University of Glasgow | School of Computing Science

A secure client-server mobile chat application implementing an elliptic curve integrated encryption system (ECIES) and other security features.

Daniel Furnivall

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

1st April 2022

Abstract

This dissertation describes the design and development process of a secure messaging Android client (and corresponding server).

The three key technical aims were the use of pseudonymous identities, self-destructing messages and the implementation of a hybrid end-to-end encryption approach using asymmetric encryption for trust establishment and shared secrets which are used to create symmetric keys used for message encryption.

The client application was developed using the Kotlin programming language and the server was implemented in Python.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: Daniel Furnivall Signature: *Daniel Furnivall*

Acknowledgements

I would like to thank my supervisor, Mark McGill for consistently providing suggestions that made me question my own thought process and adjust my approach for the better. The constant support throughout the project is something I am incredibly grateful for.

I would also like to say thanks to Ewa Wanat, Tzvi Lipschitz, Thaïs Ramdani and Matt Weston for deeply helpful stylistic guidance, Bryce Campbell for top tier music recommendations and semicolon-driven design methodology, Conal Brosnan for Kotlin evangelism, Karim Al Tom for Lebanese sweets. Additionally, I'm extremely grateful to Emer Sweeney, Stuart Allan and Michael Callan for their much-needed assistance.

Lastly, I would like to thank my wife Rona and my furry friend Gordon for providing much needed moral support throughout the development and writeup process.

Contents

Chapter 1: Introduction

Secure messaging applications aim to allow individuals to communicate with other individuals around the globe while avoiding the potential for digital eavesdropping. In theory, this means the user can enjoy the benefits of a globally connected world while preserving their own privacy. However, in practice there are many implementation challenges to be considered.

In an increasingly digital world, we are constantly creating data (and, of course, corresponding metadata). It's estimated that humanity produced somewhere in the order of 2.5 exabytes of digital data per day in 2018[1] and the growth of digital data creation by individuals is constantly accelerating.

As storage costs decrease over time[2], economic and political incentives have begun to develop for nation state actors and major organisations to develop profiling systems using large-scale data collection and mining. These "Big Data" profiling systems are already being used for targeted advertising[3], market segmentation[4], criminal investigations[5] and sentencing[6]. In the political sphere, these models have been used for increasingly effective traditional campaigning, [7] as well as (alleged) mass psychological manipulation[8] and disinformation[9] campaigns.

In a pre-digital world, an individual could avoid eavesdropping or data collection by malicious entities by simply speaking in a hushed voice, shredding documents, moving quietly or looking over their shoulder. In the current landscape, it is much more difficult to avoid surveillance without eschewing technology entirely; how secure is secure? Our mobile phones are constantly communicating our triangulated locations, and even if our web traffic is encrypted by default, browser or Internet Service Provider (ISP) metadata still contains useful profiling information.

The question of whether the average individual is able to enjoy the fruits of such data collection becomes less clear. Do these large actors have the best interests of the subjects of their data in mind? If not, methods for obfuscating or hiding sensitive data become valuable considerations for the individual.

1.1 Why are secure chat applications needed?

Secure messaging applications provide a covert means of communication between individuals or groups across a network of some kind. This can take the form of text, audio or video messaging, document or file sharing.

To begin to understand the motivations of secure chat application users, two important questions must first be answered:

1. Who desires secure messaging?
2. Who are they aiming to protect their data from?

There are many categories of potential users of such applications, and from wildly different demographics. These can be mundane and innocuous, such as the organisation of a surprise party for a friend or family member.

However, another group of potential users are those who seek to hide criminal behaviour from law enforcement organisations. A high-profile example of this would be the EncroChat network of encrypted phones, predominantly used by organised crime, which was unveiled after a Europe-wide infiltration and investigation of the network by law enforcement groups (leading to several thousand arrests) [10].

Another group who may wish to evade police or law enforcement are political dissidents. In what has become known as the "Million Dollar Dissident" [11] case, a dissident in the UAE, Ahmed Mansoor, was targeted by the NSO group (an Israeli cybersecurity firm which produces spyware for government use). He subsequently had his passport confiscated, he was beaten multiple times, his car was stolen and finally he was imprisoned by the UAE authorities - all within a week of posting anti-government posts online [12]. There are many parallels here with other groups who benefit from secure messaging - whistleblowers sharing information with journalists, and police informants who need to share data secretly with law enforcement.

In reality, there are plenty of reasons for *everyone* to use secure messaging such as protecting data in case of device theft, avoiding embarrassment, or limiting exposure to blackmail or government surveillance. Continuing the comparison to real-world communication - when speaking out loud, people don't tend to shout all the time, and tend to consciously limit and monitor who is listening to a conversation.

It should be clear that although there are many categories of potential secure messaging users, there are also adversarial parties to consider when designing these kinds of platforms, which means the development of such services is a complex undertaking. There are major tradeoffs which need to be made between usability and data security - for example, how can we preserve security of messages while also storing them on a mobile device?

1.2 Objectives

The primary development objective of this project was to develop an Android mobile application (and corresponding server) that uses complex security features including an end-to-end encryption solution that uses a combination of client-side asymmetric (Elliptic Curve Diffie-Hellman) and symmetric (AES) encryption approaches to protect data from being readable at the server layer. Additional security features included pseudonymous identity and self-destructing messages.

During the development journey of this application, the intrinsic motivation was to come to a greater understanding of the complexities, assumptions and tradeoffs involved in creating these kinds of platforms.

Chapter 2: Requirements and Analysis

Unger et al's "Systematization of Knowledge" (SoK) 2015 paper on secure messaging captured in extensive detail some of the key considerations for developing secure messaging applications. They highlighted three points which comprise the major challenges in this field; trust establishment, conversation security and transport privacy.

Trust establishment describes the problem of ensuring that users are communicating with the party they intended. This is a difficult problem, as there are tradeoffs between security and usability that need to be made. One method which has seen extensive use is key-sharing. This ensures that the recipient of a message is able to verify that it was sent by the sender. Key-sharing can be performed in a number of different ways, depending on the type of cryptographic architecture employed within the system. This will be discussed in greater depth in the next chapter.

Conversation security in this context refers to the security protocol used to encrypt the data. Unger et al lament the fact that most secure messaging solutions use only static asymmetric encryption (i.e. long-term key persistence meaning users do not need to exchange keys regularly). This is another example of the usability/security tradeoffs that need to be made for wider adoption of secure messaging.

The final challenge is transport privacy, which concerns how messages are transmitted between users. This is a complex problem, as metadata such as the identity of the sender and recipient need to exist so that the server can route messages correctly, despite also being appealing attack vectors for malicious actors. A real world corollary can be seen with the logistics of mail - for the postal service to deliver a package to the correct recipient, they require access to some form of identifying information.

2.1 Existing applications in this field

There are many existing secure chat applications available for both iOS and Android, with considerable variation in terms of encryption architecture, security and usability features. For brevity's sake, this review focuses on the three largest mobile secure messaging providers in 2022 - Signal, Telegram and Whatsapp.

The table below highlights some of the features and attributes of each of these applications:

Application	Signal	Whatsapp	Telegram
Ownership	Signal Foundation (non-profit, USA)	Facebook Corporation (USA)	Durov Brothers (Russian)
End-to-end Encryption	Double Ratchet Algorithm[13], constantly cycling symmetric session keys and Elliptic Curve Diffie-Hellman (ECDH) keypairs - this approach is called the "Signal Protocol"	Proprietary implementation of the Signal Protocol (unverifiable). Does not apply to backed-up messages stored on Google servers.	Only available in "Secret Chats", and only on mobile devices - uses Telegram's own MTProto 2.0 protocol[14]. For group chats, messages are encrypted symmetrically and are theoretically readable by the server.
Open Source	Fully open source for both client and server (with the exception of a server-side anti-spam component)	Entirely closed source	Open source client with a closed source server.
Self-destructing messages	Controlled by the sender (i.e. the messages will be deleted from the recipients device based on the sender's settings). Manageable via defaults or for individual conversations. Timespan variable.	Sender-controlled, can be managed individually or via defaults. Fixed timeframe of 7 days.	Sender-controlled, can be managed individually or via defaults. Timespan variable.
Group chats	Fully end-to-end encrypted using "client-side fanout"[15], where each message is encrypted individually to all users in the group.	Same as Signal (unverifiable due to closed source)	Encrypted between client and server but not E2EE (i.e. server can read messages).
Pseudonymous Messaging	Not available (linked to phone number)	Not available (linked to phone number)	Not available (linked to phone number, although it is possible to hide phone number from contacts after creation.)
Metadata collection	Only stores phone number used to register, date of initial registration and date of last app use.	Historical message metadata, phone contacts, device metadata and activity, blocked numbers, read receipts, full name and more.	IP addresses, phone contacts, historical message metadata.

2.2 Issues in existing applications

As can be seen in the table, Signal appears to be the most secure of the available secure messaging applications. This comes from a combination of best-in-class encryption, open source code allowing for security audits, fine-grained control of privacy options and innovative handling of security such as their use of client-side fanout for group chats. Whatsapp and Telegram both closely guard some or all of their source code, making it impossible for independent researchers to verify that their security claims are accurate. Signal also seems to have limited financial or political incentive to store metadata about the behaviour of their users due to their non-profit ownership model, which contrasts with Whatsapp's ownership by an advertising company and Telegram's use of advertising via sponsored messages.

All three applications provide fairly strong capabilities for self-destructing messages. Importantly, all three allow the sender to define the parameters for deletion, although Whatsapp does not provide timeframe granularity compared to Signal and Telegram. This means that users can feel relatively secure in the knowledge that even if the recipient is compromised their own exposure is limited.

One of the major downsides of all three applications is the lack of pseudonymous messaging. This is an understandable omission due to implicit need for chat applications to incorporate some means of social network discovery (in all cases, phone numbers and contact lists are used for this function).

This means that the development of an application that provides comprehensive end-to-end encryption and self-destructing messages alongside pseudonymous messaging features is worth pursuing.

2.3 User Personas

User personas are a user-centric design methodology which provides a convenient means of exploring and capturing user requirements, devised by Alan Cooper[16] (who also created the Visual Basic programming language). Miaskiewicz et al found that personas help with challenging assumptions, avoiding self-referential design and narrowing the target audience of the product[17]. Below are several personas which were used to guide the prioritisation and requirements gathering process.

Thaïs - a 37-year-old police informant deeply nested within a major international drug distribution network. She needs to communicate with her handlers within law enforcement securely without arousing the suspicion of her colleagues within the criminal organisation. One of the most important factors in her choice of messaging applications is self-destructing messages. This means that even if her device is compromised, there will be no evidence that she has been sharing information with police.

Ewa - a 28 year old human rights activist and political dissident in a large authoritarian state in South-East Asia. As an important agent for change within the country, she needs to alert the country's overseas diaspora, human rights organisations and international media to a new and bloody crackdown on freedom of expression that occurred. The ruling party of the country has imple-

mented state of the art surveillance technology and deep packet inspection on all outgoing web traffic. Ewa needs a means of communicating securely with her intended recipients via an insecure channel.

Tzvi - a 45 year old political "fixer" working within the ruling party of a large African state. He has been tasked with giving a veneer of legitimacy to the process of allocating government contracts to political allies through a fraudulent tendering process. This requires him to have a secure communication channel with the chosen contractor so they can ensure they are able to make the most competitive bid. Tzvi has his own political ambitions which could be tarnished by such dealings, making it vitally important for him to have plausible deniability. Using a pseudonymous identity for messaging is a key selling point for him.

2.4 Requirements

Requirements for this application were gathered after a detailed analysis of the existing products in this space. All the features in the application are present in some sense in some or all of the competitor applications analysed with the exception of pseudonymous messaging. Combining several key features (Self-destructing messages, a combination of symmetric and asymmetric encryption and pseudonymous messaging) formed the basis for the design.

Requirements were then outlined following the MoSCoW requirement prioritisation framework[18]. This method allows for prioritisation based on four priority levels:

1. "Must have" - the most basic requirements to have a functioning solution for the problem.
2. "Should have" - Requirements that the app should include, but does not absolutely need.
3. "Could have" - Requirements that could potentially be implemented.
4. "Would like to have" - Features which would be nice to include but are not strictly necessary or required.

2.4.1 MoSCoW table

Prioritisation	Requirement
Must have	User can connect to the server application via a client application.
Must have	User can send a message from one client to another client.
Must have	User can receive a message from another client.
Must have	User can create a pseudonymous username.
Must have	User identity can persist upon disconnect/reconnection events
Must have	User can see other connected users within a contacts page
Must have	Messages (sent and received) are stored locally on the client within a local database.
Must have	User can talk to more than one user independently in separate messaging sessions.

Must have	User messages reach the correct recipient only.
Should have	User can send encrypted messages which cannot be read by the server operator.
Should have	User is able to define how long to store messages in the local database.
Should have	User receives a push notification when a message is received.
Should have	User preferences (e.g. self-destruct preferences) should persist on application restart
Should have	Recipient public keys should be shared with sender as appropriate.
Should have	Private/Public keypairs should be stored securely to prevent unwanted access.
Should have	Generating keypairs should be fast while also secure.
Could have	Group messaging between multiple users.
Could have	Users can access the application via a web-based interface in addition to the mobile application.
Could have	Users can format their text using standard decorations (e.g. bold/italics/strikethrough)
Could have	Users can forward messages from one user to another user.
Would like to have	Message queueing to allow users to receive messages when they reconnect.
Would like to have	Sender-defined self-destructing messages.
Would like to have	E2EE on group chats using client side fan-out method.
Would like to have	Users can record audio messages and transmit to another user.

2.5 User stories

User stories are a means to map requirements to actionable feature development and separate workload into manageable chunks and a key tenet of Agile methodology. Indeed, a 2018 study found that approximately 90% of Agile practitioners utilised user stories in their requirements gathering process[19]. The user stories below represent the desired behaviours from the application and were used as a framework for developing the application in an agile framework.

- As a user, I want to be able to send messages from my client application to the server for handling.
- As a user, I want my message to be routed to the appropriate client device of my choosing.
- As a user, I want to create a pseudonymous username which is shown to other users.
- As a user, I want to view other connected users.
- As a user, I want messages on my device to disappear after a given time period

- As a user, I want to encrypt my messages so that the server cannot read the contents.
- As a user, I want to be able to decrypt messages that are transmitted to me from the server.
- As a user, I want my data (e.g. username) to persist after I close the application.
- As a user, I want to receive a push notification when a message is sent to my device if I am doing something else.
- As a user, I want to be able to click on a notification and be taken directly to the relevant chat window to reply to the sender.
- As a user, I want the contacts list to automatically update when a user connects or disconnects from the server.
- As the server operator, I do not want to be able to read the content of messages sent to me.
- As the server operator, I want to be able to update stored client public keys when they are changed.

The above stories formed the basis for the development process, the intricacies of which are discussed extensively in the following chapter.

Chapter 3: Design and Implementation

3.1 High Level Architecture

Some of the primary architectural considerations on this project are highlighted below:

1. Client-side encryption to allow messages to flow through the (untrusted) server.
2. Client-side self-destructing messages according to client-defined storage duration parameter.
3. Persistence of user identity and handling of disconnection/reconnection events.

The figure below represents a very high level view of the system architecture, which gives an overarching perspective of how the system fits together and some of the technological choices made. However, it does not give a comprehensive picture of the complexity of the overall system.

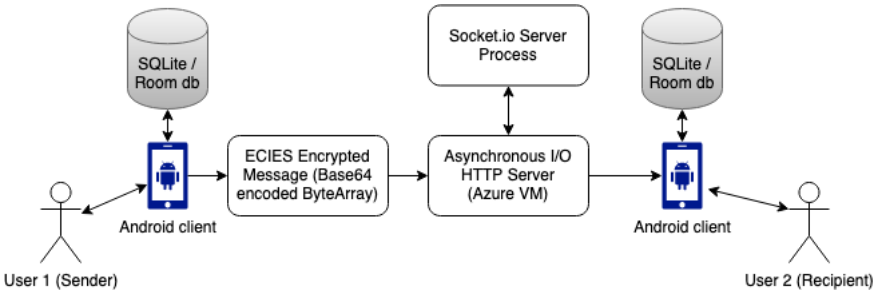

Figure 3.1: High level architecture of the system. Messages are encrypted on the sender device and passed to the (untrusted) server which routes the message to the relevant recipient device where they can be decrypted

3.2 WebSockets vs REST API

One of the major decisions required for this project was which transport protocol to use for propagating messages through the system. The two major options considered were a REST API via HTTP[20] and WebSockets[21].

REST (Representational State Transfer) is a set of design principles which assists in the development of web services which utilise a central server with one or many clients, based on the twin concepts of request and response. A client will send a request to a server and receive a response based on the content of the request. REST APIs are heavily used across many industries and mostly (though not exclusively) use the HTTP protocol to communicate between

client and server. The primary weakness of REST is that it is not optimal for circumstances where constant bi-directional communication is important between client and server. Indeed, the WebSocket Protocol Standards Track Document[21] states the problem as such:

```
Historically, creating web applications that need
bidirectional communication between a client and a server
(e.g. instant messaging and gaming applications) has required
an abuse of HTTP to poll the server for updates while
sending upstream notifications as distinct HTTP calls.
```

The WebSocket protocol is a newer system which excels in two-way communication between client and server systems. For chat applications like the subject of this project, WebSocket appears perfectly suited to the problem. Due to the comparative simplicity of bidirectional communication using WebSockets, the protocol uses significantly less energy[22] to maintain a client/server connection. This is especially relevant when considering that this is a mobile application which needs to preserve battery life. The one major downside of WebSockets is that it is a modern standard which is not necessarily fully supported across all devices or browsers yet.

During the literature review of this topic, it became apparent that there was a way to achieve the best of both worlds. Socket.io[23] is a cross-platform library which initially attempts to create a WebSocket connection and, if not possible, falls back to HTTP polling for devices which do not support it. It also provides helpful additional features like automatic reconnection. After determining that both client and server implementations existed in the desired languages (Python/Kotlin), Socket.io was subsequently included in the system design.

3.3 Encryption Implementation

As mentioned previously, the system was designed with the prevailing goal of having two clients who could communicate with each other through an insecure, untrusted environment (i.e. the server) without needing to worry about messages being vulnerable to man-in-the-middle attacks[24]. The most attractive means of achieving this is to use state of the art encryption.

An oft-repeated adage in secure software development is that a developer should "Never roll your own crypto"[25]. This is due to the fact that encryption is almost always absolutely critical to the functioning of applications. Furthermore, it is very easy for a developer's code to provide the illusion of working security features without fully understanding possible attack vectors. As such, all cryptography utilised in this project has an extensive track record of use in commercial and governmental settings.

Two of the most prevalent encryption approaches used in the modern world are asymmetric and symmetric encryption. Symmetric encryption is the simpler of the two - a secret key is used to alter the content of a piece of plain text in such a way that it's unreadable to anyone except someone who also has the secret key and can use it to decrypt the message. The major flaw with symmetric encryption is the need to share the secret key with the recipient. If the recipient does

not have access to the secret key, they cannot decrypt the message. Similarly, if the secret key is intercepted during transport of the message, a bad actor can decrypt the message. A good example of the symmetric encryption approach is the Advanced Encryption Standard (AES), which is used in this project. The AES algorithm was designed by Vincent Rijmen and Joan Daemen in 1999 and defined as a standard by the United States Government in 2001.

Asymmetric encryption is a little more complex - a key pair consisting of a public and a private key, which are separate but mathematically linked is used to encrypt and decrypt data. To send an encrypted message to the intended recipient, the sender must use their own private key and the recipient's public key. The recipient can then use their own private key and the sender's public key to decrypt the message at the other side. The primary issue with asymmetric encryption is that it's significantly more complex to implement than symmetric alternatives, and key generation is slower.

Two of the most common types of asymmetric encryption are the Rivest Shamir Adleman (RSA) algorithm[26] and Elliptic Curve Cryptography (ECC)[27]. The RSA algorithm has been in use for 45 years, and utilises the factorisation of prime numbers to produce a unique keypair. ECC is a more modern algorithm which uses the structure of elliptic curves to generate keypairs. The primary advantage of ECC over RSA is the computational requirements to generate unique keys - as the key size required for secure communication increases (i.e. when malicious actors are able to use greater computational power), the key generation process becomes prohibitively expensive. In a 2015 study, an Indian team found that RSA keypair generation was 471 times slower than the equivalent ECC keypair process when keys reached a certain size[28].

The approach taken on this project was to use a combination of both asymmetric and symmetric approaches - this combination means that the comparative speed of symmetric approaches can be harnessed for converting the plaintext to ciphertext, and the additional security provided by asymmetric encryption, ensuring that only the intended recipient can open the message as the symmetric key is independently generated by both sender and receiver using what is known as a "shared secret", based on information only held by the sender and recipient[29]. The implementation used is known as an Elliptic Curve Integrated Encryption Scheme (ECIES)[30]. ECIES is a hybrid encryption system which was devised by Victor Shoup in 2001. It comprises three key functions:

1. A key agreement function to generate "shared secrets" from a user's public key and another user's private key.
2. A key derivation function which is able to produce a key from an input of some kind.
3. A symmetric encryption algorithm which is used with the shared secret as a key to encrypt the plain text.

The advantage of ECIES is that it provides very strong encryption for secure communication within an untrusted/insecure channel while also having desirable properties for mobile devices such as much smaller keys than similar asymmetric/symmetric hybrid approaches (as shown by Martinez et al when comparing to RSA/AES hybrid systems[31])

The ECIES implementation used in this project uses the following algorithms and approaches:

- Elliptic Curve Diffie-Hellman[32] to generate a shared secret over an untrusted channel. The system was designed to ensure that alternative asymmetric encryption algorithms could be trivially swapped in as required.
- The P-521 elliptic curve outlined by the National Institute of Standards and Technology (NIST)[33]
- The SHA-512 cryptographically secure hashing function, of which the first 32 bytes are used to generate the AES key for the key derivation function.
- AES-256 for the symmetric component, generated from the shared secret.

3.4 Self-destructing Messages

One of the features implemented in the application is self-destructing messages on a client device. This allows the user to define a duration to store received and sent messages on their device, after which the specific message is deleted.

The system was designed to store messages within a local database for some degree of persistence of state between connection instances. This in itself is an example of a usability / security tradeoff and worth investigating further. On the one hand, it means that users can still read received messages after a device or application restart, but it is also vulnerable to being copied or read by an attacker with access to the physical device. Two potential solutions to this would be to encrypt the database within the application code using an option like SQLiteCipher or even to refrain from using a database entirely (i.e. providing greater security at the expense of usability). For simplicity, this project mainly focused on securing client messages from the untrusted server, but intra-device security solutions are also important to consider for future development.

This feature was implemented by prompting the user on their first time starting the application to define a duration for message storage which is passed as a parameter to a daemon process (a background thread which runs at very low priority) which sends a parametrised SQL query to the database, deleting message objects based on their timestamp metadata. Discussion of how this duration is persisted after restarts can be found in the following section.

3.5 Persistence of users after disconnects

One of the most complex challenges faced during the development of the application was persisting clients after they disconnect and reconnect from the server. This was handled in the following manner:

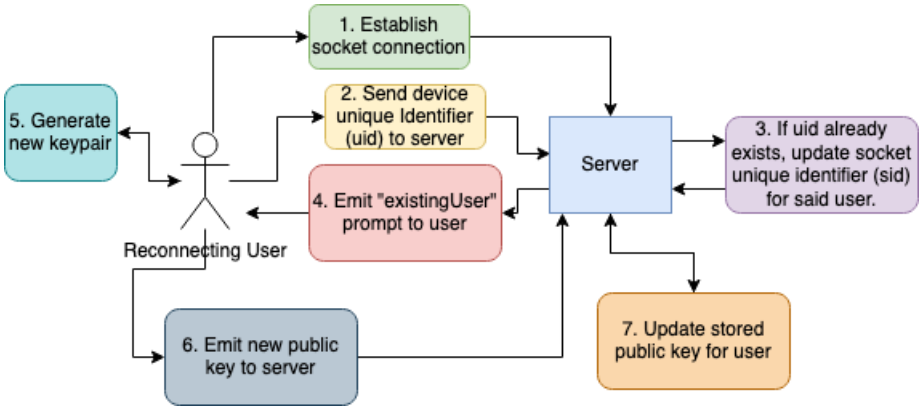

Figure 3.2: Reconnection flow for user persistence

The complexity of this process comes down to an inherent limitation of Android's built-in cryptography libraries. The system design was predicated on the assumption that it would be possible to securely store public and private keys within the Android KeyStore[34], a secure storage system within Android which is capable of storing public and private keys. However, the KeyStore does not currently support Elliptic Curve keypairs, which meant that the design had to be altered to accomodate this. What this means in practice is that a new keypair is generated on reconnection - the private key is stored locally and the public key is emitted to the server which replaces the existing stored key.

The flexibility to easily generate new keypairs is a huge advantage of elliptic curve cryptography - the key generation process is so quick that it's imperceptible to the user - something that would not be possible with RSA encryption.

3.5.1 Storage of self-destruct duration

Although there is no secure key-value store that allowed for the storage of ECDH keypairs, there is a non-secure method for persistence of key-value data (outwith the local database) using "SharedPreferences" files. These are ideal for storing benign configuration properties which persist after closing the application. This means that the initial value prompted from the user to request self-destruct message duration is able to be persisted and a returning user will be able to maintain their configuration, while also maintaining separation of concerns and reducing excessive exposure to database operations.

Storing this data may also present a mild security risk. A theoretical attacker who has control of the device could potentially alter the relevant SharedPreference file to ensure messages persist indefinitely. As mentioned previously intra-device security is a fruitful avenue for future work.

3.6 Message Propagation

The figure below describes the complex journey of a message through the messaging system. For clarity, it does not capture the self-destructing message capabilities described in the previous section.

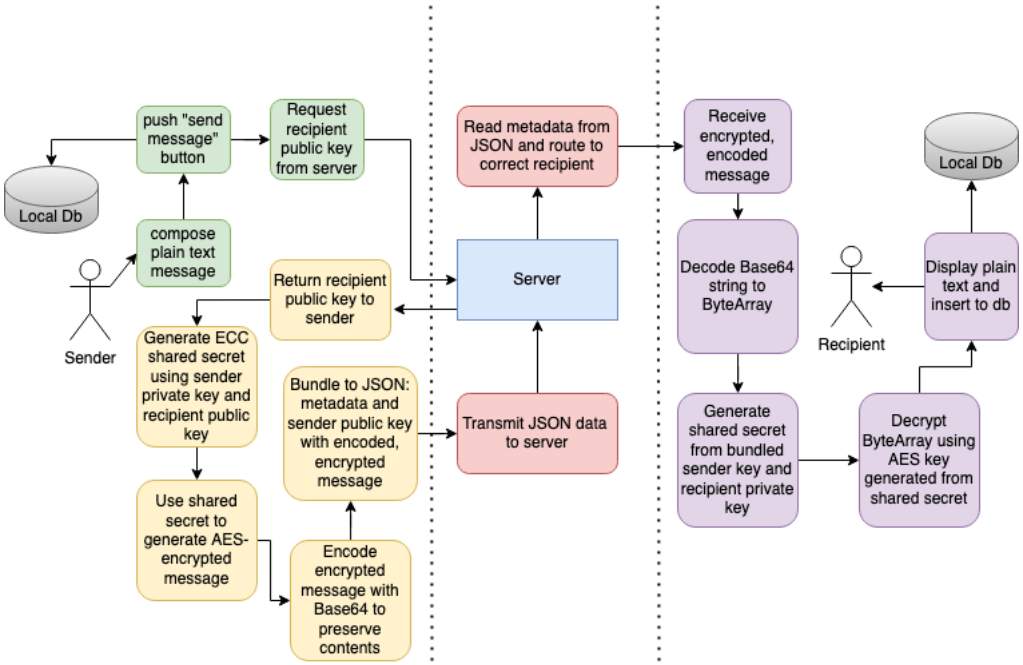

Figure 3.3: Flowchart of the journey of a message from one client to another. Green represents the initial message composition, server querying and local storage stage. Yellow represents the asymmetric and symmetric encryption and encoding stage. Red describes the server's routing process for messages. Purple shows the receipt, decryption, display and storage on the recipient's device.

3.7 Design Patterns

During the development of this project, a concerted effort was made to follow established design patterns and best practice wherever possible. It is a testament to the design of the Android development environment and documentation that this was relatively trivial to achieve.

In addition, wherever possible, adherence to the "Don't repeat yourself" (DRY)[35], "Keep it simple, stupid" (KISS), and five SOLID principles[36] was paramount.

The table below shows the design patterns which were deliberately utilised in the development of this project:

Pattern	Concise Definition	Implementation
Command Pattern[37]	Encapsulation of a request into an object	Socket.io event handler objects in both client and server
Observer Pattern[38]	When one object changes state, dependents update accordingly	LiveData queries updating views as events occur (ChatWindow and Contacts)
Adapter Pattern[39]	Allow an interface of one class to be used as another interface	ContactsAdapter for RecyclerView and MessageListAdapter for displaying chat messages.
Builder Pattern[40]	Simplifies object creation step-by-step	Push notification builder object allows for customisable notifications.
DAO Pattern[41]	Data Access Objects provide abstract interface for database, separating low level and high level operations	ChatMessageDAO allows us to specify the exact queries we can use on the database and prevents any other type of unplanned operations.
Singleton Pattern[42]	Restrict instantiation of class to exactly one instance	SocketHandler.kt in the client is an example of a class which should be (and is) a singleton.

It is also worth mentioning that the Model-View-Viewmodel architectural style was adopted throughout the application, as it is the recommended architectural pattern recommended by the Android core development team[43]. This architectural style is most clearly seen in all interactions between the database, the repository, the data access object and the ViewModel which displays data in the ChatWindow.kt activity.

3.8 User Interface and Visual Design

User interface was not a primary concern for the application, but during the requirements process, Balsamiq was used to design simple mockups of the application. These mockups do not reflect the final state of the application but they are included in the appendices for posterity.

Future development of the application will in part focus on improving the visual design.

3.9 Development tooling

There are two primary components to the overall system - one or more Android mobile client applications which communicate with a central, deployed server.

3.9.1 Client

The final client application was written entirely in the Kotlin language (v1.6.10) using the Android Studio IDE.

Development and debugging process

Traditional Android software development involves using the qEmu device emulator built into Android Studio to imitate the experience of a real user on their own device.

Debugging code during the HushChat development process was a complex endeavour, as the development machine had fairly limited RAM. As the purpose of the application was communication between two clients, it was important to be able to emulate two devices at the same time so messages could be exchanged. This was initially possible on the development machine, but became unfeasible over time as the complexity of the system grew and memory capacity became constrained.

The debugging environment eventually morphed into a single qEmu emulated device on the development machine combined with a physical android device (Google Pixel 4). This methodology became possible with the introduction of Android version 11 and Android Studio 'Bumblebee' (v2021.1), which allows the developer to run Android debugging tools (i.e the Android Debug Bridge, or ADB) via WiFi connection. This feature was extremely helpful in the development process, and it allowed for the introduction of further feature complexity that would not have been possible otherwise due to the technical limitations of the development machine.

Running a WebSocket-based application locally also presented a minor challenge, as qEmu virtual devices use the traditional localhost/127.0.0.1 address to represent their own internal loopback interface rather than that of the host development machine. This meant that to communicate with the locally hosted server, the special debugging address 10.0.2.2 was used. After the development of the server side of the application was complete, this problem was solved by deploying the server to a static IP address (via an Azure Virtual Machine instance).

Key Dependencies

- Socket.io - v2.0.0 - the developers of socket.io provide a native Java implementation. Due to Java's seamless interoperability with Kotlin, this did not cause any implementation problems with the version implemented.
- BouncyCastle - v1.67 - BouncyCastle is the cryptography API that performs a lot of the cryptographic operations within the application, including generation of elliptic curve keypairs. There is a built-in version of BouncyCastle within the Android SDK, but this does not support ECC, which meant that the client application needed to replace the inbuilt library with a more updated version. This can be seen within the codebase with the following calls within the MainActivity and ChatWindow classes:

  ```
  Security.removeProvider("BC")
  Security.addProvider(BouncyCastleProvider())
  ```

- Room ORM - v2.4.0 - Room provides an object-relational mapping (ORM) over the SQLite database built into the Android SDK. When developing the application, the main data model to consider on the client side was

that of an individual chat message, including metadata (e.g. recipient) and content. Working with Room allowed for a more streamlined development experience and reduction of boilerplate code, as it meant working directly with Kotlin (Java) entities instead of writing complex queries for inserting messages to our viewmodel. Traditional SQL queries were still used to capture relevant data to display to the user in chat windows.

3.9.2 Server

The server application was written in Python v3.9.1 in the PyCharm IDE and utilised several libraries which are highlighted below.

Key Dependencies

- python-socketio - v5.5.0 - a Python server implementation of socket.io, funded by the original socket.io developers.
- aiohttp - v3.8.1 - an asynchronous http server which (importantly) supports websockets. The socket.io process attaches itself to the aiohttp server which allows information to be transmitted and received from clients.

Deployment

To allow users to message other users, it was necessary to deploy the server application on a static IP address. To do this, a 1GB/1CPU Virtual Machine instance was used, running Ubuntu 20.04 in the West Europe region of the Microsoft Azure cloud platform.

3.9.3 Version Control / Branching Strategy

Git was the version control solution used throughout the project. Git is primarily used for team-based development but is still useful for solo development.

The "Trunk-Based Development"[44] methodology was employed throughout. This approach is a key tenet of modern continuous integration, continuous delivery (CI/CD) based workflows. Regular commits were made any time a feature was completed or reached a stable point to allow quick reversions whenever issues were introduced in the development process.

However, on reflection it would have been helpful to be able to use feature branches to test multiple approaches to achieve the same goal. This would represent a more traditional GitFlow[45] branching model.

The host of choice for the upstream repository on this project was GitHub, and the entirety of the project can be found at the following link. It is important to note that the commenting approach may appear excessive (and does not follow the "Code tells you how, Comments tell you why" commenting practice) but this was done with the intention of assisting the reader to get through the code as quickly as possible.

https://github.com/furnivall/SEng_Final_Project

Chapter 4: Evaluation & Testing

4.1 Testing

An important component of software engineering is a continuous testing process. Testing gives the developer an opportunity to ensure the application performs as intended (Quality assurance), and can provide important insights into security, performance under load and improvement possibilities. The development of this project presented some unique testing challenges partly due to its complex architecture and partly due to the limitations of the development environment.

4.1.1 Functional and load testing during development

As mentioned in chapter 3, it was initially fairly simple to test functionality of all sender client $\rightarrow$ server $\rightarrow$ recipient client interactions. As the application grew in size and complexity, this became more difficult. This was handled in two ways, the first (wireless ADB debugging) is discussed in detail in the Development and debugging process section (3.9.1).

The second method utilised was a minimal test client class written in python. This lightweight client provided most of the communication functionality of the Kotlin android client, without the performance overhead of emulation, rendering a UI or other client features. This meant in practice that in addition to testing communication functionality, it was possible to test the performance of the Socket.io connection under heavy load. On the development machine alone, it was relatively simple to run the socket.io server locally and connect over 100 test clients simultaneous constantly broadcasting messages with no performance issues.

Due to the complexity of the encryption implementation, it was unfeasible to implement the client-side encryption system on the lightweight python client. This is discussed in the next section.

4.1.2 Limitations of testing approach

The security features of the system, and particularly the encryption approach, were not tested other than by checking logs to ensure messages were unreadable at the server layer. Further investigation of the cryptography approach included within the application would likely require testers specialising in information security.

Due to the architecture of the application, developing unit test suites was impractical, as very few of the behaviours desired from the application were self-contained. End-to-end or integration testing would have been a better approach, but due to time limitations and complexity these were not pursued.

An example of the difficulty of testing this application could be seen during the development of the encryption functionality. After many wasted hours struggling to understand why the base64 string returned from the server was not

being decrypted correctly, it was determined that a trailing newline was being appended to all messages received from the server. This meant that the encrypted string representation of the plaintext was unreadable to the decryption cipher.

4.1.3 System testing

Before generating the final evaluation version of the application to share with users, the following tests were performed manually within a preconfigured Python shell instance (unless otherwise stated) to verify that all of the functionality requirements from section 2.5 were met. In addition, all functionality was manually tested using the actual client application.

Requirement	Test	Result
Sending messages to server	Send a message to server	Passed
Routing to appropriate client	Emit send_to_user event to server with JSON object containing message and recipient client	Passed
Create username	1. Emit newUsername event to server with username string to verify event triggers. 2. Emit existingUserCheck event to client with "true" parameter	1. Passed 2. Passed
View other connected users	1. Emit getUsers event to server to verify event triggers. 2. Emit users event to client to verify event triggers.	1. Passed 2. Passed
Disappearing messages	Set client self destruct duration to 1 minute. Emit send_to_user event to server, ensure display on client chatWindow activity and wait 1 minute.	Passed
Encrypted messages	Send message from client to client using application and verify that server does not receive plaintext.	Passed
Decrypt messages	Same as above, but also verify that recipient client is able to read the message	Passed
Persistence of data	Start client, connect and set username and fully close application. Restart and ensure that username and destruct duration prompts do not appear.	Partial fail. Username prompt displays very briefly before moving to contacts page.
Push notifications	Emit send_to_user event to server and verify that relevant Kotlin client receives push notification	Passed

Click notification to get to chat window	As above, but Kotlin client short-presses the notification which should open a chat window with the sender.	Passed
Updating contacts on new connections	Emit getUsers event, then connect with another client, then emit another getUsers event and check new user is included.	Passed
Server cannot read contents	Same as "Encrypted Messages"	Passed
Update stored public key on reconnect	Emit getPubKey event, then emit update_pub_key event to server, then emit another getPubKey event and verify public key has changed.	Passed

A more comprehensive automated testing approach is desirable and is a key avenue for future work.

4.2 User evaluations

No matter how comprehensively an automated testing plan is, it is essentially impossible to protect against every potential edge case. Developers are always likely to experience tunnel vision in some regard, overlooking specific user behaviours outwith their own perspective. For this reason, it is important to supplement automated testing with user testing to generate unique potential behaviours that could cause unintended consequences.

4.2.1 Methodology

The user testing approach used for this application was to generate a .APK file containing the application using Android Studio which was then distributed to testers as appropriate. Very small scale (n=3) user alpha testing was initially carried out when the application was in a working state, but still being developed (specifically before the introduction of the encryption features). This guided the development process by capturing key sticking points for users which were then prioritised as development goals. Specifically, the contacts page initially featured a manual refresh button to check for new users. The alpha testers questioned the validity of this approach, resulting in the use of a daemon process to send getUsers events to the server on a regular time interval which provided the illusion of automatically updating the contacts page on new user connections.

Formal user testing followed the implementation of the encryption algorithms. This involved supervised installation and use of the application to achieve a number of tasks shown below. Testing took place both in-person and via video calls. In all cases, a new server instance was started on the Azure virtual machine host. The terminal window showing live server logs was shared with participants to allow them to see the output produced by their actions.

1. Start application

2. Choose a self-destruct duration
3. Choose a pseudonymous username and connect to server
4. Receive a message from at least one other user
5. Receive a notification and click it to open a chat window with said user.
6. Verify that message deletion occurs according to chosen duration.
7. Send a message to another user.
8. View server logs to verify what the message data looks like on the server.
9. Fully close and restart the application
10. Verify that functionality remains intact and that username and delete duration prompts do not appear.
11. Verify that messages can still be sent and received after restart.

Testers were then invited to complete an evaluation form on Google Forms which contained a combination of demographic, exploratory and feedback questions. The form can be viewed at the following link.

`https://forms.gle/xLbLWWnRK1Hh3J7M6`

The signed and completed ethics checklist corresponding with this evaluation can be found in the appendices.

4.2.2 Demographics

Eight testers were able to complete the evaluation. Unfortunately, three additional candidates were unable to complete the testing process due to the application targeting a newer Android version than their device supported, resulting in the application being unable to install.

The testers ranged between ages 18 and 45. Half of the successful testers were between the ages of 18 and 25. All testers self-reported technological ability as “Above average” or “Average” for their age group. The majority of users felt strongly about the need for data privacy.

Interestingly, despite this, the vast majority of users stated that they regularly used chat applications with fairly poor data privacy (87.5% used Whatsapp, whereas only 12.5% used Signal). This is a strong indication that many users feel usability and network effects are larger pull factors than data security.

Users suggested many possible entities they wished to protect their data from, including employers, friends, domestic and foreign governments, advertising companies, hackers and those who wished to screenshot their messages.

Similarly, all users were asked to devise potential scenarios where a person might need secure chat messaging. Answers ranged from protecting private conversations from those that aren’t the intended recipient, avoiding targeted advertising, espionage, criminal activity, sharing credentials (e.g. bank details), infidelity, risque photography and confidential work-related data.

4.2.3 Issues highlighted

The user testing process was invaluable as it provided a great insight into features that were desired as well as unanticipated bugs and suggested improvements to existing functionality.

Several users mentioned that they would have liked the message deletion duration to be defined by the sender. This is more akin to the way it is implemented in Signal, where the sender can define exactly when the message will be deleted from the recipient's device. This would not be particularly difficult to implement, as it could simply be added to the metadata of the message. Potential issues with this approach could be differences in system time across devices (i.e. a potential attacker could change their system time backwards to ensure they could access messages indefinitely).

Furthermore, users highlighted that the implementation of self-destructing messages did not consider the "read" status of the message. In other words, the deletion timer would start at the moment of receipt, rather than when the message is actually viewed. This point is actually fairly difficult to implement, depending on how we define the reading of a message. Whatsapp, for example, has "read receipts" which are only triggered when the application is focused on the chat window where the message is received. This means that a user can treat a message as unread while knowing the content of the message via push notifications. This is a significantly complex issue and merits further work.

On the topic of push notifications, one user determined that the push notifications he received did not expire, even when clicked. This meant that even though the message is deleted from the database, the content is still viewable. This is a major security issue, as if the device is compromised, message history may be visible to the attacker. Luckily, it is not actually particularly complex to fix - the android SDK allows for programmatic expiration of messages via the NotificationManager.cancel method, using the specific notification id as a parameter.

There were also some minor UI bugs discovered on a particular user's device which had different screen dimensions than other testing devices. They concerned the centering of text on the deletion duration screen and would not be a particularly difficult issue to fix.

Another issue which several users described was the back button overriding the flow of the application. This meant that users were able to see parts of the application that they shouldn't have been able to (e.g. username set screen) by pressing back after login. This is reasonably simply solved by manipulating what is known as the "back stack" within the Android SDK, which contains previously visited views.

Despite some issues and suggestions, overall, users all gave highly positive feedback and were able to complete all allocated tasks without major loss of functionality. As such, the development process achieved all the desired outcomes and can be judged a successful endeavour.

Chapter 5: Conclusion

5.1 Project status

The aim of this project was to develop a secure messaging client for Android devices which implemented state-of-the-art encryption, self-destructing messages and pseudonymous identity, sending messages via a central server.

It can be broadly stated that all of these requirements were achieved. Indeed, as shown in section 4.1.3, all of the requirements shown in the user stories from section 2.5 were achieved (with the notable exception of the username screen briefly flashing up on reconnects). In addition, all of the "Must have" and "Should have" requrements were fulfilled from the MoSCoW table in section 2.4.1.

User evaluations were also highly positive and provided some excellent avenues for future development as highlighted in the next section. While there are remaining issues with the application and it needs some degree of polish before wider adoption, it can be seen as a success.

5.2 Future work

Further work on this project falls into two distinct groups: improvements / reworking of existing features and development of fundamentally new functionality. Trivially fixable bugs or simpler feature requests previously mentioned will not be discussed here.

As highlighted in Chapter 4, further work should certainly focus on the redesign of the self-destructing messaging feature. Adding a means of assessing "read" status before the deletion countdown starts would be a great start. Additionally, adding the capacity for senders to ensure messages expire on recipient devices (i.e. sender-defined deletion) is an important step forward and should probably have been the initial approach. If these two features can be implemented, the user could feel significantly more secure and in control of their data.

This naturally leads to the issue of improving the client-side security of the application. Most of the work in this project has focused on the desire to protect messages from being readable by anyone except the intended recipient in the transit phase, but not enough work has been spent on securing the message once it reaches the target device. Encrypting the SQLite local database would be a simple step to reduce the potential damage an attacker could cause, even with root access to the device.

On the server side, there are some major design changes that would improve the user experience significantly. Implementing message queuing and receipt callbacks would be a major improvement, as they would allow messages to be sent to an offline user easily. It is worth mentioning that this may conflict with

sender-defined self destructing messages, if combined with the read receipt approach from above. In other words, if a sender sends a message to a recipient whose device becomes compromised while they are offline for a significant period (e.g. if they are detained by law enforcement or government authorities), the messages would appear the second they reconnect to the server, which could potentially incriminate either the sender or recipient. This could potentially be handled by allowing the server to delete messages after a given timeframe if the sender allows it.

Another key area that would be good to focus on is the introduction of group chats. During the literature review for this project, the concept of client-side fanout was particularly appealing as a way of implementing end-to-end encryption within group chats. The combination of an architectural elegance and security is a deeply appealing prospect to work on.

In terms of testing, implementing end-to-end testing and integration tests for the application is a key area for further work. This allows for automatic verification of program behaviours and will make it easy to see the impacts of changes quickly without manual investigation. On reflection, not implementing this early in the development process was a major error as it likely resulted in many hours of wasted time on repetitive tasks.

5.3 Final thoughts

The project created many opportunities to learn new skills, gain understanding and solve highly interesting problems.

It was also an opportunity to understand just how complex some of the challenges involved in development of messaging applications are, despite seeming superficially simple.

Chapter 6: Appendices

6.1 Wireframes

Wireframes were generated using the Balsamiq application.

Figure 6.1: Chat window

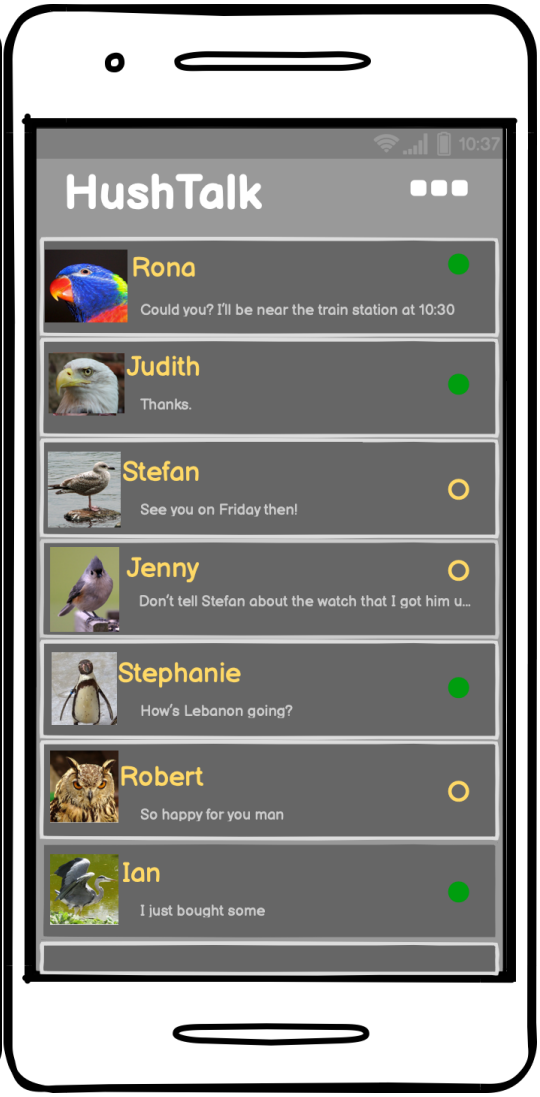

Figure 6.2: Contacts page

6.2 Ethics form

School of Computing Science
University of Glasgow

Ethics checklist form for 3rd/4th/5th year, and taught MSc projects

This form is only applicable for projects that use other people ('participants') for the collection of information, typically in getting comments about a system or a system design, getting information about how a system could be used, or evaluating a working system.

If no other people have been involved in the collection of information, then you do not need to complete this form.

If your evaluation does not comply with any one or more of the points below, please contact the Chair of the School of Computing Science Ethics Committee (matthew.chalmers@glasgow.ac.uk) for advice.

If your evaluation does comply with all the points below, please sign this form and submit it with your project.

---

1. Participants were not exposed to any risks greater than those encountered in their normal working life.
   Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that occur outside usual laboratory areas, or that require participant mobility (e.g. walking, running, use of public transport), unusual or repetitive activity or movement, that use sensory deprivation (e.g. ear plugs or blindfolds), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback

2. The experimental materials were paper-based, or comprised software running on standard hardware.
   Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, laptops, iPads, mobile phones and common hand-held devices is considered non-standard.

3. All participants explicitly stated that they agreed to take part, and that their data could be used in the project.
   If the results of the evaluation are likely to be used beyond the term of the project (for example, the software is to be deployed, or the data is to be published), then signed consent is necessary. A separate consent form should be signed by each participant.

   Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script.

4. No incentives were offered to the participants.
   The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

5. No information about the evaluation or materials was intentionally withheld from the participants.
   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

6. No participant was under the age of 16.
   Parental consent is required for participants under the age of 16.

7. No participant has an impairment that may limit their understanding or communication.
   Additional consent is required for participants with impairments.

8. Neither I nor my supervisor is in a position of authority or influence over any of the participants.
   A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. All participants were informed that they could withdraw at any time.
   All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.

10. All participants have been informed of my contact details.
    All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.

11. The evaluation was discussed with all the participants at the end of the session, and all participants had the opportunity to ask questions.
    The student must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. In cases where remote participants may withdraw from the experiment early and it is not possible to debrief them, the fact that doing so will result in their not being debriefed should be mentioned in the introductory text.

12. All the data collected from the participants is stored in an anonymous form.
    All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.

Project title: "A secure client-server mobile chat application implementing an elliptic curve integrated encryption system (ECIES) and other security features."

Student's Name : Daniel Furnivall

Student Number : 0801323

Student's Signature : Daniel Furnivall

Supervisor's Signature _____

Date : 30/03/21

Figure 6.3: Ethics checklist form

Bibliography

[1] R. Baeza-Yates and U. M. Fayyad. The attention economy and the impact of artificial intelligence. In *Perspectives on Digital Humanism*, pages 123–134. Springer, Cham, 2022.

[2] C. Walter. Kryder's law. *Scientific American*, 293(2):32–33, 2005.

[3] A. Farahat and M. C. Bailey. How effective is targeted advertising? In *Proceedings of the 21st international conference on World Wide Web*, pages 111–120, 2012.

[4] K. Pantelis and L. Aija. Understanding the value of (big) data. In *2013 IEEE International Conference on Big Data*, pages 38–42. IEEE, 2013.

[5] S. Zawoad and R. Hasan. Digital forensics in the age of big data: challenges, approaches, and opportunities. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 1320–1325. IEEE, 2015.

[6] R. Simmons. Big data and procedural justice: legitimizing algorithms in the criminal justice system. *Ohio St. J. Crim. L.*, 15:573, 2017.

[7] D. Kreiss and S. C. McGregor. The "arbiters of what our voters see": facebook and google's struggle with policy, process, and enforcement around political advertising. *Political Communication*, 36(4):499–522, 2019.

[8] H. Berghel. Malice domestic: the cambridge analytica dystopia. *Computer*, 51(5):84–89, 2018.

[9] C. Stöcker. How facebook and google accidentally created a perfect ecosystem for targeted disinformation. In *Multidisciplinary International Symposium on Disinformation in Open Online Media*, pages 129–149. Springer, 2019.

[10] P. Sommer. Evidence from hacking: a few tiresome problems. *Forensic Science International: Digital Investigation*, 40:301333, 2022.

[11] B. Marczak, J. Scott-Railton, S. McKune, B. Abdul Razzak, and R. Deibert. HIDE AND SEEK: Tracking NSO Group's Pegasus Spyware to operations in 45 countries. Technical report, 2018.

[12] M. Mazzetti, A. Goldman, R. Bergman, and N. Perlroth. A new age of warfare: how internet mercenaries do battle for authoritarian governments. *The New York Times*, 21:2019, 2019.

[13] J. Alwen, S. Coretti, and Y. Dodis. The double ratchet: security notions, proofs, and modularization for the signal protocol. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 129–158. Springer, 2019.

[14] J. Jakobsen and C. Orlandi. On the cca (in) security of mtproto. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 113–116, 2016.

[15] M. L. Jansen. A security analysis of the signal protocol's group messaging capabilities in comparison to direct messaging, 2020.

[16] A. Cooper. The inmates are running the asylum. In *Software-Ergonomie'99*, pages 17–17. Springer, 1999.

[17] T. Miaskiewicz and K. A. Kozar. Personas and user-centered design: how can personas benefit product design processes? *Design studies*, 32(5):417–430, 2011.

[18] D. Haughey. Moscow method. *Project Smart*:2011, 2011.

[19] F. Dalpiaz and S. Brinkkemper. Agile requirements engineering with user stories. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*, pages 506–507. IEEE, 2018.

[20] M. Masse. *REST API design rulebook: designing consistent RESTful web service interfaces*. " O'Reilly Media, Inc.", 2011.

[21] I. Fette and A. Melnikov. The websocket protocol, 2011.

[22] V. Herwig, R. Fischer, and P. Braun. Assessment of rest and websocket in regards to their energy consumption for mobile applications. In *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, volume 1, pages 342–347. IEEE, 2015.

[23] R. Rai. *Socket. IO Real-time Web Application Development*. Packt Publishing Ltd, 2013.

[24] A. Mallik. Man-in-the-middle-attack: understanding in simple words. *Cyberspace: Jurnal Pendidikan Teknologi Informasi*, 2(2):109–134, 2019.

[25] A. Apvrille and M. Pourzandi. Secure software development by example. *IEEE Security & Privacy*, 3(4):10–17, 2005.

[26] R. L. Rivest, A. Shamir, and L. M. Adleman. *A method for obtaining digital signatures and public key cryptosystems*. Routledge, 1982.

[27] J. Lopez and R. Dahab. An overview of elliptic curve cryptography, 2000.

[28] M. Gobi, R. Sridevi, and R. Rahini. A comparative study on the performance and the security of rsa and ecc algorithm. In *Proceedings of Conference on Advanced Networking and Applications*, 2015.

[29] H. Delfs and H. Knebl. Symmetric-key encryption. In *Introduction to Cryptography*, pages 11–31. Springer, 2007.

[30] V. G. Martınez, L. H. Encinas, et al. A comparison of the standardized versions of ecies. In *2010 Sixth International Conference on Information Assurance and Security*, pages 1–4. IEEE, 2010.

[31] V. Gayoso Martínez, L. Hernandez Encinas, and A. Queiruga-Dios. Security and practical considerations when implementing the elliptic curve integrated encryption scheme. *Cryptologia*, 39:1–26, May 2015. DOI: 10.1080/01611194.2014.988363.

[32] U. M. Maurer and S. Wolf. The diffie–hellman protocol. *Designs, Codes and Cryptography*, 19(2):147–171, 2000.

[33] M. Brown, D. Hankerson, J. López, and A. Menezes. Software implementation of the nist elliptic curves over prime fields. In *Cryptographers' Track at the RSA Conference*, pages 250–265. Springer, 2001.

[34] T. Cooijmans, J. de Ruiter, and E. Poll. Analysis of secure key storage solutions on android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 11–20, 2014.

[35] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., USA, 2000. ISBN: 020161622X.

[36] R. C. Martin. Design principles and design patterns. *Object Mentor*, 1(34):597, 2000.

[37] V. Sarcar. Command pattern. In *Design Patterns in C#*, pages 315–335. Springer, 2020.

[38] A. Eales and A. Eales. The observer pattern revisited. *Educating, Innovating & Transforming: Educators in IT: Concise paper*, 2005.

[39] R. Harmes and D. Diaz. The adapter pattern. *Pro JavaScript Design Patterns*:149–158, 2008.

[40] V. Sarcar. Builder patterns. In *Java Design Patterns*, pages 89–95. Springer, 2016.

[41] S. LONG, H. LIN, and X. CHEN. Data access object pattern. *Computer and Modernization, pp5*, 2004.

[42] K. Stencel and P. Węgrzynowicz. Implementation variants of the singleton design pattern. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 396–406. Springer, 2008.

[43] Guide to app architecture - android developers. URL: https://developer.android.com/jetpack/guide#recommended-app-arch.

[44] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[45] A. Dwaraki, S. Seetharaman, S. Natarajan, and T. Wolf. Gitflow: flow revision management for software-defined networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–6, 2015.