BumpChat

A Secure Mobile Communication System

Brian Kammourieh

04/2020

Table of Contents

1. Abstract

BumpChat is a secure messaging application for Android devices. The app makes use of the Near Field Communication (NFC) hardware on devices to securely transmit cryptographic keys. Communication between devices after pairing is handled securely by a web server, through a simple PHP driven API. Data is encrypted at all stages: at-rest on end devices, in-transit to the server, at-rest on the server. A user defined password is used to protect data on the Android device and is required to be entered on every startup. There are two primary encryption algorithms used for message security (AES-256[1]) and inbox verification with the server (RSA-2048[2]).

2. Introduction

We live in a world where we are more connected than ever. Individuals are able to communicate with each other over a vast array of services. Each service however comes with trade-offs, usually between ease-of-use and privacy. There is a subset of the global population whose privacy is constantly under threat. There are journalists, activists, and citizens of totalitarian/authoritarian regimes who can be persecuted for their beliefs. When state actors are the adversary, trusting the communication infrastructure is not an option. Some regimes have been known to access the private messages of their citizens to single out individuals based on their religion or to stop the spread of information. Whistleblowers within a government also need a secure means of communication with journalists and watchdog groups. A prime example is Edward Snowden, who implemented extreme measures to ensure the information he possessed was able to get into the hands of journalists, without the US government stopping him. The encrypted email service LavaBit was forced to shut down or else they would have been forced to turn

Commented [NM1]: The article you referred will raise sensitivity. Simple remove it.

over their private key that encrypted *all* of their user's private communications, just so the government could get the messages of a single user…widely believed to be Edward Snowden.

The goal of BumpChat is to provide a secure communication infrastructure that is both open source and privately hosted. An individual or group of individuals can spool up their own secure server on either their own hardware or through a cloud server. BumpChat is composed of a mobile messaging app for the Android platform and a backend API developed in PHP. Security is provided through the use of end-to-end encryption, at rest-encryption, RSA authentication, and AES message encryption. The addition of NFC to the pairing process removes the need to trust internet infrastructure with key data that underpins the security of communications.

One of the downsides of the highly regarded Signal [4] messaging app is the need for a public identifier that is held on their servers. This allows a third-party entity to verify that a user is part of the network by entering either their phone number or username into their contacts. Signal will verify that a user has an account with Signal without notifying the end user. With the design of BumpChat it is impossible to determine or verify two parties are communicating with each other without having both devices unlocked and in your hands.

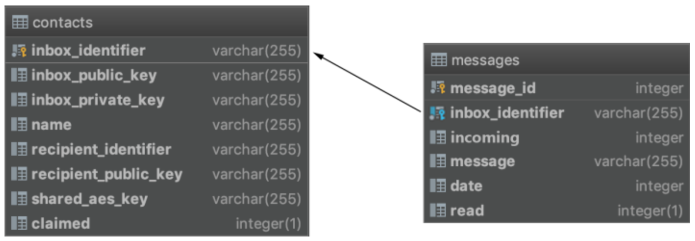
The are a few benefits to using BumpChat over similar commercial applications (Signal, WhatsApp, Facebook Messenger). They key exchange is done person-to-person without a central authority involved which puts the trust between users. A major benefit to the key exchange process is the inability to identify if a user is using the system. With all other applications, there is some identifier (email or phone number) that another user/attacker can use to find someone. With BumpChat, the identifiers are cryptographically generated and only used for communication between two parties. If the server database is ever compromised, the amount of meta-data leaked is minimal by design. It can not be determined how many inboxes are owned by a single device or even who messages are sent from. The goal is to prevent meta-analysis from leaking data about the users. By focusing on the worst-case scenario of infrastructure compromise, the system is designed to keep as much data confidential as possible. Commercial solutions focus on features where BumpChat focuses on anonymity and security.

3. Architecture

The system is split into two distinct applications. A native Android application developed in Java and a server-side API developed in PHP. Both the client and server have a local database that is encrypted at rest. The clients and server communicate over HTTPS to ensure data in encrypted in-transit as well. During the pairing process that links two clients together, they each register an inbox on the server using their individual public RSA keys. The hashed RSA public key is used as an inbox identifier which functions similar to an email address except that the client has no control over choice of identifier. The RSA key also doubles as an authentication key when a user wants to register, retrieve, or send messages.

3.1 Client Database

The database on the Android client is built on SQLite running the SQLCipher[5] extension. The SQLCipher extension allows the database to be encrypted at rest with AES-256. The database is unlocked at each application open with a user supplied password. The schema of the client database is shown below in *Figure 1*.



contacts	
inbox_identifier	varchar(255)
inbox_public_key	varchar(255)
inbox_private_key	varchar(255)
name	varchar(255)
recipient_identifier	varchar(255)
recipient_public_key	varchar(255)
shared_aes_key	varchar(255)
claimed	integer(1)

messages	
message_id	integer
inbox_identifier	varchar(255)
incoming	integer
message	varchar(255)
date	integer
read	integer(1)

Figure 1. Client Schema - SQLite

A contact represents a pairing between two devices. The *inbox_identifier* is a SHA-256 hash of the public RSA key. This allows sending and retrieving messages using a smaller identifier than the entire public RSA key. The *inbox_public_key* and *inbox_private_key* fields are the Base64 encoded strings of the respective RSA keys. The Base64 encoding allows the keys to be easily stored as string and converted back to a key when needed. The *name* field is name that the user chose for their recipient and shows on the front-end of the Android application. As the users must meet face-to-face to add each other as contacts, the *name* field can either be the other parties real name or an alias so the user knows who they are communicating with when selecting a chat. It is stored only in the encrypted database on the client and not sent to the server. The *recipient_identifier* and *recipient_public_key* are the counterparts for the recipient inbox. The *shared_aes_key* is derived from a Diffie-Hellman exchange during pairing that produces an AES256 shared key that is used to symmetrically encrypt messages. The *claimed* field denotes that the inbox has been created on the server and verified with an initial challenge response. The server will not allow any actions to happen until the inbox has been claimed.

3.2 Server Database

The database on the server is built on MariaDB[3] with Data-at-Rest encryption enabled. The server stages messages which are deleted upon retrieval.

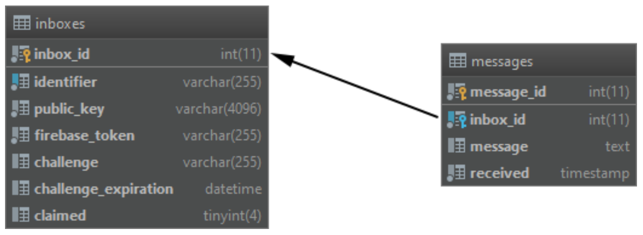


Figure 2. Server Schema – MariaDB

There are only two tables on the server, one for inboxes and one for messages. The *inbox_id* is an auto-generated primary key and is not exposed to the client. The *identifier* is the SHA-256 hash of the RSA-2048 *public_key* field. The *firebase_token* was initially designed to store a Google Firebase token which is used to send push notifications to a device. It was deemed to be a security risk as the token is always the same for a device and would allow an attacker with access to the database to determine which inboxes are linked to a single device. The *challenge* and *challenge_expiration* fields are used for authentication. A random string is generated and encrypted using the public key for the inbox. The device must respond with the decrypted challenge within 5 minutes of generation. Once the challenge has been answered, the field is cleared to defend from replay attacks. The *claimed* field denotes that the inbox has been created on the server and verified with an initial challenge response. The server will not allow any actions to happen until the inbox has been claimed.

The messages table is very simple with a foreign key *inbox_id* that references the inbox the messages were sent to. The *message_id* is an auto-generated primary key that is not exposed to the client. The *message* field holds the body of the message with a *received* timestamp that is sent to the client. Messages are cleared once they are successfully downloaded through an API call.

3.3 Communication Structures

Direct Client-Client Communication over NFC

A key feature in the program is the client key sharing process which is achieved through NFC communication. NFC allows the passing of data wirelessly between two devices in extremely close proximity, usually within ~4cm with devices facing back-to-back. The key sharing process is explained in detail in *Section 4.7 Contact Creation (Activity)*. The goal of the key sharing process is for each user to share their RSA public key with their partner and generate a shared AES key to encrypt messages sent between parties. The RSA public key is shared so the receiving party can hash it and derive the inbox_identifier of the sending party. An overview of the key sharing process is shown below.

1. **Initial Key Generation -** Each device generates an RSA-2048 key, a 2048-bit Diffie-Hellman key, and an HMAC-SHA256 salt.
    a. The RSA key is used for ownership verification of an inbox with the server. It is also used to generate the inbox_identifier by hashing the public key as described in *Section 3.2*.
    b. The Diffie-Hellman (DH) key is used in part to generate the AES shared key. The DH key allows both parties to securely generate the AES key even if the NFC communication is sniffed during the sharing process.
    c. The HMAC-SHA256 salt is a random 32-byte array that is passed into the Key Derivation Function along with the DH key to generate the AES shared key.
2. **Key Sharing** – With NFC enabled on the devices, bringing them in proximity to one another will put the devices into Beam mode. This allows a party to touch their screen and send data one-way to the recipient. The sharing process takes 3 Beams in order to complete. Each device tracks their progress in the sharing process using a bit-masked byte called TransferState. The TransferState holds three pieces of information: user sent keys, user received keys, partner verified key receipt.
    a. User sending to Partner
        i. User shares their TransferState, RSA public key, DH public key, and HMAC-

SHA256 salt with the partner.

    ii. The first party to receive uses the HMAC-SHA256 salt of the other party. In this case, the partner overwrites their local salt with that of the user.

    iii. The partner can't immediately reply back successful receipt of keys but is able to update their TransferState to denote that they successfully received keys. When the partner makes their transfer, the user will be notified that the keys were received.

b. **Partner sending to User**

    i. Partner shares their TransferState, RSA public key, DH public key, and HMAC-SHA256 salt with the user.

    ii. During this share, the user made aware through the TransferState that the partner successfully received their keys.

c. **User sending to Partner (again)**

    i. User shares their current TransferState to the partner.

    ii. This final share is needed to inform the partner that the user has successfully received the partner's keys.

3. **AES-256 Key Generation** – Now that both devices possess their partner's public DH key and a shared salt, the AES shared key can be generated. Each user follows a Diffie-Hellman key agreement using DH private key with partner's DH public key. This allows them both to derive the same 256-byte DH shared secret. The new DH shared secret along with the salt are passed into HMAC-SHA256 Key Derivation Function (as specified in RFC 5869 [7]). The AES shared key solely used to encrypt messages sent between parties.

Client-Server Communication

Communication between the client and server is used to register inboxes, send messages, retrieve messages, and clear inboxes on the server. All communications between the client and server are over

HTTPS. Below is the high-level overview of the sequence of events for primary actions and where they happen.

1. **Registering an inbox –** Inboxes are registered automatically during the key transfer process described in *Section 4.7*.
    a. Sender device generates a public key pair for a new inbox locally
    b. Sender device sends the inbox public_key to the server to create a new inbox
    c. Server responds with a challenge encrypted with the public key
    d. Sender device sends the decrypted challenge to server to claim the inbox
    e. Server acknowledges that the inbox was successfully registered at the server.
2. **Sending a message** – Messages are sent using the interface described *Section 4.8 Messages List*.
    a. Sender device requests a challenge from the server by sending their inbox_identifier
    b. Sender device sends decrypted challenge, recipient_identifier, and the message
    c. Server acknowledges that the message successfully stored at the server's message table
3. **Retrieving messages –** Messages are retrieved automatically while the *Section 4.8 Messages List* activity is open and in the foreground. New messages are retrieved from the server and added to the local database.
    a. Retrieving device requests a challenge from the server by sending their inbox_identifier
    b. Server returns a list of messages for the inbox
4. **Clearing messages** – Messages are requested to be cleared by the client software automatically after the previous step succeeds. The message_id of the last message is used to clear all messages up to that one.
    a. Retrieving device requests a challenge from the server by sending their inbox_identifier
    b. Server replies a challenge.

Commented [NM2]: What does "recipient inbox_identifier" mean? The inbox belongs to the communication between the sender and recipient. There is no recipient inbox_identifier: There are two separate things: recipient identifier and inbox_identifier

Commented [BK3R2]: I corrected this to the recipient_identifier.

Commented [NM4]: Why does the server return an error? Is this a bug in the app? Do you mean there was no message for the recipient in the inbox? If that is the case, the server should reply "no message" or reply nothing. If the source of error is something else, mention that.

Commented [BK5R4]: Errors only occur when the client failed to answer the challenge correctly. I have removed the mention of this as that is only a problem when someone tries to attack the system and not a normal occurrence.

Commented [NM6]: What if the recipient has not been up to immediately receive the message? Does the sender check whether the recipient has received the message and then request to clear the message from the server's table? Explain that.

Commented [BK7R6]: I think I have this clarified. An inbox only contains messages *to* the user. That means that the user that receives the messages is responsible for retrieving them and then deleting them once retrieved.

c. Retrieving device sends decrypted challenge, their inbox_identifier, and the max id of the message to delete

Commented [NM8]: Explain this

d. Server acknowledges that the message was successfully deleted from the server messages table

4. Implementation

4.1 Server Overview

The server software is written in PHP and is accessed over HTTPS. Currently it resides in an Ubuntu VPS that resides on my home server. HTTPS is achieved through a free 2048-bit LetsEncrypt SSL certificate. The server is accessed through a web API that supports 6 endpoints which allow access to inboxes and messages. The database is accessed through the MySQLi database driver using parameter binding to prevent from SQL injection attacks.

4.2 Server API Endpoints

Each endpoint requires input data passed over HTTP POST. This ensures that any secret data does not get accidentally stored in web server logs as would be the case with HTTP GET requests. All endpoints return data in a JSON encoded format. Some endpoints are protected which require a challenge to be generated before the endpoint is accessed. When the user requests a challenge generation, the server will send a random challenge string encrypted with the inbox's public RSA key and the client must respond with the decrypted challenge before the server allows the action to take place. In this way, an attacker cannot request or forge messages on a user's behalf.

- **Challenge/generate.php** – This generates a new challenge for an inbox and saves it to the database. It is required to generate a challenge before any protected endpoint is accessed.
  Required POST data: inbox_identifier
- **Inbox/register.php** – This creates a new inbox using the RSA public key in PEM encoded

format. Returns the initial encrypted challenge that is used to verify the inbox.

Required POST data: public_key_pem

- **Inbox/register_verify.php** – This finalizes the creation of an inbox. Required to be done before any protected endpoints are accessed.

  Required POST data: identifier, challenge_response

- **Message/clear_all.php** – Clears all messages in an inbox up to an upper message id. The upper message id is needed to protect the client from attempting to clear messages it has not retrieved yet. This is a protected endpoint.

  Required POST data: identifier, challenge_response, upper_message_id

- **Message/retrieve_all.php** – Returns all incoming messages currently staged on the server. Client is responsible for clearing messages after confirming successful delivery. This is a protected endpoint.

  Required POST data: identifier, challenge_response

- **Message/send.php** – Sends a message to a specified inbox. This is a protected endpoint.

  Required POST data: identifier, challenge_response, recipient_identifier, message

4.3 Android Client Overview

The client Android application is programmed in Java and targets Android 9 and above. The application is composed of a series of Android activities which are screens in the applications user interface. NFC hardware is required in the device for the application to function. Storage is done locally with an SQLite database file located within the apps storage area and is access through the Android Room Storage Persistence Library[11]. The library acts an abstraction layer over direct database access to the SQLite database. The library integrates with the SQLCipher encryption extension that is used to encrypt the database at-rest. The app consists of three primary sections: database unlock, message inbox, message list. Each section has activities that branch off to access specific functionality. The rest of *Section 4* details each activity individually. Currently the application is only available as an unsigned .apk file but part of the future work entails releasing through the Google Play store.

4.4 Master Password Setup (Activity)

This screen comes up on the first time the application is opened. The user is prompted to create a 12-character minimum password as shown in *Figure 3*. The input fields give visual feedback by turning green when the minimum character length has been reached and when the password confirmation matches. The **SET MASTER PASSWORD** button is unlocked once the both constraints are met as shown in *Figure 4*. The master password is used to encrypt the SQLite database and is required to be entered on every application open to decrypt the database. The user is sent to their inbox after successful decryption.

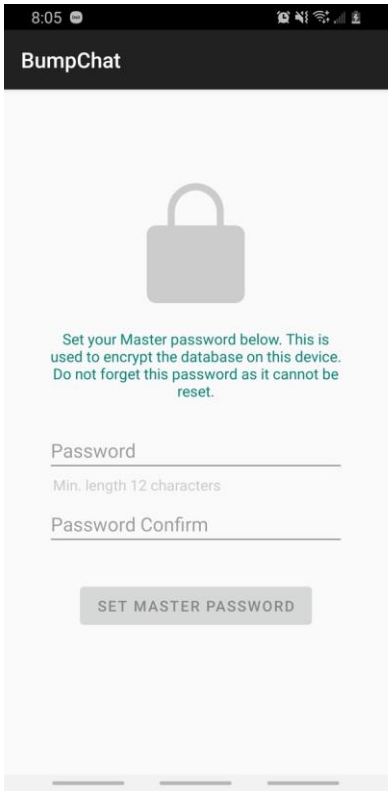

8:05
BumpChat
Set your Master password below. This is used to encrypt the database on this device. Do not forget this password as it cannot be reset.
Password
Min. length 12 characters
Password Confirm
SET MASTER PASSWORD

Figure 3. Master password setup

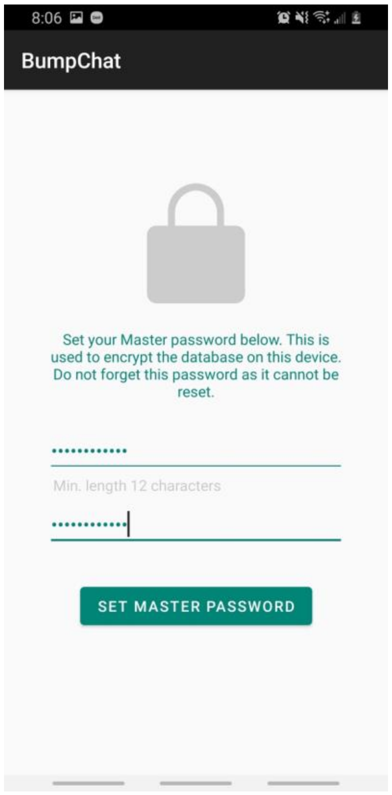

8:06
BumpChat
Set your Master password below. This is used to encrypt the database on this device. Do not forget this password as it cannot be reset.
••••••••••••
Min. length 12 characters
••••••••••••
SET MASTER PASSWORD

Figure 4. Master password feedback

4.5 Database Unlock / Login (Activity)

The login screen, shown in *Figure 5*, displays when the application is opened after the master password has been set. Once the minimum character length of 12 is reached the input field turns green and the **UNLOCK DATABASE** button is enabled as shown in *Figure 6*. *Figure 7* & *Figure 8* show failure and successful feedback respectively. The user is sent to their inbox after successful decryption.

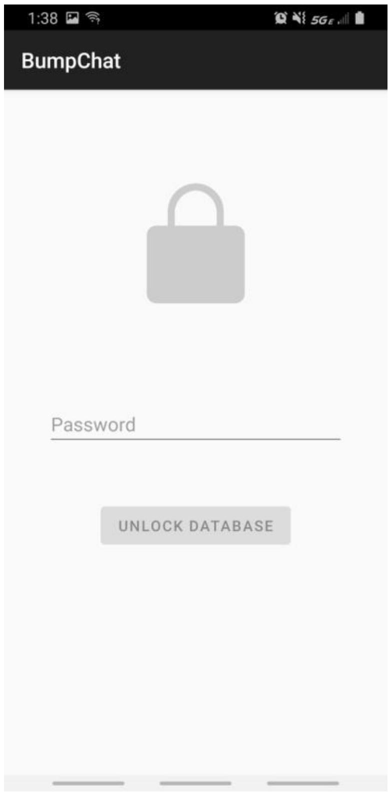


Figure 5. Database unlock prompt

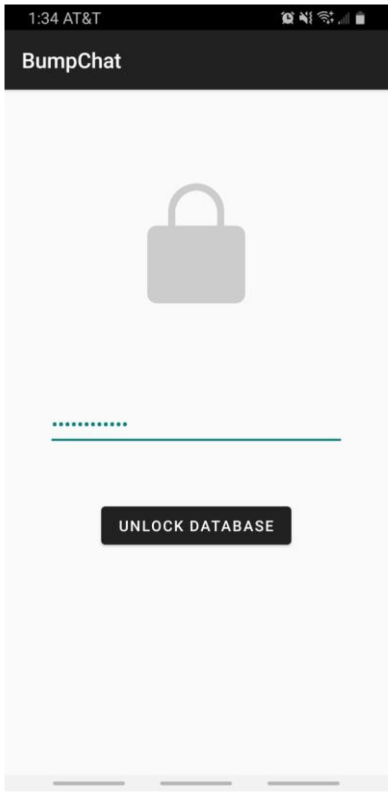


Figure 6. Database unlock ready

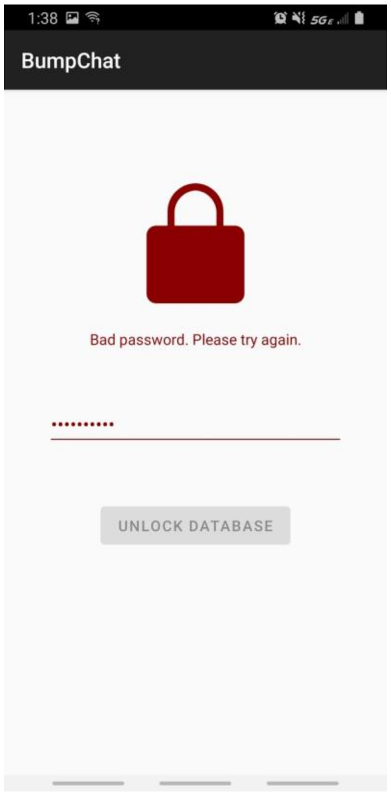


Figure 7. Database unlock failure

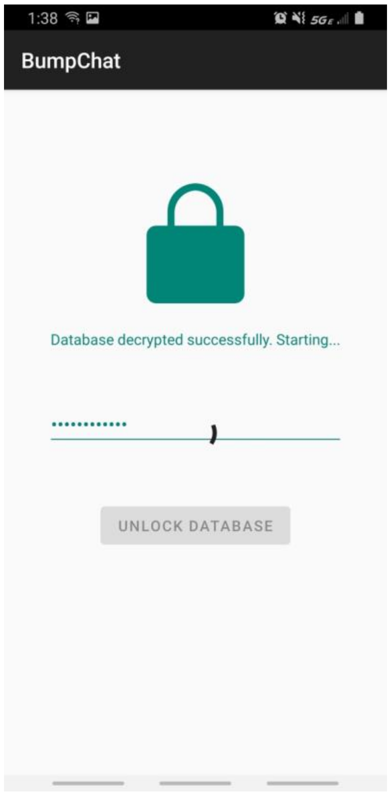


Figure 8. Database unlock feedback

4.6 Message Inbox (Activity)

The message inbox, shown in *Figure 9*, is where all the contacts can be accessed. Contacts are ordered by the last message received and show a truncated last message. New contacts can be added using the green button in the bottom right. Long pressing a contact will enable contact deletion mode shown in *Figure 10*.

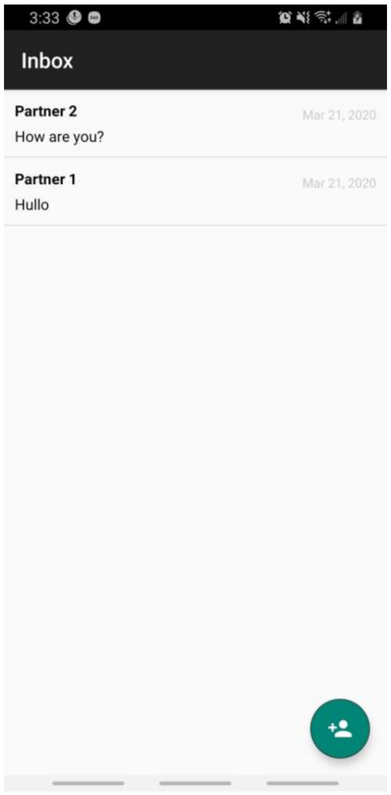


Figure 9. Inbox

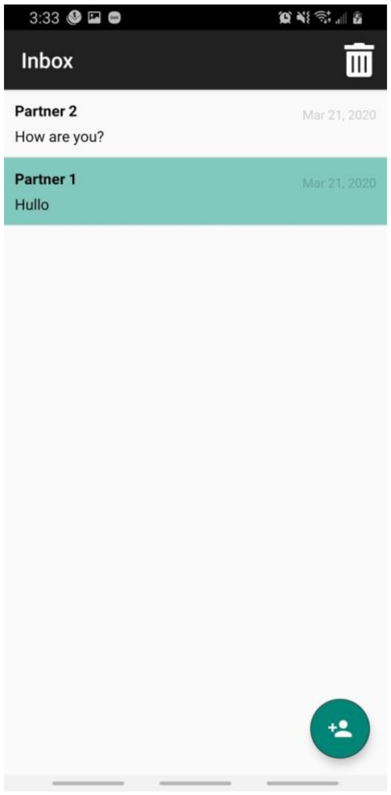


Figure 10. Inbox deletion mode

4.7 Contact Creation (Activity)

The contact creation process is the novel portion of this project. The devices pair by sending keys while in contact with one another over NFC. The steps of the pairing process are described below. The pairing screen is shown with the contact name filled in by the user *Figure 11*. The contact name and checkboxes must all be filled checked for the *SAVE CONTACT* button to be enabled.

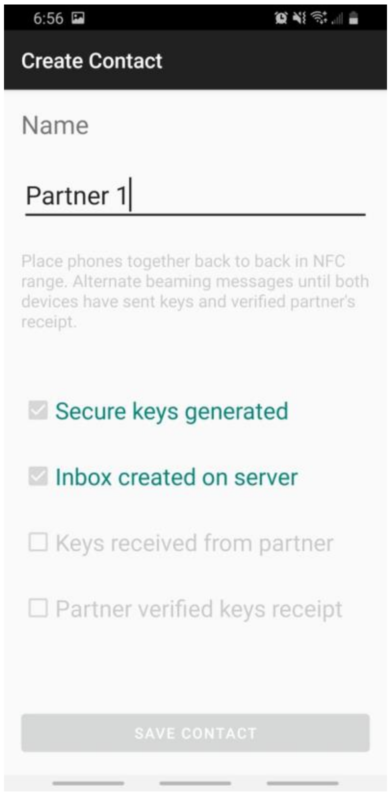
Step 1: Key Generation - The process starts with each device generating an RSA-2048 key, a 2048-bit Diffie-Hellman key, and an HMAC-SHA256 salt. Once keys are successfully generated, the first checkbox is marked as shown in *Figure 11*. The key generation happens automatically once the activity is started.

Step 2: Server Inbox Registration – Each device registers an inbox on the server using the *Inbox/register.php* endpoint. The RSA public key is sent to the server and an encrypted challenge is sent back. The client sends the decrypted challenge back to the *Inbox/register_verify.php* endpoint. The server response is displayed with a small Toast message. On a successful response, the second checkbox is marked as shown in *Figure 11*.

Step 3: Key Sharing – When the phones are placed back-to-back in close proximity (< 4 cm), they will enter an NFC beam[6] mode which allows the one-way data transfer from the initiating device. A transfer is initiated by the user touching the screen. A NdefMessage object is sent from the initiating device to the partner. The message object contains an array of NdefRecord objects which are each a byte[] array. The payload consists of 4 NdefRecords. First is a single byte that holds the transfer state of the sending party that is bitmasked to show: user sent keys, user received keys, user verified partner received keys. This first record is the one used to update the checkbox state on the UI. The next two records are the sending parties public RSA key and DH key respectively. Both keys are encoded as a byte array which are decoded on the receiving partners end. The last record is an HMAC-SHA256 salt that each user sends but only one is used. The first person to receive keys uses the partner's salt instead of their own. This allows both parties to generate the same AES-256 shared key. The AES-256 shared key is generated during at the end of sharing of the public keys using a Diffie-Hellman key agreement followed by an HMAC-based KDF. This is the key that actually encrypts the messages sent between devices as it is much less computationally intensive to encrypt using AES than RSA.

The key sharing process requires 3 transfers of data in the following order: Partner 1 -> Partner 2, Partner 2 -> Partner 1, Partner 1 -> Partner 2. The first two transfers are for key and salt exchange.

The last transfer is for Partner 2 to verify that Partner 1 received keys. *Figure 13* shows Partner 2's UI state after receipt of data from Partner 1. *Figure 14* shows Partner 1's UI state after receiving data from Partner 2. Partner 1 as this stage is now aware that Partner 2 has successfully received keys and salt. *Figure 15* shows Partner 2's UI state after the second data transfer from Partner 1 which makes Partner 2 aware of successful key receipt from Partner 1. At this stage all keys and the salt have been exchange and verified. Both devices UI will unlock the *SAVE CONTACT* button which will create a new Contact object and persist it to the local database as described in *Section 3.1*.


6:56

Create Contact

Name

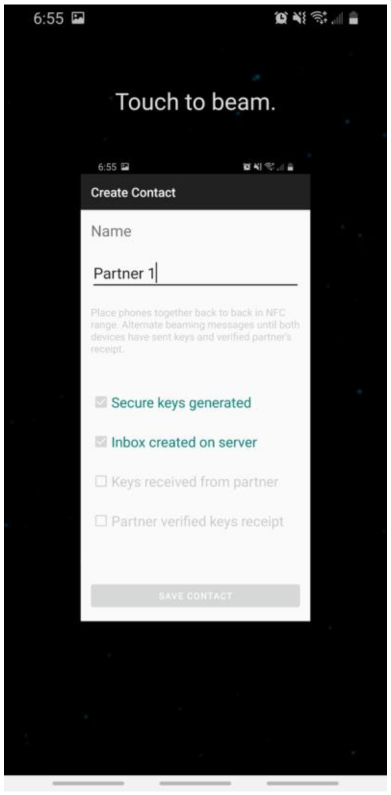
Partner 1

Place phones together back to back in NFC range. Alternate beaming messages until both devices have sent keys and verified partner's receipt.

- [x] Secure keys generated
- [x] Inbox created on server
- [ ] Keys received from partner
- [ ] Partner verified keys receipt

SAVE CONTACT

Figure 11. Create contact screen with contact name entered and successful inbox creation.


6:55

Touch to beam.

Create Contact

Name

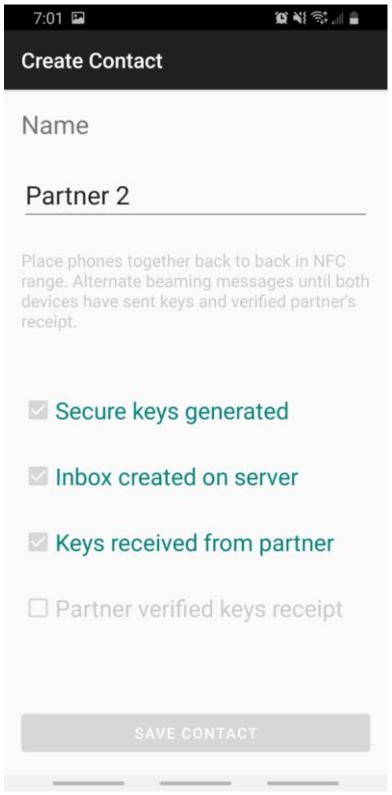
Partner 1

Place phones together back to back in NFC range. Alternate beaming messages until both devices have sent keys and verified partner's receipt.

- [x] Secure keys generated
- [x] Inbox created on server
- [ ] Keys received from partner
- [ ] Partner verified keys receipt

SAVE CONTACT

Figure 12. NFC beam view from Partner 1 before any keys have been exchanged.


7:01

Create Contact

Name

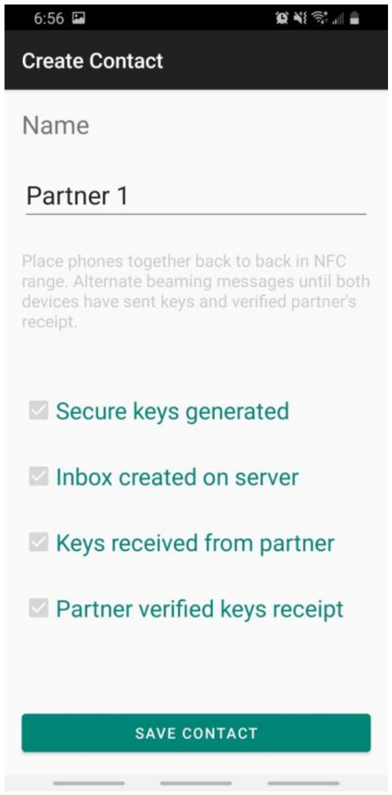
Partner 2

Place phones together back to back in NFC range. Alternate beaming messages until both devices have sent keys and verified partner's receipt.

- [x] Secure keys generated
- [x] Inbox created on server
- [x] Keys received from partner
- [ ] Partner verified keys receipt

SAVE CONTACT

Figure 13. Pairing view from Partner 2 after keys have been exchanged.


6:56

Create Contact

Name

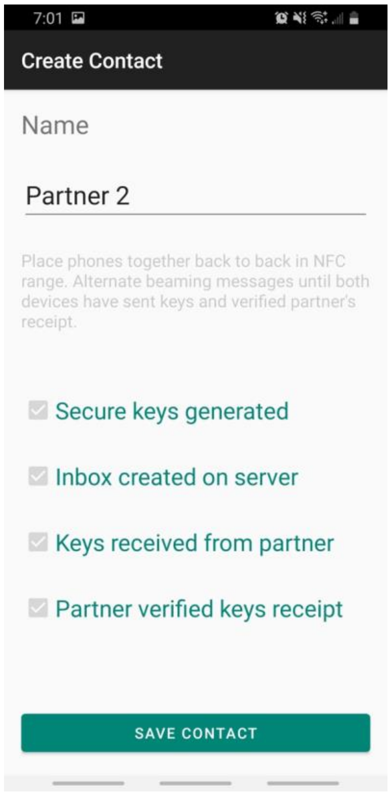
Partner 1

Place phones together back to back in NFC range. Alternate beaming messages until both devices have sent keys and verified partner's receipt.

- [x] Secure keys generated
- [x] Inbox created on server
- [x] Keys received from partner
- [x] Partner verified keys receipt

SAVE CONTACT

Figure 14. Partner 1 after receipt of keys from Partner 2.


7:01

Create Contact

Name

Partner 2

Place phones together back to back in NFC range. Alternate beaming messages until both devices have sent keys and verified partner's receipt.

✓ Secure keys generated

✓ Inbox created on server

✓ Keys received from partner

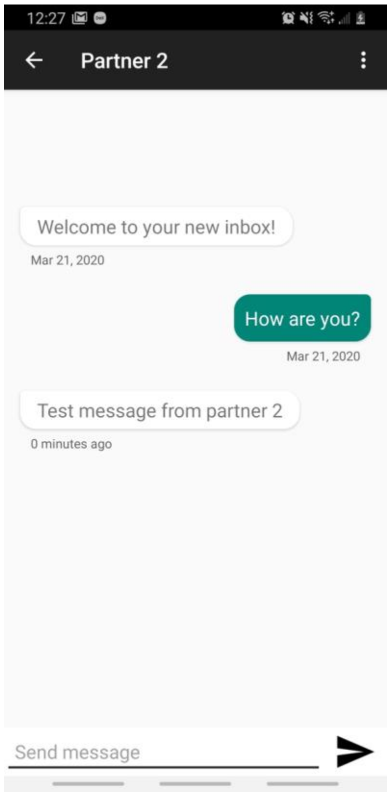
✓ Partner verified keys receipt

SAVE CONTACT

Figure 15. Partner 2 after receiving second exchange from Partner 1.

4.8 Message List (Activity)

The message list screen shows all messages with a single contact. Messages are retrieved when the screen is first loaded and are retrieved in an interval as long as the app is in the foreground. *Figure 16* shows the message history between two partners. Each message is tagged with a time difference in a readable format that changes from minutes to hours to the date. The sending of a message is shown in *Figure 17*. Pressing the paper airplane icon will encrypt the message with the shared AES-256 key. The message is addressed to the inbox identifier of the partner. The client generates a new challenge and sends the Message/send.php with the inbox_identifier, encrypted message, and decrypted challenge response. The triple dot icon in the top corner of *Figure 16* triggers the options menu when pressed as shown in *Figure 18.* The **Edit** option takes the user to the contact edit page where they can change the name of the contact. The **Clear History** option clears all messages and leaves the chat history with a single "Messages Cleared" message as demonstrated in *Figure 19*.


12:27

Partner 2

Welcome to your new inbox!

Mar 21, 2020

How are you?

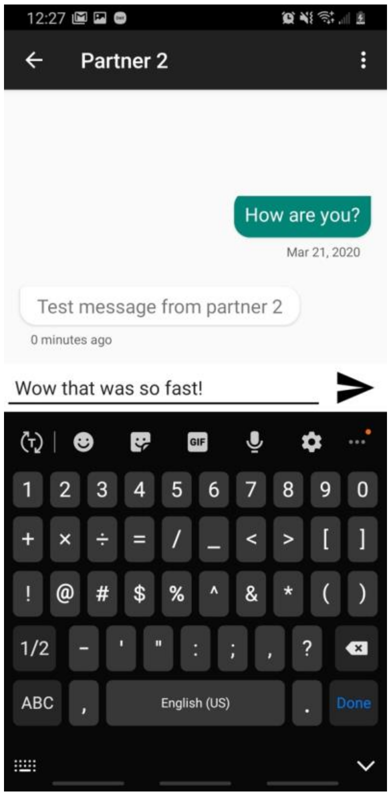
Mar 21, 2020

Test message from partner 2

0 minutes ago

Send message

Figure 16. Single contact message list


12:27

Partner 2

How are you?

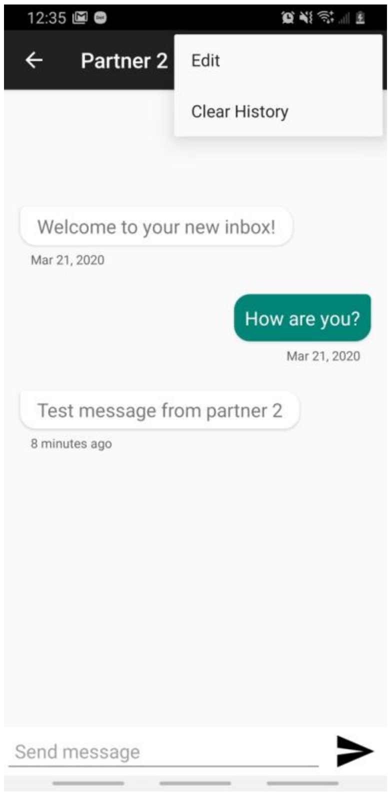
Mar 21, 2020

Test message from partner 2

0 minutes ago

Wow that was so fast!

Figure 17. User sending a message

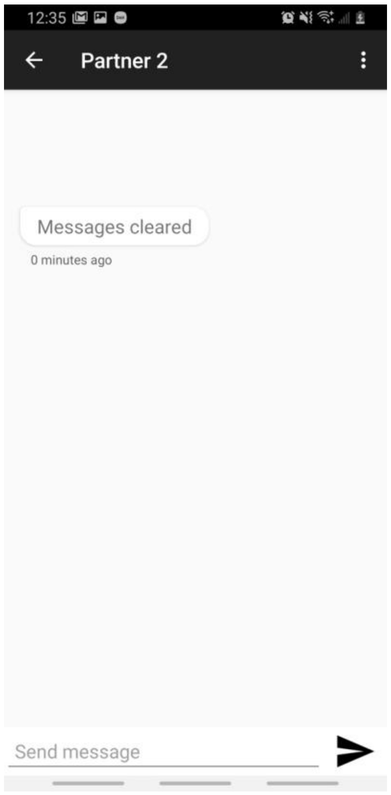


Welcome to your new inbox!
Mar 21, 2020

How are you?
Mar 21, 2020

Test message from partner 2
8 minutes ago

Figure 18. Message list options menu



Messages cleared
0 minutes ago

Figure 19. Cleared message list

4.9 Contact Edit (Activity)

The contact edit page, show in *Figure 20*, can be reached through the options menu on any chat history page. This page is solely responsible for updating the contact name that shows up in the message inbox and chat history page.

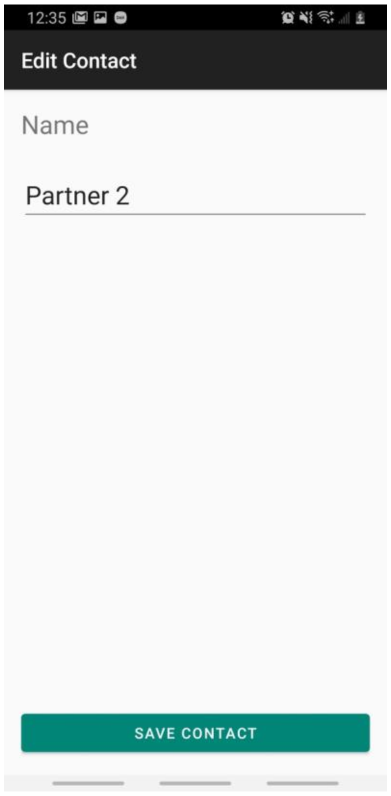


Figure 20. Contact edit page

5. Conclusion

The primary goal of this project was to build a novel key sharing mechanism over NFC. This moves the chain of trust from digital space to physical space as pairing requires physical contact between devices. This project was based on an idea I had shortly after receiving an NFC implant in my hand. The original concept was to use the implant to unlock the app and do the key sharing over the internet. However, it felt like the best use of NFC technology was for the key sharing and pairing process.

The system design looked fairly simple on paper, but the implementation details were far more exacting than initially anticipated. Security was a primary focus throughout the development process as a single weak link undermines the security of the entire project. One of the largest pain points was getting encryption consistent across Java and PHP. Formatting and encoding were critical in getting both systems to produce the same results with the same input data. The hashing of the RSA public keys are done independently on both the client and server to prevent from a chosen inbox identifier attack. The ability to generate an RSA key that hashes to a chosen inbox is outside the scope of current technology and should be for the foreseeable future. Generating 2048-bit RSA keys on Android devices is on the order of tenths of a second which makes brute-forcing not a feasible option. Serious Cryptography - A Practical Introduction to Modern Encryption by Jean-Philippe Aumasson[10] was instrumental in the design of the system. The book contains current best practices in applied cryptography which was used to select the algorithms and ciphers used. It showed the pitfalls that undermine good security and how to engineer around them. Coming from a background of limited knowledge about best security practices and limited experience in developing native Android apps, this project helped develop practical knowledge that I can use in furtherance of my career. I plan on developing this application suite further as time permits to implement the future work with plans to release this as a Free and Open-Source Software package that can be used by those that need it most.

5.1 Future Work (Client)

Adding more standard messaging features to the Android client is at the top of the list. Android has an Camera API[8] available to capture images inside of an application. Most of the work to add that feature is spent in integrating it into the UI. Another feature would be to add the sending of arbitrary files by sending the file MIME type along with Base64 byte[] encoded data format. Unlocking the app using either an NFC tag or using an NFC tag as a secondary authentication method would extend the usage of NFC inside the client while adding extra security. Adding group communication would be an interesting challenge with this model. The pairing process would need to be revamped to handle sharing keys between multiple devices. Releasing through the Google Play store would allow for signed releases and automatic updating. The only hurdle is an ongoing $25/year developer cost as Google's app review process is fairly minimal and requires answering a handful of questions to get an app released.

5.2 Future Work (Server)

Packaging the server-side software into an installer would allow individuals or groups to spool up their own communication system on hardware they control. The installer would be responsible for installing the required packages to set up the Apache server, PHP configuration, and MariaDB with the encryption settings configured. Another option is to provide a pre-configure Virtual Machine (VM) or Amazon Machine Instance (AMI) [9] that would allow a simple setup on Amazon Elastic Computing Cloud (EC2) infrastructure. The only changes on the client-side would adding a server URL input during initial setup and a configuration page to change the server URL after installation.

References

1. Advanced Encryption Standard [https://en.wikipedia.org/wiki/Advanced_Encryption_Standard]
2. RSA Encryption [https://en.wikipedia.org/wiki/RSA_(cryptosystem)]
3. MariaDB [https://mariadb.com/]
4. Signal Protocol Whitepapers [https://signal.org/docs/]
5. SQLCipher [https://www.zetetic.net/sqlcipher/]
6. NFC beam [https://developer.android.com/guide/topics/connectivity/nfc/nfc]
7. HKDF RFC 5869 [https://tools.ietf.org/html/rfc5869]
8. Android Developer Camera API [https://developer.android.com/guide/topics/media/camera]
9. Amazon Machine Instance [https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html]
10. *Serious Cryptography - A Practical Introduction to Modern Encryption* by Jean-Philippe Aumasson [https://nostarch.com/seriouscrypto]
11. Android Room Persistence Library - https://developer.android.com/topic/libraries/architecture/room