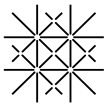

University of Basel

Implementing the Double Ratchet algorithm in Tremola, a Scuttlebutt based messaging app for Android

Bachelor Thesis

Faculty of Science
Department of Mathematics and Computer Science
Computer Networks Group
https://cn.dmi.unibas.ch

Examiner: Prof. Dr. Christian Tschudin
Supervisor: Dr. Erick Lavoie

Lars Waldvogel
lars.waldvogel@stud.unibas.ch
17-056-243

31.10.2022

Acknowledgments

I would like to thank Professor Tschudin for coming up with the idea of adding the Double Ratchet algorithm to the existing Tremola app. Dr. Erick Lavoie provided great feedback while the thesis was in the making, pointing me the right way in the process. Also, I would like to thank my father, Marcel Waldvogel, for looking over the first draft of my thesis and providing feedback on how to improve upon it.

Abstract

The Android messaging app Tremola uses the Scuttlebutt peer-to-peer gossiping protocol to transfer its messages from one user to another. This approach already supports encryption out of the box due to the properties of the Scuttlebutt protocol, where every user's identity is made up of a public/private key pair. However, should a user's key pair be compromised, all the messages they sent and received can be decrypted. Intercepting these messages is also trivial due to the nature of Scuttlebutt, where all messages are saved in an append-only log and distributed among peers.

In this thesis, we implemented the Signal protocol's Double Ratchet algorithm to provide forward secrecy and what is known as post-compromise security for these messages. This implementation took the special properties of the Scuttlebutt protocol into account to draw on its strengths, but also required some compromises to be made.

Table of Contents

1 Introduction

The Tremola app [29] enables people to exchange messages securely without requiring a central server, continuous connectivity or an internet connection at all. Instead, it is based on the Scuttlebutt (SSB) protocol [17], which relies on peer-to-peer (P2P) communication to pass the encrypted messages from one client to another. This makes it possible to use Tremola even in regions with no access to the internet.

Under the Scuttlebutt protocol, all the messages a person sends will be replicated to multiple peers to pass them on. Since we still want to preserve the privacy of these messages, they are all end-to-end encrypted (E2EE). All messages a person receives will be encrypted with the same key, namely their long-term public identity key of the Scuttlebutt protocol.

However, should an attacker compromise somebody's long-term secret identity key, for example via malware or a vulnerability in the app, they would be able to read all incoming and outgoing messages of that person. In general, using a single key to encrypt multiple messages leads to the same problem: If the key is leaked, an attacker would be able to decrypt a substantial amount of information. Therefore, using a new key for each message is a more secure option. Should an attacker eventually compromise one of the message keys through, say, a brute-force attack, they would only be able to decrypt a single message.

Two parties agreeing upon a common, secret key is possible over a reliable internet connection. The Diffie-Hellman key exchange [18] made it possible for two parties to establish a secret even over a public, possibly wire-tapped channel. Since this involves each party sending and receiving at least one message, it is not possible to negotiate multiple keys if one party should be offline. Thus, we require a way to derive new common keys even if a party should not be available for connections. One of the most practical solutions today was established by the Signal Protocol [24], which is used in many instant messengers today [21] [9] [20] [10]. The underlying Double Ratchet algorithm [25] – which is part of the Signal protocol – generates encryption keys to be used for each message. These keys are derived in such a way, that should a ratchet's state be compromised, all past keys (and thus all past messages) remain secure. This feature is known as forward secrecy. In addition to this, an attacker is only able to decrypt the next few messages (namely until a response is received) before the self-healing property of the algorithm takes effect and re-establishes secrecy.

This thesis combines the beneficial features of both the Scuttlebutt protocol and the Double

Ratchet algorithm in the Tremola app. While the Double Ratchet algorithm was designed with a central server in mind, the Scuttlebutt protocol relies solely on peer-to-peer networks. We will see that the Double Ratchet algorithm possesses additional benefits, which we are unfortunately unable to make use of due to the nature of how Scuttlebutt stores its messages, such as plausible deniability. However, we did end up with Tremola now featuring forward secrecy and the self-healing properties of the Double Ratchet algorithm.

1.1 List of contributions

1.1.1 Primary contributions

We implemented a general-purpose Double Ratchet class in Kotlin. This variant can be used for a number of different applications, which are not only limited to the Tremola app. On top of that, we designed and implemented a variant of the Double Ratchet class which is meant to be used in conjunction with the Scuttlebutt protocol. This variant was introduced into the app to provide the desired future and forward secrecy.
A preliminary security audit was performed. This was done in an effort to identify aspects which might compromise the security properties of the Double Ratchet implementation and the app in general. Our findings are listed in Appendix A.

1.1.2 Secondary contributions

We added over a thousand lines of documentation to existing code of the Tremola app. Also, some files were reformatted without changing their functionality. Additionally, our implementation features a number of unit tests. This was done in an effort to make it easier for both us and future contributors to work with the code and find bugs.

1.2 Structure

We will first lay the groundwork in Chapter 2 by explaining the different parts that make up this project. In Chapter 3, we will look at the results that we achieved and how we obtained them. Chapter 4 presents a conclusion to our work with some possible future contributions. In Appendix A, we list the found vulnerabilities and how they could be counteracted. Appendix B contains a glossary of most of the technical terms used in this thesis.

2
Background

This chapter talks about the different things that make up our thesis: the Double Ratchet algorithm, the Scuttlebutt protocol and the Tremola app. The Double Ratchet algorithm is what we are looking to introduce into Tremola. Tremola uses the Scuttlebutt protocol to exchange messages. Finally, having a basic comprehension of the Tremola app is necessary to understand what exactly we changed about it and why.

2.1 Assumptions

Throughout this thesis, we assume that current cryptographic functions are not easily broken and therefore secure. This means that a public key will not help in deriving its private key. Also, it is hard to derive a hash input even if you know its output. This is called preimage resistance. Generally, we assume that the best method to crack the cryptographic tools is brute force, and there are no methods which would trivialize breaking the cryptographic algorithms. Generating hash-collisions is considered computationally impossible. This property is commonly known as collision resistance.

2.2 Double Ratchet algorithm

The Double Ratchet algorithm (DRA) is used to encrypt and decrypt messages between two parties, each message with its individual key. This entire chapter is heavily based on the Signal protocol's specification of the Double Ratchet algorithm [25].

2.2.1 The ratchet

The name of the algorithm stems from the key-derivation function chain (KDF chain, see below) at its core. Given some input, it will produce output from which the input cannot be derived. In that way, it functions like a physical ratchet (see Fig. 2.1), which can turn one way but not the other.

Also, the KDF chain produces unique outputs for each of its states. This can be compared to the different steps in a ratchet. Every time the ratchet is used, it takes exactly one step

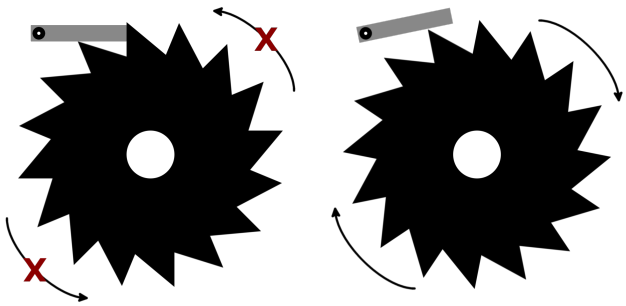

Figure 2.1: A physical ratchet can be easily turned in one direction, but not in the other.

forward and cannot be turned back. The analogy is however imperfect: Unlike a physical ratchet, a KDF chain will not come back around to one of its earlier states in practice. The number of possible states is so vast, it can be considered infinite for our purposes.

2.2.2 KDF chain

A KDF chain consists of a key derivation function (KDF), which is a function that takes a secret key and some data as input, deriving two pseudo-random outputs from them. The outputs cannot be distinguished from random data without knowing what produced them. Still, the KDF is deterministic: given the same inputs, it will always produce the same outputs. This can for example be used to derive a cryptographically strong key from a human-readable password. The input cannot be derived from the output: This is what gives the ratchet its one-way property. A KDF is depicted in Fig. 2.2.

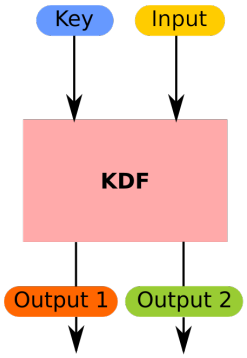

Figure 2.2: A Key Derivation Function (KDF) takes a secret key and some input data to produce two pseudo-random outputs. This computation cannot be done in reverse, i.e., this is a one-way function.

A KDF can be repeatedly used over multiple steps, forming a chain. The first output is used as the input key of the next step. The other output is our actual output, which we can use as a cryptographic key, for example. Using an output from one step as a key for the subsequent steps will produce different outputs every time. We refer to the current input key of a KDF chain as its ”state”. Thus, we have the separate steps of a ratchet, with every step featuring its own, unique state. This is one of the properties of a ratchet. Since it is

not feasible to deduce any input from the outputs, it is only possible to step forwards, not backwards. This represents the other property of a ratchet, the fact that it only turns in one direction. In Fig. 2.3, we see such a KDF chain.

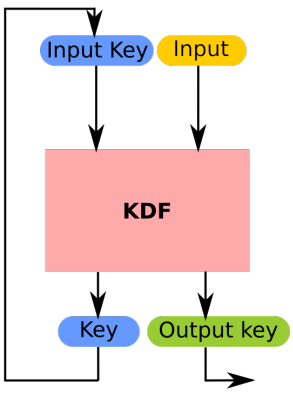

Figure 2.3: A KDF chain produces two outputs. One of its outputs is used as its next input key, the other is the output key.

The KDF chain will still require a first input key to initialize it. How the input key and the input data are chosen depends on the use of the ratchet.

2.2.3 Using symmetric ratchets for two-way communication

Two parties communicating with each other want to derive unique keys for each message they send. For this, each communication direction requires a pair of ratchets: one for sending and one for receiving. Because both ratchets compute the same sequence of keys, they are called symmetric. The KDF a symmetric ratchet uses is referred to as the chain KDF. As input data for the chain KDF, constant values are chosen. These values can be public and are typically set to `0x01` and `0x02`. It is important that these values are chosen to be distinct from one another.

Let us look at an example. Following a long-running cryptographic tradition, we will call the two parties wanting to communicate Alice and Bob. Alice and Bob both initialize their symmetric ratchets for one direction of communication. Alice initializes her sending ratchet with the same key as Bob's receiving ratchet. This process is repeated for the other communication direction as well. Having been given the same initializing input key, the output two ratchets of the same direction generate will be identical, given that they are on the same step. Fig. 2.4 depicts such a symmetric ratchet.

Whenever Alice wants to send a message to Bob, she will simply turn her sending ratchet by one step and take the derived output key as the key to encrypt her message with. The output key of a symmetric ratchet is called a message key. Once Bob receives the message, he will turn his receiving ratchet by one step and use the resulting message key to decrypt the message. Since they were both on the same step to begin with, the input fed to the KDF (the previous output key and a constant value) will produce the same output. Bob will follow the same steps whenever he sends a message to Alice, but he will use his sending

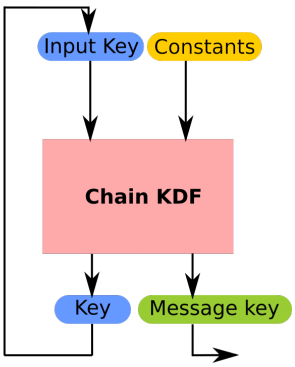

Figure 2.4: The sending and receiving ratchets take constant values as their input. The output is used as a key to encrypt a single message.

ratchet and Alice will use her receiving ratchet.
These ratchets provide forward secrecy, since the compromise of a ratchet's state in the future does not allow the decryption of previous messages. Should one of the message keys be compromised, no further message keys can be derived from it, since the output key is separate from the key fed back into the KDF chain. Should an attacker learn the input key and the constant used as inputs in the KDF chain, they would be able to produce all message keys generated by that chain from that point forward.
In the case of the Signal protocol, the specific cryptographic primitive used for the chain KDF is a keyed-hash message authentication code (HMAC) [2]. Since a HMAC function only produces one output, it is actually called twice, with two different constants as its input data. This is depicted in Fig. 2.5. In it, we can see the inner workings of the chain KDF in the Signal protocol.

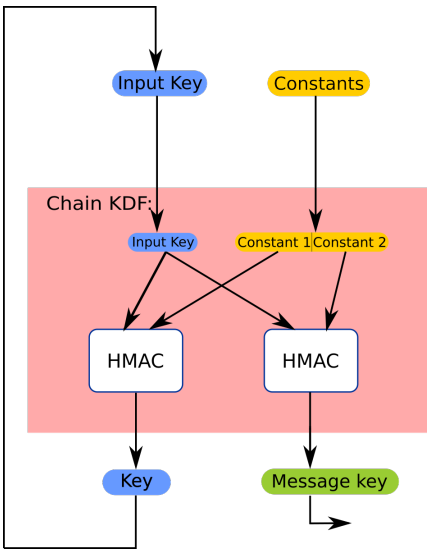

Figure 2.5: Here we see more specifically how the Signal protocol chooses to implement the chain KDF. It relies on two calls of the HMAC function with different constants.

2.2.4 Using the Diffie-Hellman ratchet for post-compromise security

The Diffie-Hellman key exchange (DHKE) [18] enables two parties to derive a common secret over a potentially wire-tapped channel. It is computationally impossible for a potential eavesdropper to derive the same secret, even if they captured all exchanged messages.
The Diffie-Hellman ratchet (also known as the root ratchet) will take the resulting secret of such a key exchange as its input. The ratchet's initial input key is a shared secret between Alice and Bob. While the Double Ratchet algorithm is indifferent as to how such a shared secret is produced, the Signal protocol has its own solution to derive it: the extended triple Diffie-Hellman key agreement protocol (X3DH) [22]. It solves two problems at the same time: First, how Alice can obtain Bob's public Diffie-Hellman key even if he is offline and second, how the two can derive a shared secret. This is achieved via the use of pre-published keys, called prekeys. Every user creates these prekeys and sends them to the central server. A sender that wants to start a new conversation will ask the server for a prekey of the desired recipient. Once the sender obtains it, they can derive the initial shared secret and have a public Diffie-Hellman key of the other party. Thus, they can initialize the root ratchet.
The output key of the Diffie-Hellman ratchet, called a ratchet key, is used to initialize a symmetric ratchet. The KDF such a ratchet uses is called the root KDF. We can see the relationship the root ratchet has with a symmetric ratchet in Fig. 2.6. The Diffie-Hellman ratchet is depicted on the left.

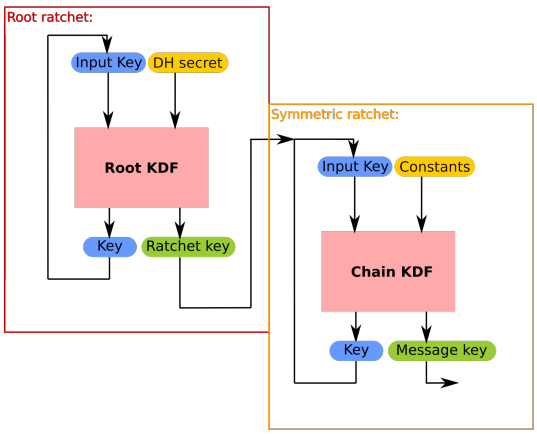

Figure 2.6: The Diffie-Hellman ratchet (left) takes the result of a Diffie-Hellman key exchange as its input. The output of the Diffie-Hellman ratchet is the initializing input key for a new symmetric ratchet (right).

This ratchet is responsible for the post-compromise security that the Double Ratchet algorithm provides [11]. Should an input key of a symmetric ratchet be compromised, all future message keys in this ratchet become known to the attacker. However, the Double Ratchet algorithm regularly replaces the symmetric ratchets by initializing new ones with its output. If an eavesdropper gains knowledge of such an output key, they can reproduce the generated ratchet and thus all message keys that it can produce. But the output that the Diffie-Hellman ratchet generates cannot be deduced from its earlier outputs or even from its earlier inputs. This is because its input stems from a DHKE. Whenever a DHKE is completed, its output is used as the input data of the Diffie-Hellman ratchet (DH ratchet) to

generate a new ratchet. From this point on, an attacker could only read messages again by compromising a new key or by deriving the shared secret of the DHKE, which is currently deemed computationally impossible. Thus, a single compromise will no longer compromise all following messages, but only the messages of the current symmetric ratchet. This property is generally referred to as post-compromise security or future secrecy [11].
In the case of the Signal protocol, the specific cryptographic primitive used for the root KDF is a HMAC-based extract-and-expand key derivation function (HKDF) [3].

2.2.5 The Diffie-Hellman key exchanges and initialization of ratchets

Diffie-Hellman key pairs have an interesting property: Given two people Alice and Bob each have their own Diffie-Hellman key pair. They can then exchange their public keys to produce a shared secret. If the keys are of the elliptic curve primitive, a scalar multiplication of Alice's public key with Bob's private key will derive the same secret as the scalar product of Alice's private key with Bob's public key. This property is central to the function of the Diffie-Hellman ratchet. Additionally, an eavesdropper cannot derive the same secret with just the two public Diffie-Hellman keys they might have captured.
Each message of Alice and Bob contains their most recent public Diffie-Hellman key. Whenever one of them receives a new public Diffie-Hellman key from their correspondent, they derive a shared secret with their existing key pair, then step their Diffie-Hellman ratchet forward to produce a new receiving ratchet. Then, they generate a new Diffie-Hellman key pair, calculate another secret with the received public key and step the ratchet forward one more time, producing a new sending ratchet. Their future messages will then contain the newly generated public key in their header.
These DHKE between the two parties are performed as often as possible. This means two new secrets are derived whenever an answer is received to one of your own messages and thus two new symmetric ratchets are initialized.

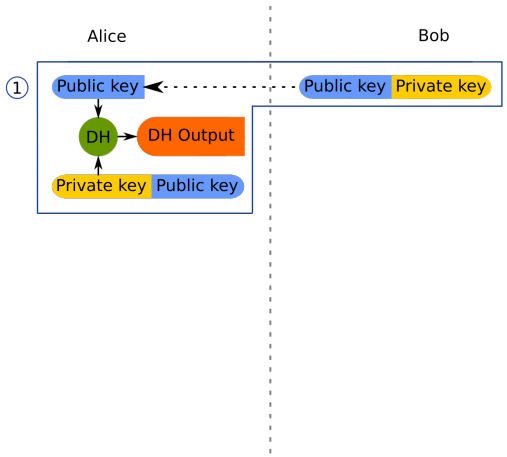

Figure 2.7: The first step of Alice and Bob exchanging their Diffie-Hellman keys. Already aware of Bob's public key and another shared secret, Alice can create a new key pair to derive the first common secret. The green "DH" represents the Diffie-Hellman operation to derive a shared secret key. The orange "DH Output" represents the secret key derived from the Diffie-Hellman key exchange, used for stepping the root ratchet forward.

Let us look at this in a concrete example. The first step can be seen in Fig. 2.7. Before using the Double Ratchet algorithm, Alice already has to know Bob's public Diffie-Hellman key in addition to a shared secret. In the setting of the Signal protocol, this is usually accomplished using the X3DH [22] over a central server. After Alice has learned of Bob's public key and the initial shared secret, she will then generate a new Diffie-Hellman key pair. These key pairs are based on elliptic-curve cryptography (ECC) – the Curve25519 primitive to be specific – and thus allow the operation of scalar multiplication on their keys. By multiplying her private key with Bob's public key, Alice will derive their shared Diffie-Hellman secret. To initialize her root ratchet, the shared Diffie-Hellman secret is used as input in combination with their initial shared secret which serves as the initial input key. The output of the root ratchet is used initialize her symmetric sending ratchet.

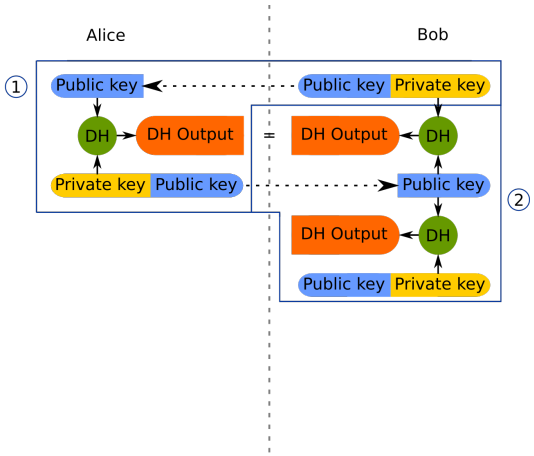

Figure 2.8: In the second step, Alice sends a message to Bob containing her public key. He can thus derive the same secret she did. Also, by initializing a new key pair, he can produce the next common secret.

The next step is depicted in Fig. 2.8. Now Alice will send her public key to Bob, who will also perform a scalar multiplication to derive the same shared secret, this time multiplying his private key with Alice's public key. This shared secret is used as the initialization key for his receiving ratchet. His receiving ratchet and Alice's sending ratchet have thus been initialized with the same values and will yield identical message keys in the steps of the same number. Whenever Alice sends a message, she will derive the key from the ratchet by ticking it one step forward. Bob will derive the same key from his ratchet and can thus decrypt the message. Alice will keep using this ratchet until she receives a reply from Bob containing a new public key in its header. All the messages produced by a single symmetric ratchets are known as a message chain.
Now, Bob will generate a new Diffie-Hellman key pair. He will multiply his new private key with the previously received public key. The derived secret is used to generate his new sending ratchet.
In Fig. 2.9 we see the third and final step. Once he sends a message to Alice, he will include his new public key in each of the message's unencrypted headers. Alice will then be able to derive the shared secret by multiplying her old private key with the newly received public

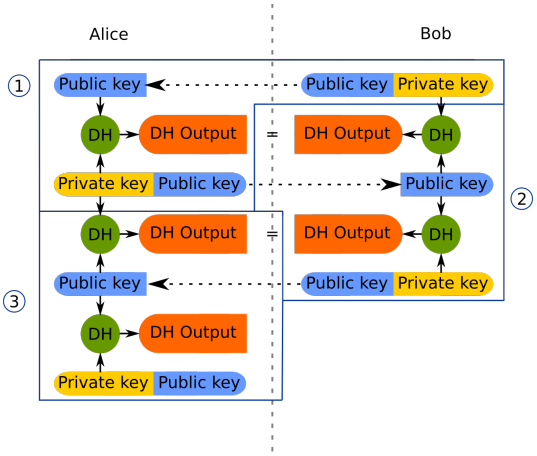

Figure 2.9: Now at the third step, Bob sends his public key to Alice. She will then be able to derive the shared secret Bob produced. Thus the cycle begins from anew.

key and initialize her receiving ratchet. Once again, her receiving ratchet and Bob's sending ratchet have been initialized with the same values and are thus identical in terms of which message keys they will produce.

Whenever Alice messages Bob again after receiving his reply, she will generate a new Diffie-Hellman key pair and begin the process described above from anew. Note that all ratchets are used only up until the point where new ratchets of the same type are produced. At that point, the old ratchets will be replaced by the new ones. Which outputs belong to which ratchets is demonstrated in Fig. 2.10.

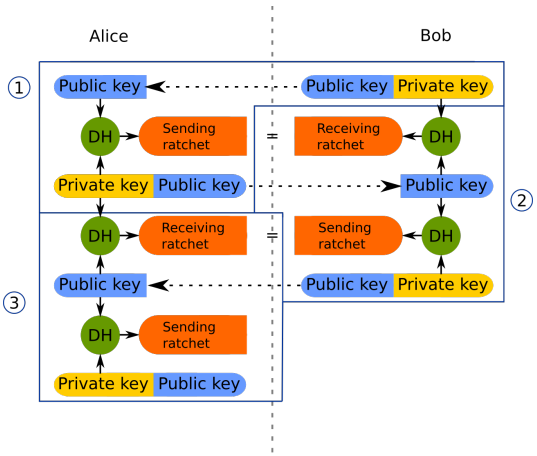

Figure 2.10: Here we can see which Diffie-Hellman outputs are used to initialize which ratchets.

2.2.6 Out-of-order messages

Over a network like the internet, not all messages are guaranteed to arrive in the order they were sent. Thus, we label each message in its header with the message's number in the current sending chain, the previous sending chain's length and the current public Diffie-Hellman key of the sender. The message's number is uniquely determined by the currently

used public key and the number of the message in the current chain (which means the amount of times the ratchet has been stepped forward since its initialization). In addition, the length of the previous message chain is included to clearly identify the amount of possible skipped messages. These three values are sent in the unencrypted header of the message. This allows the recipient to determine whether they missed any of the previous messages and how many times they have to step the root and receiving ratchets forward.
Should Alice receive a message which is out of order but still bears the current Diffie-Hellman public key, she will tick her receiving ratchet forward until it produces the appropriate message key for the sequential number of the message she just received. The intermediate keys she produced, but did not yet use, are saved, identified by their number and the public key they were associated with. Should she later receive one of the skipped messages, she will be able to decrypt it with the help of these saved message keys.
If Alice receives a reply bearing a new ephemeral public key, she checks whether she has received all messages of the previous chain which used the old public key. If she did not, she will step the ratchet forward for all lost messages and save their keys. Then she will initialize her new receiving ratchet. If the newly received message bears the number 0, she did not skip any messages in the new chain. If its number is higher, she will once again step forward the new ratchet to the right position, save the skipped keys and then decrypt the message.

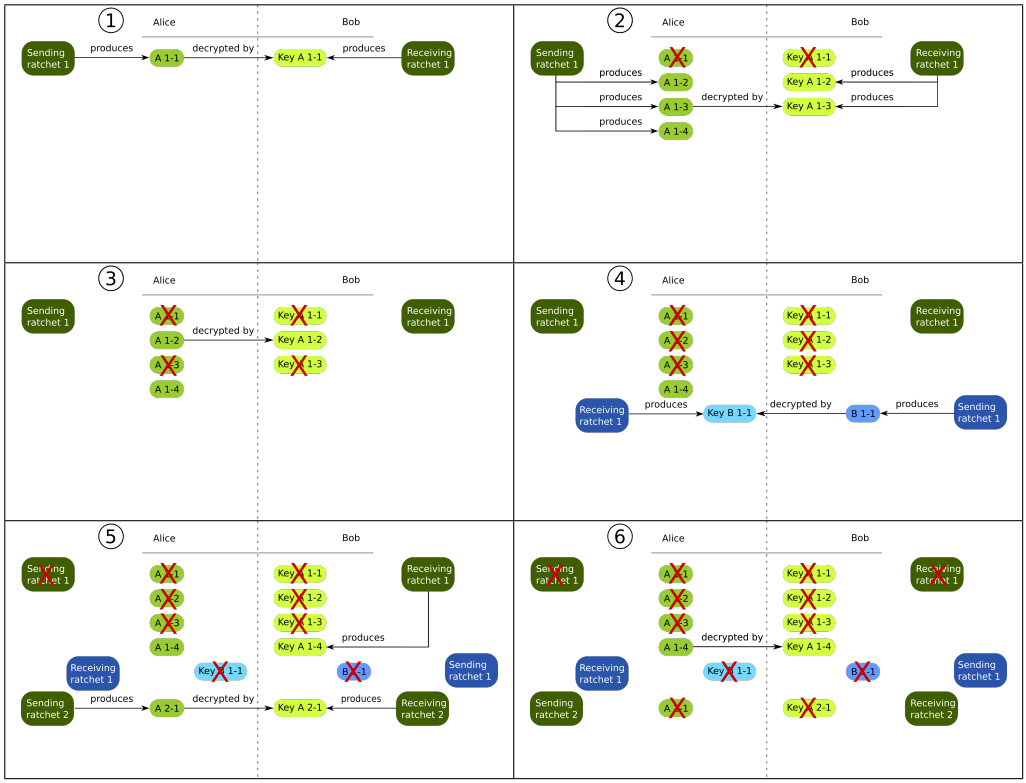

Figure 2.11: This table of figures shows the labelled steps of a scenario where messages arrive out of order. The ratchet will produce all keys up to the newest message it received and store unused keys for later decryption. A red X denotes a key that has been used, a message that has been decrypted or a ratchet which has been superseded.

Let us look at an illustrated example in the table of Fig. 2.11. Each of the figures is labeled with a step number on its upper left. In step 1, a message is sent normally. The encrypted message is denoted "A 1-1" since the sender is Alice, it is the first ratchet and the first message of that ratchet. Bob will in turn produce the key to decrypt said message "Key A 1-1". The used key and decrypted message are crossed of with a red X. In step 2, Alice produces 3 more messages, but message "A 1-3" arrives first. Bob detects that message "A 1-2" has been skipped. He will produce the key for it and store it until it is used. Then he will decrypt the received message with the next ratchet step.
In step 3, the delayed message "A 1-2" arrives. Since Bob has stored its key, he is able to decrypt it. At this point in step 4, Bob sends a message of his own, "B 1-1", containing a new public Diffie-Hellman key in its header. Alice decrypts it and initializes a new sending ratchet since a Diffie-Hellman key exchange has taken place.
In step 5, she sends another message to Bob. In the header, Bob will see that he did not receive all messages of the previous message chain and will thus produce and store "Key A 1-4". After this, he initializes a new receiving ratchet, superseding his old one, to decrypt message "A 2-1". In step 6, message "A 1-4" finally arrives. Bob can decrypt it since he stored its key.

2.2.7 Analysis

The Double Ratchet algorithm on its own does not offer unbreakable security, although it is strong when used in the right scenario in combination with the right tools.

2.2.7.1 Strengths

Security: The algorithm offers forward secrecy via the symmetric ratchets. If the input key of a symmetric ratchet is compromised, only messages produced or decrypted by that ratchet from that point onward can be read. If a ratchet key is compromised, one whole symmetric ratchet can be derived and thus all messages it handles. Any ratchet produced after this will not be compromised, since the shared secret of a DHKE is utilized to produce the next state of the root ratchet and thus any newly produced ratchet. While some messages might be compromised, the algorithm's secrecy will eventually "self-heal" and restore secrecy (see Fig. 2.12). It is the Diffie-Hellman ratchet which provides future secrecy.
Should a message key be leaked, this does not compromise any other messages. The message key is distinct from the symmetric ratchet's input key, which makes it impossible for other message keys to be derived from it.
The security of the overall algorithm has also been analyzed by security researchers [12].

Efficiency: Despite the apparent complexity of the ratchets, only few variables need to be stored and updated. The cryptographic functions used are optimized on modern chipsets, taking very little time to execute. Thus, the Double Ratchet algorithm produces little overhead and is used even in bandwidth intensive applications like video calls [13].

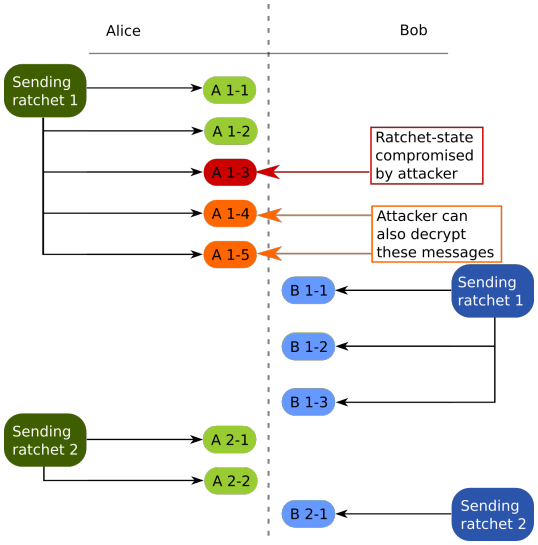

Figure 2.12: The attacker compromises the state of Alice's sending ratchet 1 at the red message. They can thus decrypt the orange messages. All other messages (blue and green) remain secret.

Offline reliability: The algorithm can easily be used even if no direct connection to the other party is available since the keys can be derived offline and out-of-order messages are handled well.

Plausible deniability: After the initialization, the Double Ratchet algorithm provides plausible deniability: This means that it is not possible to definitely prove that a message was sent by one of the parties, since both parties could forge any and all messages between them. Still, whenever Alice receives a message from Bob, she can be sure it was sent by Bob since only she and Bob know the ratchets to produce and decrypt it. However, she could easily forge a message from Bob exactly because she knows the state of the ratchets. Thus, Alice can only prove to herself that Bob sent a message, but not to anyone else. This is possible since no signatures or similar are involved, only symmetric operations.

Openness: Since the Double Ratchet algorithm is in the public domain [25], it can be used in different applications to improve security. Should interoperability be desired, an open standard makes it much easier to achieve it. It also allows for the code to be checked and improved, should a flaw be found. The algorithm can be adjusted and optimized for specific uses.

2.2.7.2 Weaknesses

No inherent authentication: The Double Ratchet algorithm does not authenticate users by itself. If users are not authenticated when the root ratchet is initialized, a man-in-the-middle attack could be executed, where an attacker Eve will pretend to be Bob and establish a connection with Alice, decrypting all the messages before forwarding them to Bob, who thinks that he is directly communicating with Alice. This way, Eve is able to not only read the exchanged messages, but also manipulate them and forge additional ones. This is

solved in the Signal protocol by using the extended three-way Diffie-Hellman key exchange (X3DH) with pre-published public keys to authenticate the users, establish a shared secret and exchange the Diffie-Hellman public key used for the initialization of the root ratchet [24]. This is only possible if both have a direct connection or access to a central server. In the latter case, one of the parties can be offline.

Device compromise: In the case that the device of one of the parties is compromised and an attacker is able to read all values of the keys, now and in the future, they are able to derive all messages sent and received from this point forward. This could be counteracted by using the Trusted Platform Module (TPM) on a device to make the individual values of the app only readable to the app itself. Android does already encrypt the local storage of an app, but an attacker might attain root rights on the device to read this content as well. Should a device be compromised in such a way, it is recommended to discard it and the communication client. Replacing them is the most secure way to handle such an incident.

Hijacking: If an attacker obtains the current private Diffie-Hellman key of one of the parties and the public key of the other party, they can possibly derive the state of the first person's ratchets. Thus, they can start flawlessly impersonating that party and hijack the conversation. Depending on the exact state, they might not require the state of all ratchets.

Denial of service: A malicious sender could send messages with a very high message number, causing the recipient to think they missed a great amount of messages. This would lead to a large amount of calculation on their device and a lot of storage might be used for the keys. This is partially counteracted by defining a maximum number of messages that can be skipped at one time. Still, the attack can exploit this by sending many messages, each skipping over a number of messages which is smaller than the maximum value. Defining a total amount of skipped messages would counteract his, but in the case where a bad connection would lead to many messages being skipped over without any malicious intent, this would also trigger the counteracting mechanism. A sane solution is to use a maximum number of messages to skip at once while only storing a limited amount of skipped keys.

Initialization problems: If both Alice and Bob send the first message at the same time, initializing their ratchets as senders, they will not be able to establish a communication channel. While this might not be too common if both are simultaneously online, it becomes a serious consideration if one of them is offline when sending the message.

Meta-analysis: The headers of the messages are not encrypted. This means that some analysis can be done via the metadata (number of messages in the chain, number of messages in the previous chain, public Diffie-Hellman key of the sender). This can be counteracted by using the variant of the Double Ratchet that features header encryption, at the cost of added complexity. This variant is also described in the Double Ratchet algorithm specification [25]. Alternatively, the messages could be encrypted on another layer.

2.3 Scuttlebutt protocol

The Scuttlebutt protocol (SSB, also known as Secure Scuttlebutt) is used to establish a peer-to-peer (P2P) network based on the way humans interact. It uses identities in the form of public/private key pairs, of which the public key is used to identify a user. Each user has their own append-only log, dubbed "feed", which is signed with their key pair. SSB uses the LibSodium crypto library for its cryptographic primitives. The following is heavily based on the Scuttlebutt protocol guide [17]. Our description aims to explain enough of the protocol for the reader to be able to understand our thesis.

2.3.1 Identities

In SSB, each user has one or more identities. Each of these identities is represented by a public/private keypair, currently in the Ed25519 primitive. They can be used to calculate shared secrets with other people, to sign and decrypt messages. Should the private key ever be compromised, an attacker can use the knowledge of said key to impersonate the user. Thus, its secrecy is of utmost importance. A feed is typically associated with a public key of such an identity.

2.3.2 Feeds

A feed is the append-only log associated with a certain identity. Only the user owning this identity is able to add to the feed since each message in the log has to be signed with their secret key. The feed contains a list of messages, which can be used for any kind of application (see Fig. 2.13). Each entry in the list contains the content, its number in the feed, a timestamp, the public key of the message's author (author ID), the ID of the previous message and a signature proving that the message was posted by the owner of the feed. These things aim to make it impossible to alter a feed after the fact. A message can contain any kind of content, be it text, images or even programs. The applications using the Scuttlebutt protocol will decide on their own what to post and what to interact with. A message can also be encrypted content, which can only be decrypted by its recipients, without hinting at how many recipients there are or who they could be. The specific implementation of Scuttlebutt limits the number of recipients of one individual encrypted message to a maximum of seven people.

2.3.3 Peer-to-peer network

Any user will have their own feed, but they will also replicate other people's feeds on their own device. This is done via peer-to-peer connections. One peer will perform a special handshake (Section 2.3.4) with another peer to establish a connection. After this, they have established an encrypted connection with authentication, which they use can use for various things. A typical example of a request is to ask the other peer whether they have a replica of a certain feed, which they will then exchange, should it be available. This is how feeds typically propagate, from peer to peer. Each peer individually decides what feeds to store and which feeds to share and ask for.

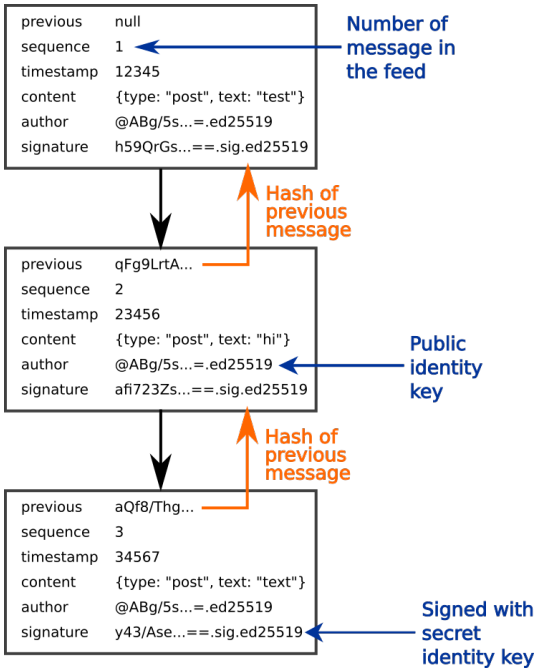

Figure 2.13: A list of messages in a Scuttlebutt feed.

Peers will broadcast their presence on a local network via UDP packets, while looking for the same kind of packets from other peers. Another way for peers to find one another is through the use of "Pubs", which are peers with public IP addresses which try to always be online. Connecting to a Pub means announcing your presence to all other people in said Pub. You will also learn who else is logged into the Pub. This way, a pub can somewhat resemble the role of what a centralized server would accomplish, by connecting peers with private IP addresses to one another. So-called "Rooms" are another way to tunnel connections from one peer to another with more privacy, but we will not go into more detail here since they are not relevant for our thesis. Should you wish to find out more about Rooms and Pubs, please visit the official Scuttlebutt Protocol guide [17].

2.3.4 The Handshake

The handshake that SSB uses to establish a connection from one peer to another has the interesting trait that no man-in-the-middle can learn the public key of either handshake participant from intercepting their messages. Among other things, it also verifies the peers' identities, provides forward secrecy and establishes a shared secret key to use for symmetric encryption of their traffic. It uses several cryptographic primitives and algorithms, among them authenticated encryption with associated data (AEAD), Diffie-Hellman key exchanges using the Curve25519 primitive and signatures using keys of the Ed25519 primitive. For this, it uses the LibSodium cryptographic library [7].

The peer initiating the handshake is known as the client, the contacted peer is referred to as the server. Before the handshake, the client must know the server's public key and the Scuttlebutt network identifier that they want to communicate on (typically the publicly known main Scuttlebutt network identifier). Should you wish to find out more about the

handshake, please refer to the official Scuttlebutt Protocol guide [17].

2.3.5 Analysis

SSB provides a nice way to build a P2P network securely. However, it still suffers from some drawbacks compared to using a centralized infrastructure.

2.3.5.1 Strengths

Decentralized: The network requires no central server to forward the messages from one device to another, relying on the gossiping aspect of transferring a feed from one user to another via various peers, should they not be able to establish a direct connection. In the event of an internet outage, an alternative route can be chosen by transferring the feeds from device to device using local networks. This property is also helpful in regions where the internet infrastructure is limited, intermittent or nonexistent.

Meeting points: For most networks, people will use private IP addresses. This makes it hard to communicate with one another if they are not in a common network. Pubs and rooms, with their public IP addresses, can be used as "meeting points" to establish a connection or to tunnel one through them.

Redundancy: Should the device of a person break, its feed is not lost if it has been replicated by other peers.

Detectable changes: A feed is signed by its owner and replicated by other peers. The owner can technically try to alter their feed by replacing a message with another. This is referred to as a fork. If the feed has been previously replicated, the people in possession of the replica can detect the fork. An example of such a scenario is depicted in Fig. 2.14.

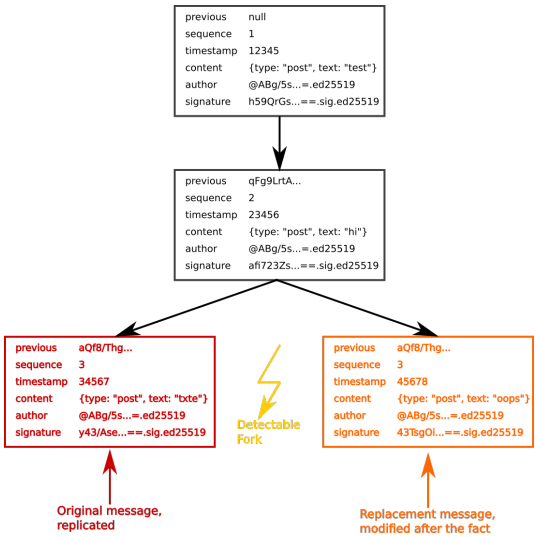

Figure 2.14: Should a message be modified after it has been replicated, this can be detected by peers.

Privacy: The exchange of private messages is supported, even for multiple recipients. While the protocol itself does not hide the fact that a private message has been sent *from* a certain user, it does not disclose who the *recipients* are, how many of them there are and what the content of the message is.

Censorship resistance: Not relying on a central infrastructure and being able to post private messages without disclosing the receiver make this network very resistant to censorship.

Authentication: The signature on each posts makes it impossible for other people to forge posts in someone else's feed when sending it to other peers. This way, everyone can be sure where the message originated from.

Openness: Since Scuttlebutt is an open protocol which does not make any assumptions on what kind of applications build upon it, it can be used for a great variety of purposes.

2.3.5.2 Weaknesses

Storage space: Having to replicate all the feeds means that a lot of storage might be used on one's device.

Availability: No central server means that availability of other peers is limited and transfer of messages can take quite long.

Difficult compromise recovery: An attacker compromising a key will be able to perfectly imitate the user. The user will have to generate a new identity and tell their peers out of band that their old ID was compromised.

No central identity verification: No central authority can check and verify that a user is who they claim to be. Out-of-band verification is needed to make sure no imposter or man-in-the-middle is acting.

No moderation: Since nobody restricts who uses Scuttlebutt and what is posted, there is no moderation to what is posted on the network. Instead, users typically decide to ignore other users in the applications SSB is used with.

2.4 Tremola

The Android app Tremola is a messenger based on the Scuttlebutt protocol. It uses JavaScript in the frontend and Kotlin in the backend [29]. The following description is based on the state of the app before we started this thesis.

2.4.1 Frontend

Tremola's user interface is based on Android `WebView`, which renders web pages. Thus, it can display an HTML page with its CSS and run the corresponding JavaScript code to make the page interactive. The data of the frontend is saved in the browser storage of the `WebView`. This data includes chat histories, contacts, settings, etc.
The frontend is really just one HTML page which is modified by JavaScript according to the scenario the app is in. The three main scenarios of the app are `Chats`, `Contacts` and `Connex`. `Chats` is a list of the chats that the user has. The number of participants in a single chat can be between one and seven, always including the user. Upon clicking on a chat, the app switches to the `Posts` scenario, which is used to represent the chat history. It displays the title of the chat, the participants, the past messages and a field at the bottom to enter a new message, which can be sent by clicking the button next to it.

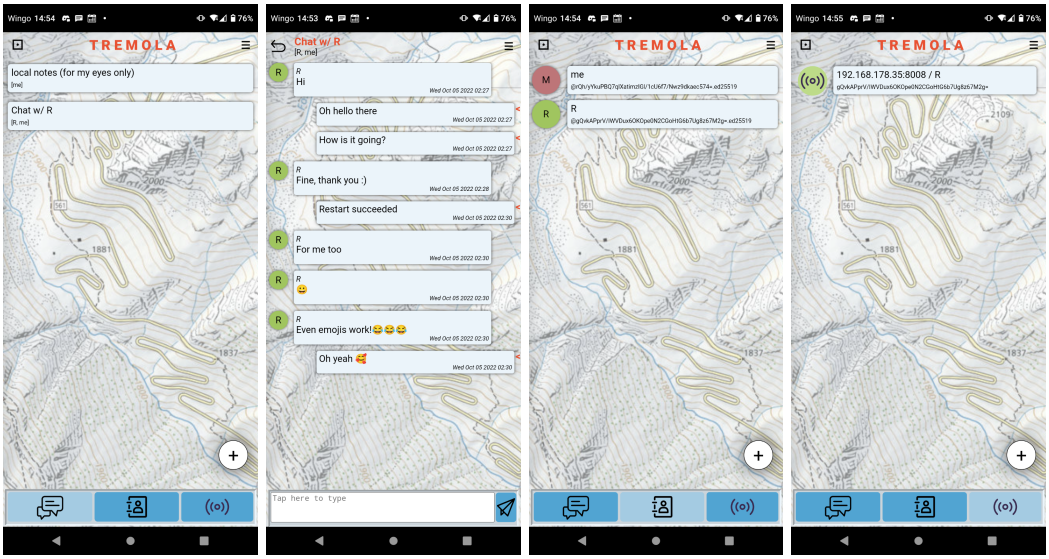

Figure 2.15: Some of the scenarios in Tremola. From left to right: `Chats`, `Posts`, `Contacts`, `Connex`.

The main scenario `Contacts` has a list of contacts that were saved. They each have an alias (chosen by the user), the corresponding SSB public key and a shortname. The shortname is a Base32 encoded string based on the hash of the SSB public key and therefore cannot be changed. New contacts can be added by entering their SSB public key, by scanning the QR code which encodes said public key or by starting a lookup with an entered shortname via the peers in the network. This will ask the available peers if they know the public key corresponding to the provided shortname. If they do, they will send the SSB public key, otherwise they will forward the message. Should no reply be received within a window of time, the lookup is considered failed.
The last main scenario, `Connex`, shows a list of peers which are on the same network as the user. They can also be added as contacts. The scenarios described here can be seen in Fig. 2.15.
Some other important functionalities of the app: A QR code can be displayed which represents the user's public key in Scuttlebutt. There is a settings menu which allows

for the customization of the app and also offers a panic button which calls the function `settings_wipe()`, which is responsible for deleting the data of the frontend and the backend in an emergency. There are of course further things the app can do and all functionalities could be explained in further detail, but we will try to keep it brief here.
Certain actions in the frontend will call functions in the backend via the `backend()` function, which will send the provided command string to the backend, where it will lead to functions being executed depending on what the command string contains exactly.

2.4.2 Backend

The backend is programmed in Kotlin. It receives commands from the frontend via the `WebAppInterface` class, which handles them and calls further backend code. The backend cannot directly send data back to the frontend, but it can execute JavaScript code in the `WebView` to alter its state.
The backend contains databases for the contacts, the feeds of the user and peers and pubs (pubs are currently non-functional in Tremola). It also stores the SSB key pair used as the identity. Tremola's backend is also responsible for the entire network operations and using the Scuttlebutt protocol. As an added bonus, it contains the lookup protocol which will try to resolve the public key belonging to a shortname by asking peers.

2.4.3 Analysis

The Tremola app offers a nice solution by implementing a messenger for Android on top of the Scuttlebutt protocol. But this rather unique approach also features some drawbacks. For Tremola's security vulnerabilities, see Appendix A.

2.4.3.1 Strengths

Scuttlebutt: The app uses Scuttlebutt and is therefore able to profit from many of its beneficial properties. For further information, please see Section 2.3.

Openness: Being open-source software, people can easily add improvements to the app as well as verifying its security themselves.

Encrypted storage: Since the app stores its data on Android, all storage of apps is encrypted and only available to the apps they belong to. Other apps have no access to the data in Tremola. However, should an attacker attain sufficient rights, they could still read it.

2.4.3.2 Weaknesses

Scuttlebutt: Using Scuttlebutt comes with its own drawbacks, see Section 2.3.

No plausible deniability: Since all messages are signed, Tremola offers no plausible deniability. Each message must have been sent by a person that has knowledge of the

private key of the sender, which is in most cases only the sender.

Single point of failure: Compromising the secret key of the Scuttlebutt identity would mean that all sent and received messages can be decrypted. No forward secrecy is offered. Additionally, an attacker would be able to decrypt all future messages and impersonate the identity they compromised.

Usability: Currently, the only way that peers can talk to each other is via local Wi-Fi. The app is designed in a way that adding more communication channels is easy, but as of now no others have been implemented.
All in all, the app is lacking many features other messaging apps such as Signal or Threema offer. Tremola seems to be more of a proof of concept at the moment, rather than a fully fledged messenger. Of course, Tremola has way fewer resources for its development, so its current state is understandable.

No blocking feature: Since there is no moderation on Scuttlebutt, people could send unsolicited messages. Users cannot currently be blocked in the app, but they can be forgotten. This would still lead to the recipient storing the messages, however.

3
Implementation

This chapter concerns itself with the work we did to implement the Double Ratchet algorithm into the Tremola app. We will first take a look at why the implementation is non-trivial. After this, we will take a look at how the app worked before our thesis. Then we will explain most of the benefits and downsides of using the Double Ratchet algorithm in Tremola. This is followed by a chapter explaining our solution. Lastly, we will measure the performance of certain aspects of the Double Ratchet algorithm to determine its impact on the user experience.

3.1 Challenges

The Double Ratchet algorithm and the Signal protocol in general were designed for use with a centralized server that is always online. Should one person be offline, the server would hold messages that are addressed to them and deliver all of them once that person comes back online. The server would then delete all information about the delivered messages it had, to increase the security of the communication. In the Signal protocol, that server will also hold the prekeys central to the initialization of the Double Ratchets.
With Scuttlebutt, no such central server exists. One peer sending a message to another might not even have seen the recipient's feed at the time of sending. There is no guarantee of anybody being online at any one time. Tremola already achieves a way to communicate securely via the Scuttlebutt network, but the leak of an identity key would jeopardize the secrecy of all the sent and received messages.
In the scenario of a messenger using a centralized server, it is no trivial matter to capture every message sent and received by a person without access to the server. In Tremola however, all messages are replicated to multiple devices across append-only logs with immutable messages. Thus, an attacker can easily reconstruct a stream of all possible sent and received messages, although they are still be encrypted. Since it is so easy for an attacker to get a hold of the encrypted messages, it is very important that they are encrypted securely.
Our goal was to introduce forward secrecy to the messages sent by Tremola. This was to be achieved via implementing the Double Ratchet algorithm adopted specifically to the Tremola use case. Since some of the preconditions that the Double Ratchet algorithms assumes are

not met over a peer-to-peer network, we had to adapt it in certain ways. The final solution turned out to be rather simplistic, but a lot of thought was put into selecting the right one. Many alternatives were considered, which will be described further on in this chapter.

3.2 Point of departure

The existing app has the user enter a message in the frontend. Upon confirming the message, it calls the `backend()` function, which passes a string to the backend's `WebAppInterface`. It contains the following arguments: the command word, the entered message encoded in Base64 and all the recipient's Scuttlebutt public keys, including the sender's. The `WebAppInterface`'s function `onFrontendRequest()` receives the string as its parameter, splits it and has a `switch` (`when` in Kotlin) case which executes different code depending on what the command word is. In this case, the command word is "`priv:post`". It then decodes the message and adds a private message with the text as its content to the sender's feed. The recipients also include the sender. The feed entry is encrypted without disclosing any of its recipients, but can be decrypted with the help of any of the recipient's private key. Upon successful decryption on a recipient's device, they send it to their `WebAppInterface`, where the `sendEventToFrontend()` function is called with the `event` object to send as its parameter. This in turn uses an `eval()` statement to execute the `b2f_new_event()` function in the frontend with the `event` object converted to a stringified JSON object. The frontend then persists the event in the `tremola` object, which is saved in browser storage. This adds a message to its respective chat. The communication between front- and backend can be seen in Fig. 3.1.

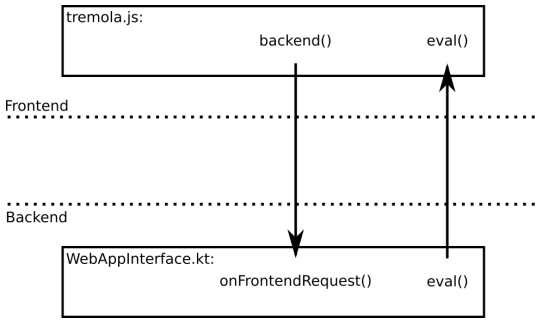

Figure 3.1: The frontend and backend of Tremola communicate with one another via certain function calls which pass messages or execute code.

3.3 Advantages and disadvantages of adding Double Ratchet algorithm

Here we discuss the benefits and downsides of using the Double Ratchet algorithm in the Tremola app. It also discusses which of the properties of the DRA we were not able to keep in our implementation and which additional benefits were granted by the app's design.
The Double Ratchet algorithm was added to Tremola without changing the UI, by using it as an additional layer of encryption in the backend. As discussed in Chapter 2, the Double Ratchet algorithm gives us the properties of forward secrecy, future secrecy.

Not all of the beneficial properties of the Double Ratchet algorithm were able to transfer over, however. The plausible deniability comes from the fact that all messages in the chat might be forged by the other party. There is no way for an outsider to conclusively determine who produced a message, even if they are given the decryption key. The communicating parties will know who produced the message by virtue of knowing which messages they did produce themselves. In SSB however, all messages are signed with your identity secret key, which makes it impossible for the other party to forge messages unless they know said key. On the other hand, this signature property allows for authentication with every message.
Adding the Double Ratchet algorithm to Tremola has its own drawbacks. For one, it adds complexity. Furthermore, having an extra step forcibly slows down the app. To determine by how much, we made some measurements in Section 3.6.
Should both parties send an initializing message while another one is on the way, the ratchets will be initialized improperly, leading to all further communication to fail (see Section 3.4.2).

3.4 Our solution

Here we will cover how we implemented the Double Ratchet in the Tremola app. We will go through it in the chronological order that we worked on it, meaning from the general implementation of the Double Ratchet to the more specific steps of tying it into the Tremola app.

3.4.1 Double Ratchet

We implemented the Double Ratchet algorithm as a Kotlin class called `DoubleRatchet`. It is a fairly default implementation of the Double Ratchet algorithm as described in the specification [25]. These are the things a user of the API should know: It needs to be initialized with a `sharedSecret` key and – depending on whether you are the sender or the recipient of the first message – with the Diffie-Hellman public key or the entire Diffie-Hellman key pair of the recipient. From that point on, messages can be encrypted and decrypted with the functions `encryptString()`, `decryptString()`, `encryptMessage()` and `decryptMessage()` (see Fig. 3.2). The `encryptString()` function will only take a single argument string which represents the plaintext to encrypt. The `encryptMessage()` function will take an `associatedData` parameter in addition to the plaintext. This associated data is signed, but not encrypted. Both of these functions return a stringified JSON object, containing the stringified JSON objects `encodedEncryptedMessage`, which contains the nonce and the ciphertext of the plaintext, Base64 encoded, and `header`, which contains the current public Diffie-Hellman key, the length of the previous sending chain and the number of the message in the current sending chain. The `decryptString()` function takes a `header` stringified JSON object and an `encodedEncryptedMessage` (also a stringified JSON object). The `decryptMessage()` function takes the same input, but an `associatedData` string as well. Both decryption functions return the decrypted plaintext. Should a decryption attempt fail, any changes made to the state will be reset to how it was before. This part is important, since the app might try to decrypt a message that has been

corrupted or try to decrypt a message it was not meant to decrypt. A `DoubleRatchet` object can be serialized with the `serialize()` method, which returns a stringified JSON object. An object can also be constructed from such a stringified JSON object. This is useful for storing the object to a file and reconstructing it after a restart of the app, for example.

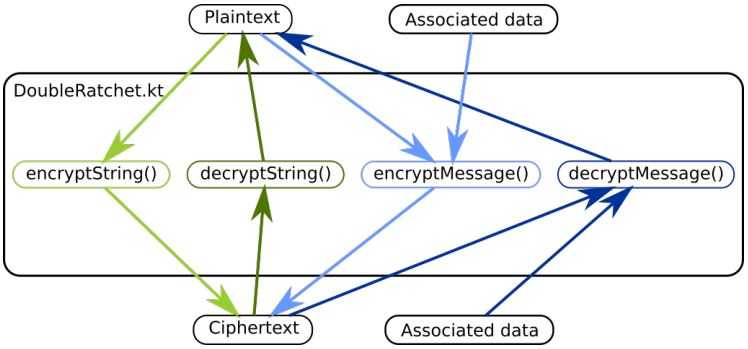

Figure 3.2: This figure depicts the way the encryption and decryption functions can be used. Please note that the associated data must be the same when encrypting and decrypting, otherwise the authentication will fail.

We used the LazySodium cryptographic library, which is an interface to the LibSodium library. LibSodium was chosen because Scuttlebutt also uses it and thus it would enable easy interoperability. For encryption, the official Double Ratchet algorithm specification recommends "an AEAD encryption scheme based on either SIV or a composition of CBC with HMAC" [25]. However, multiple vulnerabilities have been discovered in the past using the cipher block chaining (CBC) mode (see Padding Oracle attack [4]). In addition to this, Microsoft recently published an article about a vulnerability in CBC-mode, stating "Microsoft believes that it's no longer safe to decrypt data encrypted with the Cipher-Block-Chaining (CBC) mode of symmetric encryption when verifiable padding has been applied without first ensuring the integrity of the ciphertext, except for very specific circumstances." [8] While we do authenticate the received ciphertext before decrypting, we believe it is in our best interest to utilize other, more highly regarded AEAD encryption schemes not relying on CBC-mode. Thus, we used the XChaCha20-Poly1305-IETF cipher since it was recommended by the official LibSodium documentation [7] and is also included out of the box. For the Diffie-Hellman keys, we used the Curve25519 primitive, since it is also used in Scuttlebutt and recommended by the official specification of the Double Ratchet algorithm. The root ratchet KDF is recommended to be implemented with the HMAC-based extract-and-expand key derivation function (HKDF) construction [3]. Unfortunately, the LibSodium library does not implement a HKDF in the way the Signal protocol describes it (although it does include a Key Derivation API in a more general sense). Thus, we had to implement one according to the paper using SHA-512. We are aware that it is seldom recommended to implement one's own cryptographic functions, since it is very easy for a non-expert to inadvertently introduce a vulnerability. If possible, somebody with more advanced knowledge of cryptography should look through the code and thoroughly test it. Or even better, use an existing, trusted library providing the same functionality. The chain KDF is implemented

using HMAC [2] as recommended by the Double Ratchet specification, with SHA-512 as the hash function primitive.
Most of the private functions and internal workings are based upon the Python implementation code in the specification document [25]. There are some changes however: For example, the `resetToOldValues()` function which is used to reset the Double Ratchet's internal values after an unsuccessful decryption is not specified in the code, but it is mentioned in the description. Also, the Python code would often pass return values as tuples. Kotlin does not support the same functionality. We decided to use stringified JSON objects to pass between functions in these cases.

3.4.2 Initializing the Double Ratchet with Scuttlebutt

The `DoubleRatchet` class does not provide support specifically for the Scuttlebutt use case. Instead, it is a generalist implementation which could be used for many different cases. The Double Ratchet algorithm requires initialization with a shared secret and either the other device's public Diffie-Hellman key if you are the sender or your own Diffie-Hellman key pair if you are the recipient of the first message. The Signal protocol uses the extended triple Diffie-Hellman key agreement protocol (X3DH) [22] to decide on these parameters. For the Scuttlebutt network, there is no central server and a user might not have even seen the feed of the person they want to contact. They will, however, know the public key of the person they want to reach. Thus we use this key for the initialization. While the X3DH certainly is the more secure option, it is not always feasible over the SSB network. Our approach is slightly less secure, but more practicable. In Appendix A, we discuss this fact in more detail.
Let us look at how it works with our familiar actors, Alice and Bob. As the sender, Alice utilizes her own secret identity key with the public identity key of Bob to produce a shared secret. The keys are in Ed25519 (used for signing), so she transforms both of them to Curve25519 (used for elliptic-curve cryptography). Then, she can use scalar multiplication to derive the shared secret from the two. Then she uses Bob's public key, transformed to Curve25519, as the public Diffie-Hellman key. Thus, Alice has all the parameters required to initialize her Double Ratchet. Once the encrypted message reaches Bob, he will know Alice's public identity key. He can then transform his key pair and Alice's public key from Ed25519 to Curve25519. With her transformed public key and his own transformed secret key, he can take the scalar product to produce the same shared secret that Alice derived. For the Diffie-Hellman key pair, he simply uses his own transformed key pair. Thus the two can successfully initialize their Double Ratchets and use it normally. This can be seen in Fig. 3.3.
This method has the advantage of enabling communication with only the knowledge of the recipient's public identity key, nothing else. However, there is a slight drawback to this method: Should either Alice's or Bob's identity key pair be compromised, the attacker can read the first few messages that Alice sent, given that she initialized the conversation. After the Diffie-Hellman key exchanges happened, no further messages can be read. An attacker could exploit this fact by causing the parties to re-initialize their Double Ratchets over and

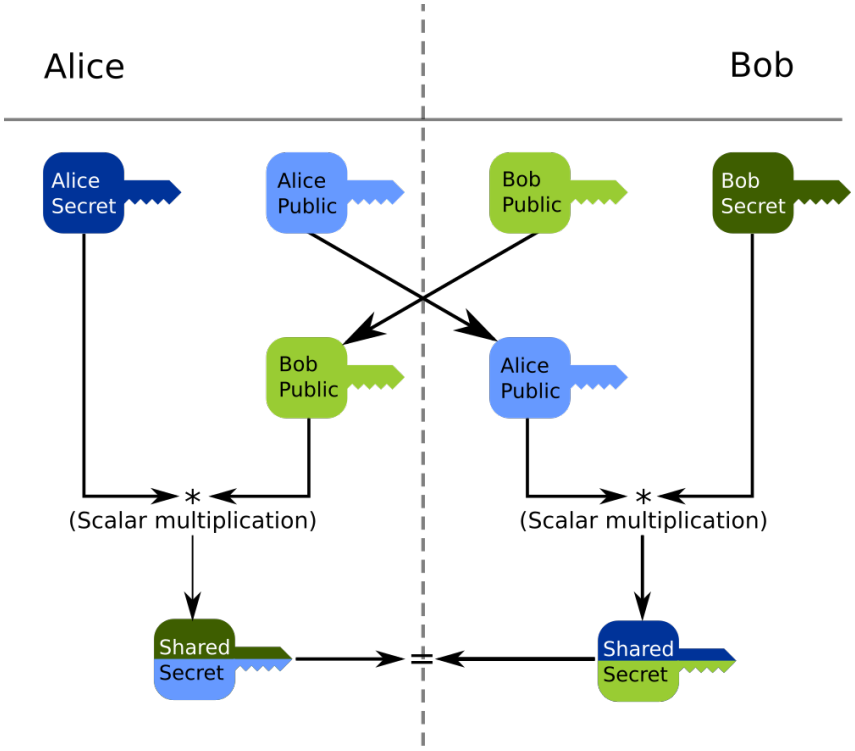

Figure 3.3: The public/private key pairs in Scuttlebutt are Ed25519 keys, which can easily be transformed to Curve25519 keys. With their own key pair and the knowledge of the correspondent's public key, Alice and Bob can derive a shared secret using scalar multiplication. This is interchangeable with an elliptic curve-based Diffie-Hellman key exchange.

over again to always read the first few messages.

Our current implementation does not allow the re-initialization of the ratchets, although it might become a consideration when trying to fix the problem of both parties initializing their Double Ratchets as senders. This only happens when both send a message without having received the other's message and without knowing one is underway. This case is rare, but realistic in scenarios where the feeds propagate very slowly (remote regions without a lot of networks). This problem is not solved by the original Double Ratchet algorithm. None of our attempts to solve it have been completely satisfactory. Our most promising option was to reset the Double Ratchet of the user with the lower public SSB identity key and initialize it as a receiving ratchet instead. This would cause some sent messages to be lost and the user should be notified. Great care is to be taken when implementing this, to avoid giving an attacker who has compromised a long-term private key the possibility of constant re-initialization.

3.4.3 SSBDoubleRatchet

The `SSBDoubleRatchet` class is a subclass of `DoubleRatchet` and is intended for use with the Scuttlebutt protocol. It features documentation in its constructors to make it more easily usable in conjunction with SSB. Additionally, it has publicly exposed functions that facilitate using it with the SSB identity keys and key pairs. The Tremola App saves

the SSB identities in an `SSBid` object. The `ssbIDToCurve()` function will take such an identity key pair object and return a Curve25519 key pair to use for initialization. The `publicEDKeyToCurve()` takes a public identity key and returns its Curve25519 counterpart. The `calculateSharedSecretEd()` and `calculateSharedSecretCurve()` take the user's `SSBid` object and the other user's public key (in Ed25519 and Curve25519, respectively) and output the shared secret key. These values can then easily be used to initialize the `SSBDoubleRatchet`.

3.4.4 Double Ratchet list

The separate Double Ratchets have to be associated with their respective chats. They should be able to be stored and retrieved. For this, we need another data structure. The resulting class is called `DoubleRatchetList`. Its only field, `list`, is a `HashMap` with strings as keys and `SSBDoubleRatchets` as its values. The strings are generated in the same way that the frontend generates the chat names with the function `recps2nm()` from `tremola.js`. It is given an array of strings, each containing a person's public SSB key, each formatted like this: "`@ABx...d5Q=.ed25519`". The function takes the array, sorts it, then joins all of them together into a single string and removes the "`.ed25519`" character sequence at the end of each ID. Thus, a unique string for every list of recipients can be derived, no matter the order of the list. This functionality is implemented in the `DoubleRatchetList` in the function `deriveChatName()`. The output of this function is used as the key in the list. Modifying the list can easily be done with the expected `get()` and `set()` functions, which also enable the array-style indexing.
For persistence, three methods are central: `serialize()`, `persist()` and `deserializeList()`. `serialize()` returns the stringified JSON object of the `DoubleRatchetList` object, `persist()` saves the current state to a file by using `serialize()` to generate the string and `deserializeList()` reads the file on the disk to reconstruct the saved `DoubleRatchetList`.
The `persist()` function has to be called before the app closes. Otherwise, a state might be have been saved to disk with different Diffie-Hellman keys than the ones which are currently used in chats. This might lead to unsuccessful decryption in all affected chats with no hope of recovery. Only re-initializing the `DoubleRatchet` could fix this problem. This is not advised, due to reasons discussed in Section 3.4.2. Instead, we chose to persist the `DoubleRatchetList` after each encryption and decryption, since its state might have changed. This is surely not the most efficient way to do it, but it is one of the most secure options regarding data loss.

3.4.5 Testing

Since the systems used for the Double Ratchet algorithm are rather complex, we needed a way to verify it works the way we expect it to. Thus, we created a test suite for each of the classes `DoubleRatchet`, `SSBDoubleRatchet` and `DoubleRatchetList`: `DoubleRatchetLocalTest.kt`, `SSBDoubleRatchetLocalTest.kt` and `DoubleRatchetListLocalTest.kt`. The unit tests are not run on the Android device, but

instead on the computer used for development. This required us to mock some Android specific libraries such as `Log` and the functions used to write to and read from files. We also had to use the `LazySodiumJava` object instead of the `LazySodiumAndroid` object for our cryptographic functions. The resulting tests are able to test the logic of the classes rather quickly since they are run on the development machine. Running them on the Android device takes a considerable amount of time. The local unit tests significantly sped up our debugging efforts, which were plentiful to say the least.
Of course, we also had to test the app on an actual Android device. For this, we created the tests `DoubleRatchetAndroidTest.kt`, `SSBDoubleRatchetAndroidTest.kt` and `DoubleRatchetListAndroidTest.kt`.
If you plan on improving the `DoubleRatchet` functionality on Tremola or using it for another purpose, it might be worth a look at the test classes, since they contain specific use cases of the `DoubleRatchet` functionality.

3.4.6 Adding it as a sandwich layer

Our goal was to improve the security without impacting the user experience. We also did not want to apply changes to the Scuttlebutt cryptographic stack, thus we chose to implement the `DoubleRatchet` capability as an in-between layer. It ended up becoming part of the Kotlin backend, but the Double Ratchet algorithm was used to encrypt and decrypt the content of the SSB message in the feed once more. This enables us to encrypt the message an additional time on top of the already pre-existing SSB private message encryption, increasing its security. It is specifically implemented in the `WebAppInterface` class. When a user sends a message, the message and its recipients are sent to the backend, to the `WebAppInterface`'s function `onFrontendRequest()` with the "`priv:post`" command as the first word in the command string. At this point, the previous version of the app used to create a message for the SSB feed and add it to it. Our implementation changes the content of the post here by looking up in the `DoubleRatchetList` whether we have a `SSBDoubleRatchet` for the chat with all of the recipients. If yes, use it to encrypt and then create the message for the feed. If there is none, it creates a new one. The Double Ratchet algorithm is currently only utilized in a two-person chat. The old implementation would only have to decrypt the message with its secret SSB key, since the sender is also one of the recipients, and then send the decrypted message to the frontend. Now there is a problem: Since we sent the message it can only be decrypted with the key we used to encrypt it with. However, we already discarded this key. Our implementation solves this problem by sending an unencrypted copy of each message back to the frontend. This approach aims to change as little as possible of the existing source code to avoid creating unexpected problems. A sender will still remain a recipient of their own messages, but will simply fail to decrypt their own message and thus ignore it.

3.5 Other solutions that were considered

We ultimately decided upon our solution by looking at different possible implementations and judging their advantages and disadvantages. We are going to list most of them here:

3.5.1 Alternatives regarding initialization

Having a complete Diffie-Hellman key exchange take place before sending the first message would improve the security of the first sending ratchet. Our current implementation simply uses the SSB public and private keys transformed to Curve25519 instead of real, random Diffie-Hellman keys. This makes the first sending ratchet deterministic, given the knowledge of one party's secret SSB key and the other party's public key. We deemed this version impractical, since it required the sender of the first encrypted message to have received the ethereal public Diffie-Hellman key of the recipient. With our use case of a remote region in mind, round trip times of a message could be considerable. Thus an extra round trip could take multiple days, delaying the sending considerably. However, it could be optionally enabled by a user in the settings of a chat to make it an "extra secure" one.
Always using prekeys instead of the public SSB key of the recipient of the first message would solve the problem of the extra full round trip (the request for an ephemeral public Diffie-Hellman key and the reply). Prekeys are pre-published Diffie-Hellman keys, signed by their creator. These prekeys could be implemented in a similar way to how the Signal protocol implements them [22], but instead of the server storing the prekeys, they would be stored and replicated on the feeds. Announcing already used prekeys could also be done on the feeds. This is once again a considerable increase in security for the first sending ratchet, although not quite to the amount of a unique ephemeral Diffie-Hellman key. It is slightly less secure because each prekey should only be used once, but two users might use the same key accidentally. One might use a certain prekey, prompting the recipient to label it "used" on their feed. The other user might be offline at the time of sending, using the same key while not knowing about the key having already been used.
This approach would not require the sender to send a message first, instead only requiring the knowledge of the recipient's feed at the time of sending. Nevertheless, we still decided against it. First, the sender might lack the knowledge of the recipient's feed by virtue of the potentially slow propagation. Second, the complexity and effort of adapting the X3DH to our use case would go beyond the length constraints of this thesis. It might be beneficial to add prekeys to Tremola, but still leaving the option of falling back to using the public SSB keys in case the recipient's feed is unknown.

3.5.2 Storing sent messages

Instead of sending an unencrypted message back to the frontend when posting a message on the feed, we could have kept the symmetric key that we used to simply decrypt the message on our feed. Alternatively, we could have a copy of our ratchets, which are one sending step behind to generate the key to decrypt the message we sent once we read it from our own feed. We deemed this to reduce the security. A second ratchet or storing the keys might offer an additional attack vector and would increase the exposure in the case of a compromise.

Instead of sending a not yet encrypted message from the `WebAppInterface` back to the frontend to add it to our chats, it might have been a better idea to simply saving each sent message only via the frontend. This should speed up the operation slightly. We did not implement this due to time constraints and since our solution did not modify anything in the frontend. We did not want to inadvertently introduce bugs into the app.

3.5.3 Using other cryptographic libraries

We used LibSodium's interface LazySodium as our main cryptographic library in the `Double-Ratchet`. However, we could have opted to use another library such as Bouncy Castle [1]. Additionally, we could have simply used the Signal protocol's Java implementation, to gain access to its HKDF function implementation [27]. We chose not to do this to guarantee the interoperability between Scuttlebutt's cryptographic functions and our implementation of the Double Ratchet algorithm.

3.5.4 Adding another scenario instead of creating a sandwich layer

We considered adding a separate "`SecretChats`" main scenario to the frontend. Only the chats made in this context would actually use the Double Ratchet algorithm. This could provide backwards compatibility for older clients since their less secure encryption mechanism is still supported as "normal" conversations in Tremola. This would have us add more to the frontend. We deemed the extra effort to create an additional scenario not worth it. We ended up adding additional security to all one-to-one chats without impacting the user experience at all.

3.5.5 Alternative methods of data transfer

Posting all messages to the feeds makes them easy for an attacker to capture and try to decrypt them at a later point in time. It is hard to miss a message this way. We did consider the possibility of adding an extra feature to the Tremola app to make it harder to track sent messages. This would still utilize the Scuttlebutt protocol's RPC to exchange messages, but they were stored locally in a hidden Tremola message log instead of the public feed. Messages would be passed in a more clandestine manner. We deemed this approach to not provide that many advantages to our approach, since the peer passing on the messages would not know who the recipients were and would pass on the messages to any peer they would also send regular feeds to. It would complicate matters, make it only possible for Tremola clients to exchange these messages from one to another without using the feeds of the users not using Tremola.

3.5.6 Double Ratchet algorithm for group chats

Implementing the Double Ratchet algorithm in group chats is no trivial matter. The arguably easiest way to implement it was to implement a group chat as multiple one-to-one chats, sending the same message to all recipients separately. There are multiple reasons for not doing it: First off, this could fill an entire separate thesis. Secondly, this simple imple-

mentation opens the way to attacks for a malicious user. One such attack is rather trivial: The malicious sender could send different messages to each recipient. The recipients have no way to know that they have received different messages [28]. Signal does counteract this by synchronizing the message keys among all group members [26]. There are also solutions like TreeKEM [15]. Implementing these is more desirable than implementing the original idea, but, once again, more complicated.

3.5.7 Storing message histories in backend

All the decrypted messages are currently stored in the frontend. We might want to store it in the backend instead for added security. While this solution is desirable, it is more convenient to leave it in the frontend. The modifications to the source code would be quite considerable.

3.5.8 Ignoring messages from non-contacts

Instead of looking at all the feeds you can find, we considered only looking at your contacts' feeds for messages. This would improve the performance, since there is less data to look through and also the security, since you will not handle any potentially malicious messages from other people. As another plus, you could limit the amount of spam you receive in the form of unsolicited messages. However, this makes it impossible for people that know your SSB ID to send you messages unless you also know theirs and have them saved as a contact. It would be a good idea to have this option in the settings if you know who will message you.

3.5.9 Disappearing messages

Disappearing messages is a feature which deletes a decrypted message on a device after a set amount of time. This would limit the amount of data leaked should the message log in the frontend be leaked. This would best be implemented as an alternative option for a chat, not as a default. We did not implement this since it is not in line with what we were setting out to do, namely implementing the Double Ratchet algorithm.

3.6 Measurements

To determine the performance impact of our changes, we chose to measure the time that certain steps take. Our measurements were taken on a Fairphone 4 with the tests from the class `TimingAndroidTest`.
Below you can find a table of the function, the average time one of these operations takes and the standard deviation for the measurements. Each operation was executed 1000 times and measured in nanoseconds. Our table displays the times in microseconds for easier reading.
You can reproduce all of these tests on your own by cloning the repository [29] and executing the respective tests yourself.
The standard deviations σ were calculated according to the following formula:

Functionality	Average [μs]	σ [μs]	σ / average [%]
unsuccessful key lookup in `DoubleRatchetList`	0.635	0.279	44
successful key lookup in `DoubleRatchetList`	0.677	0.211	31
serializing a `SSBDoubleRatchet`	31	25	82
deserializing a `SSBDoubleRatchet`	40	10	24
initializing a `DoubleRatchetList`	51	14	27
encrypting a message	85	31	37
successful decryption of a message	90	19	21
transforming an Ed25519 key to Curve25519	107	10	10
initializing a receiving ratchet	121	24	20
transforming an `SSBid` key pair to Curve25519	126	31	25
unsuccessful decryption of a message	161	37	23
calculating a shared secret with a Curve25519 key	211	15	7
setting a value in `DoubleRatchetList`	273	284	104
initializing a sending ratchet	279	50	18
calculating a shared secret with a Ed25519 key	317	32	10
persisting a `DoubleRatchetList`	324	78	24
successful decryption with 50 skipped messages	1490	574	39

Table 3.1: This table lists the average execution times of specific functionality of the app along with their standard deviations σ, both in microseconds. It also includes the ratio of the two measurements in percent.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (\mu - x_i)^2}{n}}$$

where n is the total number of measurements, x_i is the i-th measurement and μ is the mean of all measurements.

3.6.1 Analysis

From the data we can see that encrypting and decrypting a message only takes a short amount of time (below 100 μs). A unsuccessful decryption takes slightly more time at 161 μs. Under normal operation, a user will not notice the extra time it takes to encrypt and decrypt messages.

When looking at decrypting a message while skipping over 50 messages, we see a large increase however. The time it takes to generate all the keys and store them is substantial: it clocks in at almost 1.5 ms. But even in this case, the user will barely notice. However, this shows the potential for a denial-of-service (DoS) attack. While a malicious user might only spend a few microseconds on crafting a message which skips a large amount of keys, the recipient will spend potentially more than a hundredfold of the computation power on trying to decrypt said message.

Some of the other expensive operations include persisting the `DoubleRatchetList` and setting a value in it. Paradoxically, the average time it takes to execute the `set()` function is lower than the time it takes to persist the list, even though the `set()` function calls the `persist()` function when it is called. Logically, it should thus require more time. It seems the measurements are flawed. Considering the fact that both of these require I/O-interactions, it is not surprising that they are not very reliable. Especially the `set()`

function has a standard deviation larger than its average, hinting at a large amount of outliers.
The `get()` function, whether it succeeds or fails, takes only a minuscule amount of time. This is to be expected, since the primitive of the `DoubleRatchetList` is a `HashMap`, which can complete lookups in constant time.
Overall, we can see that the extra layer will not add any noticeable delay under normal operation. We also see that a DoS attack is possible, forcing the recipient to invest a substantial amount of computation time.

4
Conclusion

We achieved our goal of implementing the Double Ratchet algorithm in the Tremola app. All messages in one-to-one chats now have the property of forward secrecy and future secrecy. Group chats were left unmodified. Some adjustments had to be made to the DRA so that it can be used in conjunction with the Scuttlebutt protocol. Unfortunately, we were not able to provide plausible deniability to the messages sent with Tremola due to the nature of SSB. Also, should two people choose to send the initializing message at the same time, their ratchets would be initialized in a broken state, which renders communication impossible. On the flip side, all our implementation needs to start a chat is the recipient's public SSB identity key. The performance impact of the extra encryption is minimal.
The software we produced encompasses three primary classes: the general-purpose `Double-Ratchet.kt`, the Tremola-specific `SSBDoubleRatchet.kt` and the data structure for storing the ratchets: `DoubleRatchetList.kt`. Each of these features test classes to more easily find bugs when they are modified.
The user interface remains unchanged, which makes our changes easy to adopt for an existing user. They benefit from the added security without having to change any behaviour.
Our lightweight implementation enables users to communicate with one another easily and securely, even in regions where access to the internet is limited.

4.1 Future work

While we improved upon the existing Tremola app, there is still a lot that can be done. While most of the security and privacy improvements are listed in Appendix A, here we will list other ideas for future improvements:

X3DH: The Double Ratchet algorithm is not the only useful algorithm in the Signal protocol. Among other things, implementing the X3DH [22] in Tremola would further increase the security of the chats by making the initialization of the Double Ratchet unique. The prekeys could be stored to a user's feed and thus be propagated.

Secure group chats: While one-to-one chats had their security improved, group chats do currently not benefit from the implementation of the Double Ratchet algorithm. There exist different approaches to secure group chats, one of them being TreeKEM [15]. Another option is provided by the Signal protocol [24].

Nickname lookup: As the lookup is sill quite unwieldy, it offers no great benefit. If it is possible to transmit a pseudorandom 10 character sequence, it is most probably possible to transmit the entire SSB public key. Even if the shortname is not as long as the public key, the lookup can quite possibly fail. Directly using the public key would not fail in the same scenario, thus the benefit it offers is very much an edge case. If it were to use a more human-readable string for the lookup, say, a nickname, it would make the lookup more user-friendly. Collisions still would have to be dealt with somehow.

Further communication channels: The channels that the app can utilize should be expanded. For example, using Bluetooth and the internet instead of just detecting the peers on the same Wi-Fi network. If it could connect to the Basel Citizen Net (BACnet) [16], it would provide a way for the citizens of Basel to communicate in spite of an internet outage. Additionally, while the connection to SSB pubs is intended, it does not seem to be working as of now. Implementing this would increase the speed and distance of communication significantly.

Arbitrary group sizes: Implementing group chats of arbitrary sizes would increase the usability of the app. However, implementing it the way it is done currently would prove to be quite complex, since SSB only allows up to 7 recipients of a private message, which include the sender. Sending multiple private messages might be the way to go, but this would open the feed up to meta-analysis: Multiple private messages sent within a short period of time are probably group messages. Also, the amount of messages sent at a time will provide some information on how many participants are in a given chat. Countermeasures might include setting delays between sent messages and randomizing the number of recipients per message.

Saving sent messages in frontend: Whenever a message is sent, it is not directly saved in the frontend, but rather sent to the backend, which returns the message again as an event. This was done to limit our impact on the frontend. A more elegant but also more involved solution would be to adjust the frontend to directly save each sent message. This way, the backend would no longer have to send your own messages back to the frontend.

Add documentation: Some files still lack sufficient documentation. Should somebody have sufficient time on their hands, future developers would benefit from any additional documentation that is contributed.

UI overhaul: The general frontend of the app could be overhauled. Either by using a separate HTML document for each of the scenarios or by switching to Android's native UI

altogether. The latter would protect against most web vulnerabilities, but it might of course introduce vulnerabilities native to Android. However, we believe the latter poses a lower risk.

Vulnerabilities: In Appendix A, we discuss different vulnerabilities. A more extensive assessment would be desirable. However, the listed problems should be addressed eventually.

Code review: A general check of the logic of the app would be a great idea. Some bugs are already documented, but we are sure there are more to be found.

Bibliography

[1] Bouncy Castle documentation. https://www.bouncycastle.org/documentation.html. [Online; accessed 16-October-2022].

[2] HMAC: Keyed-Hashing for Message Authentication. RFC 2104, RFC Editor, Feb 1997. URL https://www.ietf.org/rfc/rfc2104.txt.

[3] HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, RFC Editor, Mar 2010. URL https://www.ietf.org/rfc/rfc5869.txt.

[4] Padding oracle attack. https://resources.infosecinstitute.com/topic/padding-oracle-attack-2/, Jul 2014. [Online; accessed 14-October-2022].

[5] Work with data more securely. https://developer.android.com/topic/security/data, 2022. [Online; accessed 13-October-2022].

[6] Security tips. https://developer.android.com/training/articles/security-tips, 2022. [Online; accessed 13-October-2022].

[7] Libsodium documentation. https://doc.libsodium.org/, 2022. [Online; accessed 14-October-2022].

[8] Timing vulnerabilities with CBC-mode symmetric decryption using padding. https://learn.microsoft.com/en-us/dotnet/standard/security/vulnerabilities-cbc-mode, Sep 2022. [Online; accessed 14-October-2022].

[9] Dieter Bohn. Google is rolling out end-to-end encryption for RCS in Android messages beta. https://www.theverge.com/2020/11/19/21574451/android-rcs-encryption-message-end-to-end-beta, Nov 2020. [Online; accessed 5-October-2022].

[10] Mike Butcher. Riot wants to be like Slack, but with the flexibility of an underlying open source platform. https://techcrunch.com/2016/09/19/riot-wants-to-be-like-slack-but-with-the-flexibility-of-an-underlying-open-source-platform/, Sep 2016. [Online; accessed 5-October-2022].

[11] K Cohn-Gordon, C Cremers, and L Garratt. On post-compromise security. IEEE, 2016.

[12] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A Formal Security Analysis of the Signal Messaging Protocol. Cryptology

ePrint Archive, Paper 2016/1013, 2016. URL https://eprint.iacr.org/2016/1013. https://eprint.iacr.org/2016/1013.

[13] Computerphile. Double Ratchet Messaging Encryption - Computerphile. https://www.youtube.com/watch?v=9sO2qdTci-s, Nov 2018. [Online; accessed 24-October-2022].

[14] GitHub Contributors. LazySodium Repository. https://github.com/terl/lazysodium-java, . [Online; accessed 13-October-2022].

[15] GitHub Contributors. TreeKEM for Group Key Management. https://github.com/bifurcation/treekem, Aug 2018. [Online; accessed 16-October-2022].

[16] GitHub contributors and University of Basel. Basel Citizen Net repository. https://.com/cn-uofbasel/BACnet. [Online; accessed 11-October-2022].

[17] Scuttlebutt Contributors. Scuttlebutt Protocol Guide. https://ssbc.github.io/scuttlebutt-protocol-guide/, . [Online; accessed 5-October-2022].

[18] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. doi: 10.1109/TIT.1976.1055638.

[19] The OWASP Foundation. Open Web Application Security Project. https://owasp.org/. [Online; accessed 11-October-2022].

[20] Andy Greenberg. You Can All Finally Encrypt Facebook Messenger, So Do It. https://www.wired.com/2016/10/facebook-completely-encrypted-messenger-update-now/, Oct 2016. [Online; accessed 5-October-2022].

[21] Moxie Marlinspike. WhatsApp's Signal Protocol integration is now complete. https://signal.org/blog/whatsapp-complete/, Apr 2016. [Online; accessed 5-October-2022].

[22] Moxie Marlinspike and Trevor Perrin. The X3DH Key Agreement Protocol. https://signal.org/docs/specifications/x3dh/, Nov 2016. [Online; accessed 14-October-2022].

[23] Inc. Meta Platforms. React. https://reactjs.org/, 2022. [Online; accessed 24-October-2022].

[24] Trevor Perrin and Moxie Marlinspike. The Signal Protocol Specification. https://signal.org/docs/. [Online; accessed 5-October-2022].

[25] Trevor Perrin and Moxie Marlinspike. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/, Nov 2016. [Online; accessed 5-October-2022].

[26] Trevor Perrin, Moxie Marlinspike, and GitHub Contributors. Signal's java implementation of the groupcipher. https://github.com/signalapp/libsignal-protocol-java/blob/master/java/src/main/java/org/whispersystems/libsignal/groups/GroupCipher.java, Nov 2019. [Online; accessed 16-October-2022].

[27] Trevor Perrin, Moxie Marlinspike, and GitHub Contributors. Signal's Java implementation of the HKDF. https://github.com/signalapp/libsignal-protocol-java/blob/master/java/src/main/java/org/whispersystems/libsignal/kdf/HKDF.java, Nov 2019. [Online; accessed 16-October-2022].

[28] Soatok. Threema: Three Strikes, You're Out. https://soatok.blog/2021/11/05/threema-three-strikes-youre-out/, Nov 2021. [Online; accessed 16-October-2022].

[29] Christian F. Tschudin, Etienne Mettaz, Cedrik Schimschar, and Lars Waldvogel. Tremola repository. https://github.com/cn-uofbasel/tremola. [Online; accessed 5-October-2022].

A
Security findings

During our work with the Tremola app, we discovered some potential vulnerabilities. Some of these were pre-existent while others were introduced over the course of out thesis. Here we aim to point out which kinds of vulnerabilities exist and how Tremola's security and privacy can be improved.

A.1 Metadata analysis

Using the encrypted header variant of the Double Ratchet algorithm [25] would disclose less information to an eavesdropper who can decrypt the SSB layer.
The length of a message can disclose some information on what it might contain. Splitting up long messages into shorter messages of a random length would make it more difficult to analyze, given that they are not all sent at the same time. A random delay between the messages would have to be introduced. Another option to consider would be to add random padding to a certain constant length where applicable.

A.2 Input validation

Introduce strict input validation on any data that can be altered or is provided by a user. This prevents most general attacks if done properly. For some guidance on this, refer to [6].

A.3 WebView

Since the frontend uses HTML and alters the content of it dynamically as well as enabling JavaScript, there is a possibility that cross-site scripting (XSS) could be exploited. Somebody should thoroughly check the source code to find any vulnerabilities. An alternative approach to avoid such web-based vulnerabilities altogether would be to implement the frontend in Android's default UI or using a dedicated library for building dynamic web pages, such as ReactJS [23].
The backend cannot return anything to the frontend, but it can execute statements via the `eval()` statements, which will interpret the string provided to it as JavaScript code and

execute it. Since the strings provided are built from some things which are user input, it might be possible for an attacker to get the frontend to execute arbitrary code (ACE). This should be thoroughly checked. In general, using any kind of `eval()` function is considered a bad security practice where it can be avoided. One other option would be to call fixed functions for each use case from the backend. Should this prove to be impossible for the problem at hand, very thorough and strict input checking has to be implemented to avoid code injection.
The OWASP Foundation offers details on all kinds of such attacks on their website [19].

A.4 Storing sent messages

Sent messages are encrypted by the Scuttlebutt protocol in the backend and addressed to up to 7 recipients, always including the sender. If the recipients were changed to not include the sender, a compromise of user's long-term secret would only permit the attacker to decrypt the messages the user receives. As it is now, an attacker can also decrypt the sent messages. Should the messages be from a one-to-one conversation, they are encrypted an additional time with the Double Ratchet algorithm. Group messages are not, however, and could thus be decrypted by such an attacker. As an additional benefit to removing the sender from the recipients, the group chats could include an additional user.

A.5 Cryptography

The Android Developers Security tips page [6] recommends to use `SecureRandom` to initialize `KeyGenerators` and use `KeyStore` to store them securely. This might be worth a consideration, since our implementation does not take any of these into account, instead using LazySodium's [14] default key generation methods. The keys are currently serialized to a file and stored internally. This does provide some security since Android does not make them accessible to other apps, but there are still methods for advanced attackers to retrieve these files.
Replace our own implementation of the HKDF [3] with one that has been properly tested for vulnerabilities and been implemented by experts. Alternatively, test our implementation thoroughly.

A.6 Message transfer

Since all messages are documented and replicated to different peers, they can all be retraced rather easily. All public keys of the Diffie-Hellman key exchanges are thus public, should somebody's SSB identity key pair be compromised. We suggest that should two peers be on a local network, they could exchange additional key pairs and messages via the RPC channel that SSB provides. These key pairs and messages would not be documented anywhere and could thus throw a very advanced attacker off.

A.7 Storage

When the panic button is pressed, data is mostly just unlinked. Using forensic analysis, a sufficiently advanced and equipped attacker could read the "deleted" values from disk or memory. Depending on the kind of memory and disk used, different measures have to be taken to avoid forensic analysis. Overwriting the data is imperative, but difficult on flash drives and SSDs. Low-level access to these devices is required on those kinds of storage media.

Chats with the option of disappearing messages could be a great way to further secure the frontend. Currently, all messages are stored in plaintext in the frontend. Of course, they are encrypted by the Android system, but should an attacker attain sufficient rights on the device, they could decrypt them. Thus, deleting and scrubbing messages after a certain amount of time has passed, say, 30 days or so, would limit the amount of information an attacker could obtain with such an attack.

Using a Trusted Platform Module to seal the keys that are used to decrypt the app's data would make it harder for an attacker to compromise the stored data. This does not offer perfect security, especially if an attacker has physical access to a device. Trusted Platform Modules have been reverse-engineered in the past and will in all likelihood continue to be cracked. Still, this would improve upon the current security.

In general, the Android Developer page "Work with data more securely" [5] offers advice on how to store sensitive files. This should be used to store the SSB identity keys, the Double Ratchets and other data that should not be disclosed. It might even be worth to think about encrypting the frontend `tremola` object, which contains all chat histories, among other things.

When a new public Diffie-Hellman key is received, the current implementation takes two steps to generate two new symmetric ratchets, the new receiving and the new sending ratchets. The creation of the new sending ratchet could be deferred until the first message is sent. This would reduce the time the state is in the system, increasing its security against getting leaked. This would make the system slightly more complex while adding some security.

A.8 Lookup

The lookup algorithm uses 10 characters of Base32 encoded data, which means the overall encoded bits are only 50. For a sufficiently funded attacker, producing a hash collision of such a size might be feasible, given enough time. This would lead to an attacker sending a forged public key in a response to a lookup request. This key would produce the same shortname, leading the person looking it up to believe that they found the person they were looking for, when in fact somebody is impersonating the recipient. Thus, no authentication can be guaranteed via the lookup.

The length of the shortnames could be adjusted to a suitable value that sufficiently protects against hash-collision attacks.

A.9 Authentication

Currently, every peer has to verify all other peers out-of-band to make sure they are who they claim. Implementing a web of transitive trust might make this process easier, increasing security while sacrificing a bit of privacy. Once Alice has physically verified Bob, she could post this fact on her feed, letting other people know that she guarantees for Bob's legitimacy regarding his public key. If Charlie has already verified Alice, he gains transitive trust of Bob as well. Of course, with each step that is taken, the security diminishes, but this is still better than no alternative at all. As a drawback, people publish with whom they have been in contact, leading to possible meta-analysis.

A.10 Feed corruption

Tremola allows the SSB identity to be exported. However, having the same key on multiple devices could lead to corrupt feeds (multiple feeds of the same author featuring different posts, contrary to the append-only log property). Also, being able to export the key can be seen as a security risk if the user does not know what they are doing.

A.11 Double Ratchet initialization

Instead of only using the SSB identities every time the root ratchet is initialized, people could post prekeys on their feeds to use for this purpose. Once used, users would post a message that a prekey has been used and generate new ones for their feed. The Signal protocol goes into more detail how it handles it in its centralized architecture [24]. This would require the sender to have seen the recipient's feed, however.

A.12 Skipped keys

Setting a maximum number of stored skipped message keys would protect against some denial-of-service (DoS) attacks on Tremola.

A.13 Group messages

Group chats currently rely on private messages from SSB without using the Double Ratchet algorithm. One simple approach to include the added security of the DRA in group messages would be to implement group chats as many one-to-one chats, each with their own Double Ratchets, broadcasting each message over all channels. This in itself would enable a malicious user to send different messages to different people and acting as if they all received the same one. There exist other approaches, which could be used, each with their own advantages and trade-offs. The Signal protocol has its own solution, which is worth taking a look at [24].

A.14 Logs

Removing the logging statements from the production app would increase the security of the application. Some apps with the "`READ_LOGS`" permission can read all the phone's logs, thus compromising all the data that is logged. The logs are deleted upon reboot, but can be read until then [6].

A.15 Code review

The work we did should be reviewed by an expert to check for any vulnerabilities we might have inadvertently introduced.

B

Glossary

Term	Explanation
ACE	See arbitrary code execution.
AEAD	See authenticated encryption with associated data.
Arbitrary code execution	An attack which enables an attacker to execute any kind of code on a device.
Authenticated encryption with associated data	A type of encryption whereby the ciphertext is authenticated together with some associated data.
Author ID	The public key of a feed owner in Scuttlebutt. Included in every message in the feed.
CBC	See cipher block chaining.
Chain KDF	See chain key derivation function.
Chain key derivation function	The key derivation function used by a symmetric ratchet.
Cipher block chaining	A mode of operation for block ciphers such as AES.
Code injection	An attack whereby a computer inadvertently interprets a user-supplied string as code to execute.
Cross-site scripting	An attack whereby malicious code is injected into an insecure web application, causing the browser to execute it. The web app usually alters the HTML dynamically in an insecure way.
Curve25519	An elliptic curve used in ECC. Most frequently used in DHKE.
Denial-of-service	See denial-of-service attack.
Denial-of-service attack	An attack which renders a device unavailable or unresponsive.
DHKE	See Diffie-Hellman key exchange.
Diffie-Hellman key exchange	Two parties can establish a common secret over a possibly wire-tapped channel using it. We use the elliptic-curve variant (Curve25519) in this thesis.
Diffie-Hellman ratchet	See root ratchet.
DoS	See denial-of-service attack.

Double Ratchet algorithm	An algorithm which allows two parties to encrypt and decrypt messages asynchronously, while using a distinct key for each message.
DRA	See Double Ratchet algorithm.
E2EE	See end-to-end encryption.
ECC	See elliptic curve cryptography.
Ed25519	Used for signatures. Uses EdDSA and Curve25519.
Elliptic curve cryptography	A domain in public key cryptography utilizing the mathematical primitive of elliptic curves.
End-to-end encryption	A form of encryption whereby only the end users (sender and recipient) are able to decrypt a message.
Extended triple Diffie-Hellman	A key agreement protocol used by the Signal protocol which enables secure, authenticated encryption even if the recipient of a message is offline.
Forward secrecy	A message is considered forward secret, if the disclosure of a future key does not compromise the current key.
Future secrecy	A message is considered future secret, if the disclosure of a current key does not compromise all future keys.
Identity key pair	In Scuttlebutt, a public/private key pair of Ed25519 keys is used to identify a user. It is also used for cryptographic purposes.
Injection attack	See code injection.
KDF	See key derivation function.
KDF chain	See key derivation function chain.
Key derivation function	A function taking a secret key and some input and produces a pseudorandom output.
Key derivation function chain	A chain of key derivation functions, using part of the output of a prior KDF as the key of the next KDF. Every KDF also produces a separate output key.
Long-term private identity key	See secret identity key.
Long-term public identity key	See public identity key.
Long-term secret identity key	See secret identity key.
Message chain	The list of messages produced by a symmetric ratchet.
Message key	The output of a symmetric ratchet. Used to encrypt or decrypt a single message.
Nonce	A number used once, used among other things to avoid creating the same ciphertext twice.
Post-compromise security	See future secrecy.

Prekeys	Pre-published keys used in the Signal protocol to enable users to send messages via an authenticated, secure channel even if the recipient is offline.
Pre-published keys	See prekeys.
Private identity key	See secret identity key.
Public identity key	A key of the identity key pair. It is used to verify signatures and encrypt messages to the user.
Ratchet	See key derivation function chain.
Ratchet key	The output of the root ratchet. Used to initialize a symmetric ratchet.
Receiving chain	The list of messages produced by a receiving ratchet.
Receiving ratchet	A symmetric ratchet used to produce message keys for received messages. Is the same ratchet as the senders sending ratchet.
Remote procedure call	Used in Scuttlebutt to transfer requests and replies from one device to another.
Root KDF	See root key derivation function.
Root key derivation function	The key derivation function used by the root ratchet.
Root ratchet	A ratchet utilizing the continuous Diffie-Hellman key exchanges between the parties to produce ratchet keys.
RPC	See remote procedure call.
Scalar multiplication	An operation in ECC. It allows two people to derive a shared secret key.
Scalar product	The result of a scalar multiplication. Usually a shared secret between two parties.
Scenario	A certain state of the Tremola UI. This might be a state where it displays a chat, the settings or similar.
Scuttlebutt	A peer-to-peer protocol which has its users post messages to their feeds (append-only logs) and exchange them.
Secret identity key	A key of the identity key pair. It is used to decrypt private messages sent to the user as well as to sign messages.
Secure Scuttlebutt	See Scuttlebutt.
Sending chain	The list of messages produced by a sending ratchet.
Sending ratchet	A symmetric ratchet used to produce message keys for sent messages. Is the same ratchet as the receiver's receiving ratchet.
Shortname	A short string used to identify an SSB public key in Tremola. Is based upon the hash of the key.
SSB	See Scuttlebutt.
SSB ID	See public identity key.
Symmetric ratchet	Each user has two symmetric ratchets, one for sent messages and one for received messages. Outputs message keys.

WebView	A UI element in Android responsible for rendering web pages.
X3DH	See extended triple Diffie-Hellman
XSS	See cross-site scripting.

Table B.1: A glossary of terms used in the thesis.

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis: Implementing the Double Ratchet algorithm in Tremola, a Scuttlebutt based messaging app for Android

Name Assessor: Prof. Dr. Christian Tschudin

Name Student: Lars Waldvogel

Matriculation No.: 17-056-243

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: Stein am Rhein, 31.10.2022 Student: L. Waldvogel

---

Will this work, or parts of it, be published?

- ○ No
- ◉ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: 31.10.2022

Place, Date: Stein am Rhein, 31.10.2022 Student: L. Waldvogel

Place, Date: _____ Assessor: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis