Analysing the Signal Protocol

A manual and automated analysis of the Signal Protocol

Dion van Dam

Analysing the Signal Protocol

A manual and automated analysis of the Signal Protocol

Dion van Dam

21 August 2019

Supervisors:
dr. V. Moonsamy *(Radboud University)*
ir. C.N.I.W. Schappin *(Deloitte Netherlands)*

Second reader:
dr.ir. E. Poll *(Radboud University)*

Radboud University

ABSTRACT

This thesis is the first to provide an automated analysis of the Signal Protocol. It is a protocol that is used by, among others, Signal and WhatsApp. Despite the extensive usage of this protocol, the amount of research done in this field is limited. We hope that this research will function as a stepping stone for further analyses on the implementation of the Signal Protocol.

The implementation of the Signal Protocol is not well documented and under-analysed. Although the authors of the Signal Protocol have recently written several documents to explain the protocol, it cannot be used as a specification as the documents are too high-level.

To address this problem, we will first analyse the implementation manually. We will also use other academic papers that have analysed the theoretical protocol to get a better understanding of the specification. Our contribution to academia is an implementation analysis and several models (state machines) of the protocol. We will use an existing tool to generate models from the implementation automatically. These models will give better a insight into the session management algorithm of the Signal Protocol. The advantage of using automated testing techniques is that new versions of the implementation can be tested while requiring little to no changes in the testing tool.

ACKNOWLEDGEMENTS

First of all, I want to thank Veelasha, my supervisor from the Radboud University, for all her feedback and to get me into contact with people involved in the Signal development and software verification. Secondly, I want to thank Colin, my supervisor from Deloitte, for the feedback and useful discussions; and of course thanks for the fun week at the summer school. Also, thank you Erik to be the second reader.

Furthermore, I want to thank Deloitte for allowing me to write my thesis at the company. I also want to thank the Radboud University and the teachers for providing this excellent master programme. Besides, thank you Arris, Dennis and Henrie for taking the time to read and discuss my thesis and help me improve it. Thanks Karthikeyan Bhargavan, Jan Rubín and Trevor Perrin to share your knowledge of the Signal Protocol. Thanks to all the people I have met at the Cyber Risk department, it was great fun!

Thanks to my friends for the much-needed breaks. A big thanks to my parents and sister for the great support over the years. They were always there for me when I needed it. Last but not least, I want to thank my girlfriend, 斌. She supported me in doing my master's, even though this meant that during this time we could not be together. Thank you for your patience and support during hard times.

CONTENTS

CHAPTER
ONE
INTRODUCTION

The Signal Protocol is a cryptographic protocol that secures the conversations of over 1.5 billion people [6]. This and similar protocols, such as OTR and OMEMO, are fundamental to support the right to privacy and freedom of speech as described in Article 12 and 19 of the Universal Declaration of Human Rights, *"no one shall be subjected to arbitrary interference with his privacy"* and *"everyone has the right to freedom of opinion and expression"* respectively [34]. These rights are often not seen as absolute rights and may be restricted for national security. An example of recent tension is the new legislation in India, which requires companies to screen messages of users to check on unlawful behaviour. This legislation makes applications that implement the Signal Protocol unlawful. When the legislation is implemented, it would mean that more than 200 million users[1] could not secure their conversations any longer [36].

Whereas other research has been conducted on the theoretical specification, this research is the first that will use automated testing techniques on the implementation. Research on the implementation is essential, as an implementation could have bugs or wrong interpretations of the specification. Therefore, even when the specification has been formally proven to be correct, it does not necessarily mean that the implementation is secure as well. Hence, this project will test how the protocol is implemented.

1.1 History of Signal

The Signal Protocol is a protocol developed by Open Whisper Systems in 2013 [9]. At that time the protocol was still known as the TextSecure Protocol. The protocol provides end-to-end encryption and is used by Signal[2] and WhatsApp[3] [37]. Facebook Messenger[4] uses the Signal Protocol only for secret conversations [10].

Signal was first known as a proprietary application called TextSecure, which was used for sending encrypted text messages in 2010 and was developed by Moxie Marlinspike and Stuart Anderson of Whisper Systems [14, 16]. The start-up company also produced an app for encrypted voice calls called RedPhone [8]. Whisper Systems was acquired by Twitter in 2011 and made TextSecure open-source a month after the acquisition [14]. RedPhone was shut down by Twitter in the same period [13] but was rereleased in July 2012 as open-source project [39].

[1]The amount of WhatsApp users in India [36]
[2]An open-source messaging application developed by Open Whisper Systems https://signal.org/
[3]A proprietary messaging application owned by Facebook https://www.whatsapp.com/
[4]A proprietary messaging application owned by Facebook https://www.messenger.com/

In 2013 Moxie Marlinspike created Open Whisper Systems to develop TextSecure further [20]. The first version of the Signal application was released in 2014 for iOS, called Signal, and in 2015 they renamed the Android version from TextSecure to Signal [23, 24]. In 2016 the protocol (TextSecure Protocol) was renamed to the Signal Protocol for consistency.

Figure 1.1 shows the development of Signal. The figure shows how it started as two different projects, RedPhone and TextSecure. TextSecure has been redeveloped as instant messaging application and later renamed to Signal. The figure also shows that the old TextSecure application has been further developed as SMSSecure. Moreover, the figure shows that RedPhone has been merged into Signal as (video) calling feature.

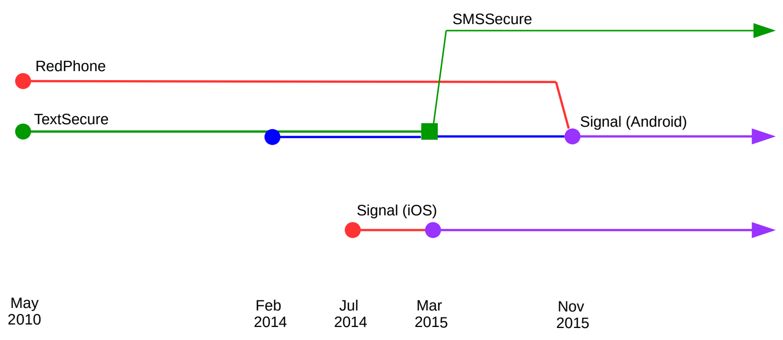


Figure 1.1: Timeline of the Signal development[5]

1.2 Properties in Security Messaging Protocols

Traditionally there are three essential concepts in information security, namely *confidentiality*, *integrity* and *authenticity* (CIA). CIA is often complemented with *non-repudiation*. For example, pretty good privacy (PGP) offers *confidentiality* by encryption, *integrity* and *non-repudiation* by signatures and *authenticity* when one knows the public key belongs to that particular person. However, these properties are not enough. One would also want that when one key is comprised, only one message is compromised and not preceding and succeeding messages — called *forward secrecy* and *future secrecy* respectively. Furthermore, *repudiation* (also known as *cryptographic deniability*) is also a crucial property for private messaging as one may not want to be held responsible for private conversations. These properties are further explained in chapter 2.

1.3 Research Questions

This thesis will address the following questions:

1. What are the properties of the Signal Protocol and are these still valid in the implementation?

2. How can the high-level protocol be modelled such that it gives a better understanding of the implementation?

[5]https://en.wikipedia.org/wiki/Signal_(software)#/media/File:Signal_timeline.svg

1.4 Research Scope

Part of this project's scope is to research the security guarantees of the protocol. This includes checking if the guarantees hold under all conditions. Additionally, part of the scope is to find out if the protocol is implemented as it should — which is hard as the protocol is not fully specified. Therefore, we will use existing documentation and check if it is consistent and argue about the security of undocumented behaviour. The research will cover the Java implementation of the Signal Protocol and the Signal server.

Not part of the scope is explicitly searching for vulnerabilities. An error in the code could, for example, make a message readable for a third-party or cause a cross-site scripting (XSS). However, the goal of this project is to get a better understanding of the protocol implementation. An inconsistency with the documentation may lead to a vulnerability, but it is not the focus of this thesis.

To be more precise, this thesis will focus on the source code of Signal Android and Signal server, but also on the state machine of the protocol. In this way, we get a high-level overview of the protocol. The figure below shows how the state machine could look like. As the figure shows the state machine of the protocol, parts of it might still be unclear but will become more evident in the next chapters. Note that this is a concept state machine. The actual state machine of the protocol will be built with the testing tools and discussed in chapter 5.

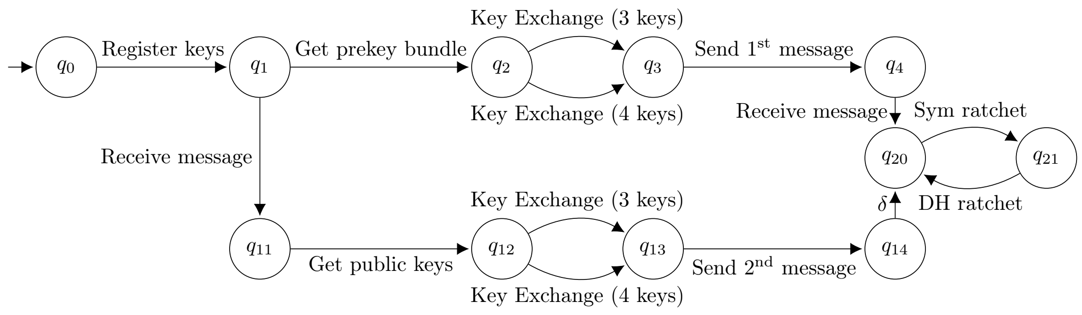


Figure 1.2: Global state machine of the Signal Protocol

1.5 Thesis Outline

The Signal Protocol and fuzzing is introduced in chapter 2. An overview of the related work is given in chapter 3. The implementation analysis is described in chapter 4. We present the fuzzing set-up and results in chapter 5. Lastly, we discuss the findings in chapter 6, describe the future work in chapter 7 and conclude the work in chapter 8.

CHAPTER
TWO

PRELIMINARIES

This chapter introduces the Signal Protocol and the testing technique fuzzing. The knowledge will be used in the later chapters where the actual work is presented.

2.1 Signal Protocol

The Signal Protocol is a protocol that has been built to have private and group conversations over the internet. The design considerations of the protocol have been leaning towards the ease of use. To quote the developer, Moxie Marlinspike, "If it's 'like PGP,' it's wrong" and "The user doesn't know what a key is" [22]. Therefore, the application takes care of all the key management, including key generation, key registration and key exchange.

Before diving into the details of the Signal Protocol, we first give a list of definitions. These definitions may not all be clear at this point but will help to understand the terminology used in the later sections. Table 2.1 describes the (public) asymmetric keys, and table 2.2 describes the cryptographic functions that are used in the protocol. The private exponents are represented as the inverse of the public key, e.g. ik^{-1}.

Key	**Explanation**
ik_A	Alice's identity key
ek_A	Alice's ephemeral key
ik_B	Bob's identity key
spk_B	Bob's signed pre-key
opk_B	Bob's one-time pre-key

Table 2.1: Asymmetric keys

Function	**Explanation**
$\mathsf{DH}(x, y)$	Diffie-Hellman key exchange where x is a private key and y a public key
$\mathsf{KDF}(x)$	One-way function to derive a new key from x
$\mathsf{Enc}(x, y)$	Encrypt message x with key y
$\mathsf{Sig}(x, y)$	Sign message x with key y

Table 2.2: Cryptographic functions

2.1.1 Signal Protocol Phases

To make it easier for the reader, we describe the protocol as a multi-phase protocol. The phases are the registration phase, the key agreement phase and the conversation phase. We will first explain these phases globally, and in the later sections, the phases will be discussed in detail.

Key Registration Phase

The registration phase includes sending several public keys to the server. This makes it possible for users to start a conversation while the other party is offline. When Alice wants to start a conversation with Bob, she can request his public keys from the server.

Key Agreement Phase

The key agreement phase starts when Alice receives Bob's public keys from the server. Alice can use his public keys together with her private keys to generate a shared secret key. She will then send Bob a message encrypted with the shared secret key. Bob will, on receiving Alice's message, fetch her public keys from the server. Finally, he will calculate the same shared secret key based on his private keys and Alice's public keys. This key agreement is possible by using an extended version of the Diffie-Hellman key exchange, as will be explained in section 2.1.2.

Conversation Phase

When both parties have the shared secret key, the conversation phase starts. This phase can be split into two sub-phases, the symmetric ratchet phase, and the Diffie-Hellman ratchet phase.

The symmetric ratchet phase is used to derive a new message key from the shared secret key. When Alice sends multiple messages to Bob without receiving a reply, every message will be encrypted with a new key. This new key is based on the previous key, so that both Alice and Bob, but nobody else could calculate it (at least when the previous key has not leaked).

The Diffie-Hellman ratchet phase is to generate a new shared secret key. When Bob sends a new message to Alice, he generates a new ephemeral key pair. Bob uses this key to derive a new shared secret key. He sends his ephemeral public key to Alice so that Alice can also calculate the new shared secret key. This shared secret key will then be used again in the symmetric ratchet phase to generate new message keys.

2.1.2 Extended Triple Diffie-Hellman

The Signal Protocol uses an extended version of Diffie-Hellman, called extended triple Diffie-Hellman (X3DH). For the sake of completeness, we first explain plain Diffie-Hellman briefly. Alice and Bob both generate a public key pk which is based on the agreed generator g, modulus m and their randomly chosen private key sk (also called secret key). Alice and Bob will then share their public key over an (possibly insecure) channel. After this exchange both can derive a shared secret key ssk. This key can then, for example, be used for communication using symmetric cryptography. Figure 2.1 shows the protocol [7, 17].

X3DH has been developed by Open Whisper Systems to support that one party could be offline during the key exchange. Something which is not possible in the case of plain Diffie-Hellman. We will use the paper of Moxie Marlinspike to describe the protocol [29].

Let ik_A, ek_A be the public keys of Alice and ik_B, spk_B and opk_B be the public keys of Bob, where ik is the identity key, ek the ephemeral key, spk the signed pre-key and opk the one-time pre-key. The identity keys are long-term public keys. An ephemeral key is a key that can only be used once. Pre-keys are keys that are published before the protocol starts. One-time pre-keys

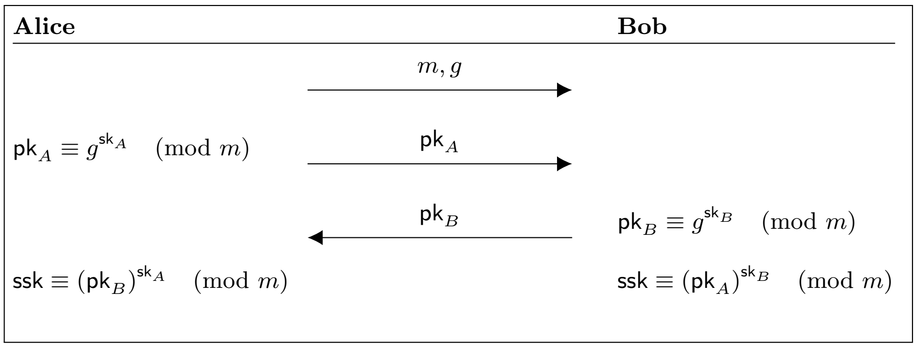

Alice
Bob
m, g
$\mathsf{pk}_A \equiv g^{\mathsf{sk}_A} \pmod m$
pk_A
pk_B
$\mathsf{pk}_B \equiv g^{\mathsf{sk}_B} \pmod m$
$\mathsf{ssk} \equiv (\mathsf{pk}_B)^{\mathsf{sk}_A} \pmod m$
$\mathsf{ssk} \equiv (\mathsf{pk}_A)^{\mathsf{sk}_B} \pmod m$

Figure 2.1: Diffie-Hellman — Standard version

will be published in sets to the server as the server will send one of these keys every time another user wants to start a conversation. The signed pre-key is a pre-key which is signed with the private exponent of the identity key, i.e. $\mathsf{Sig}(\mathsf{spk}_X, \mathsf{ik}_X^{-1})$.

For the protocol to be able to work offline, every user will send their public keys to the server. For example, when Bob signs up, he will send ik_B, spk_B, the signature $\mathsf{Sig}(\mathsf{spk}_B, \mathsf{ik}_B^{-1})$ and $(\mathsf{opk}_B^1, \mathsf{opk}_B^2, \mathsf{opk}_B^3, \ldots)$ to the server. The identity key only has to be sent once, the signed pre-key has to be renewed every week or month. The server will request new one-time pre-keys when there are almost none left.

When Alice wants to start a conversation, she requests the pre-key bundle of Bob from the server. The pre-key bundle exists of Bob's identity key ik_B, his signed pre-key spk_B, the signature of spk_B and one of the one-time pre-keys opk_B^x. The server will delete opk_B^x after sending it with the pre-key bundle. When there are no one-time pre-keys of Bob left, the pre-key bundle will be sent without an one-time pre-key.

On receiving the pre-key bundle, Alice first verifies the signature of spk_B. When the verification succeeds, Alice will generate her ephemeral key pair. Then she will calculate the shared secret key ssk with key derivation function (KDF) $\mathsf{KDF}(x)$ as shown in the equation below. After the calculation, Alice will delete ek_A^{-1} and all the Diffie-Hellman output values.

$$k_1 = \mathsf{DH}(\mathsf{ik}_A^{-1}, \mathsf{spk}_B) \tag{2.1}$$
$$k_2 = \mathsf{DH}(\mathsf{ek}_A^{-1}, \mathsf{ik}_B) \tag{2.2}$$
$$k_3 = \mathsf{DH}(\mathsf{ek}_A^{-1}, \mathsf{spk}_B) \tag{2.3}$$
$$k_4 = \mathsf{DH}(\mathsf{ek}_A^{-1}, \mathsf{opk}_B^x) \qquad \text{Only when received } \mathsf{opk}_B^x \tag{2.4}$$
$$\mathsf{ssk} = \mathsf{KDF}(k_1 \| k_2 \| k_3 \| k_4) \tag{2.5}$$

$\mathsf{DH}(x, y)$ is an elliptic curve Diffie-Hellman function that calculates a shared secret key based on two keys. $\mathsf{KDF}(x)$ is a key derivation function based on RFC5869 [19]. k_1 and k_2 provide mutual authentication, whilst k_3 and k_4 provide forward secrecy. Figure 2.2 depicts the relation.

Alice sends the initial message to Bob consisting of her identity key ik_A, her ephemeral key ek_A, an identifier $\mathrm{ID}_{\mathsf{opk}_B^x}$ so that Bob knows which one-time pre-key Alice used and ik_A concatenated with ik_B encrypted with ssk, i.e. $\mathsf{Enc}(\mathsf{ik}_A \| \mathsf{ik}_B, \mathsf{ssk})$.

Bob, who receives the initial messages, will also calculate the shared secret key in the same way Alice did. Bob will then decrypt the message that Alice has sent and checks if the value is correct. When it is correct, Bob will delete the one-time pre-key that was used. After this, Alice and Bob could reuse ssk for the following messages or keys derived from ssk.

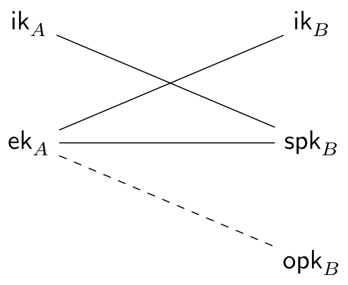


Figure 2.2: Diffie-Hellman key relation

Figure 2.3 shows a simplified version of X3DH, i.e. it does not show when keys have to be deleted and how the calculations shall be done.

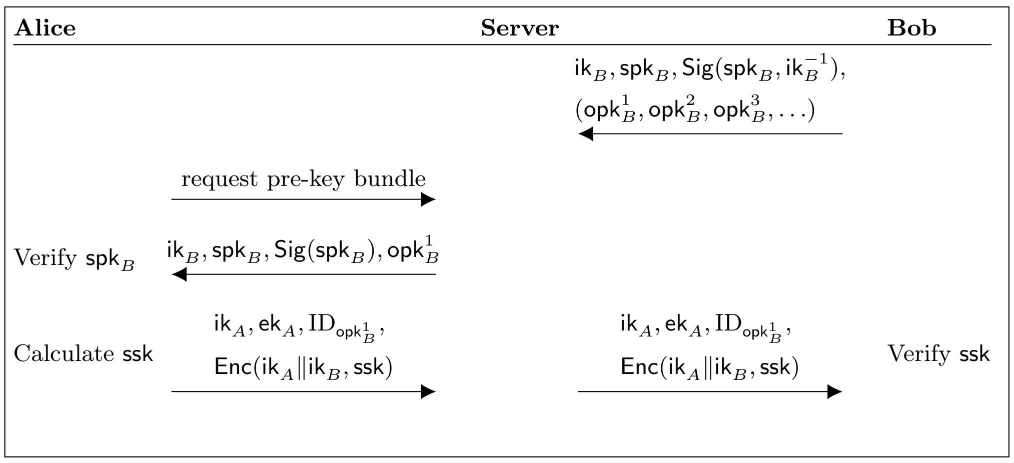

Alice
Server
Bob
$\mathsf{ik}_B, \mathsf{spk}_B, \mathsf{Sig}(\mathsf{spk}_B, \mathsf{ik}_B^{-1}),$
$(\mathsf{opk}_B^1, \mathsf{opk}_B^2, \mathsf{opk}_B^3, \ldots)$
request pre-key bundle
Verify spk_B
$\mathsf{ik}_B, \mathsf{spk}_B, \mathsf{Sig}(\mathsf{spk}_B), \mathsf{opk}_B^1$
Calculate ssk
$\mathsf{ik}_A, \mathsf{ek}_A, \mathrm{ID}_{\mathsf{opk}_B^1},$
$\mathsf{Enc}(\mathsf{ik}_A \| \mathsf{ik}_B, \mathsf{ssk})$
$\mathsf{ik}_A, \mathsf{ek}_A, \mathrm{ID}_{\mathsf{opk}_B^1},$
$\mathsf{Enc}(\mathsf{ik}_A \| \mathsf{ik}_B, \mathsf{ssk})$
Verify ssk

Figure 2.3: X3DH protocol

Note for both Diffie-Hellman and X3DH one cannot guarantee the identity of the other party when the public key (identity key) is not manually verified.

2.1.3 Double Ratchet Algorithm

The Double Ratchet algorithm, formerly known as Axolotl Ratchet [27], is an algorithm developed by Moxie Marlinspike and Trevor Perrin. The algorithm is used in combination with X3DH, where X3DH is used to start a conversation and the Double Ratchet algorithm for the ongoing conversation. We discuss this algorithm based on the paper written by Moxie Marlinspike [27].

Just like X3DH, the Double Ratchet algorithm uses a KDF function. The Double Ratchet algorithm, however, uses the output of the KDF as input for the next key generation using again the KDF. This is called the KDF chain, also shown in figure 2.4.

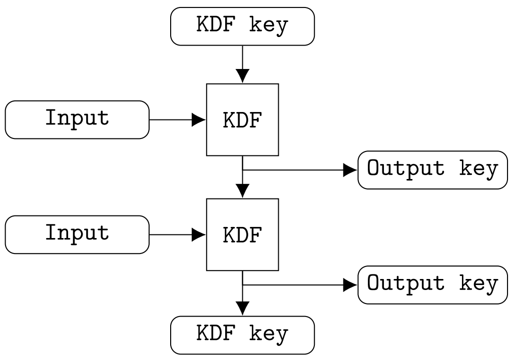


Figure 2.4: KDF chain

The KDF chain provides some interesting properties, namely *resilience*, *forward secrecy* and *future secrecy*. Resilience means that the output key looks random to an adversary. Forward secrecy means that all keys that are generated before key k cannot be calculated when key k is compromised. This is guaranteed as the KDF is a one-way function. Future secrecy means that all keys that are generated after key k cannot be calculated when key k is compromised. Note that this can only be guaranteed when there is enough entropy, i.e. when only the KDF output is used as input, then future secrecy does not hold.

The Double Ratchet algorithm distinguishes two ratchets, the Diffie-Hellman ratchet (figure 2.7) and the symmetric-key ratchet (figure 2.5). The Diffie-Hellman ratchet uses a Diffie-Hellman key to provide entropy for the KDF, while the symmetric-key ratchet ratchet does not have any entropy. Thefore, the Diffie-Hellman ratchet provides future secrecy while the symmetric-key ratchet does not. The generation of a new key is called a Diffie-Hellman/symmetric-key ratchet step.

These two ratchets are mixed in the protocol, hence the name Double Ratchet algorithm. The output keys of the Diffie-Hellman ratchet are used as input for the symmetric-key ratchet, and the symmetric-key ratchet keys are then used to encrypt messages. The Signal implementation generates a new Diffie-Hellman ratchet key for every message to provide future secrecy. Computerphile[1], an initiative from the University of Nottingham, has a great introduction video that explains how these ratchets work.

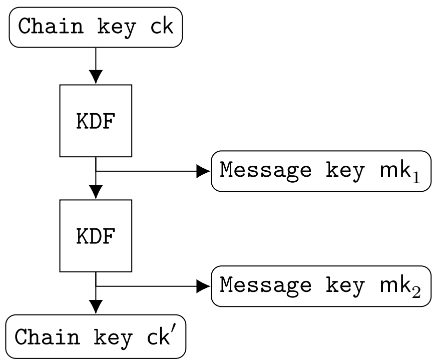


Figure 2.5: Symmetric-key ratchet

[1] https://youtu.be/watch?v=9sO2qdTci-s

Figure 2.6 shows the Diffie-Hellman ratchet on protocol level. Both Alice and Bob will generate a ratchet key pair. When Bob wants to send a message to Alice, he will send his ratchet's public key pk_B^1. Alice will then calculate a new shared secret key from her ratchet's private key and Bob's public key, i.e. $\mathsf{ssk}_1 = \mathsf{DH}(\mathsf{sk}_A^1, \mathsf{pk}_B^1)$. Alice can then send a message with her public key pk_A^1, so that Bob can perform a ratchet step, i.e. he can calculate the same shared secret key $\mathsf{ssk}_1 = \mathsf{DH}(\mathsf{sk}_B^1, \mathsf{pk}_A^1)$. When Bob wants to send a new message, he generates a new ratchet key pair, namely $(\mathsf{pk}_B^2, \mathsf{sk}_B^2)$. He uses this key pair to calculate a new shared secret key with Alice's previous public key, $\mathsf{ssk}_2 = \mathsf{DH}(\mathsf{sk}_B^2, \mathsf{pk}_A^1)$. When Bob sends a message to Alice with his new public key pk_B^2, Alice will calculate ssk_2 and also generate a new ratchet key pair so that she can calculate $\mathsf{ssk}_3 = \mathsf{DH}(\mathsf{sk}_A^2, \mathsf{pk}_B^2)$.

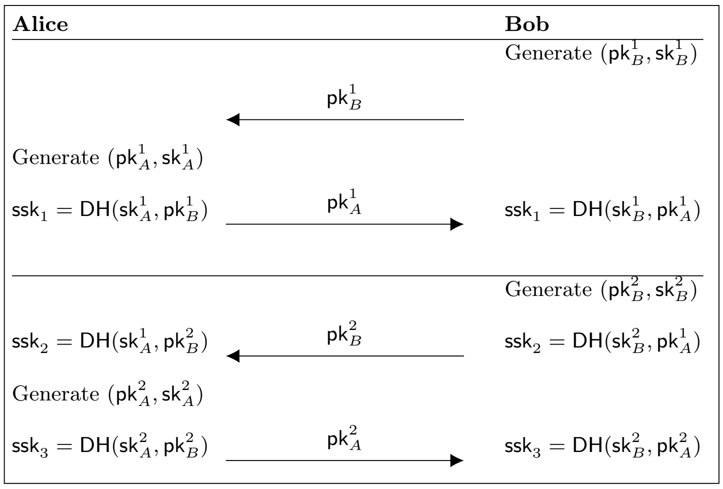

Alice
Bob
Generate $(\mathsf{pk}_B^1, \mathsf{sk}_B^1)$
pk_B^1
Generate $(\mathsf{pk}_A^1, \mathsf{sk}_A^1)$
$\mathsf{ssk}_1 = \mathsf{DH}(\mathsf{sk}_A^1, \mathsf{pk}_B^1)$
pk_A^1
$\mathsf{ssk}_1 = \mathsf{DH}(\mathsf{sk}_B^1, \mathsf{pk}_A^1)$
Generate $(\mathsf{pk}_B^2, \mathsf{sk}_B^2)$
$\mathsf{ssk}_2 = \mathsf{DH}(\mathsf{sk}_A^1, \mathsf{pk}_B^2)$
pk_B^2
$\mathsf{ssk}_2 = \mathsf{DH}(\mathsf{sk}_B^2, \mathsf{pk}_A^1)$
Generate $(\mathsf{pk}_A^2, \mathsf{sk}_A^2)$
$\mathsf{ssk}_3 = \mathsf{DH}(\mathsf{sk}_A^2, \mathsf{pk}_B^2)$
pk_A^2
$\mathsf{ssk}_3 = \mathsf{DH}(\mathsf{sk}_B^2, \mathsf{pk}_A^2)$

Figure 2.6: Diffie-Hellman ratchet on protocol level

It is trivial to see that the protocol in figure 2.6 is vulnerable to a man-in-the-middle attack (MITM) as the ratchet key pairs does not prove the identity of Alice or Bob. Therefore, the first shared secret key has to be mixed with the output of X3DH. Figure 2.7 shows how this is done by deriving a key using the KDF. The output keys, the receiving chain key ck_r and the sending chain key ck_s, are used as input for the symmetric-key ratchet from figure 2.5. As the names imply the sending chain keys are used to derive keys to encrypt messages for the other party, and the receiving chain keys are used to derive keys that decrypt messages from the other party.

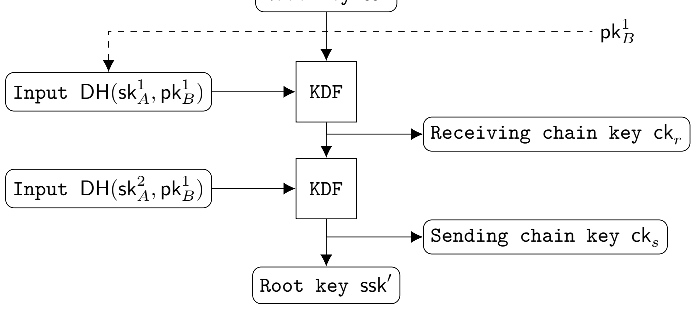

Root key ssk
pk_B^1
Input $\mathsf{DH}(\mathsf{sk}_A^1, \mathsf{pk}_B^1)$
KDF
Receiving chain key ck_r
Input $\mathsf{DH}(\mathsf{sk}_A^2, \mathsf{pk}_B^1)$
KDF
Sending chain key ck_s
Root key ssk'

Figure 2.7: Diffie-Hellman ratchet from Alice's perspective

Figure 2.8 shows the Double Ratchet algorithm from Alice's perspective. In the figure, Alice first receives the public key pk_B^1 of Bob. Alice uses this key to do a Diffie-Hellman ratchet step and thereby she calculates the first receiving chain key ck_r^1. Bob sends Alice three messages, and Alice can decrypt these messages by doing three symmetric-key ratchet steps to calculate mk_r^1, mk_r^2 and mk_r^3. Alice then wants to send two messages to Bob. First, she generates a new ratchet key pair $(\mathsf{pk}_A^2, \mathsf{sk}_A^2)$ and sends pk_A^2 to Bob. Second, she does a Diffie-Hellman ratchet step to calculate the sending chain key ck_s^1. Finally, she calculates the two message keys mk_s^1 and mk_s^2 by doing two symmetric-key ratchet steps [21].

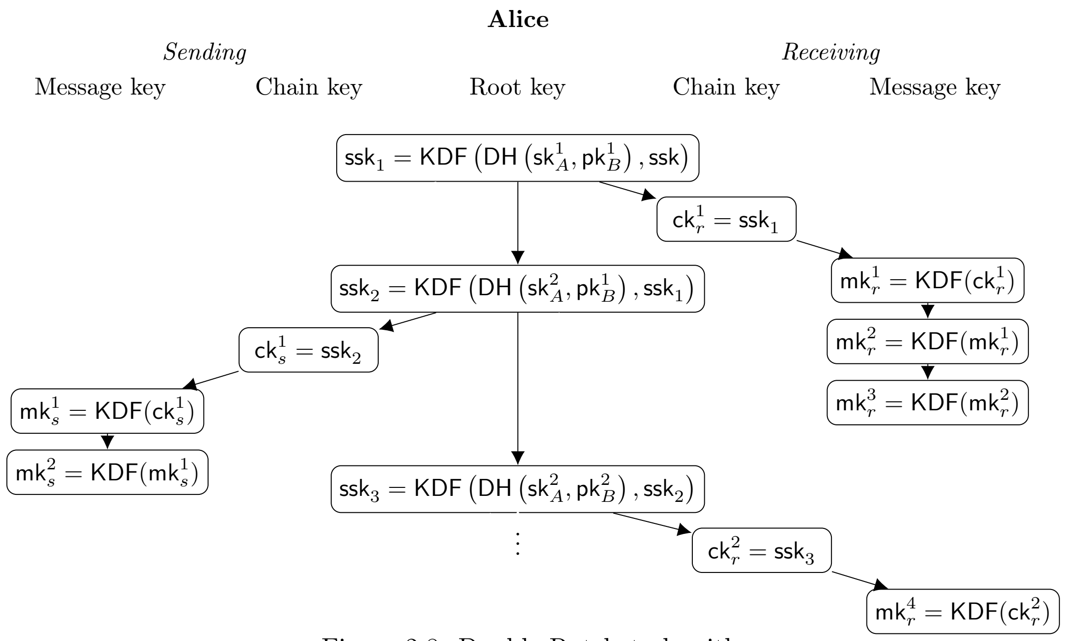

Alice
Sending
Receiving
Message key
Chain key
Root key
Chain key
Message key
$\mathsf{ssk}_1 = \mathsf{KDF}\left(\mathsf{DH}\left(\mathsf{sk}_A^1, \mathsf{pk}_B^1\right), \mathsf{ssk}\right)$
$\mathsf{ck}_r^1 = \mathsf{ssk}_1$
$\mathsf{ssk}_2 = \mathsf{KDF}\left(\mathsf{DH}\left(\mathsf{sk}_A^2, \mathsf{pk}_B^1\right), \mathsf{ssk}_1\right)$
$\mathsf{mk}_r^1 = \mathsf{KDF}(\mathsf{ck}_r^1)$
$\mathsf{ck}_s^1 = \mathsf{ssk}_2$
$\mathsf{mk}_r^2 = \mathsf{KDF}(\mathsf{mk}_r^1)$
$\mathsf{mk}_s^1 = \mathsf{KDF}(\mathsf{ck}_s^1)$
$\mathsf{mk}_r^3 = \mathsf{KDF}(\mathsf{mk}_r^2)$
$\mathsf{mk}_s^2 = \mathsf{KDF}(\mathsf{mk}_s^1)$
$\mathsf{ssk}_3 = \mathsf{KDF}\left(\mathsf{DH}\left(\mathsf{sk}_A^2, \mathsf{pk}_B^2\right), \mathsf{ssk}_2\right)$
$\vdots$
$\mathsf{ck}_r^2 = \mathsf{ssk}_3$
$\mathsf{mk}_r^4 = \mathsf{KDF}(\mathsf{ck}_r^2)$

Figure 2.8: Double Ratchet algorithm

2.1.4 Sesame Algorithm

The Signal Protocol manages sessions with the Sesame algorithm [28]. An algorithm which is specially developed for an asynchronous and multi-device setting. We introduce the need of this algorithm by giving three examples. One, Bob may have multiple devices, meaning that Alice should encrypt her message for all of Bob's devices. Note that Bob could add or delete devices at any given time. Two, as the protocol is asynchronous, both Alice and Bob could start a session at the same time, which means they must agree on which of the two sessions to use for further communication. Three, Bob could reset his state (e.g. by resetting his phone) or restore from a backup, which causes the sessions to be out-of-sync. Apart from these examples specific to the Signal Protocol, the algorithm must also be resistant to general network issues. Messages might not arrive or might be received out-of-order. Moreover, an adversary could tamper with messages on the network.

To explain the Sesame algorithm we first give a list of definitions (table 2.3). The algorithm makes a distinction between users, devices and sessions. A user could have one or more devices, and every device could have none to multiple sessions. The algorithm also makes a distinction between active and inactive sessions. There can be at most one active session at any given time. When a new session is started, the current active session is moved to the list of inactive sessions.

Variable	**Definition**
UserID	The identifier of the user, e.g. a username or telephone number
DeviceID	The identifier of the device (should be unique per user)
SessionID	The identifier of the session
MessageID	The identifier of a message
UserRecord	A record containing a set of DeviceRecords
DeviceRecord	A record containing an active session and/or a list of inactive sessions
MessageRecord	A record containing the information of one message
MAXLATENCY	The amount of time before a UserRecord or DeviceRecord will be deleted

Table 2.3: Sesame definitions

On the server-side, we have of course one or multiple servers and mailboxes. To keep it simple, we assume one server in our further analysis. The server has a mailbox per device, which contains the messages sent to this particular device. Once the messages are retrieved from the mailbox, they will be deleted. When a device sends a message to another device's mailbox, the UserID and DeviceID of the sender will be stored on the server.

Before we discuss encryption and decryption, we first further elaborate the device state. Every device stores a set of UserRecords containing the users it has started a conversation with. A UserRecord or DeviceRecord may be marked stale, which means the corresponding user or device has been deleted, but the record is being kept to decrypt delayed messages. A stale record will be deleted after MAXLATENCY has been reached.

To send a message, the Sesame algorithm will encrypt the plain text for a given UserID. The message will be encrypted for every non-stale device of the user, given there is an active session for the device. The client then sends the encrypted messages together with the recipient's UserID and DeviceID(s) to the server. Upon retrieval, the server checks if the UserID and DeviceID(s) are currently in use. If this is the case, the server will accept the encrypted messages. Otherwise, it will reject the messages and warn the sender. The server will either warn that the user does not exist, or that the device list has been updated. In the latter case, the server will inform the sender of the updated device list. The sender will, on receiving the warning, mark the applicable UserRecord(s) as stale in the case of a non-existing user. In the other case, it will encrypt the message with the updated device list and mark the old DeviceRecord(s) as stale.

To receive a message, the Sesame algorithm fetches the encrypted message and the sender's UserID and DeviceID from the server. If the message is an initiation message and the recipient does not have a session with the sender's device, then the public key is extracted from the message header and a new DeviceRecord is created with the initiation message. Otherwise, the message is decrypted with the applicable session. If there is no such session, then the message is discarded.

For both sending and receiving messages, errors could occur. For sending messages, this could be an invalid server response or another error during the session creation. When this happens, the sender should stop encrypting this message and discard changes made to the state. For receiving messages, errors could occur during the parsing and decryption of messages. The receiver should stop the process, discard the message and also discard the changes to the state.

The paper does not discuss how a client can recover from simultaneously initiated sessions and lost or out-of-order messages; it only discusses a solution to diverged sessions. Furthermore, the paper discusses the Sesame algorithm as an independent algorithm. Especially out-of-order and lost messages are interesting combined with the Double Ratchet algorithm, as this could mean that some ratchet keys are lost. Below is described how one can recover from diverged sessions and in chapter 5, we will analyse how diverged sessions are implemented.

The following could be performed to recover from diverged sessions. A sender will store a MessageRecord for each recipient device. This MessageRecord stores the plain text message, a unique MessageID, the recipient's UserID and the SessionID. When the recipient cannot decrypt the message, it will ask the sender if it can resend the message by sending back the MessageID. The sender then checks if the MessageID refers to one of the MessageRecords and if the UserID is the same as the one in the MessageRecord. If not, the retry request is discarded. Otherwise, when there is an active session in the DeviceRecord, the message will be encrypted and send to the server. If such a session does not exist or if the session has the same SessionID as in the MessageRecord, the device first queries the server. If the UserID or DeviceID does not exist, the records will be marked stale. If they do exist, the device will encrypt the message with the keys it got from the server. Moreover, if the session already exists, the device will send a new initiating message.

2.1.5 Sealed Sender

To complement the end-to-end encryption it is also necessary to keep as less data as possible. This is difficult as one normally authenticates over Transport Layer Security (TLS). It is, of course, fixable by not having to authenticate to the server. However, this gives the problem that it is easy to spoof or to perform denial of service (DOS) attacks. Sealed sender, introduced in October 2018, makes use of short-lived certificates and delivery tokens to solve these problems [30].

The sender will get a new certificate every x time. The x is not specified in the document, only stated is that the certificate is short-lived. This certificate has the phone number and identity key of the sender together with an expiry date. It will be encrypted together with the message. The receiver can validate the authenticity of the certificate on retrieval.

The delivery token is derived from the profile key. To be able to send a message, the sender should proof to the sever it knows the delivery token of the sender. This can only be the case when the two parties have talked before, thus have shared their profile with each other and in this way prevent most of the abuse.

2.1.6 Private Contact Discovery Service

Private contact discovery service is a service that has only been rolled out recently. The service runs in a secure enclave and Signal clients can connect to it remotely. The client can test if the code in the enclave is the same as the online source code — as the enclave supports this functionality. To retrieve the contacts using Signal, the client will send an encrypted version of its contacts. The server will respond with the intersection of all registered users and the contacts send by the client. As the service runs in an enclave, Open Whisper Systems cannot learn about the contacts of the user [26].

2.2 Fuzzing

Fuzzing is a technique that can be used to test applications in a largely automated fashion. The technique works by sending semi-random input to the application and observing the output. For example, one could send special values like negative values, newlines and long strings to the application and then observe if it crashes [31].

There are different kind of fuzzers, namely mutation-based, generation-based (also called grammar-based), evolutionary and white-box fuzzers. Mutation-based fuzzers mutate valid inputs. This can be done by observing the network traffic and then changing some of the bits. Generation-based fuzzers generate inputs based on the file or protocol specification. While mutation-based and generation-based fuzzers are considered black-box testing, evolutionary fuzzing is a grey-box technique. An evolutionary fuzzer tries random mutations; if the mutation leads to a new path, the mutation will be saved [31]. White-box fuzzing analyses the source code by symbolic execution, i.e. analysing what inputs will execute which branches of the code. Based on this analysis, interesting inputs can be generated automatically [15].

2.2.1 Mutation-based Fuzzer

Based on the example of Zeller, Gopinath, Böhme, Fraser and Holler [41], we will illustrate how mutation-based fuzzing works. Assume we have a program that takes a URL as input and parses it. We send the following string:

```
https://www.google.com/search?q=fuzzing
```

Assume that the URL is parsed into different parts, e.g. the protocol is `https://`, the location is `www.google.com`. Now, if we want to craft a valid URL, e.g. one that starts with `https://`, then the actual chance of this happening is $1/93^8$ where 93 represent all the possible ASCII characters.

Using an existing URL and mutating it will reduce this number greatly. A mutation in this sense is inserting, changing or deleting a character in the string. Mutating the protocol part of the URL could lead to another valid protocol, e.g. `https://` $\Rightarrow$ `http://`. The chance of this happening is $1/3 \cdot 1/8 = 1/24$ as we can choose between three operations (e.g. deleting) and there are eight possible characters to delete. When mutating a string multiple times, it will look similar to this:

```
0 mutations:  http://www.google.com/search?q=fuzzing
5 mutations:  http://www.cloogle.com/seaRchq=fuz:ing
10 mutations: http:/L/www.cloWglej.com/seaRchqfuz:in
15 mutations: http:/L/wwcloWglej.com/seaR3hqf,u:in
20 mutations: gtTps/L/SwcloWglej.cdm/seaR3hqf,u:in
```

Figure 2.9: Applying mutations to a URL

2.2.2 Generation-based Fuzzer

The idea of reusing valid input can be further extended by using the specification of the protocol to be fuzzed. To stay with the example of URLs, we can use the URL specification to create valid URLs. To do this, we will write a simple grammar. Figure 2.10 defines this grammar and is loosely based on the book of Zeller et al. [40]. The grammar is in no way meant to be correct nor exhaustive.

```
URL         := PROTOCOL | ://  | NAME | . | DOMAIN | PATH | QUERY
PROTOCOL    := https    | http | ftp
NAME        := google   | bing | duckduckgo
DOMAIN      := com      | nl
PATH        := /        | /ID
ID          := search   | q    | fuzzing
QUERY       := ?PARAMS  | ϵ
PARAMS      := PARAM    | PARAM&PARAMS
PARAM       := ID=ID
```

Figure 2.10: URL grammar

The grammar can be used to generate URLs as input for a fuzzer. Several valid URLs that can be generated by this grammar are shown in figure 2.11. Therefore, generation-based fuzzing is a powerful tool to craft valid inputs easily. The method also has its drawbacks, e.g. the `NAME` could be almost anything, which is hard to capture in a grammar, but much easier to generate with mutations. Another caveat is the loops that can exist in grammars, e.g. the grammar defined in figure 2.10 can, in theory, have unlimited `PARAMS`. However, this is easily fixed in practice by only allowing to loop x amount of times.

```
http://bing.nl/search
ftp://duckduckgo.com/
http://bing.com/q?q=search
https://google.nl/fuzzing?search=q&fuzzing=fuzzing&q=search
```

Figure 2.11: Generate URLs based on a grammar

2.2.3 Evolutionary Fuzzer

The grammar that we defined for URLs was far from complete. It shows the difficulty of manually defining a complex input. This would be even worse for programs that are not well documented. It would be easier to automatically derive valid inputs by observing how the input is processed.

Valid inputs can be generated by using evolutionary algorithms. Such an algorithm is based on biological evolution. First, we choose one or multiple inputs values to start. Second, we will mutate these values to produce new values. When the new values are valid, we will mutate these values further; otherwise we will go on to mutate the old values.

Figure 2.12 shows how an evolutionary algorithm works based on the URL example. We start with a valid URL and mutate it. To know which URLs are valid, we feed the mutated URLs to the program and observe the output. Only when the mutated URL is valid, this new URL will be mutated further.

Clearly, the method depicted in figure 2.12 would not work to find URLs with FTP as protocol. This could be solved by providing more input strings, by mixing valid URLs or by increasing the number of invalid mutations before the branch is not processed any further.

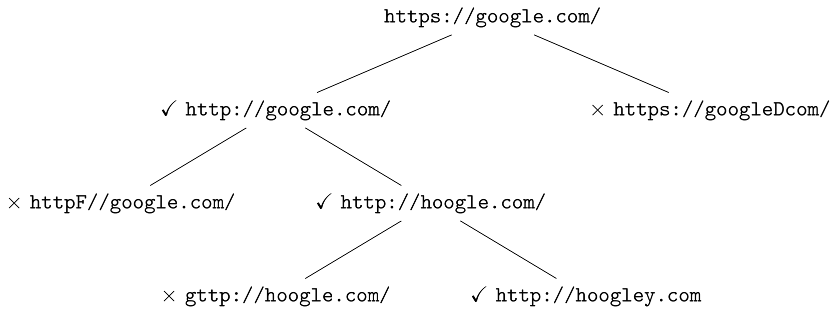


Figure 2.12: URL mutation based on an evolutionary algorithm

2.2.4 White-box Fuzzer

Instead of guessing valid inputs, another method is to analyse the code by symbolic execution. The idea is to use a symbolic value for variables (e.g. `protocol = λ`) instead of a concrete value (e.g. `protocol = "https://"`). Algorithm 2.1 shows a part of the parser. When the code is symbolically executed, then the following code is evaluated: `if protocol == λ`, which could be either true or false. Both branches will be taken and hence eventually all paths will be executed.

Algorithm 2.1 URL parser

```
if protocol is HTTP then
    ...
else if protocol is HTTPS then
    ...
else if protocol is SSH then
    ...
end if
```

Figure 2.13 shows all possible paths from algorithm 2.1. As there are four paths, there are four different inputs that are relevant; other inputs will not trigger anything new. This is thus much more effective than, e.g. mutation-based fuzzing which produces many URLs that do not execute new paths. Moreover, white-box fuzzing can find the edge cases. However, large programs do have a lot of possible paths, and in practice, not all paths will be taken due to the limited time and space [15, 31].

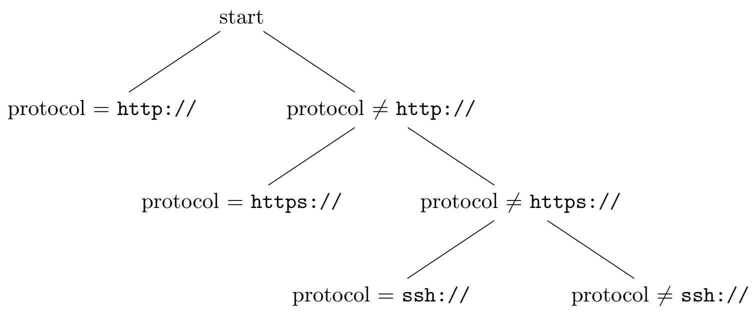


Figure 2.13: URL parser cases

CHAPTER
THREE

RELATED WORK

This chapter discusses the work that is closely related to this thesis. The related work is meant to give an overview of the current state of the research regarding the Signal Protocol and protocol state fuzzing but is not meant to be exhaustive. Furthermore, it also serves as a foundation to get a better understanding of the Signal Protocol and to find techniques that could be used for this thesis.

3.1 Protocol Verification

Several researchers have analysed the Signal Protocol. Frosch, Mainka, Bader, Bergsma, Schwenk and Holz [12] were the first to verify the protocol when it was still called TextSecure. Cohn-Gordon, Cremers, Dowling, Garratt and Stebila [5] have been the first to do a formal security analysis with the Signal Protocol as a multi-stage key exchange protocol. Kobeissi, Bhargavan and Blanchet [18] have used ProVerif and CryptoVerif to model the protocol and to verify it formally. Rubín [32] did a security analysis on the Java protocol implementation. This paper will discuss the papers of Frosch, Cohn-Gordon and Rubín.

3.1.1 Proving TextSecure Secure

TextSecure has been analysed and formally verified by Frosch et al. in 2016 [12]. The formal verification includes the X3DH and the Double Ratchet algorithm individually, but not as composition. They have found two flaws in the implementation and found that the deniability property was not fully satisfied.

Implementation Flaws

One of the implementation flaws resided in the export function. This function would export the password in plain text on the Android device. The password was needed to register one-time pre-keys with the server. The paper states that this feature has already been removed in the meantime.

Another flaw is the vulnerability to the unknown key-share attack (UKS). This flaw is an attack based on the working of the key agreement protocol, where Alice thinks she has the key of Bob while in fact, she has the key of Dave [3]. Frosch et al. explain this by a TextSecure example. In the example, Bob wants to play a joke on Charlie. Bob knows that Charlie will invite him for a party — as Alice already told him. To trick Charlie, Bob will replace his key with the key of

Dave. When Charlie invites Bob for his party using TextSecure, Bob will forward this message to Dave. For Dave, it will look like Charlie has sent this message. So, Charlie thinks he invited Bob, while he actually invited Dave. In 2016, the time the paper was written, the issue was not yet resolved. However, the authors have contacted the developers of TextSecure, and they have also provided methods to solve the issue.

Security Proof

Forward secrecy, *future secrecy* and *deniability* are important properties of the TextSecure Protocol. Frosch et al. have verified under what circumstances these properties hold.

Forward secrecy is satisfied as long as the private keys on the Android telephone are kept secret. As the protocol mixes a long-term shared secret key with short-term keys, both parts have to leak, i.e. when only the long-term key leaks, one cannot calculate the actual message keys.

The paper claims that *future secrecy* is satisfied as long as the long-term private key does not leak. Otherwise, it would be possible to impersonate that party.

Deniability is not satisfied, the messages are encrypted and authenticated using symmetric cryptography, which makes that Alice and Bob can both generate the same encrypted message, which gives deniability. Moreover, Alice or Bob could even hold a whole conversation locally, by first downloading the pre-key bundle of the other party and then imitating the other party locally. Although all this would provide deniability in theory, in practice, the server needs the identity of the user for the correct delivery of messages. Therefore, a user cannot deny involvement in a conversation.

3.1.2 Formal Security Analysis of the Signal Protocol

Cohn-Gordon et al. [5] were the first to conduct a formal analysis of the X3DH key agreement and Double Ratchet algorithm as a multi-staged key exchange protocol. They have accomplished this by modelling a tree of stages to describe the chains used in Signal. They have captured the security properties of Signal with adversarial queries and freshness conditions.

In their paper, they have proven that the Signal Protocol is secure under the gap Diffie-Hellman assumption and the assumption that all KDFs are random oracles. This proof has been conducted in the random oracle model. Moreover, the attack of Frosch's paper, the UKS is not considered an attack in this model. This because the model of Cohn-Gordon does not include identities, as this is also the case in the Signal Protocol. Therefore, the session identifier of Alice's session with Eve is the same as Alice's session with Bob.

The model made several simplifications. Features implemented in Signal that are not considered part of the Signal Protocol such as header encryption are not modelled. The out-of-band verification channel, the mechanism to verify the other party's public key, is not analysed. Out-of-order decryption, the mechanism to decrypt messages that are received later in the chain than expected, is also not analysed. Signal can handle when Alice and Bob start a conversation at the same time; however, this is not modelled. At last, a more complex issue, the model of Cohn-Gordon does not catch attacks where the adversary alters the identity key and inserts a malicious signed pre-key. This because they do not consider the signatures but enforce authentication of the pre-keys. Hence, as the Signal Protocol uses the identity key for both signing the pre-key and for the X3DH an attacker could alter this but the model will not catch it.

3.1.3 Implementation Analysis of the Signal Protocol

Rubín is the first to analyse the Signal Android application. In his thesis, he discusses the theoretical protocol and the Java implementation. Afterwards, he discusses potential security vulnerabilities. In his implementation analysis, he documents a lot of essential functions of the implementation, which is very useful for our project. We will not discuss his documentation here but will use it for our analysis in chapter 4.

The conclusion of Rubín is that he could not find weaknesses in the protocol. The protocol uses secure cryptographic standards, and the implementation of the protocol is also rather robust. Just as Cohn-Gordon, Rubín analysed the documented header encryption variant of the Double Ratchet algorithm. A variant that should be more secure as additional information is also encrypted. However, he could not find this version in the implementation.

3.2 Protocol State Fuzzing

Fuzzing, but also protocol state fuzzing has successfully been used to find flaws in applications. An example is the paper by Fiterau-Brostean, Vaandrager, Poll, de Ruiter, Lenaerts and Verleg [11] in which they have used protocol state fuzzing to check three Secure Shell (SSH) implementations. De Ruiter and Poll [33] have also used protocol state fuzzing to infer state machines from TLS implementations. Protocol state fuzzing is not limited to network protocols. Aarts, de Ruiter and Poll [1] have used this technique to test bank cards that implement the Europay, Mastercard, Visa (EMV) standard. In the next paragraphs, we will discuss the paper about TLS fuzzing and another paper that discusses the effectiveness of several methods to prevent vulnerabilities.

3.2.1 Fuzzing TLS Implementations

Multiple TLS implementations have been tested by de Ruiter and Poll [33]. They have used state machine learning to deduce state machines from both the client and server TLS implementations. To accomplish this, they have fuzzed different sequences of messages to infer the state machine. This is why they called their version of state machine learning protocol state fuzzing.

To perform protocol state fuzzing, de Ruiter and Poll used LearnLib[1] — a Java framework for automata learning developed at TU Dortmund. The LearnLib framework offers numerous learning algorithms. This research project used (a modified version of) Angluin's L* algorithm. To send messages with LearnLib to the system under test (SUT), one should create a grammar (like in section 2.2.2). As the messages created from the grammar are abstract, de Ruiter and Poll have implemented an adapter that translates these abstract messages to concrete messages that the TLS implementation can understand. When sending these messages, LearnLib will develop hypotheses for the state machine. The hypotheses are based on the responses from the SUT. The hypotheses are checked on equivalence with the implemented state machine. However, this state machine is unknown; therefore, the check is done with an (improved version) of Chow's W-method. This method can be used to proof the generated state machine correct to a given upper bound.

This research project found new flaws in three out of the nine tested implementations. They argue that protocol state fuzzing can find logic errors in the implementation. However, this method is not suitable to find intentional backdoors. Moreover, this method would also not find errors in, for instance, message parsing.

[1]https://learnlib.de/

3.2.2 Testing Techniques to Prevent Vulnerabilities

This section discusses the effectiveness of testing techniques based on the Heartbleed[2] bug. The paper discusses multiple techniques including multiple types of fuzzing. The Heartbleed bug was found in OpenSSL implementations and would let attackers obtain information from memory remotely. As an attacker was able to control the length of the payload, which could be longer than the actual payload, the attacker could read further than this payload in memory, i.e. a buffer over-read. Interestingly is that many testing techniques would not find a bug like Heartbleed.

Wheeler [38] discussed multiple methods that do not work to find the Heartbleed bug. Static analysis would not work, for the simple reason that the OpenSSL code is too complex. Dynamic analysis would also not work, as it is not designed to find these sorts of vulnerabilities. Fuzzing is a technique that mostly looks for crashes, something which will not happen with a buffer over-read. Wheeler also states that most testing techniques will test correct input, which is in contrast to Heartbleed and most other security vulnerabilities, which come from incorrect input.

The following methods are methods that would work to find a bug such as Heartbleed according to Wheeler. Negative testing, as these tests are designed to let the software fail. Fuzzing could also find the bug when extra tools are used that can detect these buffer over-reads. An example of such a tool is Address Sanitizer (ASan), an open-source tool by Google to detect, amongst other things, buffer overflows. Also, when the output during fuzzing is examined, Heartbleed can be found. This is the way how Codenomicon[3] found the vulnerability. Another method is by human review. A person could do a review to analyse if all input is validated correctly. Static analysis could be used when giving the analyser more context, e.g. by annotating the code. Microsoft's source-code annotation language (SAL) is an example of such a tool. The last method described by Wheeler is formal methods. For this, a formal specification should be made, and then the code should be checked to prove that the code meets the specification.

[2]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

[3]A company founded in 2011 that built fuzzing tools. Acquired by Synopsys in 2011

CHAPTER

FOUR

ANALYSIS OF PROTOCOL IMPLEMENTATION

This chapter gives an overview of the Signal code, with a focus on the Signal Protocol Java library and Signal server. The chapter also gives an overview of how the whole system interacts.

4.1 Overview

The Signal Protocol is an instant messaging protocol developed for use with Android and iOS. Open Whisper Systems also created a desktop version that can work independently from the smartphone client after linking. Figure 4.1 shows how the applications interact with the Signal server. This server has many dependencies, which will be explained in more detail later in this chapter.

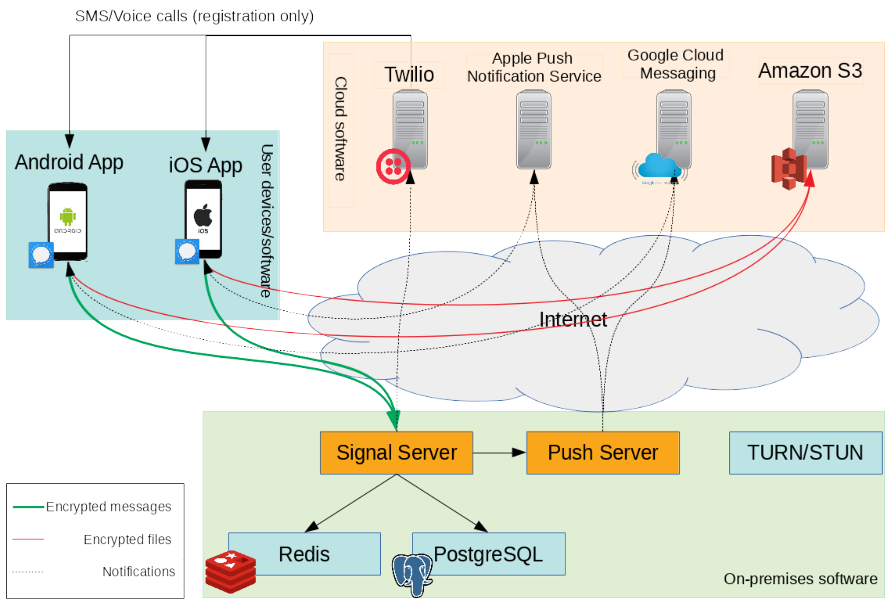


Figure 4.1: Signal Architecture[1]

4.1.1 Signal Application

The Signal application for Android and iOS are made out of several layers. The lowest layer consists of the cryptographic functions. These functions are used in the protocol layer, which implements the Signal Protocol. As the protocol is stateful, these separate protocol functions have to be combined to start an actual conversation, which is done in the service layer. The Signal application uses these layers to build a smartphone application that can start end-to-end encrypted conversations (it also supports other features, such as encrypted video calling).

There is also a desktop application written in JavaScript. This application can be used independently, but only after it is linked to an Android or iOS device. When installed, the desktop application will generate a key pair. The public key will be encoded as a QR code that can be scanned by the Android or iOS app. The smartphone then encrypts the identity key with the public key of the desktop app and uploads it to the Signal server. The encrypted identity key will be downloaded by the desktop app and decrypted locally. From that point, the desktop client can be used even when the phone is switched off.

In this thesis, the desktop and iOS app are left out of scope. The iOS codebase will still be briefly discussed in the following paragraphs to show the differences with the Android codebase. However, the in-depth analysis, starting from section 4.2, will only discuss the Java application, as this is the application that will be fuzzed in chapter 5.

Cryptographic Functions Layer

The `KDF()` of both Java and C are built in the protocol library. The Java implementation of Curve25519 (`curve25519-java`[2]), the elliptic curve used for key agreement, is implemented by Open Whisper Systems. The C implementation of the elliptic curve is forked from Adam Langley's implementation (`curve25519-donna`[3]) of 2008. The GitHub page of Adam Langley states the following: "Note: this code is from 2008. Since that time, many more, great implementations of curve25519 have been written, including several amd64 assembly versions by djb. You are better served now by NaCl or libsodium.". Although this could mean that the implementation has unresolved bugs, there are no open issues that imply this. The latest commit is from 14 November 2015.

Protocol Library Layer

The Signal Protocol Android library is developed in Java (`libsignal-protocol-java`[4]), and the library for iOS in C (`libsignal-protocol-c`[5]). These libraries use the cryptographic functions to implement the Signal Protocol. As this library does not have states, it cannot be used to start a conversation.

Service Layer

The repository also contains a service library in Java (`signal-service-java`[6]). This service layer implements the states by using the Signal Protocol Java interfaces from the library. The application programming interface (API) of the service layer is used in the Signal Android app. There is no such service layer for iOS.

[1]https://sorincocorada.ro/signal-messanger-architecture/
[2]https://github.com/signalapp/curve25519-java
[3]https://github.com/agl/curve25519-donna
[4]https://github.com/signalapp/libsignal-protocol-java
[5]https://github.com/signalapp/libsignal-protocol-c
[6]https://github.com/signalapp/libsignal-service-java

4.1.2 Signal Server

The Signal server used to register keys and relay messages is developed in Java. The Signal server (`Signal-Server`[7]) is also available on the GitHub repository. The dependencies of the server can be found in the configuration file. PostgreSQL is used as database management system, Remote Dictionary Server (Redis) as key-value database, Twilio for the phone number verification using text messages and Amazon Web Services (AWS) to store attachments and profiles. Furthermore, Apple Push Notification Service (APNs) and Google Cloud Messaging (GCM) are used for sending push notifications [25].

4.2 Signal Android

First, we analyse the source code of Signal Android regarding key and session creation. We present this analysis by using sequence diagrams.

4.2.1 Key Generation

The key generation is done in the smartphone application using the library functions. Figure 4.2 shows how the key generation is implemented in Android. To create a registration ID, the random number generator of Java is used. Then the private identity key is loaded, as this key pair is already created before the registration. After the key is loaded, the pre-keys are generated. The application will create 100 one-time pre-keys and one signed pre-key using `generatePreKeys()`. The signed pre-key is signed with the private identity key using `generateSignedPreKey()`. This concludes the key generation phase.

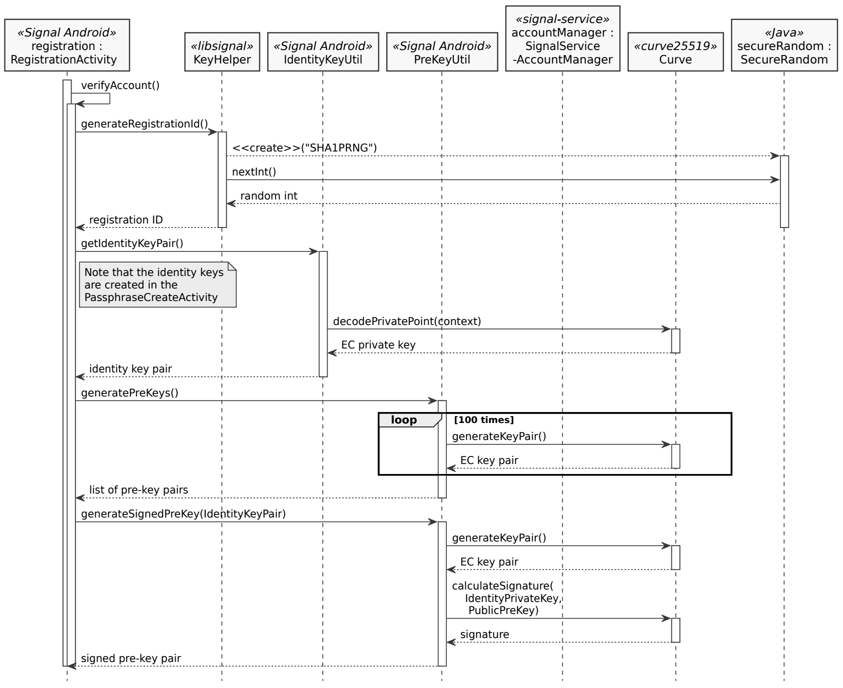

«Signal Android» registration : RegistrationActivity
«libsignal» KeyHelper
«Signal Android» IdentityKeyUtil
«Signal Android» PreKeyUtil
«signal-service» accountManager : SignalService -AccountManager
«curve25519» Curve
«Java» secureRandom : SecureRandom
verifyAccount()
generateRegistrationId()
<<create>>("SHA1PRNG")
nextInt()
random int
registration ID
getIdentityKeyPair()
Note that the identity keys are created in the PassphraseCreateActivity
decodePrivatePoint(context)
EC private key
identity key pair
generatePreKeys()
loop
[100 times]
generateKeyPair()
EC key pair
list of pre-key pairs
generateSignedPreKey(IdentityKeyPair)
generateKeyPair()
EC key pair
calculateSignature(IdentityPrivateKey, PublicPreKey)
signature
signed pre-key pair

Figure 4.2: Key generation

[7]https://github.com/signalapp/Signal-Server

4.2.2 Session Creation

The session creation (figure 4.3) for one-to-one messaging is started by the `process()` call. This call is done from the Signal service class `SignalServiceMessageSender`. First, the identity of the other party is checked with `isTrustedIdentity()`. As Signal uses the trust on first use (TOFU) principle, a key is also trusted when the database has no entry of this public key. Only when there is a mismatch with the public key in the database, the other party is considered untrusted. When this is the case, or when the signed pre-key is `null` or the signature cannot be verified, an exception will be thrown.

After the checks, the session (if existing) will be loaded from the session store. Otherwise, a new session will be returned from the `loadSession()` function. To start the X3DH, an ephemeral key pair has to be created with `generateKeyPair()`. All the keys will then be set as a parameter to the `AliceSignalProtocolParameters` class, which is used to start the key agreement.

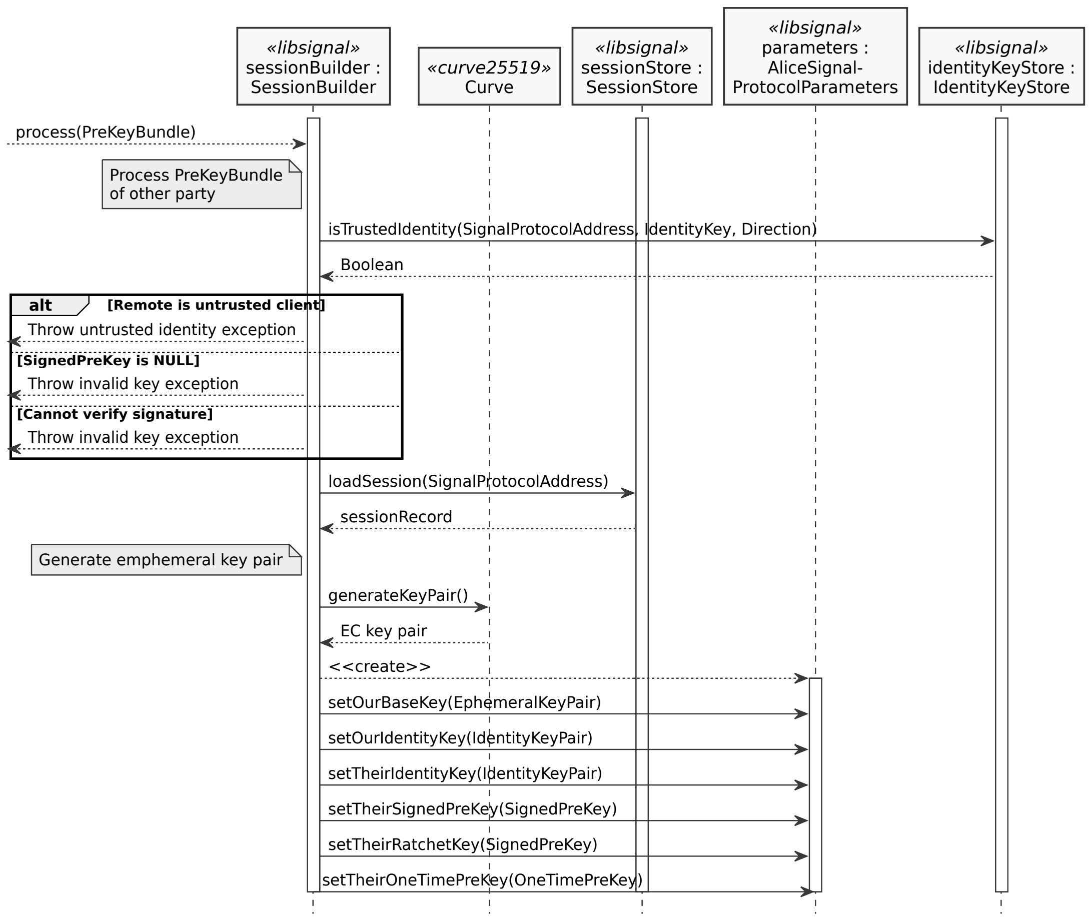

«libsignal» sessionBuilder : SessionBuilder
«curve25519» Curve
«libsignal» sessionStore : SessionStore
«libsignal» parameters : AliceSignal-ProtocolParameters
«libsignal» identityKeyStore : IdentityKeyStore
process(PreKeyBundle)
Process PreKeyBundle of other party
isTrustedIdentity(SignalProtocolAddress, IdentityKey, Direction)
Boolean
alt [Remote is untrusted client]
Throw untrusted identity exception
[SignedPreKey is NULL]
Throw invalid key exception
[Cannot verify signature]
Throw invalid key exception
loadSession(SignalProtocolAddress)
sessionRecord
Generate emphemeral key pair
generateKeyPair()
EC key pair
<<create>>
setOurBaseKey(EphemeralKeyPair)
setOurIdentityKey(IdentityKeyPair)
setTheirIdentityKey(IdentityKeyPair)
setTheirSignedPreKey(SignedPreKey)
setTheirRatchetKey(SignedPreKey)
setTheirOneTimePreKey(OneTimePreKey)

Figure 4.3: Session creation

Note that the `loadSession()` function is not implemented in the library, it is merely an interface that should be implemented by the application that wants to run a Signal client. The Signal Android app implements the interface in the class `TextSecureSessionStore`.

4.2.3 Key Agreement

X3DH is implemented in the library for both Alice and Bob. This distinction is made as the key agreement is different for the conversation starter and the conversation receiver (see figure 2.2 in subsection 2.1.2).

Figure 4.4 depicts how the key agreement looks like from Alice's perspective. First, the `SessionBuilder` class calls the `initializeSession()` function. The `RatchetingSession` then initialises the session by generating a ratchet key pair with `generateKeyPair()`. This key pair will not be used during the key agreement, but has to be generated for the Double Ratchet algorithm to work. Then all needed keys are loaded. In the actual implementation it is done in the following way:

```
secrets.write(Curve.calculateAgreement(
    parameters.getTheirSignedPreKey(),
    parameters.getOurIdentityKey().getPrivateKey()))
```

However, to keep the diagram simpler, we show this as `getKeys()` and `write(key)`. The X3DH is calculated using the agreement function of the elliptic curve library using multiple `calculateAgreement()` calls. Note the optional key agreement with the one-time pre-key at the bottom of the figure.

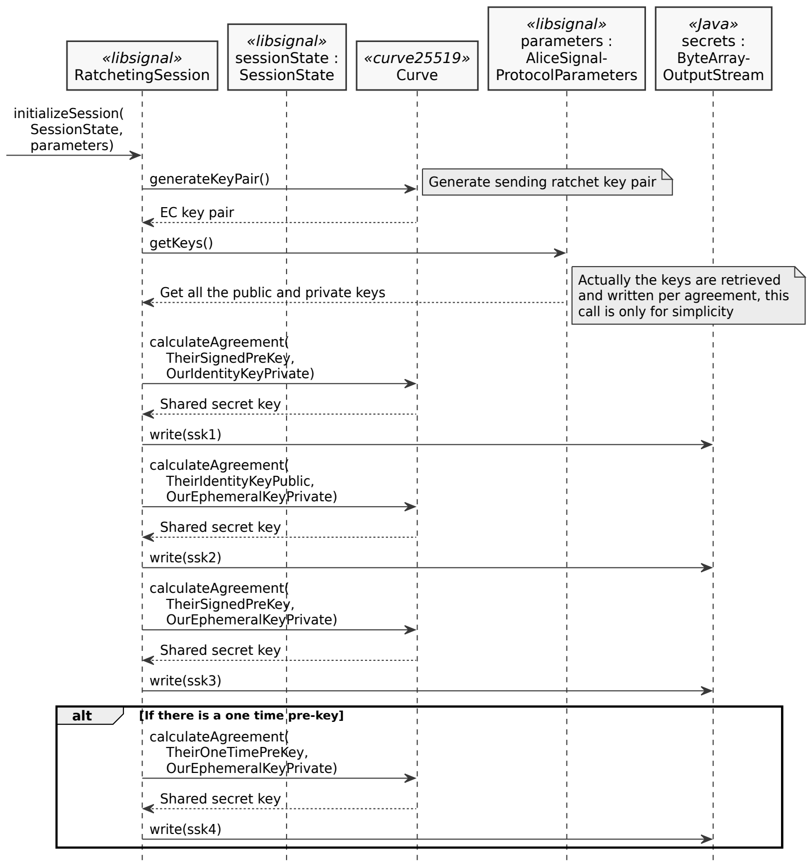

«libsignal» RatchetingSession
«libsignal» sessionState : SessionState
«curve25519» Curve
«libsignal» parameters : AliceSignal-ProtocolParameters
«Java» secrets : ByteArray-OutputStream
initializeSession(SessionState, parameters)
generateKeyPair()
Generate sending ratchet key pair
EC key pair
getKeys()
Actually the keys are retrieved and written per agreement, this call is only for simplicity
Get all the public and private keys
calculateAgreement(TheirSignedPreKey, OurIdentityKeyPrivate)
Shared secret key
write(ssk1)
calculateAgreement(TheirIdentityKeyPublic, OurEphemeralKeyPrivate)
Shared secret key
write(ssk2)
calculateAgreement(TheirSignedPreKey, OurEphemeralKeyPrivate)
Shared secret key
write(ssk3)
alt
[If there is a one time pre-key]
calculateAgreement(TheirOneTimePreKey, OurEphemeralKeyPrivate)
Shared secret key
write(ssk4)

Figure 4.4: Key agreement

The actual shared secret key that will be used as input for the Double Ratchet algorithm is calculated with the `calculateDerivedKeys()` function, as shown in figure 4.5. This function derives a key from the Diffie-Hellman secrets of figure 4.4. The first half of the shared secret key is used as root key and the second half as chain key. These keys will be saved in the `sessionState`. For the receiving chain, the ratchet key of the other party (Bob) will be saved, and for the sending chain, the ratchet key of Alice.

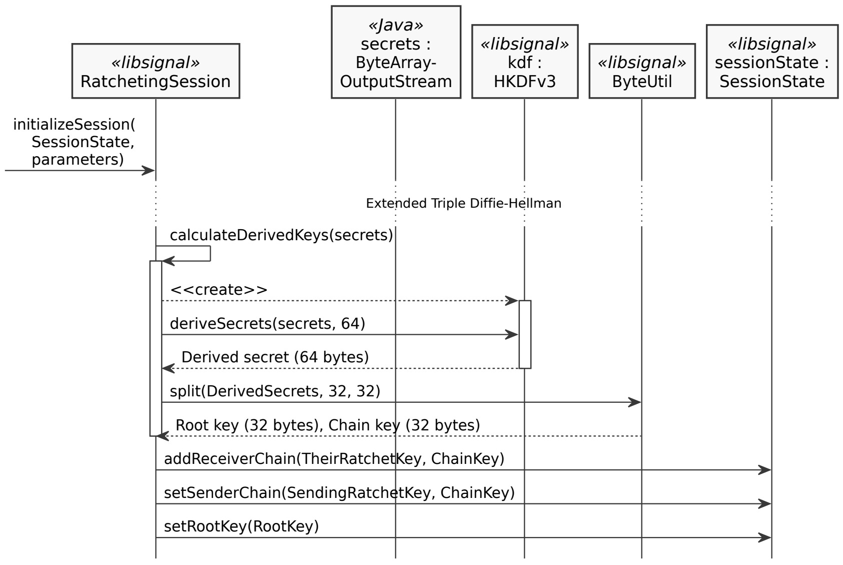

«libsignal» RatchetingSession
«Java» secrets : ByteArray-OutputStream
«libsignal» kdf : HKDFv3
«libsignal» ByteUtil
«libsignal» sessionState : SessionState
initializeSession(SessionState, parameters)
Extended Triple Diffie-Hellman
calculateDerivedKeys(secrets)
<<create>>
deriveSecrets(secrets, 64)
Derived secret (64 bytes)
split(DerivedSecrets, 32, 32)
Root key (32 bytes), Chain key (32 bytes)
addReceiverChain(TheirRatchetKey, ChainKey)
setSenderChain(SendingRatchetKey, ChainKey)
setRootKey(RootKey)

Figure 4.5: Key derivation

4.2.4 Double Ratchet

Messages are encrypted and decrypted using the `encrypt()` and `decrypt()` functions in the class called `SessionCipher`. To be able to use these functions, the session has to be established first with the `SessionBuilder` class (see section 4.2.2). Note that `encrypt()` and `decrypt()` are not called in the library, but only in the Signal service. Hence, the starting call in figure 4.6 and 4.7 is from the Signal service.

Figure 4.6 depicts the start of the `encrypt()` function. The function first loads the session and the session state. The function will then load all the needed keys from the session state with the functions `getSenderChainKey()`, `getMessageKeys()` and `getSenderRatchetKey()`. At last, other session information is loaded, i.e. the counter and session version.

Figure 4.7 shows the actual encryption. This is done with the `getCiphertext()` function in class `SessionCipher`. A `SignalMessage` is then created which stores the ciphertext together with the keys. To conclude this phase, the next chain key is generated with `getNextChainKey()` and saved in the session state `setSenderChainKey()`. The identity of the other party is checked, just like in figure 4.3. Finally, the identity is saved with `saveIdentity()`, the session stored using `storeSession()`, and the `SignalMessage` ciphertext is returned.

Note that after every encryption a new chain key is generated with `getNewChainKey()`. This means a new Diffie-Hellman secret is used for every encrypted message, and thus guarantees future secrecy.

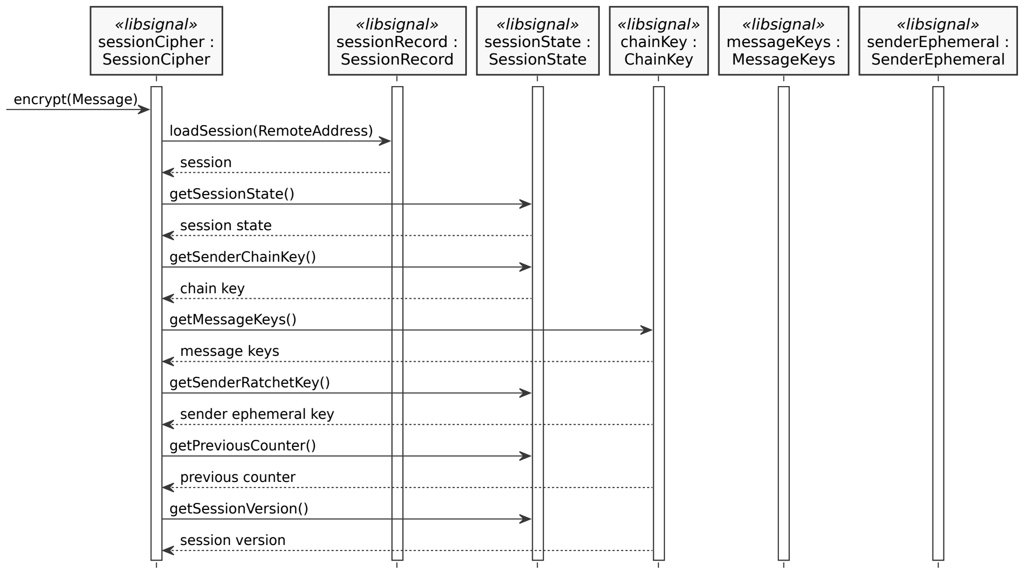

«libsignal» sessionCipher : SessionCipher
«libsignal» sessionRecord : SessionRecord
«libsignal» sessionState : SessionState
«libsignal» chainKey : ChainKey
«libsignal» messageKeys : MessageKeys
«libsignal» senderEphemeral : SenderEphemeral
encrypt(Message)
loadSession(RemoteAddress)
session
getSessionState()
session state
getSenderChainKey()
chain key
getMessageKeys()
message keys
getSenderRatchetKey()
sender ephemeral key
getPreviousCounter()
previous counter
getSessionVersion()
session version

Figure 4.6: Encrypt function : Load keys

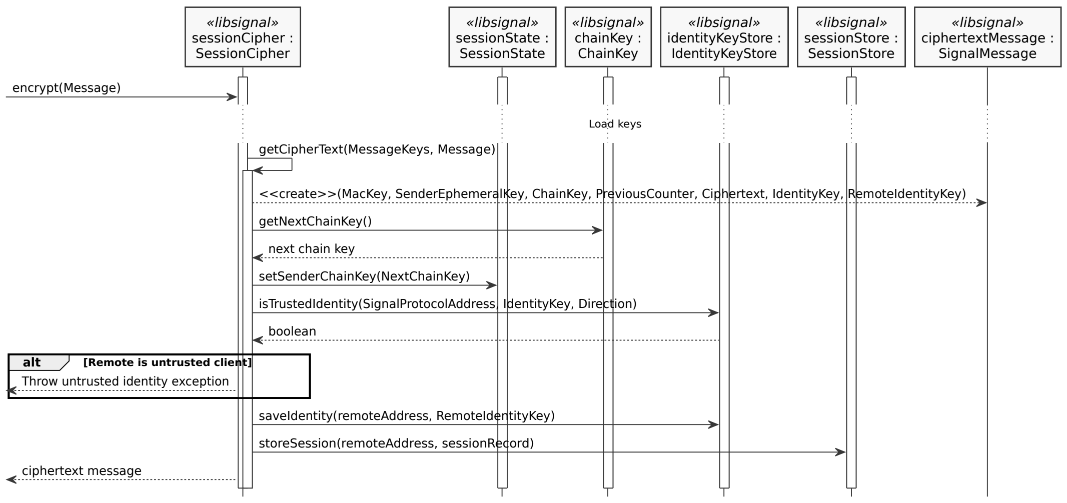

«libsignal» sessionCipher : SessionCipher
«libsignal» sessionState : SessionState
«libsignal» chainKey : ChainKey
«libsignal» identityKeyStore : IdentityKeyStore
«libsignal» sessionStore : SessionStore
«libsignal» ciphertextMessage : SignalMessage
encrypt(Message)
Load keys
getCipherText(MessageKeys, Message)
<<create>>(MacKey, SenderEphemeralKey, ChainKey, PreviousCounter, Ciphertext, IdentityKey, RemoteIdentityKey)
getNextChainKey()
next chain key
setSenderChainKey(NextChainKey)
isTrustedIdentity(SignalProtocolAddress, IdentityKey, Direction)
boolean
alt
[Remote is untrusted client]
Throw untrusted identity exception
saveIdentity(remoteAddress, RemoteIdentityKey)
storeSession(remoteAddress, sessionRecord)
ciphertext message

Figure 4.7: Encrypt function : Generate ciphertext

4.3 Signal Server

For the Signal server we analysed the source code and the API that can be used to send messages to the server.

4.3.1 Registration

The registration starts with a simple GET request. The user can either request for a verification code through `sms` or `voice` using the `transport` option. The `number` should be an E.164 encoded telephone number, e.g. `+31612345678`.

```
GET /v1/accounts/{transport}/code/{number}
```

Figure 4.8: Request verification code (GET request)

The server, figure 4.9, responses to this request by checking the input values (`transport` and `number`). When these are correct, the server will generate a verification code using Java's SecureRandom library. This code will be stored together with the current time so that there is a time limit on how long this code can be used. Finally, the code is sent by either a text message or through a call.

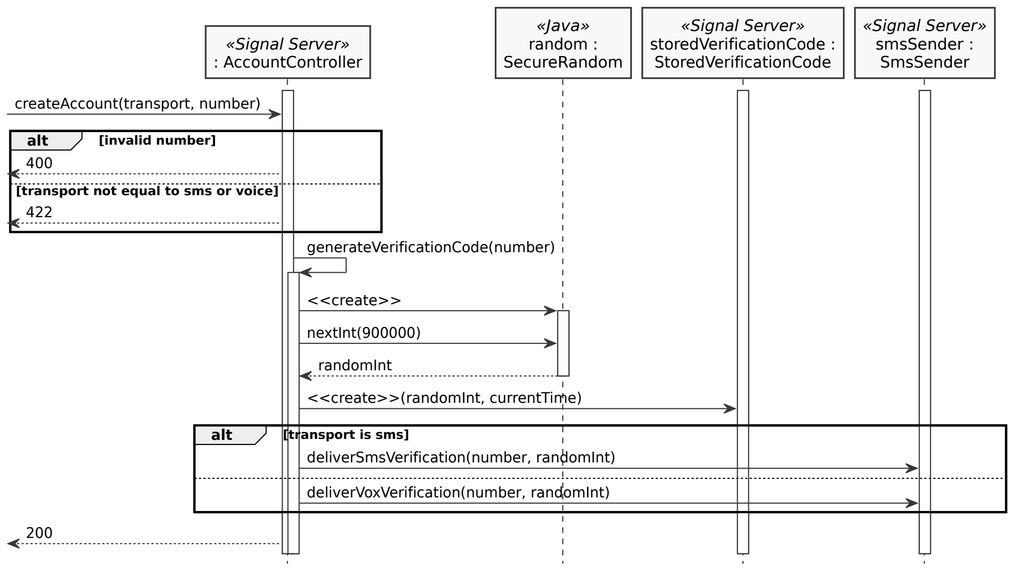


Figure 4.9: Request verification code

The user can confirm the verification code by sending a PUT request to the server. This request contains the values `verification_code`, the authentication token `token`, a registration ID `registration_id` and a key called `signaling_key`. In practice, some extra variables have to be sent as well, but these are less important — see appendix B.3 for the full request.

The authentication token is a Base64 encoded string consisting of `{number}:{password}`, where `password` is a random ASCII string of 24 bytes. This token has to be sent with all the following requests as well, making it a kind of session key. The `registration_id` is a 14 bit random integer which functions to identify the device. The `signaling_key` is a Base64 encoded 32 byte AES key and 20 byte HMAC-SHA1 MAC key.

```
PUT /v1/accounts/code/{verification_code}
Authorization: Basic {token}
{
  "registrationId" : {registration_id},
  "signalingKey" : "{signaling_key}",
}
```

Figure 4.10: Confirm verification code (PUT request)

The server first checks if the verification code and the header are valid. If so, a new account will be created with the information given to Signal such as the number, name, password, ID and key. More notably is the other information that Signal stores, especially when thinking of Signal's focus on privacy. The server also stores the time of account creation, last seen status precise on the day, and the user agent of the user.

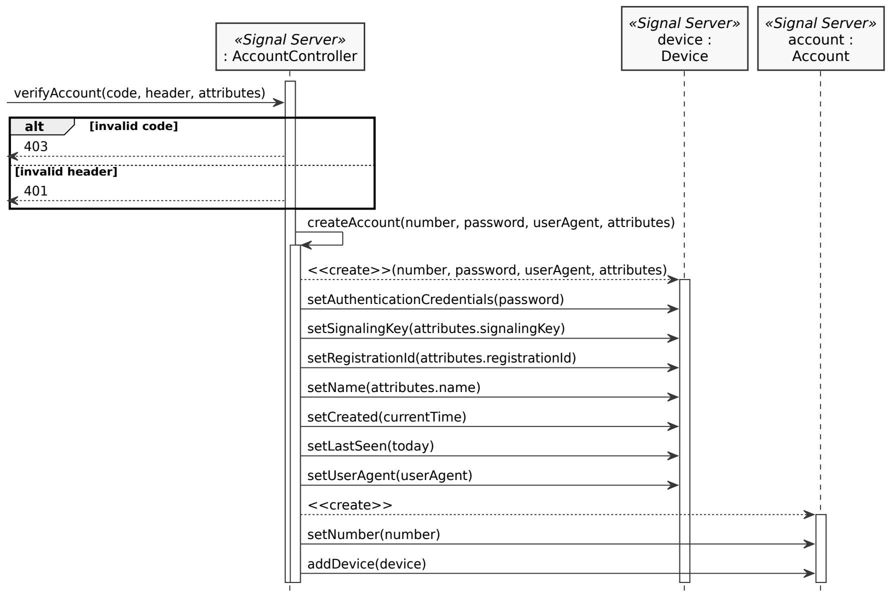

«Signal Server»
: AccountController
«Signal Server»
device :
Device
«Signal Server»
account :
Account
verifyAccount(code, header, attributes)
alt
[invalid code]
403
[invalid header]
401
createAccount(number, password, userAgent, attributes)
<<create>>(number, password, userAgent, attributes)
setAuthenticationCredentials(password)
setSignalingKey(attributes.signalingKey)
setRegistrationId(attributes.registrationId)
setName(attributes.name)
setCreated(currentTime)
setLastSeen(today)
setUserAgent(userAgent)
<<create>>
setNumber(number)
addDevice(device)

Figure 4.11: Confirm verification code

After the account is created, the user can upload the keys to the server. The `identity_key` and `public_keys` are both Curve25519 Base64 encoded public keys with a size of 33 bytes. A client will publish 100 pre-keys when registering, see figure 4.2, however, this is not a restriction. Along with these keys, a signed pre-key will be sent with a `signature` over the `public_key`. The `key_id` is an identifier of 3 bytes.

```
PUT /v2/keys/
Authorization: Basic {token}
{
  "identityKey":"{identity_key}",
  "preKeys" : [
    {"keyId" : {key_id}, "publicKey" : "{public_key}"},
    {"keyId" : {key_id}, "publicKey" : "{public_key}"},
    ...
  ],
  "signedPreKey" : {
    "keyId" : {key_id},
    "publicKey" : "{public_key}",
    "signature" : "{signature}"
  }
}
```

Figure 4.12: Register keys

4.3.2 Messaging

Messaging is in principle fairly easy. The difficulty mostly arises from the multi-device and the asynchronous setting. For instance, messages may be sent to an already deleted device, and one should make sure that messages can only be fetched by the user for which the message is meant.

Figure 4.13 shows how a message can be sent. The `token` generated during the registration should be used to authenticate to the server. An array of messages is used to make it easy to send a message when the receiver has multiple devices. For this reason, the `device_id` of the receiver should be specified. The message `encrypted_msg` is uniquely encrypted per device and Base64 encoded.

```
PUT /v1/messages/{destination_number}
Authorization: Basic {token}
{
  "messages" : [{
    "type" : 1,
    "destinationDeviceId" : {device_id},
    "body" : "{encrypted_msg}",
    "timestamp" : {millis_since_epoch}
  }]
}
```

Figure 4.13: Send a message

The most important task of the server is checking the header of the message. In figure 4.14 we see several error codes. First is error `404` when the user does not exist. Second, error `409` when there is a mismatch between the devices. The server will send the missing and/or extra devices in the response. Error `410` will be returned when there are stale devices; the list of stale devices will be sent in the response. The latest is `400` when the destination is invalid. If the header is valid, the messages will be wrapped in the `Envelope` class and sent to the receiver.

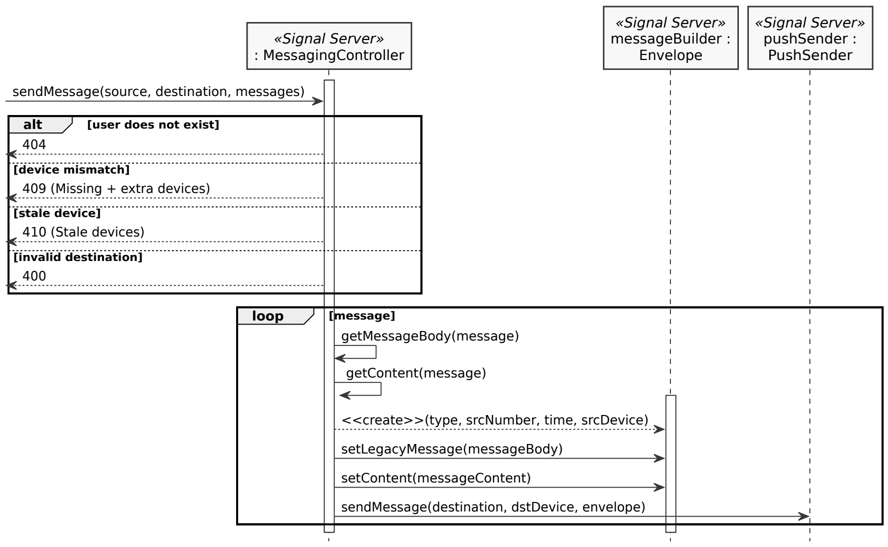


Figure 4.14: Messaging

4.4 Vulnerabilities

An implementation analysis would not be complete without describing the previously found vulnerabilities. The common vulnerabilities and exposures (CVE) system lists one bug in the Android app, two bugs in the iOS app and four bugs in the desktop application. The following CVE descriptions are from the CVE Details[8] website.

CVE-2018-9840 (Signal iOS) "allows physically proximate attackers to bypass the screen locker feature via certain rapid sequences of actions that include app opening, clicking on cancel, and using the home button."

CVE-2018-16132 (Signal iOS) "The image rendering component (. . .) fails to check for unreasonably large images before manipulating received images. This allows for a large image sent to a user to exhaust all available memory when the image is displayed, resulting in a forced restart of the device."

CVE-2018-10994 (Signal Desktop) "`js/views/message_view.js` (. . .) allows XSS via a URL."

CVE-2018-11101 (Signal Desktop) "allows XSS via a resource location specified in an attribute of a SCRIPT, IFRAME, or IMG element, leading to JavaScript execution after a reply."

CVE-2018-14023 (Signal Desktop) "allows information leakage."

CVE-2019-9970 (Signal Desktop & Signal Android) "are vulnerable to an IDN homograph attack when displaying messages containing URLs. This occurs because the application produces a clickable link even if (for example) Latin and Cyrillic characters exist in the same domain name, and the available font has an identical representation of characters from different alphabets."

All found vulnerabilities are not related to the protocol itself but are related to the features added in the applications. When looking at the paper [33] of de Ruiter and Poll in which they have fuzzed multiple TLS implementations, they argue that protocol state fuzzing would not find errors in the parsing of messages or deliberate backdoors. However, protocol state fuzzing could find mistakes in the state machine of an implementation. This gives way to a whole different set of vulnerabilities that are not (yet) found in one of the Signal implementations. Our attempt to fuzz the Signal Protocol will be described in chapter 5.

[8]https://www.cvedetails.com/

CHAPTER

FIVE

PROTOCOL STATE FUZZING

This chapter describes the test set-up that is used to fuzz the Signal Android application and Signal server. Moreover, it describes the alphabet and the adapter (also known as test harness) used in the test set-up. The alphabet is an abstract definition of the messages that can be sent to Signal. These abstract messages have to be parsed by the adapter to actual messages that can be understood by the Signal client and server.

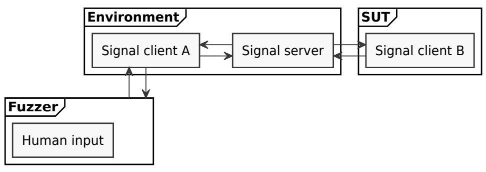
5.1 Test Set-up

To test Signal Android and Signal server the following is needed:

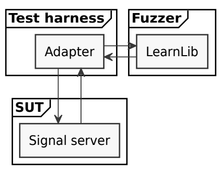
- Signal client (two, to test Signal Android only)
- Signal server
- LearnLib (Signal server only)
- Adapter (Signal server only)

Figure 5.1 shows the set-up for client and server. For Android, manual fuzzing will be used. LearnLib will be used as fuzzer for the server. LearnLib is a generation-based fuzzer which generates input from an alphabet. The alphabet is converted to actual input using the adapter. Output of the SUT is converted back to the alphabet. LearnLib can create a state machine of this input and output.

The Signal Android application could either be emulated on a desktop computer or run on an Android phone. The Signal server could run on any device supporting Java. While the Signal client can run without dependencies, the Signal server depends on AWS for cloud storage and Twilio for the phone number verification. Subscriptions have to be bought for both services to be able to start the server.



(a) Android Client



(b) Server

Figure 5.1: Test set-up architecture

5.2 LearnLib

LearnLib can infer Mealy machines from the implementation. A Mealy machine is a finite-state machine where the output values are controlled by the input values and the current state. The difference with a deterministic finite automaton (DFA) is that we specify both input and output on a transition. For example, if we are in state q_0 and there exists a transition to q_1 with the label a/x, then when we receive an a as input, the machine will output x. Formally, a Mealy machine is defined as a tuple $\mathcal{M} = \langle Q, q_0, I, O, \delta, \lambda \rangle$, where

- Q is a finite, non-empty set of states,
- $q_0 \in Q$ is an initial state,
- I is a finite set called the input alphabet,
- O is a finite set called the output alphabet,
- δ is a transition function defined as $\delta : Q \times I \rightarrow Q$, and
- λ is an output function $\lambda : Q \times I \rightarrow O$ [35].

LearnLib uses a modified algorithm of Angluin's L* algorithm to infer Mealy machine [2]. The algorithm assumes a teacher and a learner. The teacher knows of a Mealy machine $\mathcal{M}$ and the learner only knows the alphabet I and O of $\mathcal{M}$. The learner can learn the model by querying the following:

- A *reset query*. The teacher will reset its state to the initial state q_0.
- An *output query* $i \in I$. The teacher will do a transition $q \xrightarrow{i/o} q'$, where q is the current state and q' the new state. The output $o \in O$ will be returned to the learner.
- An *equivalence query* $\mathcal{H}$, where $\mathcal{H}$ is the hypothesis Mealy machine. The teacher will answer yes when $\mathcal{H} \equiv \mathcal{M}$, else the teacher will give a counterexample $\lambda_{\mathcal{M}}(a) \neq \lambda_{\mathcal{H}}(a)$, where $a \in I$ [1].

However, as the Signal applications are viewed as black-box, the equivalence queries can only be approximated. Chow's W-method executes this approximation [4]. Given a hypothesis $\mathcal{H}$, the upper bound for the W-method is the number of found states plus a specified depth. Counterexamples will, therefore, only be up to the same length as the upper bound. If no counterexamples are found, the hypothesis is assumed to be correct [33].

5.3 Hypothesis

As a finite-state machine is an abstract view of the implementation, we will have to choose the right level of abstraction to be able to generate a sensible model. Since we treat the implementation as a black-box, the key agreement and encryption would be transitions in the state machine. In other words, it is not possible to look at the internals of, e.g. the encryption function, and therefore these are not states in the state machine. The model should thus be on the same level as the Sesame algorithm; as this gives us the states in the model.

Although Mealy machines are deterministic, the hypotheses below are nondeterministic to keep the models clean. However, one should keep in mind that all possible inputs from the input alphabet I can be sent at any time. All the non-defined transitions will output an error and will not change the current state. Note that these models are *not* the hypotheses $\mathcal{H}$, but merely our expectation of what the models will look like.

5.3.1 Signal Server

The Signal server keeps track of all registered devices. A user can only register when it verifies the telephone number first. Figure 5.2 shows that it starts with requesting a verification code c. This code is sent to telephone number t. After confirming code c, one can upload its own keys k, get the pre-key bundle of user u and get the list of contacts that are also using Signal by uploading one's own hashed contact list l.

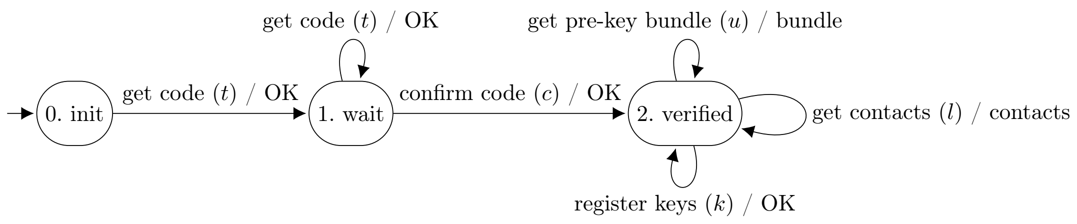


Figure 5.2: Signal server registration (Hypothesis)

In figure 5.3 we simplify registration as the 'register' transition. When the Signal server receives a message for device A, it will store this message on the server until the device fetches the message. The Signal server also keeps track of device changes. So when the server receives a message intended for device A, but the owner of device A added an new device X, then the server will reject the message. Along with the rejection, the server will pass the list of missing devices.

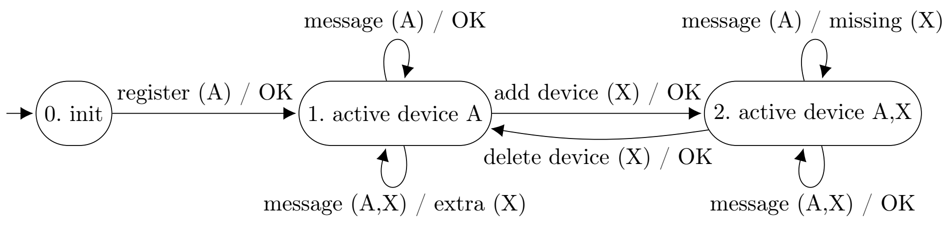


Figure 5.3: Signal server (Hypothesis)

5.4 Results

In this section, we present simplified versions of the models as generated by LearnLib. The models are simplified as otherwise they do not fit nicely on paper and are hard to read. The full versions can be found in appendix C. After every description of the model, we will summarise the differences between the model and the hypothesis.

5.4.1 Signal Server

We have edited the server code slightly to make it possible to fuzz. The server has rate limiters in place to stop an adversary from making many requests. However, this would make the model nondeterministic as after so many requests the server would give a different output for the same input. As the version of LearnLib that we are using does not support nondeterminism, we have deleted the rate limiters. Moreover, to be able to automate the registration, we send the verification code in the response body of the HTTP request instead of through the SMS service of Twilio. The responses in the model, such as `200`, `204` and `401` are HTTP status codes.

Registration

Figure 5.4 shows that the registration phase is implemented correctly. One cannot register keys or request a pre-key bundle when its device is not yet verified. This thus means that one must have a valid telephone number to register — at least when assumed the verification code cannot be obtained in any other way.

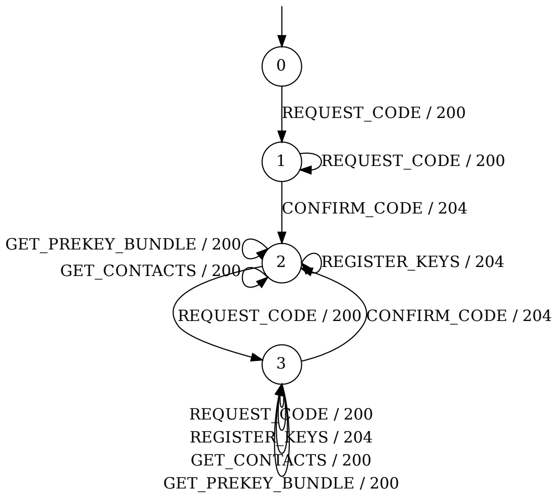


Figure 5.4: Registration

Conlusion The difference with the hypothesis from figure 5.2 is that this model has an extra state. This is because it is possible to register a new device, e.g. switching to a new telephone, which should make it possible to request a new verification code while already being registered. We did not take this situation into account in our hypothesis. This extra state gives us an extra edge case, what if the SIM card is stolen or the number gets assigned to a new person, will the new device be able to decrypt all the data? Note that the Sesame algorithm can recover from diverged sessions as described in section 2.1.4. This will be answered in section 5.4.2.

Messaging

We have generated multiple models to be able to focus on the different aspects of the Sesame algorithm. The problem with fuzzing the server is that the amount of states can be infinite as the state of the server is based on the registered devices. Therefore, there does not exist a correct model as it exists for the registration phase. The generated models look into specific situations, such as sending messages when the device does not exist or is deleted and how the protocol handles a multi-device setting.

Figure 5.5 shows how basic messaging works. The transition from state 0 to state 1 and state 2 register device A and B respectively. In state 3 both, devices are registered. In this state, all messages get accepted by the server. In state 1, messages to device B are rejected by the server with a `404` error, and in state 2 the messages for device A are rejected. One cannot find out if the device was ever registered, as after deletion (e.g. see the transition from 2 to 1) the server will also give a `404` error. Appendix C.4 shows a similar model but with more devices.

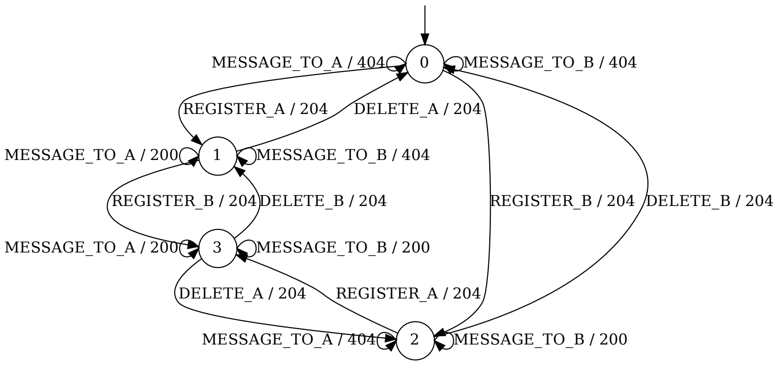


Figure 5.5: Basic Messaging

Figure 5.6 is a model where the sender tries to send a message that is meant for multiple devices to A; see section 4.3.2 for the syntax. This message gives the `409` error in state 1 and 3. The log of our adapter shows the following response `{"missingDevices":[],"extraDevices":[2]}`. It means that the server has rejected the message and that the client should try again without adding the message to the device with ID 2.

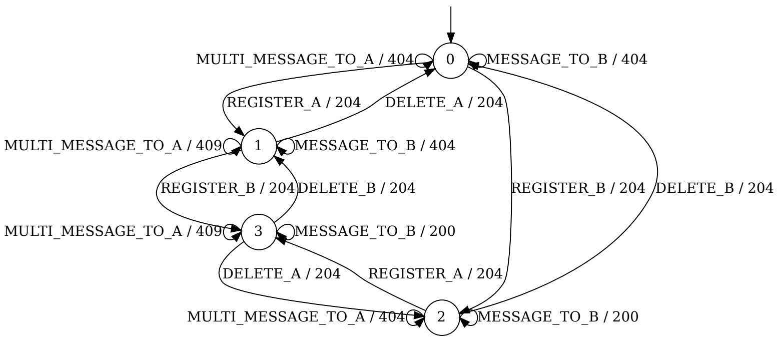


Figure 5.6: Multi-Device Messaging

Conclusion The hypothesis we have built is for one user, while the generated models are for two users. We have done this as we wanted to keep the hypothesis clean but wanted to test the server with multiple users. The transition on state 1 of the hypothesis, `message (A) / OK`, is the same as `MESSAGE_TO_A / 200` in state 1 of figure 5.5. The other transition on state 1, `message (A,X) / extra (X)`, is the same as `MULTI_MESSAGE_TO_A / 409` in state 1 of figure 5.6. We were, however, not able to reproduce `message (A) / missing (X)` from the hypothesis in the model of figure 5.6. In the `DeviceController.java` class of the server one can generate a verification code for an extra device. While after adding the second device, it is findable in the server database, one can still not send messages to this device.

5.4.2 Signal Android

We use manual fuzzing to fuzz the Android client, as using the automated tool LearnLib is infeasible to test features from the Sesame algorithm. One should automate the process of starting the application, typing in the telephone number, entering the verification code, sending messages to another user, deleting the account and clearing the cache of the Android app. For manual fuzzing, we have done all these steps by hand.

Session Restoration

The Sesame algorithm should be able to recover from diverged sessions, as stated in the paper of Marlinspike and Perrin [28] and discussed in section 2.1.4. The implementation of this feature is crucial as a wrong implementation could leak messages to another user. For example, when the SIM card is stolen or the telephone number is reassigned and the session is restored, then messages will leak — see also section 5.4.1.

For the session restoration of the Android client, there are four distinct cases, namely

1. restoring from backup with account deletion,
2. restoring without backup with account deletion,
3. restoring from backup without account deletion,
4. restoring without backup without account deletion.

To analyse the first case, we have registered two accounts at the Signal server. We then started a conversation by sending some messages from both sides. On one of the accounts we enabled the backup function by going to the settings `Chats and media > Chat backups`. After the backup has finished, we deleted the account by disabling `Advanced > Signal messages and calls`. When the account is deleted, we send messages with the other telephone. The messages are now unable to be sent, and the client receives the notification ‘sending has failed’ — which means there is no need for session restoration.

For the second case, we first cleaned the database and Android phones. We then registered two accounts as in the first case. Starting a session by sending some messages. This time, we did not enable the backup function, but immediately deleted the account. After deletion, the other device again received the notification ‘sending has failed’ when trying to send a message.

In the third case, we started similar to the second case, cleaning, registering and starting the conversation. We enabled the backup function, and after the backup was finished, we deleted the Android application without deleting the account. Now, when sending a message with the other device, messages can be sent but are not delivered. After sending several messages, we reinstalled the Signal application and restored from backup. However, unexpectedly, the messages are not delivered after reinstallation.

For the final, fourth case, we started in the same fashion as the earlier cases. We did *not* enable the backup function, but deleted the Android application after starting the conversation. Just like in the third case, when sending some messages with the other device, the messages are not delivered. After reinstalling the Signal application, messages sent during this inactive period are not delivered. Moreover, the other device will receive a warning that the key of the other party has changed. The sender should confirm before messages can be sent again.

To summarise, in the first case, messages received during the period Signal was not active, i.e. deleted from the Android client, will fail to deliver. The sender will receive a notification ‘sending has failed’. The second case yields the same result as the first case. At the third case, messages received during the inactive period will be lost. Unlike the first two cases, a user can still send messages to the client; however, the messages will never be received. In the final case, messages will be lost, just like in the third case. After re-registration, the sender will receive a warning that the key of the other party has changed.

This analysis shows that the session recovery as described in the Sesame algorithm paper [28] is not implemented in the Android client. Something which is, in essence, wrong, as the Signal implementation is thus not the same as documented. However, the feature does give a new attack surface, which is at this moment not present in Signal Android. This attack is not explained in the paper, nor is the mitigation of this attack given. Therefore, an implementation that does want to implement the Sesame algorithm must be aware of the problem as described at the beginning of this section.

CHAPTER

SIX

DISCUSSION

In this chapter we discuss our observations and their limitations.

6.1 Observations

Our major observation is that not all the properties of the Signal Protocol are met in practice. Even for some of the properties that are met, there are some caveats. Furthermore, we found that the Sesame algorithm is not implemented in the Signal Android application like Marlinspike and Perrin documented it [28]. We first describe the observations and then discuss the solutions.

6.1.1 Deniability

The deniability property is not met. As found by Frosch et al. (section 3.1.1), even though symmetric cryptography is used to encrypt messages, deniability is not met as a user still authenticates to the server and thus cannot deny involvement in a conversation. Of course, Open Whisper Systems could promise not to log any of this data, though this does not give the guarantee that the deniability property requires.

Sealed sender as described in section 2.1.5 looks promising to resolve this problem. The feature has been introduced in October 2018, but it has not been formally analysed yet.

6.1.2 Forward and Future Secrecy

Although (a weaker version of) forward secrecy and future secrecy are met, supported by research of Frosch et al. (section 3.1.1) and Rubín (section 3.1.3), we argue that this conclusion is too simple when looking at the implementation. First, let us look again how Frosch et al. argue that keys can be revealed, "When claiming perfect forward secrecy for TextSecure, we first have to define which keys could be revealed. Since the TextSecure server does not store any private or secret end-to-end communication keys, those keys can only be revealed from a TextSecure app". They then show how in some particular cases keys could be extracted as they are not yet deleted from the device — thus breaking forward secrecy. However, they forget to note that on most devices, erased data is not really erased.

It does sound odd that an adversary comprises a device and then tries to retrieve the private keys instead of just reading the messages on the phone. Therefore, we argue that the notion of forward secrecy does not mean a lot when having access to the device itself. In this same way forward secrecy can also be easily bypassed by the organisation that created the message application. For instance, the organisation could perform analyses of the messages on the client-side where messages are in plain-text. Then metadata could be inferred from the messages and sent back to the server. This will still guarantee forward and future secrecy, but will defy the meaning of end-to-end encryption.

We, therefore, advocate for a stronger notion of both forward secrecy and future secrecy that deals with this situation. We define strong forward secrecy as “All messages that are sent before message m cannot be retrieved when message m is compromised”, and strong future secrecy as “All messages m that are sent after message m cannot be retrieved when message m is compromised”. Meaning for forward secrecy that messages and keys must not be retrievable in any way after erasure. It also implies that only one message and one key can be stored at a time. For future secrecy, it means the device must be able to recover from a device compromise after every message. These two notions, however, sound too strict and do not deal well with the example above. We could also keep the current notion and add that the device is uncompromisable and no information from the messages is inferred. This does not give us the restriction of having to erase messages and keys instantly; however, this does require special hardware[1].

6.1.3 Sesame Algorithm

The basic components of the Sesame algorithm are implemented as documented by Marlinspike and Perrin [28]. One can only retrieve a pre-key bundle of a user or send messages once registered. Messages that are sent to non-existing users or to wrong devices are rejected appropriately.

The session recovery mechanism of the Sesame algorithm is not implemented. On one side this means information is not able to leak to an adversary, but on the other side, it means that we were not able to analyse the implications of such implementation, see section 5.4.2.

6.2 Threats to Validity

In our thesis, we focused on the higher level layers, so parts of the lower level remain untouched, such as the elliptic curve implementation. However, the thesis of Rubín does focus on the lower layers. Our research is thus a good addition to the existing implementation analysis.

For protocol state fuzzing, one should choose the ‘right’ level of abstraction, which is not a trivial task as it mainly depends on the purpose of the research. The abstraction level is reflected in the chosen alphabet. This means that one should be careful that all the necessary inputs and outputs are defined; otherwise, some cases will not show up in the finite-state machine. Therefore, the chosen abstraction level, the alphabet and the interpretation of the model are very important to assess the value of the research.

We did not look at group chats, the iOS and desktop codebase, nor the implementation features of Signal such as sealed sender and the private contact discovery service. Research on these parts may influence the observations. As written at the beginning of this chapter, sealed sender may solve the problems with deniability.

[1]Tinfoil Chat is an application that meets the requirements with exception of physical compromise https://github.com/maqp/tfc

CHAPTER
SEVEN

FUTURE WORK

In this chapter, we describe the most relevant future work that could still be done in this field.

7.1 Desktop and iOS Client

Most of the research, including this paper, focuses on the Android version of Signal. Notably, the paper of Frosch et al. [12], Cohn-Gordon et al. [5], and Rubín [32] are based on this version. Therefore, analysing the desktop or iOS version could help to get a full picture of Signal.

7.2 Fuzzing on a Different Level

In this thesis, we have focused mostly on the Sesame algorithm. One could also take another approach and search for vulnerabilities in one of the implementations. An easy target is the Unicode renderer, which has been vulnerable for XSS attacks before — see *CVE-2018-11101*[1]. Also, the image and video viewer could be fuzzed. Open Whisper Systems recently even made the attack surface bigger by adding support to search for GIFs online[2] and previewing links[3]. The buffer overflow in the Voice over Internet Protocol (VoIP) stack of WhatsApp targeted all platforms (*CVE-2019-3568*[4]). Signal also offers this service and fuzzing should be able to find buffer overflows with the right toolset as described in section 3.2.2. So it is interesting to search for similar kind of vulnerabilities in Signal.

7.3 Signal's New Features

Signal brings out new features from time to time. Two of the most interesting features are sealed sender and private contact discovery service. Sealed sender is a feature to hide the sender from the Signal server. Normally, a Signal client will connect and authenticate to the server over TLS. This authentication, however, gives the problem that the server can learn who is talking to whom. Sealed sender is designed to hide the sender, but also to prevent spoofing and still give control to the server to apply abuse protection.

[1] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11101
[2] https://signal.org/blog/giphy-experiment/
[3] https://signal.org/blog/i-link-therefore-i-am/
[4] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-3568

Private contact discovery service is a service to find friends on Signal without Signal needing to have access to all contacts in the address book. For this to work the Signal server uses a secure enclave, which allows performing computations in this enclave without Signal being able to learn anything about it. The Signal client can even test if the source code in the enclave is the same as the open-source version. The enclave then finds all the contacts of the user in the list of registered users and sends this result encrypted to the client.

Both features are not trivial to implement and would benefit from some more in-depth analysis. Problems with the sealed sender could, for example, be that IP addresses still leak. Questions arise from using the secure enclave, how does the list of registered users get into the enclave and are there any side-channel attacks possible?

7.4 Key Compromise

Frosch et al. [12] describe that in special cases forward secrecy can be broken by retrieving the not yet deleted keys from a Signal client. A future research project could test the feasibility of this attack by trying to retrieve the private keys from the device. This test also gives an even better insight into the key deletion policy of Signal. Another research question could be, are the keys still retrievable after erasure?

7.5 Fetching and Dropping Messages

There are two questions not yet answered in this thesis due to time constraints. One of them is, how does Signal Android fetches messages from the server, does the server send the messages or does the client poll for messages? In appendix B.3 one can see that on registration the variable `fetchesMessages` can be set to either true or false. An interesting scenario would be when Signal Android polls for new messages and authenticates with the authorisation code generated at registration. When an adversary intercepts this authorisation code, the adversary could use the code to authenticate to the server. The adversary can then fetch messages from the server, which will be deleted on retrieval. This attack will thus lead to lost messages for the user.

The second question is, what happens when messages are lost? In section 2.1.3, we have seen that keys to encrypt messages depend on the parent key. What if a previous message with this parent key gets lost? One would, in theory, not be able to perform a ratchet step. For instance, when the previous shared secret key is lost, the next could not be calculated with a Diffie-Hellman ratchet as it depends on the previous shared secret key, see also figure 2.8. One could research what happens in practice by dropping some of the messages and look if some messages could not be decrypted.

7.6 Available Documentation and Code

Further research could benefit from the documentation and code written for this project. All can be found on GitLab, `https://gitlab.com/dvandam_thesis`. Due to lack of documentation, much time was spend to set up a Signal server. Therefore, we forked the project and added documentation and a sample configuration, which can be found under the name *Signal Server v1.88*. On this same GitLab page, the configuration for protocol state fuzzing can be found, called *Statelearner*. Furthermore, the attempt to fuzz Signal Android with LearnLib can be found in the repository called *Signal Adapter v4.25.10*; it contains a working server to receive LearnLib messages and an auto-typer to simulate a user typing messages. Full quality images of the models and diagrams can be found under *Signal Protocol — MAIN*.

CHAPTER
EIGHT

CONCLUSION

In this thesis, we have answered two research questions. “What are the properties of the Signal Protocol and are these still valid in the implementation?”, which we have answered by analysing existing literature and the implementation analysis of chapter 4. The second question, “How can the high-level protocol be modelled such that it gives a better understanding of the implementation?”, we have answered by fuzzing and analysing the model (chapter 5).

We have found, by analysing existing literature, that deniability is not satisfied. Furthermore, by analysing the implementation, we found that the notion of forward secrecy and future secrecy is too weak when looking at the code. Previous research stated that in exceptional cases, forward secrecy was broken as multiple keys were kept in storage. We argue that there is a more significant problem, the first being that erasure on most systems only erases the link to the address and not the data itself. More importantly, when having access to the device, one could already read all the messages. Therefore, we have introduced a stronger notion in chapter 6 that also includes device compromise.

By using manual fuzzing we have observed that the Sesame algorithm is not entirely implemented. The mechanism to recover from diverged sessions is lacking. From a functionality perspective, it means Signal is missing one of their documented features. With Signal’s focus on ease of use, it seems logical to implement this feature as it prevents loss of messages. However, from a security perspective, it makes sense not to implement this feature or to implement it in such a way that an adversary cannot abuse it — implying such an implementation exists.

We have also figured that the usefulness of protocol state fuzzing is very depended on the ability to automate the process. Protocol state fuzzing is easy to execute on the server-side as we can send messages by using the API and we can restart the process by dropping the database with only a few lines of code. To fuzz the client-side, one needs to be able to mimic a user and be able to clear the cache of the Android client, which is much harder to set up.

Future work could be conducted on both the iOS version and the desktop version, as these are less analysed in the literature. Fuzzing could be used to analyse, e.g. the Unicode renderer, the JPEG viewer and the video calling service. Implementation analysis could be done on the new features sealed sender and the private contact discovery service. Also, to research the feasibility of compromising forward secrecy as described by Frosch et al., one could try to compromise the Signal keys from a device.

APPENDIX

ONE

DATABASE STRUCTURE

A.1 Public Information

Information stored in the database for each person that registers at Signal. See section 4.3.1.

```
(id, number, data)
2 +27783XXXXXX
{
  "number":"+27783XXXXXX",
  "devices":
  [{
    "id":1,
    "name":null,
    "authToken":"ec69925615b61656e9dfcf8d21365f6d0d47a0a9",
    "salt":"184883909",
    "signalingKey":
      "NMAEPTbvVFz4xcqnZuIRP1i3T7JMFQeRLL3XH
       ilquLLjw/7WXGY6+rwJISiUAWiRK/936A==",
    "gcmId":
      "APA91bEqQMIDee0Dqi9VxeCYTMmQ9lQYbA7mN
       AEfwkfs0Dk2o1P-nm6XKcxXsi1JrYlLhEgn2A
       4-Bp0PfwquifrceGLnRqB3w-_I_n4fIhdQm_-
       X-yldxkd0Dam0jcAA6iCd7LjMJ-X0",
    "apnId":null,
    "voipApnId":null,
    "pushTimestamp":1562314176636,
    "fetchesMessages":true,
    "registrationId":5615,
    "signedPreKey":
    {
      "keyId":1251893,
      "publicKey":
        "BWWeP5LtpwJghIHcDxhf7riStbUesdykux71+ayAL4Ay",
      "signature":
        "yVVTngxTIo+1vGN5b9Wk+O0N9aZ+RFQJj9+PVv0eiy1/
         Z05l+xl7r1WxiTiSkzGEN25QIoaLNOF1Nr6ldfIbDg"
    },
    "lastSeen":1562284800000,
    "created":1562314174141,
    "voice":true,
    "video":true,
    "userAgent":"OWA"
  }],
  "identityKey":"BbnlEWBP+qfztfnACDimGbswZZp0a2wxPCpKtm6bE3tc",
  "name":
    "6mJtQs6LD1mEKbyazxPwj9xxFFUZ8SeAqcvHw
     4fP3Toh9mw8mK94JWVXRXgfmXIyX/jSzVq/",
  "avatar":"profiles/B0VsDmc0kz_80e6jj01LVg",
  "avatarDigest":null,
  "pin":null
}
```

A.2 Public Keys

Table containing all the public keys of every device registered at Signal. See section 4.2.1.

```
(id, number, key_id, public_key, last_resort, device_id)

2   +31611XXXXXX 12056054 BX0hNur2vN6kHUH6poHLWILAkYAAl0NFB4DFdW3VTRAv 0 1
3   +31611XXXXXX 12056055 BVzoNI69s3wpGyz0E2KSLHwHglqULGSVgrQq2mzVRL0Y 0 1
4   +31611XXXXXX 12056056 Bd7fuUxl0Cy60jClv8vz+cJ0DgnpugIb0pkLkia1r5sF 0 1
5   +31611XXXXXX 12056057 BbRf3JW2RSQNZ/y1ZzbIVUA7FPQ7m83eSjD956LEkC4a 0 1
...
96  +31611XXXXXX 12056148 BSaZEfXA8HciaPYt8fe2NNflA0lyvPn9AAmLCMYF30sE 0 1
97  +31611XXXXXX 12056149 BaYmSvv0rpi/uBd8Sn1avJsMW+Yxfob95UL054sVhiZP 0 1
98  +31611XXXXXX 12056150 Bc/Hb8ETpa0y/ZlFLh5x8MghMsSgVvChgbHTN0NLhCVM 0 1
99  +31611XXXXXX 12056151 BYNKeZ8Yhb3rHT0sPSC3bBH04mildchgZ/4BDOQHr5d0 0 1
100 +31611XXXXXX 12056152 BT904i4H+wBi909FeaYLwUzmpjz/RC2U2Mn5mTLzRKY5 0 1

102 +27783XXXXXX 3244537  BTQ+JA4gtCYvXtAndEQQpDg2xi71bEKOCrPyUKKZh5Ir 0 1
103 +27783XXXXXX 3244538  BUxtxJZK5Wm8kB7MM6VroeqeuBijnLP5VP24T2jMsypi 0 1
104 +27783XXXXXX 3244539  BabYl5+Ito1QwLqlsUWMFTPW02CSFkHfbB0+KpYoHHFu 0 1
105 +27783XXXXXX 3244540  BeNj/U+xyzs1ATdIPGq9hpk07k4h1n2UQob+Xeht3jwc 0 1
...
196 +27783XXXXXX 3244631  BT59fA6CW400MxwEA1E7DkYPgbGJaaeuzGvjNx/K4Wg0 0 1
197 +27783XXXXXX 3244632  BWW1QZ/iktEnZ0zTl3C8mBTr94WtrxXeXk0MU0RYXeNW 0 1
198 +27783XXXXXX 3244633  BdH67vDHTAAwSlXMG24bc/6eUE2A9P/q6WgtPj+EIqRI 0 1
199 +27783XXXXXX 3244634  BQTo1AMBatxYcst14+PwQ6xs9EmRMFJH7gay3NXLmGYp 0 1
200 +27783XXXXXX 3244635  BVQeyJir64Wns9zzHAJsIIfXWwViDRICXnVcfMNAFrsX 0 1
```

APPENDIX
TWO

MESSAGE STRUCTURE

B.1 Server Log

Log showing the kind of information that the Signal server logs. The log contains all the requests done to the server. See section 4.3.

```
"PUT /v2/keys/signed HTTP/1.0" 401 49 "-" "okhttp/3.9.0" 89
"PUT /v1/directory/tokens HTTP/1.0" 401 49 "-" "okhttp/3.9.0" 13
"PUT /v1/accounts/gcm/ HTTP/1.0" 401 49 "-" "okhttp/3.9.0" 19
"PUT /v1/directory/tokens HTTP/1.0" 401 49 "-" "okhttp/3.9.0" 9
"GET /v1/accounts/sms/code/+27783XXXXXX HTTP/1.0" 200 0 "-" "okhttp
    ↪ /3.9.0" 3323
"PUT /v1/accounts/code/463343 HTTP/1.0" 204 0 "-" "okhttp/3.9.0" 308
"PUT /v2/keys/ HTTP/1.0" 204 0 "-" "okhttp/3.9.0" 606
"PUT /v1/accounts/gcm/ HTTP/1.0" 204 0 "-" "okhttp/3.9.0" 301
"PUT /v1/directory/tokens HTTP/1.0" 200 120 "-" "okhttp/3.9.0" 89
"PUT /v1/profile/name/ClwolzrHTsQNPVMhxNEnM8jGtH8zIukdjVvfjp%2
    ↪ FDRmPD73KLRh2Lt7qsjHupkscqHR7SaR1P HTTP/1.0" 204 0 "-" "okhttp
    ↪ /3.9.0" 258
"GET /v1/profile/form/avatar HTTP/1.0" 200 795 "-" "okhttp/3.9.0" 232
"GET /v1/websocket/?login=+27783XXXXXX&password=HypBjFZKf7znjoUqNPDaoXhw
    ↪ HTTP/1.1" 101 0 "-" "okhttp/3.9.0" 66
"GET /v1/profile/+31611XXXXXX HTTP/1.0" 200 0 "-" "-" 0
"GET /v2/keys/+31611XXXXXX/* HTTP/1.0" 200 319 "-" "okhttp/3.9.0" 36
"PUT /v1/messages/+31611XXXXXX HTTP/1.0" 200 0 "-" "-" 0
org.whispersystems.textsecuregcm.push.GCMSender: GCM Failed: org.
    ↪ whispersystems.gcm.server.AuthenticationFailedException
```

B.2 Request a Verification Code

The GET request to ask a verification code by SMS. See section 4.3.1.

```
GET /v1/accounts/sms/code/+27783XXXXXX HTTP/1.0

X-Signal-Agent: OWA
Accept-Encoding: gzip
User-Agent: okhttp/3.9.0
```

B.3 Confirm a Verification Code

The PUT request to confirm the verification code received by either SMS or call. See section 4.3.1.

```
PUT /v1/accounts/code/284920 HTTP/1.0

Authorization: Basic KzI3NzgzWFhYWFhYOkFnNXNEczZUZ29DMWRvRmU5alpZMkRMUw==
X-Signal-Agent: OWA
Content-Type: application/json; charset=utf-8
Accept-Encoding: gzip
User-Agent: okhttp/3.9.0

{
  "fetchesMessages" : false,
  "pin" : null,
  "registrationId" : 5615,
  "signalingKey" : "NMAEPTbvVFz4xcqnZuIRP1i3T7JMFQeRLL3XHilquLLjw/7WXGY6+
     ↪ rwJISiUAWiRK/936A==",
  "video" : true,
  "voice" : true
}
```

B.4 Registering Keys

The PUT request to register the keys generated by the Signal client at the server. See section 4.2.1 and 4.3.1.

```
PUT /v2/keys/ HTTP/1.0

Authorization: Basic KzI3NzgzWFhYWFhYOkFnNXNEczZUZ29DMWRvRmU5alpZMkRMUw==
X-Signal-Agent: OWA
Content-Type: application/json; charset=utf-8
Accept-Encoding: gzip
User-Agent: okhttp/3.9.0

{
  "identityKey":"BaUQEan9RhY/0ina2Ey8GTbAICWCLifMmwphLpXouYsC",
  "preKeys":
  [
    {"keyId" : 11494154, "publicKey" : "BeKIi8JcPDJtUEj5E9o2iILZVOvxLHp
       ↪ +/2pQgUmrvTOG"},
    {"keyId" : 11494155, "publicKey" : "BXLGjH/+kCMfoZQWkcxeplNOS3oqzel/7
       ↪ fXkzhO+04lN"},
    {"keyId" : 11494156, "publicKey" : "BaaybIOIkJucecjAL/
       ↪ WJb59P77KvCtHddSpYnSvaEDId"},
    ...
  ],
  "signedPreKey":
  {
    "keyId" : 15273084,
    "publicKey" : "BXOhm+95qNypSut1WwiGk1ZGH8oHaZIqFIBqXlYSbyxD",
    "signature" : "MszpBIcvcpEEU0iBpM/t//
       ↪ xLmTabIaBxWUMySnLPlxA1f6ACWKX89T1je/
       ↪ KJQWZ7KY57dX7QJOgkZoqYQzqIAw"
  }
}
```

B.5 Retrieve Contacts

The PUT request to retrieve the contacts that are also using Signal. See section 4.3.1.

```
PUT /v1/directory/tokens HTTP/1.0

Authorization: Basic KzI3NzgzWFhYWFhYOkFnNXNEczZUZ29DMWRvRmU5alpZMkRMUw==
X-Signal-Agent: OWA
Content-Type: application/json; charset=utf-8
Accept-Encoding: gzip
User-Agent: okhttp/3.9.0

{"contacts":["tOLmkASkS/Ogfg","MBd3hr+qHflKOA","131SHxvDHqkZeg"]}
```

B.6 Retrieve Pre-key Bundle

The GET request to retrieve the pre-key bundle from another Signal user. See section 4.3.1.

```
GET /v2/keys/+31611XXXXXX/* HTTP/1.0

Authorization: Basic KzI3NzgzWFhYWFhYOkFnNXNEczZUZ29DMWRvRmU5alpZMkRMUw==
X-Signal-Agent: OWA
Accept-Encoding: gzip
User-Agent: okhttp/3.9.0

SERVER RESPONSE:
{
  "identityKey":"Bbqgqsljd0xcMK8zCU/Ex12/A0OQAiakaxHg8pXd85BW",
  "devices":
  [
    {
      "deviceId":1,
      "registrationId":13975,
      "signedPreKey":
      {
        "keyId":4590740,
        "publicKey":"BfSBjiwTiiShoQb3PuTKHANgNI+KCSiDHfL6cmetFO0d",
        "signature":"yrpFYdPCl0d9cDsLVUm9JQPlIp5APKNuvHdKnVS
           Oab7m6JuMNe6Xp1LcQMDG4mu3cRfpirCZ2aT7FQtAxVLWBA"
      },
      "preKey":
      {
        "keyId":12146509,
        "publicKey":"BRszJOJohHaI7zDMM3h3o95z/tllmqPmnOd7SjbzD41n"
      }
    }
  ]
}
```

APPENDIX

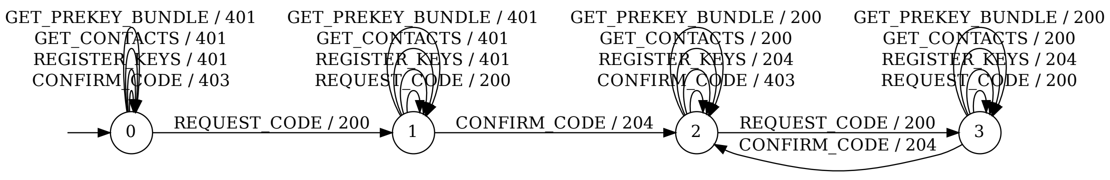
THREE

MODELS

The models are also available on https://gitlab.com/dvandam_thesis/main/tree/master/models.

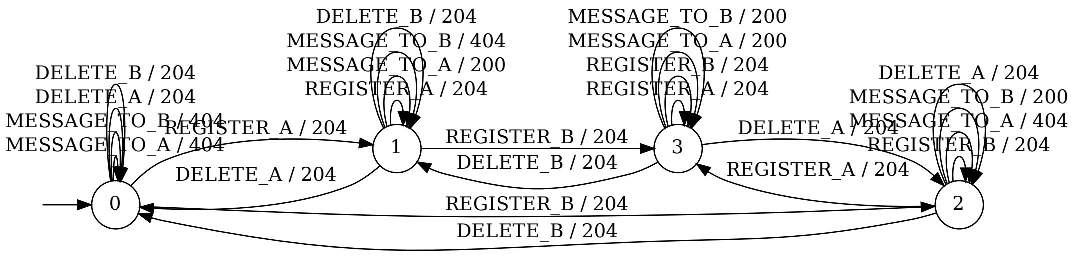
C.1 Signal Server Registration

Model that describes the registration phase when a Signal client wants to register. See section 5.4.1.



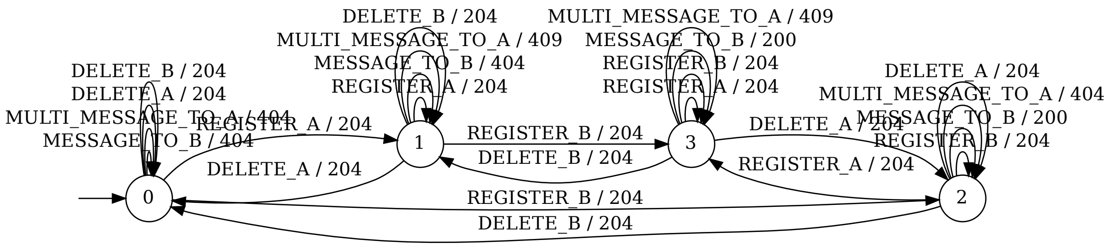
C.2 Signal Server Messaging (2 devices)

Model that describes the possible states of the server when two clients are connected. See section 5.4.1.



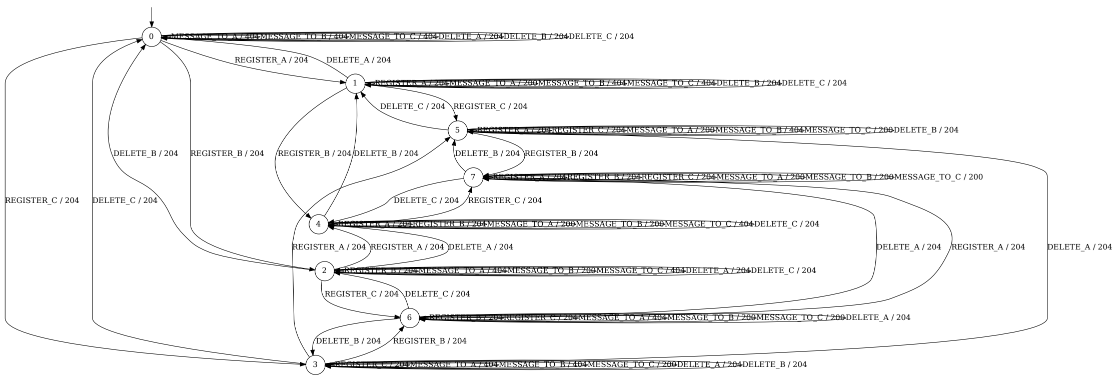
C.3 Signal Server Multi-Device Messaging (2 devices)

Model that describes the possible states of the server when two client, one with multiple devices, are connected. See section 5.4.1.



C.4 Signal Server Messaging (3 devices)

Model that describes the possible states of the server when three clients are connected. See section 5.4.1.


0
MESSAGE_TO_A / 404MESSAGE_TO_B / 404MESSAGE_TO_C / 404DELETE_A / 204DELETE_B / 204DELETE_C / 204
REGISTER_A / 204
DELETE_A / 204
1
REGISTER_A / 204MESSAGE_TO_A / 200MESSAGE_TO_B / 404MESSAGE_TO_C / 404DELETE_B / 204DELETE_C / 204
DELETE_C / 204
REGISTER_C / 204
5
REGISTER_A / 204REGISTER_C / 204MESSAGE_TO_A / 200MESSAGE_TO_B / 404MESSAGE_TO_C / 200DELETE_B / 204
DELETE_B / 204
REGISTER_B / 204
REGISTER_B / 204
DELETE_B / 204
7
REGISTER_A / 204REGISTER_B / 204REGISTER_C / 204MESSAGE_TO_A / 200MESSAGE_TO_B / 200MESSAGE_TO_C / 200
DELETE_C / 204
REGISTER_C / 204
4
REGISTER_A / 204REGISTER_B / 204MESSAGE_TO_A / 200MESSAGE_TO_B / 200MESSAGE_TO_C / 404DELETE_C / 204
REGISTER_A / 204
REGISTER_A / 204
DELETE_A / 204
2
REGISTER_B / 204MESSAGE_TO_A / 404MESSAGE_TO_B / 200MESSAGE_TO_C / 404DELETE_A / 204DELETE_C / 204
REGISTER_C / 204
DELETE_C / 204
6
REGISTER_B / 204REGISTER_C / 204MESSAGE_TO_A / 404MESSAGE_TO_B / 200MESSAGE_TO_C / 200DELETE_A / 204
DELETE_B / 204
REGISTER_B / 204
3
REGISTER_C / 204MESSAGE_TO_A / 404MESSAGE_TO_B / 404MESSAGE_TO_C / 200DELETE_A / 204DELETE_B / 204
REGISTER_C / 204
DELETE_C / 204
DELETE_B / 204
REGISTER_B / 204
DELETE_A / 204
REGISTER_A / 204
DELETE_A / 204

BIBLIOGRAPHY

[1] Fides Aarts, Joeri De Ruiter and Erik Poll. 'Formal Models of Bank Cards for Free'. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, Mar. 2013, pp. 461–468. ISBN: 978-0-7695-4993-4. DOI: 10.1109/ICSTW.2013.60.

[2] Dana Angluin. 'Learning regular sets from queries and counterexamples'. In: *Information and Computation* 75.2 (1987), pp. 87–106. ISSN: 10902651. DOI: 10.1016/0890-5401(87)90052-6.

[3] Simon Blake-Wilson and Alfred Menezes. 'Unknown Key-Share Attacks on the Station-to-Station (STS) Protocol'. In: *Lecture Notes in Comptuer Science*. 1999, pp. 154–170. DOI: 10.1007/3-540-49162-7_12.

[4] T.S. Chow. 'Testing Software Design Modeled by Finite-State Machines'. In: *IEEE Transactions on Software Engineering* SE-4.3 (May 1978), pp. 178–187. ISSN: 0098-5589. DOI: 10.1109/TSE.1978.231496.

[5] Katriel Cohn-Gordon et al. 'A Formal Security Analysis of the Signal Messaging Protocol'. In: *Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017* November (2017), pp. 451–466. ISSN: 13484214. DOI: 10.1109/EuroSP.2017.27.

[6] Josh Constine. *WhatsApp hits 1.5 billion monthly users. $19B? Not so bad.* Jan. 2018. URL: https://techcrunch.com/2018/01/31/whatsapp-hits-1-5-billion-monthly-users-19b-not-so-bad/.

[7] W. Diffie and M. Hellman. 'New directions in cryptography'. In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654. ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638.

[8] Max Eddy. *RedPhone (for Android)*. Sept. 2014. URL: https://www.pcmag.com/review/308056/redphone-for-android.

[9] Ksenia Ermoshina, Francesca Musiani and Harry Halpin. 'End-to-End Encrypted Messaging Protocols: An Overview'. In: *Springer*. Vol. 275. 5304. 2016, pp. 244–254. ISBN: 978-3-319-70283-4. DOI: 10.1007/978-3-319-45982-0_22.

[10] Facebook. *Secret Conversations*. URL: https://www.facebook.com/help/messenger-app/1084673321594605/ (visited on 05/02/2019).

[11] Paul Fiterău-Broştean et al. 'Model learning and model checking of SSH implementations'. In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software - SPIN 2017* (2017), pp. 142–151. DOI: 10.1145/3092282.3092289. URL: http://dl.acm.org/citation.cfm?doid=3092282.3092289.

[12] Tilman Frosch et al. 'How Secure is TextSecure?' In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Mar. 2016, pp. 457–472. ISBN: 978-1-5090-1751-5. DOI: 10.1109/EuroSP.2016.41.

[13] Caleb Garling. *Twitter Buys Some Middle East Moxie*. Nov. 2011. URL: https://www.wired.com/2011/11/twitter-buys-moxie/.

[14] Caleb Garling. *Twitter Open Sources Its Android Moxie*. Dec. 2011. URL: https://www.wired.com/2011/12/twitter-open-sources-its-android-moxie/.

[15] Patrice Godefroid, Michael Y Levin and David Molnar. 'SAGE: Whitebox Fuzzing for Security Testing'. In: *Communications of the ACM* 55.3 (Mar. 2012), p. 40. ISSN: 00010782. DOI: 10.1145/2093548.2093564.

[16] Andy Greenberg. *Android App Aims to Allow Wiretap-Proof Cell Phone Calls*. May 2010. URL: https://www.forbes.com/sites/firewall/2010/05/25/android-app-aims-to-allow-wiretap-proof-cell-phone-calls/.

[17] Bart Jacobs and Joan Daemen. *Computer Security : Public Key Crypto*. 2016. URL: https://www.sos.cs.ru.nl/applications/courses/security2016/asymmetric.pdf.

[18] Nadim Kobeissi, Karthikeyan Bhargavan and Bruno Blanchet. 'Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach'. In: *Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017* (2017), pp. 435–450. DOI: 10.1109/EuroSP.2017.38.

[19] Hugo Krawczyk and Pasi Eronen. 'HMAC-based Extract-and-Expand Key Derivation Function (HKDF)'. In: (2010). URL: https://www.ietf.org/rfc/rfc5869.txt.

[20] Moxie Marlinspike. *A New Home*. Jan. 2013. URL: https://signal.org/blog/welcome/ (visited on 07/02/2019).

[21] Moxie Marlinspike. *Advanced cryptographic ratcheting*. 2013. URL: https://signal.org/blog/advanced-ratcheting/ (visited on 07/02/2019).

[22] Moxie Marlinspike. *Contributing to Signal Android*. 2018. URL: https://github.com/signalapp/Signal-Android/blob/3f9ddaf409f50b3730e95756a172c3551f7e4839/CONTRIBUTING.md (visited on 04/04/2019).

[23] Moxie Marlinspike. *Free, Worldwide, Encrypted Phone Calls for iPhone*. 2014. URL: https://signal.org/blog/signal/ (visited on 07/02/2019).

[24] Moxie Marlinspike. *Just Signal*. 2015. URL: https://signal.org/blog/just-signal/ (visited on 07/02/2019).

[25] Moxie Marlinspike. *Signal Server Config*. 2018. URL: https://github.com/signalapp/Signal-Server/blob/8d72515a3098b5c634c19442a88aab17b9ac9508/config/sample.yml (visited on 08/04/2019).

[26] Moxie Marlinspike. *Technology preview: Private contact discovery for Signal*. Sept. 2017. URL: https://signal.org/blog/private-contact-discovery/.

[27] Moxie Marlinspike and Trevor Perrin. *Double Ratchet Algorithm*. Tech. rep. 2016, p. 35. URL: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[28] Moxie Marlinspike and Trevor Perrin. 'The Sesame Algorithm: Session Management for Asynchronous Message Encryption'. In: *Signal* (2017). URL: https://signal.org/docs/specifications/sesame/sesame.pdf.

[29] Moxie Marlinspike and Trevor Perrin. *The X3DH Key Agreement Protocol*. Tech. rep. 2016, p. 11. URL: https://www.whispersystems.org/docs/specifications/x3dh/.

[30] Open Whisper Systems. *Technology preview : Sealed sender for Signal*. Oct. 2018. URL: https://signal.org/blog/sealed-sender/.

[31] Erik Poll. *Fuzzing*. Nijmegen, 2018. URL: https://cs.ru.nl/E.Poll/ss/slides/11%7B%5C_%7DFuzzing.pdf.

[32] Jan Rubín. 'Security Analysis of the Signal Protocol'. MA thesis. Czech Technical University in Prague, 2018, p. 85. URL: https://dspace.cvut.cz/bitstream/handle/10467/76230/F8-DP-2018-Rubin-Jan-thesis.pdf.

[33] Joeri De Ruiter and Erik Poll. 'Protocol State Fuzzing of TLS Implementations'. In: *USENIX Security Symposium* (2015), pp. 193–206. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter.

[34] United Nations General Assembly. 'Universal Declaration of Human Rights Preamble'. In: *United Nations Declaration of Human Rights* (1948), pp. 1–8. ISSN: 1364-2987. DOI: 10.1080/13642989808406748. arXiv: arXiv:1011.1669v3.

[35] Frits Vaandrager. 'Model learning'. In: *Communications of the ACM* 60.2 (Jan. 2017), pp. 86–95. ISSN: 00010782. DOI: 10.1145/2967606. URL: https://dl.acm.org/citation.cfm?id=2967606%20http://dl.acm.org/citation.cfm?doid=3042068.2967606.

[36] Kurt Wagner. *WhatsApp is at risk in India. So are free speech and encryption.* Feb. 2019. URL: https://www.recode.net/2019/2/19/18224084/india-intermediary-guidelines-laws-free-speech-encryption-whatsapp.

[37] WhatsApp. *WhatsApp Encryption Overview*. Tech. rep. WhatsApp, 2017. URL: https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.

[38] David A. Wheeler. 'Preventing Heartbleed'. In: *Computer* 47.8 (2014), pp. 80–83. ISSN: 00189162. DOI: 10.1109/MC.2014.217.

[39] Whisper Systems. *RedPhone is now Open Source*. 2012. URL: https://web.archive.org/web/20120731143138/http://www.whispersys.com/updates.html (visited on 08/03/2019).

[40] Andreas Zeller et al. 'Fuzzing with Grammars'. In: *Generating Software Tests*. Saarland University, 2019. URL: https://www.fuzzingbook.org/html/Grammars.html.

[41] Andreas Zeller et al. 'Mutation-Based Fuzzing'. In: *Generating Software Tests*. Saarland University, 2019. URL: https://www.fuzzingbook.org/html/MutationFuzzer.html.

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

API	application programming interface. 21, 26, 42
APNs	Apple Push Notification Service. 22
ASan	Address Sanitizer. 19, 56, *Glossary:* Address Sanitizer
AWS	Amazon Web Services. 22, 31
CIA	confidentiality, integrity and authenticity. 2
CVE	common vulnerabilities and exposures. 30
DFA	deterministic finite automaton. 32
DOS	denial of service. 12
EMV	Europay, Mastercard, Visa. 18
GCM	Google Cloud Messaging. 22
KDF	key derivation function. 6–9, 17, 21, 56, 60, *Glossary:* key derivation function
MITM	man-in-the-middle attack. 9
PGP	pretty good privacy. 2, 56, *Glossary:* pretty good privacy
Redis	Remote Dictionary Server. 22
SAL	source-code annotation language. 19
SSH	Secure Shell. 18
SUT	system under test. 18, 31
TLS	Transport Layer Security. 12, 18, 30, 40
TOFU	trust on first use. 23

UKS unknown key-share attack. 16, 17, 57, *Glossary:* unknown key-share attack

VoIP Voice over Internet Protocol. 40

X3DH extended triple Diffie-Hellman. 5, 7, 9, 16, 17, 23, 24, 53, 57, 60, *Glossary:* extended triple Diffie-Hellman

XSS cross-site scripting. 3, 40

GLOSSARY

address sanitizer An open-source tool developed by Google to detect memory corruption bugs. 19, 56

extended triple diffie-hellman An extended version of the Diffie-Hellman key exchange developed to support that one of the parties is allowed to be offline. 5, 57, 60

key derivation function A one-way function that can be used to derive keys by giving a key as input and optionally extra bits used for entropy. 6, 56

pretty good privacy Software developed by Phil Zimmerman to encrypt and sign messages using asymmetric cryptography. 2, 56

unknown key-share attack An attack where Alice thinks she has the key of Bob while in fact she has the key of Eve. 16, 57

adapter In this sense it is a translator between two application. 31, 61

alphabet An abstract definition of messages used by LearnLib. 31, 39

authentication Proof that you are actually communicating with the person you think you are communicating with. 6

Axolotl Ratchet The former name of the Double Ratchet algorithm. 7

Base64 A scheme to represent binary data in ASCII format. 27–29

black-box testing A form of testing that sees the program as a black-box and thus can only see in- and output. 13, 32, 59

buffer over-read Bug that makes it possbile to read memory after the buffer. 19

cryptographic deniability Not being able to proof to a third party that the other party had said something. 2, 58

Curve25519 An elliptic curve that can be used for elliptic curve Diffie-Hellman. 21, 28

deniability See: cryptographic deniability. 16, 17, 38, 39, 42

Diffie-Hellman A protocol to agree on a shared secret key over a (possibly) insecure channel. 4–10, 25, 53, 58–60

Diffie-Hellman ratchet An algorithm where both parties generate new key pairs by turns and use these to do a Diffie-Hellman key exchange. The shared secret key will then be used as chain keys. This gives the future secrecy property. See: Figure 2.7. 8, 9, 41, 53, 59, 60

Double Ratchet algorithm An algorithm that uses a Diffie-Hellman ratchet and a symmetric-key ratchet to encrypt and decrypt messages. 7, 8, 10, 11, 16–18, 24, 25, 53, 58

E.164 A standard to define telephone numbers, starting with a country code of maximum 3 digits and a subscriber number of maximum 12 digits. 26

edge case Cases that are at the boundary, e.g. `x >= 0` and `x = 0` are sometimes confused and can give errors. 15

elliptic curve An mathematical function that can be used for public-key cryptography when used over finite fields. 6, 21, 24, 39, 58

end-to-end encryption Encryption from client to client, this means the server does not has to be trusted. 1, 12, 39

ephemeral key A key in the Signal Protocol that may only be used once by the conversation starter. 4–6, 23

evolutionary algorithm An algorithm that is based on biological evolution. An input is mutated and the mutation is then checked. If the mutation is valid or better than the previous, it will be further mutated. If not, the previous input is mutated again. 14, 15, 53, 59

evolutionary fuzzer A fuzzing technique that generates its input based on a evolutionary algorithm, i.e. mutate valid input and check if the mutated input is still valid. 13

finite-state machine A model that can be used to describe the states of a system. 32, 39, 60

forward secrecy All keys that are generated before key k cannot be calculated when key k is compromised. 2, 6, 8, 17, 38, 39, 41, 42, 60

future secrecy All keys that are generated after key k cannot be calculated when key k is compromised. 2, 8, 17, 25, 38, 39, 42, 59

fuzzer Automated software testing technique by providing semi-random input to an application. 3, 4, 13, 14, 18, 19, 30, 31, 34, 36, 40, 42

generation-based fuzzer A fuzzing technique that generates its input based on a manually defined grammar. 13, 14, 31, 59

grammar-based fuzzer See: generation-based fuzzer. 13

grey-box testing Combination of white-box and black-box testing. 13

Heartbleed A bug in OpenSSL that made it possible to access memory behind the buffer boundary. 19

identity key The long-term key in the Signal Protocol. 4–6, 12, 17, 22, 61

KDF chain Using the output of the KDF as input for the next KDF. See: Figure 2.4. 7, 8, 53, 61

LearnLib A tool that can infer state machines. 18, 31–33, 36, 41, 58

Mealy machine A finite-state machine that defines both input and output on a transition. 32

mutation-based fuzzer A fuzzing technique that generates its input by mutations, e.g. inserting, changing and deleting characters in a string. 13, 15

one-time pre-key A key in the Signal Protocol that is published in sets, and which can be used by conversation starter to start the extended triple Diffie-Hellman. 4–6, 16, 22, 24

one-way function A function from which the inverse is not feasible to compute, i.e. when having y it is not feasible to calculate x from $f(x) = y$. 4, 8

Open Whisper Systems A company founded in 2013 by Moxie Marlinspike. 1, 2, 5, 12, 20, 21, 38, 40, 60

OpenSSL Open-source implementation of SSL and TLS. 19, 59

perfect forward secrecy See: forward secrecy. 38

pre-key bundle . 33, 34, 39, 47

private contact discovery service A service that makes it possible to find contacts without the server having to know the full address book. 12, 39–42

private key The key of a key pair that only the owner must known. 4, 5, 17, 61

protocol state fuzzing Inferring a state machine by sending semi-random messages to an application an obversing the output. 16, 18, 30, 39, 41, 42, 61

public key The key of a key pair that may be known by everyone. 4, 5, 23

random oracle An oracle that has a random response for every unique query. 17

random oracle model Used to proof a system secure instead of using hash functions. 17

ratchet key pair The key pairs used to execute the Diffie-Hellman ratchet, i.e. used to agree on new shared secret keys using the Diffie-Hellman key exchange. 9, 10, 24

ratchet step The generation of a new key for either the Diffie-Hellman ratchet or the symmetric-key ratchet. 8–10, 41

receiving chain The keys that will be used to derive decryption keys. 9, 10, 25

RedPhone A proprietary encrypted voice calling application developed by Whisper Systems. 1, 2

resilience The output key looks random to an adversary. 8

sealed sender A function created by Open Whisper Systems to stay anonymous. 12, 38–42

secret key See: private key. 5

secure enclave A separated space, either physically or virtually, which other processes are not allowed to read from. 41

sending chain The keys that will be used to derive encryption keys. 9, 10, 25

sequence diagram A diagram defined by the UML standard describing how objects interact. 22

Sesame algorithm An algorithm that is used to manage sessions. 10, 11, 32, 34, 36–40, 42

shared secret key A key that is shared by two persons. 5, 6, 9, 17, 25, 41, 58–60

Signal An instant message application available for Android, iOS, Windows, macOS and Linux. III, V, 1–3, 8, 12, 17, 18, 20–23, 25, 26, 28, 30–33, 36–48, 53, 61

Signal Protocol The protocol used by Signal to offer end-to-end encryption. III, V, 1–5, 10, 16, 17, 20, 21, 30, 38, 42, 53, 59–61

signed pre-key A key in the Signal Protocol that is signed with the identity key of the owner. 4–6, 17, 22, 28

state machine An abstract model that models the states of a computer program. III, 3, 18, 30–32, 53, 60

state machine learning See: protocol state fuzzing. 18

symbolic execution Following all the paths in the code by using symbolic values (e.g. $x = \lambda$). 13, 15, 61

symmetric-key ratchet An algorithm that uses the KDF chain to derive new keys. These keys will be used as message keys. See: Figure 2.4. 8–10, 53, 59, 60

test harness See: adapter. 31

TextSecure The former name of Signal. 1, 2, 16, 17, 38

TextSecure Protocol Old name of the Signal Protocol. 1, 2, 17

Whisper Systems A company founded in 2010 by Moxie Marlinspike and Stuart Anderson. The company was acquired by Twitter in 2011. 1, 60

white-box fuzzer A fuzzing technique that relies on symbolic execution. 13, 15

white-box testing A form of testing that tests the internal structure of the program. 59