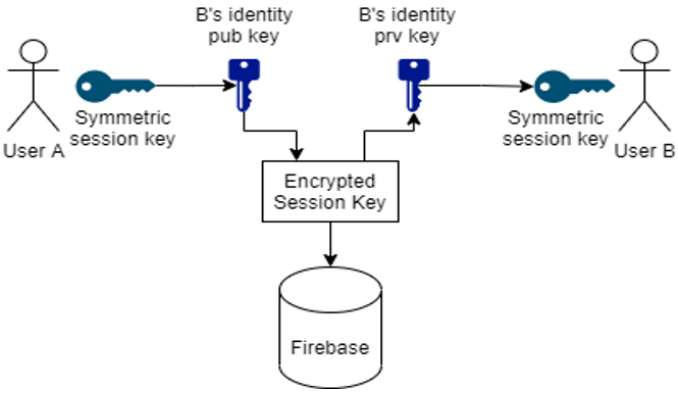

Figure 4: Hybrid Encryption Scheme

First, User A generates a random 256-bit AES symmetric session key, which is used to encrypt messages. The initial session key for a chat is generated when the chat is created. User A saves the session key locally to SharedPreferences, with a key id made from

the database chat id and the key creation timestamp. The key must be confidentially sent to User B, so User A uses User B's public key to encrypt the session key and uploads it to the database. When User B opens the chat, the program will retrieve the session key from the database, decrypt it with User B's private key, and save the session key locally to SharedPreferences. Thus, both User A and User B will have the same session key saved to their devices.

2.3.2 Session Key Retrieval. Session keys are the symmetric AES keys used for message encryption and decryption. Each of a user's chats will have different session keys, and the session key for a chat will switch to a new one after seven days by uploading a new encrypted key to the database. Encrypted session keys are stored in the database, but they cannot be used until they are retrieved and stored locally on a user's device. Thus, this section details the scheme for retrieving keys from the database.

Figure 5 demonstrates the process for session key retrieval. First, a user opens a chat and queries the database for all session key documents for the given chat id. The important information retrieved from each document is the key id (chat id + timestamp) and the encrypted session key. Second, the program checks if each of the key ids are stored in SharedPreferences. If a key id is present, that signifies that the key was already retrieved from the database previously and does not need to be decrypted again. Otherwise, if the key id is not in SharedPreferences, the session key is decrypted with the private key and saved to SharedPreferences. Lastly, all necessary chat session keys are loaded to the chat.

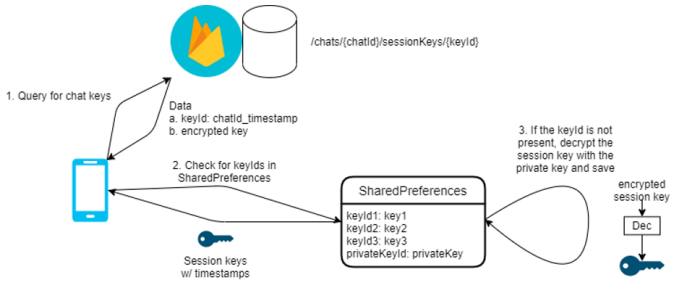

Figure 5: Session Key Retrieval

To decrypt messages, the application reads the encrypted chat messages from the database. The message timestamps are compared to the session key timestamps to select the correct one. The check is done starting from the last session key to the first session key, and if the message timestamp is greater than the key timestamp, that key is selected.

2.4 Challenge-Response Authentication

Inside a chat, User A can verify that User B is truly User B through challenge-response authentication (Figure 6. Challenge-response makes use of Firebase's notification system. First, User A generates a challenge string, which is encrypted with a chat's latest session key. The challenge information is uploaded to the challenges collection in the database. Writing to the challenges collection triggers Google Cloud Functions, which sends a notification to User B's device. If User B presses "accept", a broadcast receiver will attempt to decrypt the challenge. If the challenge is properly decrypted, that signifies User B had the correct session key, and to get the correct session key, User B had to have the correct private key to retrieve it from the database. User B then re-encrypts the result with the session key, updates the challenge in the database, triggering a notification to User A. User A decrypts the result, and if it is the same as the original, User B is verified.

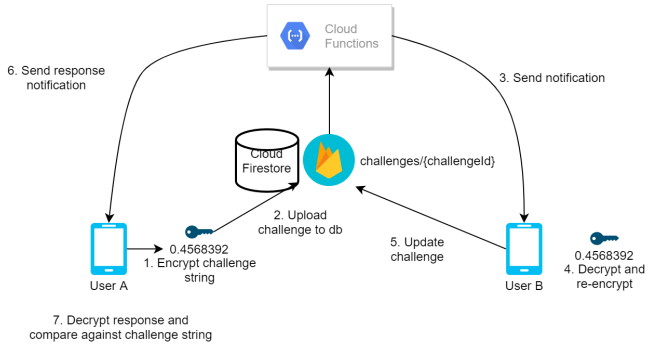

Figure 6: Challenge-Response

2.5 Database Design

All data stored to Cloud Firestore is stored in collections of documents. Each document represents one object, and each document has fields that describe the object's attributes. Each document also has a string id. Additionally, each document can have its own subcollections.

The database for EncChat is divided into three major collections: *users, chats,* and *challenges.*

Figure 7 shows the *users* collection. Each document has the following attributes:

Document ID User id from Firebase Authentication
deviceToken Device id from Firebase Cloud Messaging
publicKey Base64-encoded 2048-bit RSA public key

In addition, the *users* collection has a sub-collection called *userChats,* which contains denormalized chat data for the sake of faster data loading on the home page.

Figure 8 shows the *chats* collection. Each document contains the following attributes:

Document ID Combination of the user ids of the two chatters
members User ids of the two chatters
timeCreated Creation timestamp

Within the documents are two important subcollections, *messages,* which holds encrypted chat messages, and *sessionKeys,* which holds encrypted session keys. Figure 9 shows the attributes for each message document. The most important ones are the **message** field, which holds the encrypted message, and the **time** field, which holds the message send time. Figure 10 shows the attributes for each session key document. Most importantly, the **sessionKey** field shows the encrypted session key, and the **decrypter** field shows the user id of the person who will decrypt the key.

Figure 11 shows the *challenges* collection, which is used in challenge-response authentication. Important attributes include:

challenge An encrypted challenge string

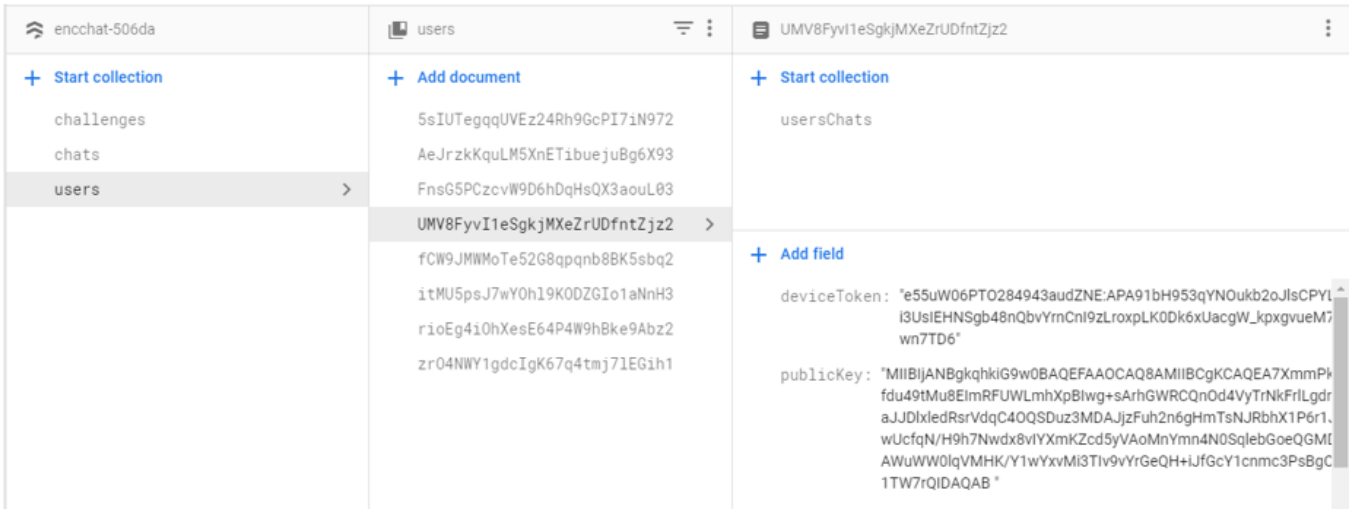

Figure 7: Users Collection

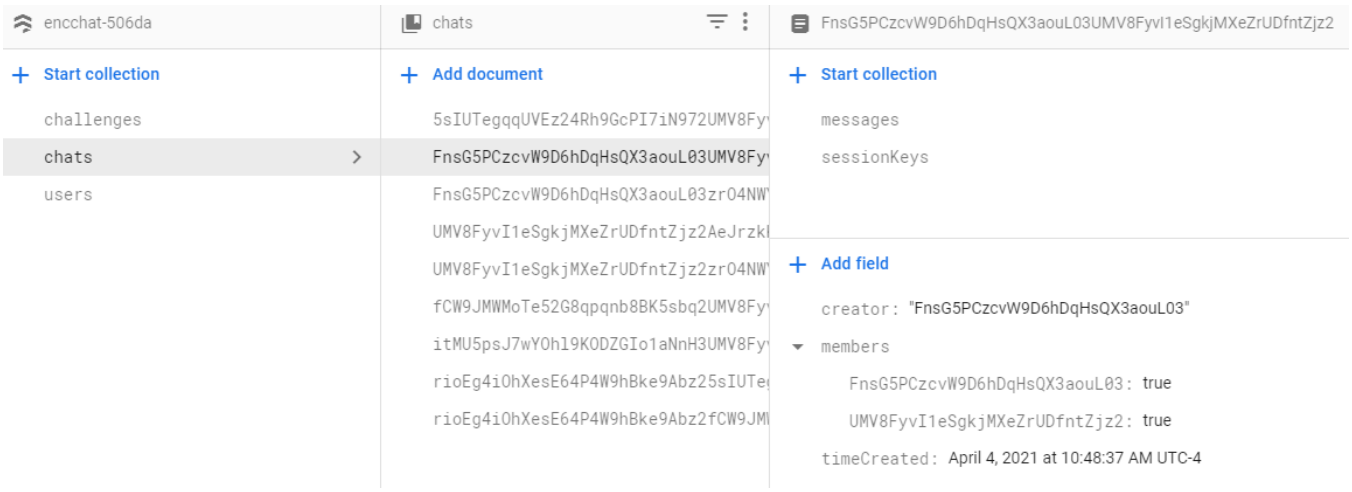

Figure 8: Chats Collection

receiverDeviceToken The Firebase Cloud Messaging token of the person who will receive the initial notification

approved Represents that the notification receiver responded to the challenge

challegneResponseData The encrypted challenge string that the notification receiver responded with

Challenge-response authentication is based around notifications, which involve Google Cloud Functions (Figure 12). One cloud function, *notificationForChatMessage* is for normal notifications that are sent when a chat message is received and the app is in the background. The other two cloud functions, *notificationForChallenge* and *notificationForChallengeResponse*, are triggered based on the creation or update of the *challenges* database collection.

3 IMPLEMENTATION DETAILS

EncChat is a native Android application. Thus, it was developed using Java. In addition, some pages, primarily the activities (pages) are written using Kotlin, which is based on Java and uses Java packages, but syntactically is similar to scripting languages like Python. The UI was developed with Material Design, which allows UI components to be laid out in an XML file.

One of the significant tools in EncChat's development was Firebase (package: com.google.firebase), which is a suite of tools provided by Google. EncChat utilizes the following services from Firebase:

Firebase Authentication An authentication service that generates credentials for user without giving up password information to the app owner. Used for email-based login and creating user id.

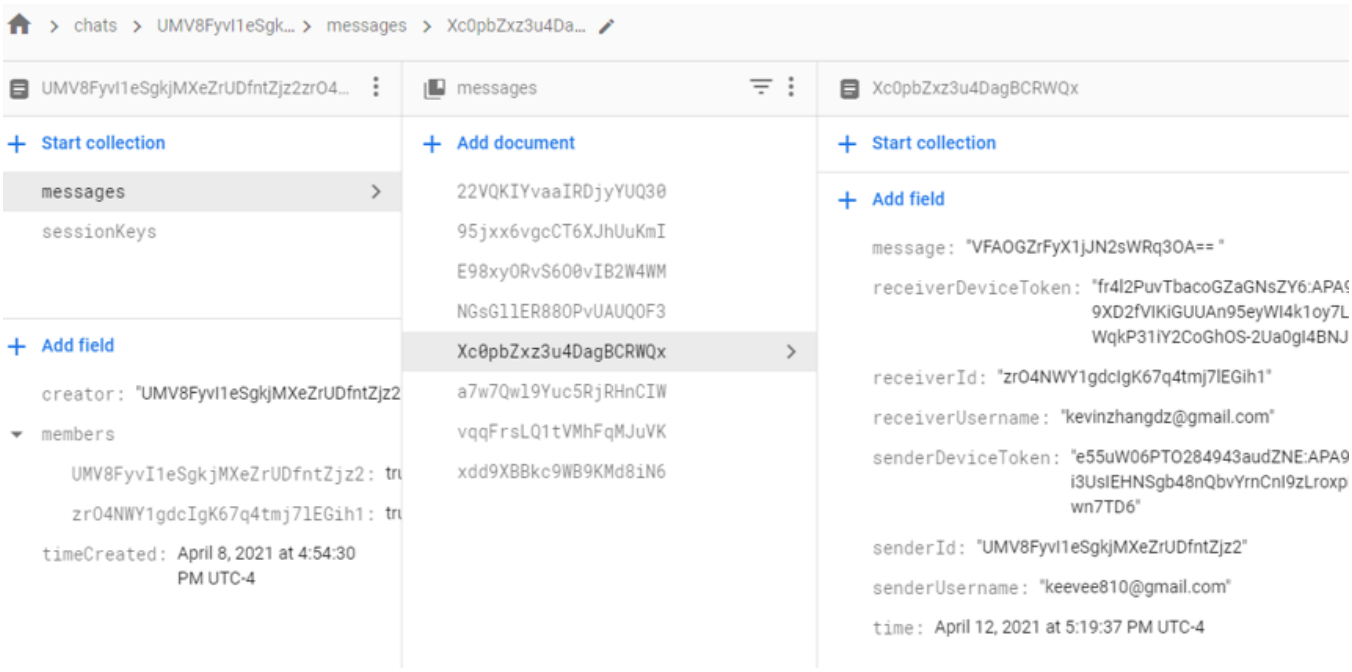

Figure 9: Chats Collection - Messages Sub-Collection

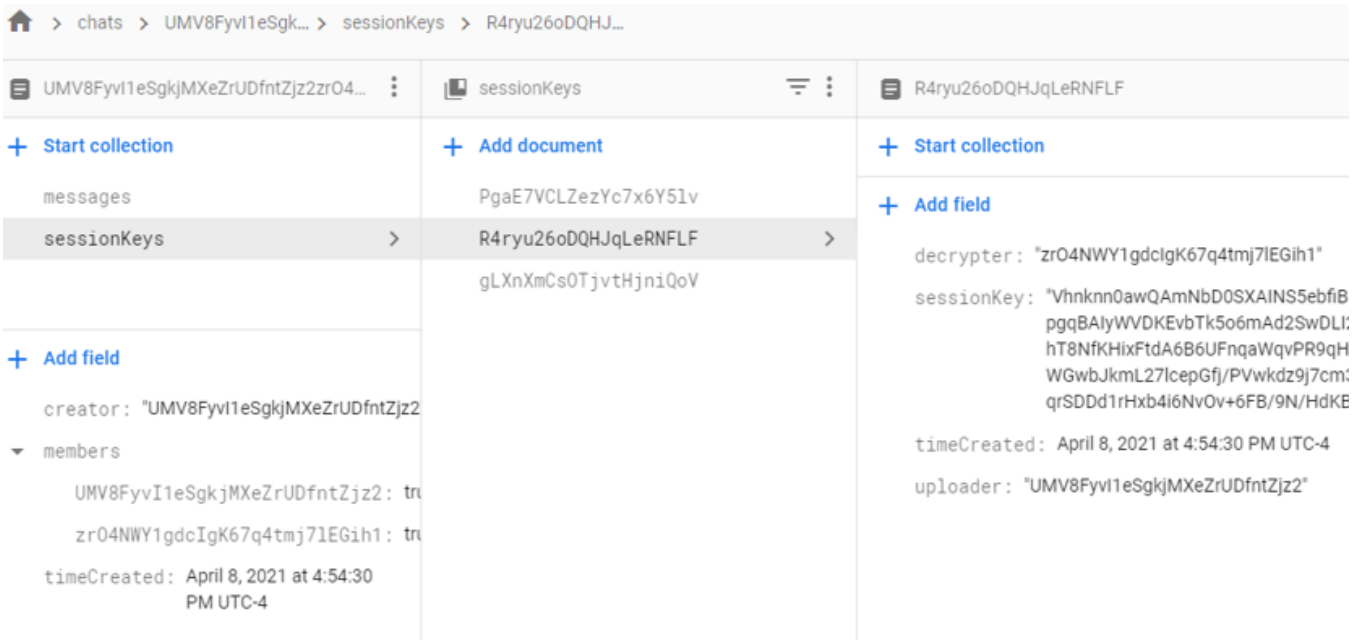

Figure 10: Chats Collection - Keys Sub-Collection

Cloud Firestore A cloud-based document database. Documents represent an object of data, and data is stored in string-value pairs. Used primarily to store user and chat information.

Firebase Cloud Messaging Allows for the transfer of data between client devices and the server. Used to generate device tokens and send notifications.

Additionally, EncChat uses some key Java packages, which are as follows:

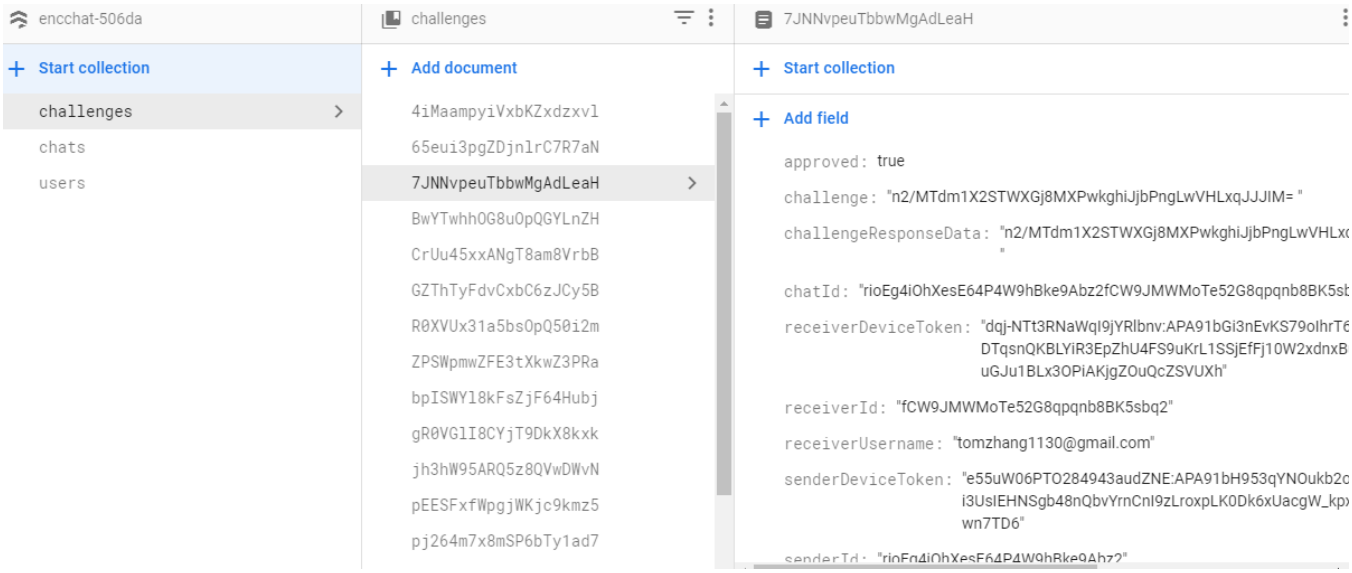

Figure 11: Challenges Collection

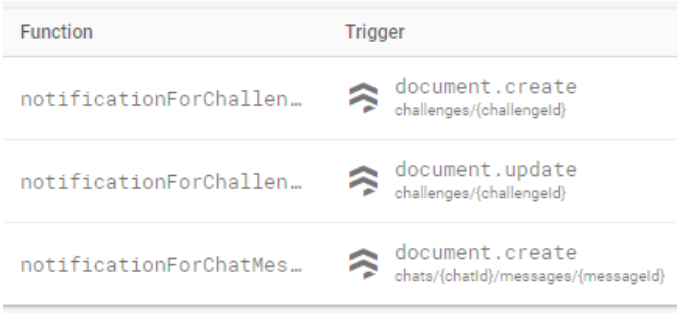

Figure 12: Cloud Functions

java.security Provides Key classes, such as PrivateKey and PublicKey.

javax.crypto Provides the Cipher class, which is used to perform encryption and decryption. Used for the "RSA/ECB/PKCS1Padding" and "AES/ECB/PKCS5Padding" algorithms.

android.content.SharedPreferences Allows use of SharedPreferences, which stores string-value pair information inside the app. Used to save session keys locally to the device.

Also, challenge-response authentication used Google Cloud Functions to send notifications. Cloud functions trigger when certain collections of documents in the database are modified. The functions were written using TypeScript, a variation of JavaScript, and deployed to the Firebase server using Node.js.

EncChat's source code is separated into three major packages: *objects, utils* and *activities*. Objects consists mostly of data classes. Utils contains various reusable tools, and it is where most cryptography-related code is located. Activities contains the code for each page within the app and makes use of the cryptographic functions. The file structure is as follows, but less important files are excluded, and only only the classes most directly related to the app's cryptographic functions will be described in detail.

Objects

- User
- Chat
- ChatMessage
- SessionKey

Utils

- CryptoHelper
- RSAAlg
- SecretKeyAlg
- SharedPreferencesHandler

Activities

- MainActivity (login)
- RegistrationActivity
- HomePageActivity
- ChatActivity

3.1 Objects

Describing the core objects in the application will provide an overview of the actors working to make the chat work. Two users will have one chat. Each chat has chat messages. Each chat message will be encrypted and decrypted by session keys. These concepts can be represented as classes. In the Objects module, classes are mainly data classes that hold information, which is manipulated in other files, such as the activities. First, Figure 13 displays the attributes of the User class. Some important attributes include the *id,* which is the id string generated by Firebase Authentication, and *publicKey* and *privateKey,* which store the RSA public and private keys that are unique to each user. The *regenerated* attribute helps identify if

a user logged in from a new device, and the *deviceToken* attribute is used to identify the device.

```
public class User implements Parcelable {
    private String id;
    private String username;
    private String password;
    private Key publicKey;
    private Key privateKey;
    private Key signedPrekeyPublic;
    private Key signedPrekeyPrivate;
    private boolean regenerated; //represer
    private String deviceToken;
```

Figure 13: User Class

The attributes of the Chat class are shown in Figure 14. The *id* attribute is created by combining the ids of the two chatters. In terms of other attributes, the Chat class is primarily used on the Home Page in order to display a user's existing chats. Thus, its attributes show the last information in the chat, such as the last message. Furthermore, information on the other user in the chat is stored as well. The current user does not have to be specified, because the data can be retrieved the from the User class.

```
public class Chat {
    private String id; //firestore document id
    private String lastUsername;
    private String lastMessage;
    private Date lastMessageTime;
    private String otherUserId;
    private String otherUserUsername;
```

Figure 14: Chat Class

The ChatMessage class (Figure 15) contains information about each message sent in a chat. The *message* attribute contains the encrypted message string, while the *time* attribute stores the message send time, which is important when selecting a decryption key.

```
public class ChatMessage {
    private Date time;
    private String message;
    private String senderId;
    private boolean ownMessage;
```

Figure 15: Chat Message Class

The SessionKey class (Figure 16) is used to hold the AES session key bytes. The *keyId* is composed of the chat id and creation timestamp, which is held in *timeCreated*.

```
public class SessionKey {
    private String keyId;
    private byte[] sessionKey; //
    private Timestamp timeCreated;
```

Figure 16: Session Key Class

3.2 Utils

3.2.1 CryptoHelper. The core class that handles cryptographic tasks is the CryptoHelper class. Most functions in the class are general, static functions, that can be called whenever encryption is needed. The first task that CryptoHelper handles is key encoding and decoding. For keys to be properly saved, locally or to Firestore, they must be converted to a string through Base64 encoding. When the key needs to be used, it is decoded from a string. Implementation details for encodeKey() and decodeKey() are shown in Figures 17 and 18 respectively. Notably, public and private keys require different key specifications (X509 or PKCS8) to be properly loaded.

```
/**
 * Base64 encodes a public or private key
 * @param key - Any Key
 * @return String - The base64 encoded key.
 */
static public String encodeKey(Key key){
    try {
        byte[] keyBytes = key.getEncoded();
        String keyStr = Base64.encodeToString(keyBytes, Base64.DEFAULT);
        return keyStr;
    }
    catch (NullPointerException e){
        //e.printStackTrace();
        Log.d("CryptoHelper encodeKey","Key was null");
        return null;
    }
}
```

Figure 17: encodeKey()

Furthermore, CryptoHelper contains encryptMessage() (Figure 19) and decryptMessage() (Figure 20), which are the main functions used to encrypt and decrypt messages when using the chat page. The encryption function produces a Base64-encoded ciphertext string, and the decryption function produces the plaintext string. The session key is passed in as bytes. These functions perform cryptography using the SecretKeyAlg class, which uses the AES algorithm.

CryptoHelper also helps the chat page determine which session keys a message should be decrypted with through the selectSessionKey() function (Figure 21). The function passes in a list of the session keys and a message's timestamp. The latest session key is at the end of the list. The message timestamp is compared to each key's timestamp through a backwards for loop, and if the message timestamp is greater, that signifies the correct session key.

Apart from the detailed functionalities, CryptoHelper provides the code for key generation and encryption of session keys.

```java
/**
 * Base64 decodes an encoded key string
 * @param keyStr - Base64 encoded key string
 * @param keyType - PUBLIC or PRIVATE
 * @param alg
 * @return Key - The public or private key
 */
static public Key decodeKey(String keyStr, KeyType keyType, String alg) throws NoSuc
    try {
        byte[] sigBytes = Base64.decode(keyStr, Base64.DEFAULT);
        KeyFactory keyFact = KeyFactory.getInstance(alg);

        Key key = null;
        //creates a public key
        if (keyType == KeyType.PUBLIC) {
            X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(sigBytes);
            key = keyFact.generatePublic(x509KeySpec);
        }
        //creates a private key
        else if (keyType == KeyType.PRIVATE) {
            PKCS8EncodedKeySpec pkcs8KeySpec = new PKCS8EncodedKeySpec(sigBytes);
            key = keyFact.generatePrivate(pkcs8KeySpec);
        }
        return key;
    }
    catch (NullPointerException e){
        Log.d("CryptoHelper decodeKey","Key was null");
        return null;
    }
}
```

Figure 18: decodeKey()

```java
/**
 * Symmetrically encrypt a text message
 * @param message - The text message to encrypt
 * @param secretKey - Symmetric key for encryption
 * @return String - The encrypted message after base64 encoding
 */
static public String encryptMessage(String message, byte[] secretKey){
    //perform the encryption
    SecretKeyAlg secretKeyAlg = new SecretKeyAlg(secretKey);
    byte[] encryptedMessageBytes = new byte[0];
    try {
        encryptedMessageBytes = secretKeyAlg.encrypt(message).get("ciphertext");
    } catch (Exception e) {
        e.printStackTrace();
    }
    String encryptedMessage = Base64.encodeToString(encryptedMessageBytes, Base64.DEFAULT);

    System.out.println("Encrypted Message: ${encryptedMessage}");

    return encryptedMessage;
}
```

Figure 19: encryptMessage()

```java
/**
 * Symmetrically decrypt a base64 encoded text message
 * @param encodedMessage - The text message to decrypt. It must be base64 decoded first
 * @param secretKey - Symmetric key for encryption/decryption
 * @return String - The decrypted plaintext
 */
static public String decryptMessage(String encodedMessage, byte[] secretKey){
    //base 64 decode the message
    byte[] message = Base64.decode(encodedMessage, Base64.DEFAULT);

    SecretKeyAlg secretKeyAlg = new SecretKeyAlg(secretKey);
    String plaintext = null;
    try {
        plaintext = secretKeyAlg.decrypt(message);
    } catch (Exception e) {
        e.printStackTrace();
    }

    return plaintext;
}
```

Figure 20: decryptMessage()

```java
static public SessionKey selectSessionKey(List<SessionKey> sessionKeyList, Timestamp messageTime){
    // Start from highest and iterate down.
    for (int i = sessionKeyList.size() - 1; i >= 0; i--){
        SessionKey currentSessionKey = sessionKeyList.get(i);

        //If message timestamp higher than the key timestamp, use it
        if(messageTime.getSeconds() >= currentSessionKey.getTimeCreated().getSeconds()){
            return currentSessionKey;
        }
    }
    return null;
}
```

Figure 21: selectSessionKey()

3.2.2 RSAalg/SecretKeyAlg. The main encryption algorithms that EncChat uses are in the RSAAlg and SecretKeyAlg (AES) classes. Both functions implement an interface called CryptoAlg, which means both classes have an encrypt() and decrypt() function. Implementation details for SecretKeyAlg are shown in Figure 22. The Cipher class from javax.crypto is used to encrypt and decrypt. RSAAlg's implementation is mostly similar to SecretKeyAlg, but it allows a public or private key to be used for encryption.

```java
@Override
public HashMap<String, byte[]> encrypt(String textToEncrypt) throws NoSuchPaddingExcepti
    final SecretKeySpec keySpec = new SecretKeySpec(secretKey, "AES/ECB/PKCS5Padding");
    final Cipher cipher = Cipher.getInstance(ENCRYPTION_ALG);
    cipher.init(Cipher.ENCRYPT_MODE, keySpec);

    //encrypt using the secret key
    byte[] ciphertextBytes = cipher.doFinal(textToEncrypt.getBytes());

    //build hashmap
    HashMap<String, byte[]> ciphertext = new HashMap<String, byte[]>();
    ciphertext.put("ciphertext", ciphertextBytes);
    return ciphertext;
}

@Override
public String decrypt(byte[] bytesToDecrypt) throws NoSuchPaddingException, NoSuchAlgori
    final SecretKeySpec keySpec = new SecretKeySpec(secretKey, "AES/ECB/PKCS5Padding");
    final Cipher cipher = Cipher.getInstance(ENCRYPTION_ALG);
    cipher.init(Cipher.DECRYPT_MODE, keySpec);

    //decrypt
    byte[] plaintextBytes = cipher.doFinal(bytesToDecrypt);

    //convert to string
    String plaintext = new String(plaintextBytes);

    return plaintext;
}
```

Figure 22: SecretKeyAlg encrypt() and decrypt())

3.2.3 SharedPreferencesHandler. Finally, one of the last key Util classes is SharedPreferencesHandler, which retrieves and saves data into SharedPreferences. Some of the data it saves includes the identity private key and chat session keys. SharedPreferences is accessed through the android.content.SharedPreferences library. Each important piece of data has a "save" function, which writes data to SharedPreferences, and a "get" function, which loads data from SharedPreferences. For example, session keys have a saveSessionKey() function (Figure 23) and getSessionKey() function (Figure 24). To save a value, an id is generated. The session key id is the chat id and key timestamp. Then, the id and Base64-encoded key are saved with the SharedPreferences.Editor class. Then, getting the key just requires the id and the getString() function.

```java
/**
 * Saves a chat's session key. Needs a timestamp
 * @param chatId - the id of the chat the session key is used for
 * @param keyCreationTime - the Timestamp for when the session key was generated
 * @param sessionKey - the actual session key used for encrypting the chat
 */
public void saveSessionKey(String chatId, Timestamp keyCreationTime, byte[] sessionKey){
    SharedPreferences.Editor editor = sharedPref.edit();
    //encode key
    String encodedKey = Base64.encodeToString(sessionKey, Base64.DEFAULT);

    //save to sharedPreferences. id is chat id + timestamp as seconds
    String sessionKeyId = chatId + "_" + keyCreationTime.getSeconds();
    editor.putString(sessionKeyId, encodedKey);
    editor.apply();
    Log.d("saved session key id", sessionKeyId);
}
```

Figure 23: saveSessionKey()

```
/**
 * Gets a chat's session key. Needs a timestamp
 * @param chatId - the id of the chat the session key is used for
 * @param keyCreationTime - the Timestamp for when the session key was generated
 * @return byte[] - the actual session key used for encrypting the chat
 */
public byte[] getSessionKey(String chatId, Timestamp keyCreationTime){
    String encodedSessionKey = sharedPref.getString(chatId + "_" + keyCreationTime.getSeconds(), "");
    Log.d("Encoded session key", encodedSessionKey);
    //if you didn't get any value back, don't try to decode
    if(!encodedSessionKey.equals("")){
        byte[] sessionKey = Base64.decode(encodedSessionKey, Base64.DEFAULT);
        return sessionKey;
    }
    return null;
}
```

Figure 24: getSessionKey()

3.3 Activities

3.3.1 MainActivity/RegistrationActivity. The MainActivity handles user login, while the RegistrationActivity handles user registration. Both interact with Firebase Authentication through the firebase.auth.FirebaseAuth package. Users register using the createUserWithEmailAndPassword() function, and they login using signInwithEmailAndPassword(). Registration creates a FirebaseUser object, which contains login information. However, only the id from the FirebaseUser is used, and it is saved into a User object. The id is used to query the database for additional user information, which is again saved to the User object. The User object is then passed to the HomePageActivity.

3.3.2 HomePageActivity. On the home page, a user's chats, with the latest message, are displayed in unencrypted form. On the page load, the loadChats() function retrieves all the chats for the current user id. For each chat, the app retrieves the correct session key from SharedPreferences with SharedPreferencesHandler's getCorrespondingChatKey() function. That key is used to decrypt the message with CryptoHelper's decryptMessage() function.

The home page can also be used to create a chat. The createChat() function creates a new chat document in Cloud Firestore, using the id of the current user and the id of the other user to generate the chat id. A session key is generated, encrypted, and uploaded to the database as well. Database calls are performed using the firebase.firestore.FirebaseFirestore package.

The home page also generates new session keys for chats. In the checkForExpiredKey() function, a new encrypted session key is uploaded to the database if the latest chat key is more than seven days old. The key is also saved locally to SharedPreferences.

3.3.3 ChatActivity. The chat activity is core functionality of the app, allowing two users to communicate with encrypted messages. The chat activity performs two major actions: sending encrypted messages and reading encrypted messages. Sending messages is performed with the sendMessage() function, which utilizes the CryptoHelper's encryptMessage() function and uploads the encrypted chat message to the database in the following document collection: chats/chat document id/messages

To read a message, the app queries for the messages collection in the database. First, the loadSessionKeys() function is called to get session keys from the database. The session key documents provide key ids, and the keys are retrieved from SharedPreferences using the SharedPreferencesHandler. All keys are loaded into the sessionKeys list. If a key is not in SharedPreferences, it is decoded using the CryptoHelper. Then, the setUpChatListener() function reads the chat messages from the database, and selects the correct session key to decrypt each message by using CryptoHelper's selectSessionKey() function. Each decrypted message is pushed to a chat message adapter, which displays the message to the screen.

4 COMPARISON

EncChat is not the only mobile end-to-end encrypted chat application. Two popular ones are Signal and WhatsApp, which are available for Android and iOS. Both applications have had significantly more time and effort put into their application design and encryption scheme, and as a result, they are generally superior to EncChat. However, both can be used as examples to improve EncChat, and flaws with those applications can help reveal what EncChat should not do in the future.

Both Signal and WhatsApp are based on the Signal Protocol, which is a security protocol specifically designed to provide end-to-end encryption for chat. Like EncChat, the Signal Protocol is a hybrid encryption scheme, but it uses a larger amount of public and private keys to guarantee mutual authentication and forward secrecy.

To initiate a chat with the Signal Protocol, two users perform Extended Triple Diffie-Hellman (X3DH) [2]. The purpose of X3DH is to generate a secret symmetric key between two users. Both users have public-private key pairs called the identity key pair (IK), signed prekey pair (SPK), and the one-time prekey pairs (OPK1, OPK2, etc.). Generally, in the key exchange, only the public keys are involved.

Suppose User A wants to communicate with User B. First, User B must push his public keys to the key server. User B pushes IK_B, SPK_B, and one of the OPK_B. Additionally, User B generates a prekey signature, by signing the SPK with his IK. The signature will be used to provide authentication.

Afterwards, User A retrieves B's keys from the server. To verify the information is truly User B's, the prekey signature is verified by decrypting it with IK_B and comparing it to SPK_B. If the keys match, the information is valid and the process can proceed. Then, User A creates a temporary ephemeral key pair, EK_A and performs the triple Diffie-Hellman (Figure **??**). Each Diffie Hellman achieves a different goal, such as mutual authentication or forward secrecy. The secret key SK is created by applying a key derivation function (KDF) on the Diffie Hellman results.

User A then sends the IK_A and EK_A to User B, and User B can perform triple Diffie-Hellman to get the exact same secret key.

The Signal Protocol's secret key derivation scheme has the same goal as EncChat's session key creation scheme: generating a 32-byte symmetric key for two users to use. However, the Signal Protocol has the advantage of forward secrecy. For EncChat, safety of session keys depends on the safety of a user's private key. If the private key is compromised, session keys are compromised as well. However, because the Signal Protocol uses an ephemeral key in its key agreement algorithm, it ensures that every secret key will have an element of randomness to it, making each key tougher to crack. The introduction of randomness in the Signal Protocol highlights a potential vulnerability of EncChat: if an attacker wants to attack EncChat, it is much more efficient to attack the user private keys. While the amount of effort needed to crack a 2048-bit RSA key is

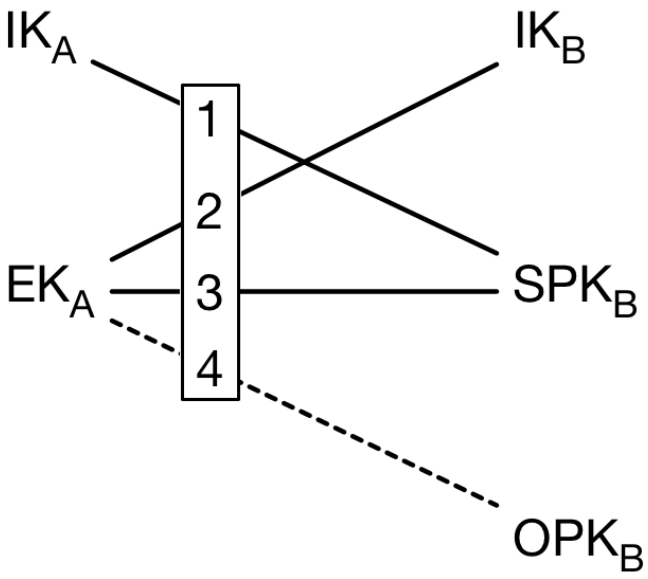

Figure 25: Triple Diffie-Hellman

still incredibly high, it may be better to switch EncChat to Diffie-Hellman-based key exchange with forward secrecy in the future, so that encryption is less reliant on a single point.

One small weakness that EncChat and the Signal Protocol both share is that they do not give anonymity preservation. Both requires users to upload public keys to key servers. Therefore, each user is clearly identified. Lack of anonymity preservation is not a serious issue as long as encryption keys are safely kept, but it can make it easier for attackers to choose the specific people they want to target. Furthermore, messages are passed to devices through the servers. Since the server is a logically centralized server, if an attacker wants to intercept messages, it is easier to obtain them than with a decentralized server. To solve these issues, one tactic EncChat could use is to switch to a peer-to-peer decentralized chat scheme. This scheme would be tough to implement because it would require complete restructuring, as well as the removal of Cloud Firestore. However, it will be possible through use of Android BroadcastReceiver and WifiP2PManager classes, which will allow devices to work as both clients and servers.

5 CONCLUSION

Ultimately, EncChat provides a fully functional end-to-end encrypted chat application. Chats messages are encrypted with switching session keys, and session keys are transferred through encryption by public key. User transparency is held because users do not need to manually deal with keys and can use the chat as if it were an unencrypted chat. EncChat's switching session keys provide security by preventing attackers who have obtained one session key from fully reading a chat. The weak point of the encryption scheme is the identity private key; if it is leaked, attackers will have full access to a chat. However, due to the large key size (2048 bit), private keys are still too computationally intensive to brute force. While other encrypted chat applications are still superior, EncChat offers sufficient security, while remaining a fairly low profile application.

5.1 Future Improvement

Future improvements can be divided into feature improvements and structural improvements. Feature improvements will add new features, while structural improvements will improve the application flow to provide better cryptography. First, some feature improvements are listed.

Currently, EncChat only supports chats between two users. One area of improvement is to add group chat support, where multiple users can chat. For group chat to work, the system would need a way to send session keys to each user in the chat. One approach would be to encrypt the session key with the public key of each other user in the chat. Therefore, a chat between one user and three other users would upload three encrypted session keys to Cloud Firestore. Then, each user would decrypt their session key with their private key.

Another area of improvement would be to support multiple types of data upload. Currently, the chat only supports text messages, but it would be beneficial to include video, audio, and image messaging as well. Sending encrypted multimedia messages is much more difficult due to the size of the data. A naive approach would be to encrypt it, Base64 encode the data to a string, and upload it to Cloud Firestore, but data like videos is much too large to do so realistically. Alternatively, one approach demonstrated by Google Messages [3] is the encrypt the data with an AES key, and upload the data to a content store (e.g. Firebase Storage). Then, the sender sends a message with encrypted file metadata, the file encryption key, and a link to the file. The receiver decrypts the message, and uses the link and key to decrypt the file.

Finally, one structural improvement would be to reduce the amount of unencrypted metadata stored on Cloud Firestore. The most important types would be the user ids and message timestamps. User ids are important because they are used to start chats with other users, and chat ids are derived from user ids. Message timestamps are crucial because they are used to check against the key timestamps in SharedPreferences and grab the correct key for message decryption. Because of the way chats query for users and messages from the database, supporting encryption of the metadata would require a complete restructuring of the application.

REFERENCES

[1] [n.d.]. Firebase Authentication. https://firebase.google.com/docs/auth.
[2] Moxie Marlinspike. 2016. The X3DH Key Agreement Protocol. https://signal.org/docs/specifications/x3dh/. (11 2016).
[3] Emad Omara. 2020. Messages End-to-End Encryption Overview. (11 2020).