ETH zürich

ETH Library

Continuous authentication in Secure Messaging

Master Thesis

Author(s):
Poirrier, Alexandre

Publication date:
2021

Permanent link:
https://doi.org/10.3929/ethz-b-000474066

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Continuous authentication in Secure Messaging

Master Thesis

A. Poirrier

Tuesday 9th March, 2021

Supervisor: Prof. Dr. K. Paterson
Advisors: Dr. B. Dowling, Dr. F. Günther

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

Messaging applications providing end-to-end encryption such as Signal can be vulnerable to Man-in-the-Middle (MitM) attacks, in which case authenticity may no longer be guaranteed. For Signal, authenticity relies on out-of-band protocols such as a comparison of long-term public keys. When a MitM attacker is detected, the usual response is to close the session and reopen a new one to restore security. Because long-term keys may have been compromised users need to change their long-term keys to be certain that security is restored. However such an operation is costly and requires human interaction.

This thesis addresses this issue by providing users with some degree of certainty on whether their long-term secrets have been compromised. It defines a security model capturing a notion of demonstration of knowledge of long-term secrets, in which an attacker tries to gain a MitM position without being detected by honest parties and without revealing their knowledge of a long-term secret. In this security definition the attacker has two goals: first they need to breach the authenticity of a session without being detected if users rely only on the in-band channel for detection. If users use an out-of-band channel for detecting the presence of adversaries, then the second goal of attacker is that users make a wrong choice on whether the attacker has used a long-term secret to remain undetected in-band. The attacker has full control of the network and can leak session secrets and long-term secrets.

This thesis also offers modifications on the original Signal protocol. Those modifications have two objectives: the first one is to detect the first type of attackers in-band, which allows users to close and reopen a session to lock the attacker out. The second one is to detect an attacker using their knowledge of long-term secrets to avoid in-band detection thanks to an out-of-band protocol. This enables parties to change their long-term secrets.

This paper proves formally the security of the proposed protocol according to the security definition offered.

It also presents a Java implementation based on the official Signal library. Practical tests verify that the original implementation does not guarantee security while the modifications achieve to restore security. The paper also presents an analysis with benchmarking of the space and computational overhead introduced by the modifications.

Contents

Contents iii

1 Introduction 1
1.1 Related work . . . 2
1.2 Contributions . . . 3
1.3 Report outline . . . 4

2 Signal protocol 5
2.1 Notations . . . 6
2.1.1 Cryptographic notations . . . 6
2.1.2 Stages and states . . . 6
2.1.3 Keys . . . 7
2.2 Registration phase . . . 7
2.3 X3DH . . . 8
2.4 Asymmetric ratcheting . . . 8
2.5 Symmetric ratcheting . . . 10
2.6 Out-of-order messages . . . 11

3 Continuous Man-in-the-Middle detection 13
3.1 High-level security definition . . . 13
3.2 Messaging scheme . . . 15
3.3 Signal is a messaging scheme . . . 17
3.4 Formal security game . . . 21
3.5 Signal (in)security . . . 26

4 Modified Signal protocol 29
4.1 Modification overview . . . 29
4.1.1 Recording ciphertexts . . . 30
4.1.2 Authentication steps . . . 31
4.2 Out-of-order messages . . . 34

4.3 Protocol soundness 34
4.4 Detecting long-term secret compromise 37
4.5 Authentication steps protocol as a messaging scheme 37

5 Security of the new Signal protocol 41
5.1 Cryptographic primitives 41
5.2 Security proof 43
5.2.1 Upper bound for false negatives 45
5.2.2 Upper bound for false positives 48

6 Extensions and limitations 51
6.1 Immediate decryption 51
6.1.1 Weaker security game 51
6.1.2 Extending the Signal protocol 52
6.2 Multi-party setting 53
6.3 Simultaneous session initiation 54
6.4 Deniability 55
6.5 Application to other settings 55

7 Implementation 57
7.1 Original implementation 57
7.2 Tests 59
7.3 Extending the protocol 63
7.3.1 Presentation of the modifications 63
7.3.2 Test results 65
7.4 Overhead introduced by authentication steps 67
7.4.1 Space and computation overhead 67
7.4.2 Benchmarking the space overhead 68
7.5 Observations on the official implementation 70
7.5.1 Design of an attack breaking post-compromise security 70
7.5.2 Explanation behind this weaker property 71
7.5.3 Fixing post-compromise security 72

8 Conclusion 73

Bibliography 75

Chapter 1

Introduction

The establishment of a secure communication channel over an adversarially-controlled network between two parties can typically be separated into two steps: first parties execute a key exchange protocol with long-term asymmetric keys, whose public part is publicly distributed, to derive a shared secret. This initial shared secret is then used to open a session in which parties use symmetric cryptography to get a secure channel [22].

One such protocol is Signal [17], which uses the X3DH protocol [19] as its initial key exchange to share an initial secret and then the Double Ratchet protocol to exchange messages securely. Signal is a messaging protocol – implemented in the Signal app itself, in Facebook Messenger [10] and Whatsapp [27] – the latter having several billions of users [26].

A unique property of messaging protocols is that sessions have a long lifetime, which can typically live for several months or years. This is a substantial problem as the presence of a Man-in-the-Middle (MitM) attacker could break security for the whole duration of the session if the attacker remains undetected [9].

Moreover, when detecting a compromised session, the standard response would be to close the session and open a new one. However, this does not necessarily restore security in the event of a long-term secret being compromised.

This paper addresses both issues, and derives a solution for detecting an adversary modifying a session between two users and forcing the adversary to reveal their knowledge of a long-term secret, which enables parties to refresh their long-term keys if compromised.

1.1 Related work

The Signal protocol has been designed and widely deployed before any security analysis was given. It is only years later that its creators, Marlinspike and Perrin, published a detailed description of both components of the Signal protocol:

- the X3DH (eXtended Triple Diffie-Hellman) protocol [19], the initial key exchange,
- the Double Ratchet protocol [17], which is at the core of the end-to-end encryption of Signal.

The very first analysis of Signal comes from Cohn-Gordon et. al [6]. They present an in-depth formal security analysis of the specific Signal protocol, based on its implementation due to the lack of formal specification. They focus on the Multi-Stage Key Exchange protocol aspect of Signal and define security definitions in term of key indistinguishability. The proposed model evaluates a tree of stages, which represents the various chains in Signal. That paper proves that the Signal protocol is secure according to the given security definition, from which usual properties such as forward secrecy or post-compromise security are derived. While the threat model is quite strong (capturing security against a network active adversary which can compromise long-term and medium-term keys as well as devices' states), the adversary is constrained using 'freshness' predicates. Long- and medium-term keys are supposed authenticated out-of-band: key impersonation attacks like the one described in [13] are excluded, as well as MitM attacks following state corruption as the paper focuses exclusively on keys exchanged by both parties.

Following those publications, there has been numerous analysis of the Signal protocol. Bellare et. al [2] have studied the security of *single, one-side ratcheting*. The considered protocol is a simplification of Signal. In the security game, the receiver is never corrupted and does not need to update its state. This allows the authors to come up with a simple secure solution but which does not extend well to the complete Signal protocol. However, Bellare et. al offer one of the first formalisation of the security properties of end-to-end communication. They also separate Ratcheted Encryption from Ratcheted Key-Exchange, by reducing the analysis of Signal's security to the security of its keys.

Concentrating on the Double Ratchet part of the Signal protocol, Alwen et. al [1] provide a formalised definition of 'secure messaging', which captures forward secrecy, post-compromise secrecy and immediate decryption. They also build a generic Signal protocol which aligns with their definition of secure messaging, from which the actual Signal protocol can be implemented. They exclude from their analysis the X3DH key exchange mechanism, by

assuming parties share a secret from the beginning. Furthermore, the definition of post-compromise security is quite restrictive as it forces the attacker to remain completely passive after compromise. As a session is considered established and long-term keys are not used in the Signal protocol beyond first key exchange, compromise of long-term secrets is not relevant in that context.

As the definitions of post-compromise security given by [6] and [1] both focus on secrecy, Dowling and Hale [8] offer a new security model including post-compromise security for authentication. They prove the original fingerprinting protocol is not secure as an attacker can compromise a device and gain an active MitM position by making the states of both devices diverge. Minor modifications to the protocol can provide a solution to restore security. This solution creates fingerprints derived from the Key Derivation Function (KDF) at the core of the Double Ratchet protocol. However authentication still happens out-of-band when parties meet, with the possibility of having an adversary controlling the phone outputs, *i.e.* the adversary can display wrong information on the device or input information instead of the user.

1.2 Contributions

While Dowling and Hale's paper [8] use an out-of-band channel for verifying authenticity of parties in a session, this paper offers protocol modifications to detect if an adversary is present by using the in-band channel. This detection procedure is secure assuming the adversary has not compromised a long-term secret.

Moreover, this paper presents an out-of-band detection step which coupled with the previous modifications can reveal if the attacker has used a long-term secret to avoid detection in-band. This allows users to know if their long-term secret has been compromised so they can refresh it only if necessary.

The protocol modifications have been implemented on top of the official Java library for the Signal protocol. Concrete test sets show that the original Signal implementation does not provide security, which is ensured with the modifications presented. The paper also proposes a benchmarking analysis of the computational and space overhead introduced by those modifications, using a database of SMS messages and given different channel reliabilities and conversation layouts.

1.3 Report outline

The remaining of this paper is organised as follows. Section 2 presents the original Signal protocol. Section 3 defines the security notion for detecting adversaries and Section 4 modifies the Signal protocol presented in Section 2 to ensure security according to the definition given in Section 3. Security of the modified Signal protocol is proven in Section 5. Section 6 evaluates the limitations of the protocol defined in Section 4 and provides some extensions to overcome the encountered issues. Finally, Section 7 describes the Java implementation of the protocol modifications, as well as the different tests and benchmarking performed. Finally Section 8 summarizes the contributions of this thesis.

Chapter 2

Signal protocol

This section gives an overview of the Signal protocol, inspired from [6]. Signal is an asynchronous messaging protocol, meaning it enables two parties to communicate even if the receiver is offline when messages are sent. It aims at providing end-to-end secure encryption, with specific security properties such as forward secrecy and post-compromise security.

On a high level, the protocol consists of four distinct stages:

- *Registration*: users publish on a server their public key material, referred as pre-key bundle, composed of their public identity (long-term) key, a medium-term key signed using the identity key and ephemeral keys. More details are provided in Section 2.2.
- *Session establishment*: When one user (Alice) wants to open a session with another (Bob), she fetches the corresponding pre-key bundle on the semi-trusted server and uses it to derive a *root key*. When she sends her first message to Bob, she attaches some public material with the ciphertext so Bob can also derive the same root key. This root key is used as an initial shared secret for the session. Details are provided in Section 2.3.
- *Asymmetric ratchet*: when a user (Alice) receives a message from the other (Bob) containing a new public *ratchet key*, she performs a Diffie-Hellman computation combining the newly received public ratchet key with her last private ratchet key, deriving a new *root key* and receiving *chain key*. She then generates a new ratchet key-pair, combines the private part with the received public ratchet key to derive a new root key and sending chain key. Section 2.4 provides more detailed information.
- *Symmetric ratchet*: given the chain keys derived from the asymmetric ratchet, users can derive *message keys* to encrypt or decrypt messages

using an AEAD scheme. More details are provided in Section 2.5.

An overview of the Double Ratchet protocol (asymmetric and symmetric ratcheting) can be found in Figure 2.2.

2.1 Notations

In the remainder of this paper, sessions are initiated by Alice who wants to communicate with Bob. Signal is a complex protocol, involving different stages and keys. This section presents notations used in this paper.

2.1.1 Cryptographic notations

This paper uses the following notations:

- $x||y$ represents the concatenation of byte sequences x and y.
- $\mathtt{DH}(x,\ y)$ represents the shared secret output from an Elliptic Curve Diffie-Hellman function involving keys x and y. x and y may represent one public and one private key to precise how the computation is performed, or both public keys if this precision does not matter. In the Signal protocol, the curve will either be the X25519 or X448 [15] elliptic curve.
- $\mathtt{KDF}(x)$ represents the 32 bytes output of a HKDF algorithm [14] execution given the following inputs:
  - input key material: $F||x$ where F is a bytestring containing enough 0xFF bytes to get a correct input length,
  - salt: null bytes sequence.
- $\mathtt{SIG}_{sk}(x)$ represents an XEdDSA [16] signature of x produced with private key sk. Verification is performed by using $\mathtt{Vfy}_{pk}(x, \sigma)$ where pk is a public key and outputs 1 if the verification succeeds and 0 otherwise.
- $x \xleftarrow{\$} \mathtt{Proc}()$ signifies x is chosen at random from the image of procedure $\mathtt{Proc}$. The distribution is determined by the procedure and is uniform if not specified.
- $x \in_R S$ means x is chosen uniformly at random from set S.

2.1.2 Stages and states

A session is divided in *epochs*: an epoch is a unidirectional stream of messages sent by one party, without receiving a reply from the other party.

Public key	Private key	Description
ipk^A	ik^A	A's identity (long-term) key-pair
$prepk^B$	$prek^B$	B's medium-term (signed) prekey pair
epk^A	ek^A	Ephemeral key pair for the initial handshake
rpk_i^A	rk_i^A	A's ratchet key pair on epoch i
	sk_i	Root (state) key on epoch i
	$ck_{i,j}$	j^{th} chain key on epoch i
	$mk_{i,j}$	Message key for message j of epoch i

Table 2.1: Keys used in the Signal protocol. If the public key is absent then the description is of a symmetric key.

Epochs are numbered in increasing order[1], such that even epoch numbers belong to Alice sending messages and odd epoch numbers to Alice receiving messages (recall Alice is defined as the initiator of the session). Inside an epoch, messages are also monotonically increasing from 0. The current local state of Alice in epoch i while sending or receiving message j is denoted as $\pi_{i,j}^A$. This local state does not includes long-term secret information, which is stored in another set LTS_A.

For Signal, at most one message is exchanged and accepted for an epoch i and index j. The corresponding plaintexts are denoted $pt_{i,j}^U$ (for user U) and the ciphertext $c_{i,j}$.

2.1.3 Keys

Signal distinguishes several types of keys, summarized in Table 2.1. Keys are written in italics and end with the letter k. For private/public key-pairs, public keys end with pk and have the same prefix than their private counterpart. If necessary, the user who generated the key is written in superscript and the stage number in subscript.

There can be several ephemeral keys generated by each parties.

2.2 Registration phase

Each party P generates upon installation of the messaging application:

- a long-term identity key-pair (ipk^P, ik^P),
- a medium-term prekey pair $(prepk^P, prek^P)$,
- multiple ephemeral key-pairs $(epk_{(i)}^P, ek_{(i)}^P)$.

[1]In the specification and the implementation, this is not the case and epochs are instead referred by the public ratchet key chosen by the sender of the epoch. However epochs can be numbered by integers increasingly, which simplifies the theoretical considerations.

The public long-term key, the public prekey signed using the long-term key and the ephemeral public keys are published on a server. Those keys constitute the prekey bundle of party P. From time to time, each party adds ephemeral keys and changes the medium-term key.

Ephemeral keys are to be used only once in the X3DH protocol. After use they are discarded.

2.3 X3DH

When Alice wants to open a session with Bob, she fetches the pre-key bundle Bob uploaded on the server during registration. She then generates an ephemeral key-pair (epk^A, ek^A) and performs the following computations:

$$DH_1 = \mathtt{DH}(ik^A, prepk^B), \; DH_2 = \mathtt{DH}(ek^A, ipk^B),$$

$$DH_3 = \mathtt{DH}(ek^A, prepk^B), \; DH_4 = \mathtt{DH}(ek^A, epk^B)$$

The fourth computation is optional and depends whether there remains an ephemeral key in Bob's prekey bundle. A summary of performed Diffie-Hellman computations is given on Figure 2.1.

Finally, Alice computes the initial shared secret:

$$sk_0 = \mathtt{KDF}(DH_1 || DH_2 || DH_3 || DH_4)$$

Once again, the fourth value is not used if there is no ephemeral key in Bob's prekey bundle.

In her first message, Alice will include in the associated data (recall messages are encrypted using an AEAD scheme, which will be presented in Definition 2.1) her identity public key ipk^A, her ephemeral public key epk^A and an identifier for Bob to know which ephemeral key epk^B has been used. Upon receiving this first message, Bob will be able to compute the same Diffie-Hellman shared secrets and finally compute the same initial shared secret sk_0 than Alice.

2.4 Asymmetric ratcheting

At the core of the Signal protocol is the Double Ratchet protocol. As the name suggests, the protocol uses two 'ratchets', which are *KDF chains*. A KDF chain holds an internal state, which can be initialised using a key. Then by passing an optional input to the KDF chain, the internal state is updated and an output value is produced thanks to a KDF.

To communicate, Alice and Bob both use 3 KDF chains: one root chain, one sending chain and one receiving chain. Asymmetric ratcheting advances the

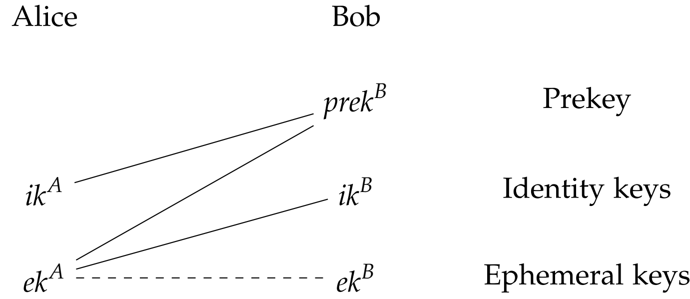


Figure 2.1: X3DH Diffie-Hellman keys. The dashed line is not used if there is no ephemeral key in the pre-key bundle downloaded from the server.

root chain, whose outputs are used to initialise the sending and receiving chains.

In this setting, we assume Alice and Bob share a secret sk_i. This secret value is used as the internal state of both root KDF chains.

Asymmetric ratcheting is triggered when a user receives a message containing a new ratchet key. Let's assume Bob receives a ciphertext from Alice containing a new ratchet key rpk_i^A.

Then Bob combines the newly received ratchet public key with his own private ratchet key:

$$DH_i = \mathtt{DH}(rk_{i-1}^B,\ rpk_i^A)$$

If this is the first message from Alice, Bob uses his medium-term key as a ratchet key: $rpk_{-1}^B = prepk^B$.

Then Bob advances his root KDF chain by providing DH_i as input. The output is a chain key, which will be used to initialise his receiving KDF chain.

$$sk_{i+1},\ ck_{i,-1} = \mathtt{KDF}(sk_i||DH_i)$$

He then discards his ratchet key, and generates a new ratchet key-pair $(rpk_{i+1}^B, rk_{i+1}^B)$. He performs the same operation as above to initialise his sending KDF chain:

$$DH_{i+1} = \mathtt{DH}(rk_{i+1}^B,\ rpk_i^A)$$

$$sk_{i+2},\ ck_{i+1,-1} = \mathtt{KDF}(sk_{i+1}||DH_{i+1})$$

The generated ratchet key rpk_{i+1}^B will be included in the associated data (of the AEAD scheme, see Definition 2.1) of every message sent from this point, until he changes the key.

To initialise her very first sending chain, Alice performs the second part of the asymmetric ratcheting to derive her first sending chain, by using Bob's medium-term key as a ratchet key.

Both KDF root chains of Alice and Bob are synchronised (meaning they have the same internal state). The Diffie-Hellman computations ensure they provide the same inputs to their root chains, and therefore Alice's sending chains are synchronised with Bob's receiving chains and vice-versa.

Figure 2.2 illustrates this exchange, and how asymmetric ratcheting is combined with symmetric ratcheting to send messages.

2.5 Symmetric ratcheting

Symmetric ratcheting is triggered when a user wants to encrypt or decrypt a message. Symmetric ratcheting assumes asymmetric ratcheting has been performed and the sending or receiving chain has been initialised at some point.

To derive a message key for encryption or decryption, a user advances their receiving or sending KDF chain without providing any input. The output of the KDF chain is the message key:

$$ck_{i,j},\ mk_{i,j} = \mathtt{KDF}(ck_{i,j-1})$$

The message key is then used to encrypt (or decrypt) a message using a symmetric AEAD scheme. The associated data contains the following information:

- the public ratchet key of the sender;
- the message number;
- the total number of messages sent in the previous sending epoch;
- additional data identifying the session (identity keys, session version, etc).

An AEAD scheme is defined as follows:

Definition 2.1 *An* AEAD scheme *is a set of two algorithms* ($\mathtt{Enc}$, $\mathtt{Dec}$)*:*

- $\mathtt{Enc}$ *takes as input a key k, associated data AD and a plaintext m and outputs a ciphertext* $c \xleftarrow{\$} \mathtt{Enc}(k, m, AD)$*,*
- $\mathtt{Dec}$ *takes as input a key k, a ciphertext c and associated data AD and outputs a plaintext* $m \leftarrow \mathtt{Dec}(k, c, AD)$*.*

The AEAD scheme verifies the correctness property:

$$\forall k \in \mathcal{K},\ \forall AD,\ m \in \{0,1\}^*,\ \texttt{Dec}(k, \texttt{Enc}(k, m, AD), AD) = m$$

`Dec` *may output a special symbol* $\perp$ *which corresponds to a decryption error.*

Symmetric ratcheting steps are the small KDF chains at the right of Figure 2.2.

As the sending chain of the sender is synchronised with the receiving chain of the receiver thanks to the asymmetric ratchet, the index of the message in the epoch ensures the same key is derived on both sides. The correctness of the AEAD scheme thus ensures messages are correctly decrypted.

2.6 Out-of-order messages

As the message number (as well as its epoch) are sent alongside the message in the associated data, a user receiving a message can deduce if some message has been missed, and in that case can store the corresponding keys to ensure decryption once the corresponding message arrives.

Such keys and message identifier are stored in a dictionary

$$MKSKIPPED : \mathbb{N} \times \mathbb{N} \to \mathcal{K}$$

(where $\mathcal{K}$ is the key space).

A parameter determines how long (in term of number of messages and number of epochs) such keys are stored on the device before being erased[2].

[2] In the official Signal implementation, a maximum of 2000 keys are stored in the state. Keys older than 5 epochs are also automatically deleted.

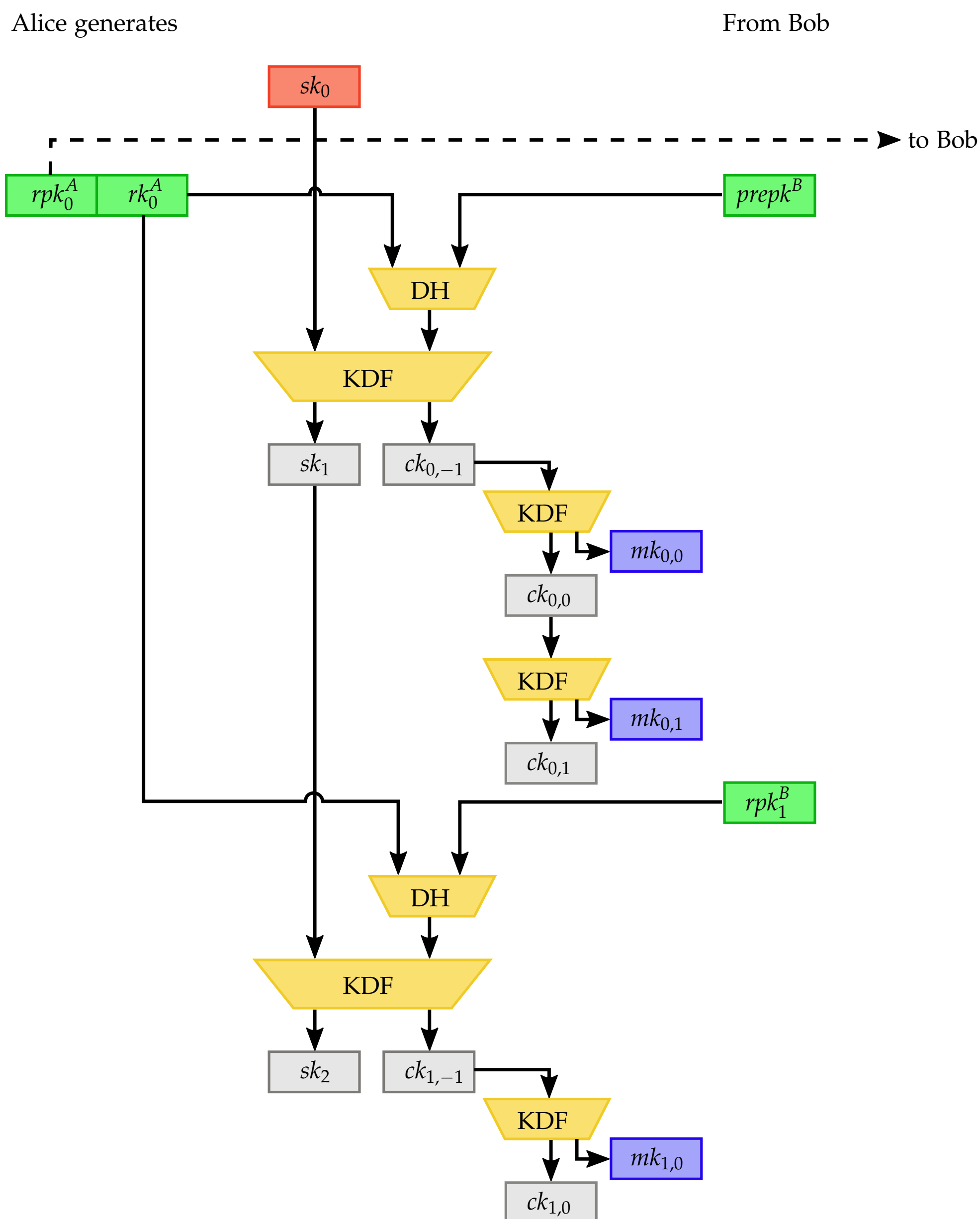


Figure 2.2: Double ratchet overview, from Alice's perspective. Asymmetric ratchet uses shared secret (in red) and ratchet keys (in green) to produce next shared secret and chain key (in grey). Symmetric ratcheting uses a chain key to produce the next chain key and a message key (in blue). In epoch 0 Alice uses message keys to send messages and in epoch 1 to decrypt received messages.

Chapter 3

Continuous Man-in-the-Middle detection

This section presents and formalises a security model capturing the notion of *demonstrating knowledge of long-term secrets* for asynchronous messaging protocols. This security model takes the form of a security game in which two honest parties communicate. They are in presence of an attacker who has full control of the network and can compromise sessions as well as long-term secrets in order to break the secrecy or authenticity of the session. The attacker's first objective is to break authenticity by injecting a message without being detected by the communicating parties if they only use the in-band channel. If users can use an out-of-band channel to detect some session tampering, the attacker can still win if parties incorrectly guess if the attacker knows a long-term secret.

3.1 High-level security definition

In messaging protocols such as Signal, an active attacker compromising a session state may gain access to a MitM position, as stated in the original Signal specification [18]:

> The attacker could substitute her own ratchet keys via continuous active MitM attack, to maintain eavesdropping on the compromised session.

Such a position breaks both secrecy and authenticity for the remainder of the session. If such an attacker is discovered, the typical response would be to close the session and reopen a new one. However this does not lock the attacker out if they know the long-term secrets of parties. This means that long-term keys also need to be refreshed in this event.

This section presents a security definition which enables users to detect

MitM adversaries by using the in-band channel if they do not use long-term secrets, and which forces the attacker to prove their knowledge of long-term secrets if users use an out-of-band channel.

Given the possibility of attackers to disclose all secrets, an attacker could impersonate parties and therefore the attacker is indistinguishable from the honest peer from a user's point of view. Therefore a winning condition for the attacker is to be able to inject a message while remaining undetected by honest parties in-band, without ever compromising a long-term secret.

The attacker has a second winning condition. Indeed, if a message is injected (and different from a honestly produced message), then users can by using an out-of-band channel compare their transcripts and deduce some attacker is present. As stated in the introduction, the usual response is to close the session, but to restore security parties need to know if they need to refresh their long-term secrets. Such an operation is expensive and should be avoided if not necessary. Indeed, changing a long-term secret would require to authenticate it (with out-of-band means) to every other peer, which involves human interaction.

Therefore messaging schemes should implement an out-of-band detection procedure, which uses an out-of-band channel to detect if an adversary is present and knows a long-term secret. The attacker can win if parties make the wrong decision. It means the out-of-band detection procedure should verify the following properties:

- if an attacker compromises and uses a long-term secret, users should be able to detect it (no false negatives);
- if the attacker does not disclose a long-term secret, parties should decide that no long-term secret is compromised (soundness property).

The security definition is expressed through a game, in which an attacker interacts with oracles simulating communication between two parties. In this game, the attacker's objective is to tamper with the communication, by injecting at least one forged ciphertext, without parties being aware that this ciphertext is not legitimate. Moreover, once parties meet, the attacker can win the game if parties make an incorrect decision about the knowledge of a long-term secret of the attacker.

In practice, this corresponds to the following scenario. Parties face a powerful adversary who can corrupt the session state of one or both devices. By corrupting a device, it can inject messages, pretending to be the other party. As parties may not meet or compare messages for a long time, the attacker wishes to remain undetected for as long as possible and retain its MitM position.

An example of secure messaging scheme may compare messages and state regularly, using long-term secrets to authenticate such messages. An attacker wishing to remain undetected would be required to use long-term secrets. That would be a proof of their knowledge of some long-term secret, which will be discovered once parties meet and detect tampering.

This paper presents such a scheme in Section 4, whose security is proved in Section 5.

In the following, Section 3.2 formally defines messaging schemes. Section 3.3 shows how the original Signal protocol presented in Section 2 fits this definition of messaging scheme. Section 3.4 presents the formal security game and Section 3.5 explains why the original Signal protocol is insecure.

3.2 Messaging scheme

This section defines formally messaging schemes. A messaging scheme consists of several algorithms to create users (Register), initiate sessions between them (InitState) and let them create and receive messages (Send and Recv).

The definition supports an arbitrary number of users, but only two-parties sessions (no group chat).

In addition to the four traditional messaging procedures, StartAuth is a procedure which can be used to make a user require in-band authentication. This procedure can leave the state unchanged if the messaging scheme does not support in-band authentication. Another additional procedure, DetectOOB, compares the states of participants in a session out-of-band and decides if an adversary knowing a long-term secret is present.

Definition 3.1 (Messaging Scheme) *A* messaging scheme MS *consists of six probabilistic algorithms:*

$$\mathtt{MS} = (\textsc{Register},\ \textsc{InitState},\ \textsc{Send},\ \textsc{Recv}, \textsc{StartAuth}, \textsc{DetectOOB})$$

Those algorithms have the following signature:

- Register *creates a user U and outputs long-term information and secret as well as medium-term information and secret* $(LTI_U, LTS_U, MTI_U, MTS_U) \xleftarrow{\$}$ Register(),
- InitState *takes as input the long- and medium-term secret of a user U, the long-term information of a user V and optionally some medium-term information for user V and creates an initial session state for U to communicate with V:* $\pi_U \xleftarrow{\$}$ InitState$(LTS_U, MTS_U, LTI_V, MTI_V)$,

- SEND *takes as input the state* π_U *and private long-term information* LTS_U *of the sender as well as a message m and outputs a new state, a ciphertext and a message index* $(\pi'_U, c, idx) \xleftarrow{\$} \text{SEND}(\pi_U, LTS_U, m)$*,*
- RECV *takes as input the state* π_U *and long-term private information* LTS_U *of the receiver as well as a ciphertext c and outputs a new state, a plaintext and an index* $(\pi'_U, m, idx) \xleftarrow{\$} \text{RECV}(\pi_U, LTS_U, c)$*,*
- STARTAUTH *takes as input a state* π_U *and outputs a new state* $\pi'_U \xleftarrow{\$} \text{STARTAUTH}(\pi_U)$*,*
- DETECTOOB *takes as input two states* π_A *and* π_B *and outputs a bit* $d \xleftarrow{\$} \text{DETECTOOB}(\pi_A, \pi_B)$.

RECV *may return an error (*$\perp$*) instead of the plaintext which signals the ciphertext has not been accepted. Moreover, it may send a* **Close** *exception which signifies the party closes the connection.*

REGISTER creates users, INITSTATE creates a blank state for initiating a new session. SEND and RECV procedures allow to send and receive messages. Private information is passed to SEND and RECV to allow users to use long-term secrets to create ciphertexts or modify their state.

To create a session between two users, the initial state should be created with INITSTATE. For protocols such as Signal in which the initiator retrieves data from a server, this data can be included in the MTI_V parameter.

States hold an `auth` flag which is initially set to **None**. STARTAUTH is a special procedure which may set this flag to some value in the state of the party, indicating the party wants to perform an authentication step. If no authentication steps are implemented, as in the original Signal protocol, this method does not change the state. The authentication step is *passed* once the `auth` flags of both parties are back to **None**.

Note that in the remainder of this paper, if some variable x can be **None**, then x can be considered as a boolean variable being **True** if x is not **None** and **False** otherwise.

DETECTOOB is another special procedure which simulates parties using an out-of-band channel to compare their local states and decide if their communication has been tampered with or not.

On top of Definition 3.1 which describes the syntax of a messaging scheme, Definition 3.3 adds some semantics. Schemes studied in this paper satisfy this correctness definition.

Before defining correctness for messaging schemes, the following definition introduces the notion of *matching states*.

Definition 3.2 (matching states) *If the following applies for two users A and B:*

- Register *outputs* $(LTI_A, LTS_A, MTI_A, MTS_A)$ *for user A (resp. for user B),*
- *A creates her initial state* $\pi_A \leftarrow$ InitState$(LTS_A, MTS_A, LTI_B, MTI_B)$*,*
- *B creates his initial state* $\pi_B \leftarrow$ InitState$(LTS_B, MTS_B, LTI_A, \textbf{None})$*.*

Then states π_A *and* π_B *are* matching*.*

This enables to define correct messaging schemes.

Definition 3.3 (Correct messaging scheme) *Let* MS *be a messaging scheme. Let also A and B be two users created with the* Register *algorithm.* MS *is* correct *if it follows the following properties:*

1. *The index idx returned by* Recv *is efficiently computable from the ciphertext;*
2. *Indexes come from a totally ordered set;*
3. *If idx is returned by* Send(π_A, LTS_A, m)*, then idx is greater than every index corresponding to a message sent or received using* π_A*;*
4. Recv *returns plaintext* $\perp$ *if the ciphertext corresponds to an index which has already been decrypted;*
5. *If* Recv *returns plaintext* $\perp$*, then the state remains unchanged;*
6. *If states* π_A *and* π_B *are matching, A uses* Send *to create* $(\pi'_A, c, idx) \leftarrow$ Send(π_A, LTS_A, m) *from a plaintext m, and B inputs this ciphertext to create* $(\pi'_B, m', idx') \leftarrow$ Recv(π_B, LTS_B, c)*, then* $m = m'$*,* $idx = idx'$ *and states* π'_A *and* π'_B *are still matching;*
7. *Given two matching states, if one of them has* $\pi.auth \neq \textbf{None}$*, then there exists a finite number of calls to* Send *and* Recv *such that both states get back to* $\pi.auth = \textbf{None}$*.*

Property 1 makes immediate decryption possible. Properties 2 and 3 ensure the indexes come from a totally ordered set and are distributed increasingly. Moreover property 4 makes sure messages decrypted correspond to different indexes. Property 5 ensures bad ciphertexts do not break the scheme.

Property 6 ensures this soundness property propagates to all messages in the communication. Note that this soundness property is weaker than what Signal offers, as immediate decryption is not included with this definition. The reader may refer to [1] to get a more precise definition of soundness property for the Signal protocol.

Property 7 rules out schemes where an authentication step can never end.

3.3 Signal is a messaging scheme

The Signal protocol presented in Section 2 matches the Definition 3.1 of a messaging scheme.

Indeed, the different algorithms of Definition 3.1 can be defined in a simplified version as described on Figure 3.1. Notations on this figure follow notations from Section 2.1.1 and use AEAD procedures from Definition 2.1.

Throughout the paper, procedures from the messaging scheme are in Small-Capitals with CamelCase. Helper procedures are in `TeletypeFont`, also in CamelCase. Oracles from security games are written sansSerif.

`KeyGen` is a special procedure sampling key-pairs uniformly at random. The X3DH computations are not detailed here. The one in `Connect-Send` returns a new root key and an index to identify which ephemeral key has been used. The one in `Connect-Recv` returns the same root key. More details can be found on Section 2.3.

The `Recv-Epoch` procedure takes a state as parameter, and returns true if the state is currently in receiving epoch. Its actual implementation is not explicitly shown, as it would require the state to hold more variables which are not useful for this paper. The `badIdx` procedure verifies if the given pair (i, j) is correct given the state, *i.e.* the pair has not already been decrypted using this state and corresponds to the peer's sending epoch.

There are two helper functions:

- `AsymRtch` takes as input a state, an optional private ratchet key and a ratchet public key and performs one turn of the asymmetric ratchet (modifying the state given as input). It uses the ratchet keys given in parameter, and if no private ratchet key is provided, one is randomly generated.
- `IdxMgmt` takes as input a state and associated data from a ciphertext. It advances the state to derive the message key corresponding to this associated data, by computing intermediary keys and storing them in the state if necessary.

DetectOOB is voluntarily left unimplemented as there are no strategy in the Signal protocol to detect compromise or tampering out-of-band. As shown in Section 3.5, any implementation will result in an insecure scheme.

The implementation presented in Figure 3.1 is simplified. For instance, signatures for prekey bundles are not included in the description. A full implementation would require to verify those signatures at the beginning of `Connect-Send`. Also the state should conserve some information on whether the party is the initiator of the connection or not, in order to correctly implement the `badIdx` and `Recv-Epoch` functions. Simple checks for consistency of parameters are also omitted, for instance for Send and Recv there is no check to verify the state and the long-term secrets belong to the same user.

The Signal protocol verifies the correctness definition given in Definition 3.3. Indeed, message indices are included in the associated data (*i.e.* appears in

1 **procedure** REGISTER():
2 $n \in_R \mathbb{N}$
3 $(ipk^P, ik^P) \xleftarrow{\$}$ `KeyGen()`
4 $(prepk^P, prek^P) \xleftarrow{\$}$ `KeyGen()`
5 $\forall i \in [\![1, n]\!], (epk^P_{(i)}, ek^P_{(i)}) \xleftarrow{\$}$ `KeyGen()`
6 $MTI_P \leftarrow \left(prepk^P, epk^P_{(1)}, ..., epk^P_{(n)}\right)$
7 $MTS_P \leftarrow \left(prek^P, ek^P_{(1)}, ..., ek^P_{(n)}\right)$
8 **return** $(ipk^P, ik^P, MTI_P, MTS_P)$

1 **procedure** STARTAUTH(π_P):
2 **return** π_P

1 **procedure** INITSTATE($LTS_U, MTS_U, LTI_V, MTI_V$):
2 **if** MTI_V:
3 **return** `Connect-Send`$(LTS_U.ik^U, LTI_V, MTI_V)$
4 $\pi_U \leftarrow \{LTI_V : LTI_V, MTS_U : MTS_U\}$
5 **return** π_U

1 **procedure** SEND(π_U, LTS_U, m):
2 **if** `Recv-Epoch`(π_U):
3 $\pi_U \xleftarrow{\$}$ `AsymRtch`$(\pi_U,$ **None**$, \pi_U.rpk^V)$
4 $(\pi_U.i, \pi_U.j_U) \leftarrow (\pi_U.i + 1, 0)$
5 **else:**
6 $(\pi_U.ck^U, \pi_U.mk) \leftarrow$ `KDF`$(\pi_U.ck^U)$
7 $\pi_U.j_U + +$
8 $AD \leftarrow \pi_U.AD \cup \{(\pi_U.i, \pi_U.j_U), \pi_U.tot, \pi_U.rpk^U\}$
9 $\pi_U.AD \leftarrow \emptyset$
10 $c \xleftarrow{\$}$ `Enc`$(\pi_U.mk, m, AD)$
11 **return** $(\pi_U, (c, AD), (\pi_U.i, \pi_U.j_U))$

1 **procedure** RECV(π_U, LTS_U, c):
2 $\pi'_U \leftarrow \pi_U$
3 $(c, AD) \leftarrow c$
4 **if** $\pi_U.MTS_U$:
5 $\pi'_U \leftarrow$ `Connect-Recv`$(LTS_U.ik^U, \pi_U.MTS_U, \pi_U.LTI_V, AD)$
6 **try:**
7 `IdxMgmt`(π'_V, AD)
8 **except BadIdx:**
9 **return** $(\pi_V, \perp, (AD.i, AD.j))$
10 $m \leftarrow$ `Dec`$(\pi'_V.mk, c, AD)$
11 **if** $m == \perp$:
12 **return** $(\pi_V, \perp, (AD.i, AD.j))$
13 **return** $(\pi'_V, m, (AD.i, AD.j))$

Figure 3.1: Simplified implementation of the Signal protocol following the messaging scheme definition. Auxiliary functions are defined on figure 3.2

1 **procedure** Connect-Send(ik^U, LTI_V, MTI_V):
2 $\pi_U \leftarrow \{auth : \textbf{None}, i : -1, j_U : 0, j_V : 0, tot : 0, LTI_V : LTI_V\}$
3 $(epk^U, ek^U) \xleftarrow{\$}$ KeyGen()
4 $(\pi_U.sk, kid) \xleftarrow{\$}$ X3DH(ik^U, LTI_V, MTI_V, ek^U)
5 $\pi_U.rpk^V \leftarrow MTI_V.prepk^V$
6 $\pi_U.AD \leftarrow (epk^U, kid)$
7 **return** π_U

1 **procedure** Connect-Recv(ik^U, MTS_U, LTI_V, AD):
2 $\pi_U \leftarrow \{auth : \textbf{None}, i : -1, j_U : 0, j_V : 0, tot : 0\}$
3 $\pi_V.sk \leftarrow$ X3DH(ik^U, MTS_U, LTI_V, AD)
4 $\pi_U.rk^U \leftarrow MTS_U.prek^U$
5 **return** π_U

1 **procedure** AsymRtch(π_P, rk, rpk):
2 **if** $\neg rk$:
3 $(\pi_P.rpk^P, \pi_P.rk^P) \xleftarrow{\$}$ KeyGen()
4 $rk \leftarrow \pi_P.rk^P$
5 $tmp \leftarrow$ DH(rk, rpk)
6 $(\pi_P.sk, \pi_P.ck^P) \leftarrow$ KDF($\pi.sk || tmp$)
7 $(\pi_P.ck^P, \pi_P.mk) \leftarrow$ KDF($\pi_P.ck^P$)
8 **return** π_P

1 **procedure** IdxMgmt(π_U, AD):
2 **if** badIdx($\pi_U, (AD.i, AD.j)$):
3 **raise BadIdx**
4 **if** $(AD.i, AD.j) \in \pi_U.MKSKIPPED$:
5 $\pi_U.mk \leftarrow \pi_U.MKSKIPPED.pop((AD.i, AD.j))$
6 **else:**
7 RecordSkipped (π_U, AD)
8 $(\pi_U.ck^U, \pi_U.mk) \leftarrow$ KDF($\pi_U.ck^U$)
9 $\pi_U.j_V + +$

1 **procedure** RecordSkipped(π_U, AD):
2 **if** $AD.i > \pi_U.i$:
3 $\pi_U.tot \leftarrow \pi_U.j_U$
4 **while** $\pi_U.j_V < AD.tot$:
5 $(\pi_U.ck^U, \pi_U.mk) \leftarrow$ KDF($\pi_U.ck^U$)
6 $\pi_U.j_V + +$
7 $\pi_U.MKSKIPPED[(\pi_U.i, \pi_U.j_V)] \leftarrow \pi_U.mk$
8 $\pi_U \leftarrow$ AsymRtch($\pi_U, \pi_U.rk^U, AD.rpk$)
9 $(\pi_U.i, \pi_U.j_V) \leftarrow (AD.i, 0)$
10 **while** $\pi_U.j_V < AD.j$:
11 $(\pi_U.ck^U, \pi_U.mk) \leftarrow$ KDF($\pi_U.ck^U$)
12 $\pi_U.j_V + +$
13 $\pi_U.MKSKIPPED[(\pi_U.i, \pi_U.j_V)] \leftarrow \pi_U.mk$

Figure 3.2: Auxiliary functions used in the Signal implementation.

cleartext), therefore are efficiently computable. They come from $\mathbb{N}^2$ which is a totally ordered set with the lexicographical order. The index management for Signal (described in detail in [1] for instance) verifies property 3. The `badIdx` method explicitly performs the check from property 4, and by reading the implementation of RECV, the reader can make sure every time $\perp$ is returned for the plaintext the state remains unchanged.

The soundness property 6 can also be proven. The reader may refer for instance to [2] to understand why keys are correctly derived and the Signal protocol is sound. This was also quickly explained in Section 2.

In this original version, states are never authenticating (*i.e.* $\pi.auth$ is always **None**). Therefore property 7 is immediate.

3.4 Formal security game

This section presents the formal security game corresponding to the notion briefly presented in Section 3.1.

In this game, the attacker is active on the network and can corrupt devices, *i.e.* disclose their current state. It can also disclose long-term secrets, which activates a flag signifying that the attacker knows them.

The attacker has two ways of winning the game: it can either break authenticity without being detected or make a user incorrectly decide that the attacker knows long-term secrets.

For the first winning condition, the attacker needs first to tamper with the communication, *i.e.* to inject a forged message to one party which will be successfully decrypted. Then it needs to launch the in-band detection procedure using the STARTAUTH function (which is accessible through an oracle) and be undetected by this procedure. In practice, this procedure may be triggered automatically and periodically during the communication.

When the attacker stops, an out-of-band detection step is triggered. If the attacker successfully injected a message and passed an authentication step, but the out-of-band detection does not detect an adversary knowing a long-term secret, the adversary wins.

The second winning condition is triggered if the out-of-band detection step detects an adversary knowing a long-term secret but the adversary never used the compromise oracle.

The security game creates two users, Alice and Bob, and lets the attacker interact with them using oracles to simulate a communication. Because the final objective is to detect long-term secret compromise, the long-term secrets are distributed honestly to parties. However medium term secrets are only generated and can be tampered with.

The attacker has access to the following oracles:

- createState-A /B creates the initial state of a party given some prekey bundle provided by the attacker,
- transmit-A /B takes a plaintext as input and simulates one party sending it,
- deliver-A /B takes a ciphertext as input and simulates one party receiving it,
- corruptState-A /B returns the current state of the party,
- auth-A /B makes one party request authentication,
- corruptLTS-A /B leaks the long-term secret of the party.

Those oracles are defined on Figure 3.3. The game itself and the security notion are defined in the following definition.

Definition 3.4 (Detection game) *Let $\mathcal{A}$ be a probabilistic polynomial-time adversary against a messaging scheme* `MS`*. It has access to oracles defined on Figure 3.3. The security game is the following:*

1. **game** `Detection-Game(`$\mathcal{A}$, `MS):`
2. $(LTI_A, LTS_A, MTI_A, MTS_A) \xleftarrow{\$} \texttt{MS}.\textsc{Register}()$
3. $(LTI_B, LTS_B, MTI_B, MTS_B) \xleftarrow{\$} \texttt{MS}.\textsc{Register}()$
4. $\pi_A, \pi_B \leftarrow \textbf{None}, \textbf{None}$
5. $win \leftarrow \textbf{False}, closed \leftarrow \textbf{False}, compromised \leftarrow \textbf{False}$
6. $trans_A, trans_B \leftarrow \emptyset, \emptyset$
7. $lastrecv_A, lastrecv_B, authidx \leftarrow 0, 0, 0$
8. $inj_A, inj_B, authinj, passinj \leftarrow \emptyset, \emptyset, \emptyset, \emptyset$
9. $\mathcal{A}^{oracles_{\texttt{MS}}}(LTI_A, LTI_B, MTI_A, MTI_B)$
10. `detectTrial()`
11. **return** $win \wedge \neg closed$

`detectTrial` *is defined in Algorithm 2.*

The advantage *of adversary $\mathcal{A}$ against the messaging scheme* `MS` *in the tampering game is:*

$$\texttt{Adv}(\mathcal{A}) = \Pr\left[\text{Security-Game}(\mathcal{A}, \texttt{MS}) = 1)\right]$$

The messaging scheme `MS` *is said* secure *if for all adversaries $\mathcal{A}$,* `Adv`$(\mathcal{A})$ *is negligible.*

The game defines internal variables to keep track of the communication and of the adversary's actions:

- π_A, π_B are the states of both legitimate users.
- *win* is a flag representing if the adversary has met the winning conditions.
- *closed* is a flag representing the state of the connection (if it is closed or not).
- *compromised* records if the adversary has compromised either of the parties' long-term secrets.
- $trans_U$ is a set holding ciphertexts created by a legitimate user U.
- $lastrecv_U$ represents the highest message index received by user U.
- *authidx* is an index used during authentication steps which represents the index of the last message to authenticate.
- inj_U is a set containing messages injected to user U (which user U accepted) that are yet to be authenticated.
- *authinj* is a set used during authentication steps which holds all injected messages currently being authenticated.
- *passinj* is a set containing all injected messages that successfully passed authentication.

Those variables are updated in the oracles defined in Figure 3.3.

Figure 3.3 shows the implementation of oracles available to the adversary in the game. Each oracle presented on this figure has a counterpart for the other party, which is omitted in the figure. The figure shows the oracles with Alice sending messages and Bob receiving them.

transmit-A /B is a wrapper around SEND, which records ciphertexts created legitimately by users by storing them in the *trans* sets (see line 8 of transmit-A on Figure 3.3).

Similarly, deliver-A /B is a wrapper around RECV. If decryption happens correctly (*i.e.* the message is accepted), the oracle checks if the message is legitimate, *i.e.* is in the corresponding *trans* set. If that is not the case, it means that the message has been successfully injected and is therefore added to the *inj* set (see lines 11 and 12 of deliver-B on Figure 3.3). It also manages $lastrecv_B$ (see lines 5 and 6 of deliver-B on Figure 3.3).

The lines 4 to 7 in transmit-A in Figure 3.3 prevent a trivial attack from the attacker. For this attack, the attacker could inject some messages to a party and make their peer generate the exact same message. That way the message would be considered by the game as injected, but will be honestly

1 **procedure** createState-A(MTI):
2   `assert`$(\neg\pi_A)$
3     $\pi_A \xleftarrow{\$}$ InitState$(LTS_A, MTS_A, LTI_B, MTI)$

1 **procedure** transmit-A(m):
2   `assert`(π_A)
3     $(\pi_A, c, idx) \xleftarrow{\$}$ Send(π_A, LTS_A, m)
4   **if** $c \in inj_B \cup authinj \cup passinj$:
5         $inj_B \leftarrow inj_B \setminus \{c\}$
6         $authinj \leftarrow authinj \setminus \{c\}$
7         $passinj \leftarrow passinj \setminus \{c\}$
8     $trans_B.append(c)$
9   **return** c

1 **procedure** corruptState-A():
2   **if** π_A:
3   **return** π_A
4   **return** MTS_A

1 **procedure** auth-A():
2   `assert`$(lastrecv_A > 0 \wedge \neg\pi_A.auth \wedge \neg\pi_B.auth)$
3     $\pi_A \xleftarrow{\$}$ StartAuth(π_A)
4     $authinj, authidx \leftarrow inj_A, lastrecv_A$

1 **procedure** corruptLTS-A():
2     $compromised \leftarrow$ **True**
3   **return** LTS_A

1 **procedure** deliver-B(c):
2   `assert`(π_B)
3   **try:**
4         $(\pi'_B, m, idx) \xleftarrow{\$}$ Recv(π_B, LTS_B, c)
5   **if** $m \neq \bot \wedge idx > lastrecv_B$:
6             $lastrecv_B \leftarrow idx$
7   **if** $\neg\pi_B.auth \wedge \pi'_B.auth$:
8             $authinj \leftarrow authinj \cup \{c \in inj_B | c.idx \leq authidx\}$
9   `CheckAuthStepPassed()`
10        $\pi_B \leftarrow \pi'_B$
11   **if** $c \notin trans_B \wedge m \neq \bot$:
12            $inj_B[idx] \leftarrow c$
13   **return** m
14   **except Close:**
15        $closed \leftarrow$ **True**

Figure 3.3: Oracles in the security game accessible by the adversary. The `MS.` prefixes for functions of the messaging scheme are omitted. The `CheckAuthStepPassed` function is defined in Algorithm 1. Each oracle has a counterpart whose implementation is similar by swapping A and B in the implementation.

generated later and therefore the injected message is the same as a legitimate ciphertext. Therefore those ciphertexts are removed from injected sets.

This attack is excluded from the winning conditions, as there is no real breach of authenticity as the attacker only injects messages that get honestly generated.

The attacker can require one party to start an authentication step thanks to the auth-A /B oracles. This oracle can only be called when no parties are currently authenticating (with the $\neg\pi_A.auth \wedge \neg\pi_B.auth$ condition). This condition is enforced by the game to restrict the adversary from starting an authentication step if one is already ongoing. In practice, this condition can be enforced by messaging schemes by scheduling authentication steps in advance.

When Alice starts an authentication step, *authinj* is filled with all messages that were injected to her. Also, the index *authidx* is set to the highest index amongst messages she has received. Messages authenticated will be those with index lower than *authidx*.

When Bob receives the first authentication message, all injected messages to Bob with index lower than *authidx* are added to *authinj*. This is what happens in lines 8 and 9 of deliver-B on Figure 3.3.

Whenever the adversary calls deliver-A /B, the `CheckAuthStepPassed` function is called (defined in Algorithm 1). This function checks if the adversary has successfully injected a message and passed an authentication step. In that case, it adds the injected messages that were successfully authenticated in *passinj* and removes them from *authinj* and *inj* sets.

1 **procedure** `CheckAuthStepPassed():`
2 | **if** $authinj \neq \emptyset \wedge \neg\pi_A.auth \wedge \neg\pi_B.auth$**:**
3 | | $passinj \leftarrow passinj \cup authinj$
4 | | $inj_A \leftarrow inj_A \setminus authinj$
5 | | $inj_B \leftarrow inj_B \setminus authinj$
6 | | $authinj \leftarrow \emptyset$

Algorithm 1: `CheckAuthStepPassed` checks if the adversary succeeded in injecting a message which passed an authentication step.

More precisely, the `CheckAuthStepPassed` function performs two checks:

- it checks if *authinj* is not empty. This set contains messages successfully injected by the adversary which are in the authentication process or were already authenticated.
- it also checks if $\pi.auth$ is **None** for both parties, which means that no authentication step is currently happening.

If both conditions are met, this means that the attacker has successfully injected a message which has been authenticated (as it is in *authinj* but there is no authentication happening).

The *win* flag can only be set to **True** in the `detectTrial` function, which is defined on Algorithm 2. This happens either if parties output **True** to the out-of-band detection step but the long-term secret was not compromised or if they output **False** but communication was successfully tampered with and an authentication step passed.

In this game, the `detectTrial` function is called when the adversary stops. This strengthens the adversary by allowing them to choose the optimal moment to call the function (while in practice, this happens when users decide to perform the check out-of-band which may be at any point during the communication).

```
1 procedure detectTrial():
2 |   assert(π_A ∧ π_B ∧ ¬π_A.auth ∧ ¬π_B.auth)
3 |   d ← DetectOOB(π_A, π_B)
4 |   if d ∧ ¬compromised:
5 |   |   win ← True
6 |   elif ¬d ∧ passinj ≠ ∅:
7 |   |   win ← True
```

Algorithm 2: Out-of-band detection step performed at the end of the game.

3.5 Signal (in)security

Signal fails to meet the security definition of the security game defined in Definition 3.4 because the post-compromise security property guaranteed by the protocol only concerns privacy, but not authentication. In other words, an attacker compromising the state can impersonate parties and no mechanism prevents that if parties do not have access to an out-of-band channel.

Concretely, the following attacker wins the game with probability 1.

Proposition 3.5 *Let $\mathcal{A}$ be the following adversary:*

```
1  attacker A(LTI_A, LTI_B, MTI_A, MTI_B):
2      m_0, m_1 ≠ m'_1, m_2 ∈_R {0,1}*
3      createState-A(MTI_B)
4      createState-B(None)
5      c_0 <-$ transmit-A(m_0)
6      deliver-B(c_0)
7      c_1 <-$ transmit-B(m_1)
8      (c_1, AD_1) ← c_1
9      π_A ← corruptState-A()
10     tmp ← DH(AD_1.rpk^B, π_A.rk^A)
11     (_, ck) ← KDF(π_A.sk||tmp)
12     (_, mk) ← KDF(ck)
13     c'_1 <-$ Enc(mk, m'_1, AD_1)
14     deliver-A((c'_1, AD_1))
15     auth-A()
16     c_2 <-$ transmit-A(m_2)
17     deliver-B(c_2)
```

If the messaging scheme MS *of Definition 3.4 is the Signal protocol as described in Section 2 (and whose simplified implementation is given in Section 3.3), then attacker $\mathcal{A}$ wins the game with probability 1.*

Proof The proof is quite straightforward. First the attacker opens a legitimate session between both users and uses transmit-A and deliver-B to send a message from Alice to Bob. Then, using transmit-B, the attacker makes Bob create message 0 of epoch 1 for plaintext m_1.

However, instead of delivering the created ciphertext, the attacker drops it and forges her own ciphertext instead. To do so, she corrupts the local state of Alice using corruptState-A and uses the private ratchet key rk^A of Alice and the shared root key sk to derive a valid message key.

Indeed, by looking at the implementation of the Recv procedure on figure 3.1, the computations performed by the attacker lead to the same message key than the ones performed by Alice once she receives c'_1 when the attacker calls deliver-A.

Because c_1 and c'_1 correspond to different plaintexts, and because of the correctness of the Signal protocol, $c_1 \neq c'_1$, which means $(c'_1, AD_1) \notin trans_A$.

Finally, the attacker calls `auth-A`. As (c'_1, AD_1) has been successfully injected, $(c'_1, AD_1) \in inj_A \subset authinj$. Then the attacker transmits one message from Alice to Bob. When Bob receives the message, `CheckAuthStepPassed` is called. As StartAuth does not change the state, both conditions $authinj \neq \emptyset$ and $\neg\pi_A.auth \wedge \neg\pi_B.auth$ are satisfied, and therefore $(c'_1, AD_1) \in passinj$.

When `detectTrial` is called, the attacker wins whatever the implementation of DetectOOB is. Indeed, if DetectOOB outputs $d = 1$, then as $\mathcal{A}$ never compromised a long-term secret, *compromised* is **False** and therefore the attacker wins. On the other hand, if DetectOOB outputs $d = 0$, then as *passinj* is not empty the attacker also wins the game.

Thus the attacker wins with probability 1. □

Chapter 4

Modified Signal protocol

This section presents modifications to the Signal protocol. Those modifications transform the original Signal protocol to a new protocol whose objective is to detect, without using any out-of-band channel, a breach of authenticity for active adversaries who control the network and may corrupt devices. Only adversaries knowing a long-term secret should be able to avoid detection. Therefore if parties later use an out-of-band channel and discover some authenticity issue which has not been detected by the protocol, they can deduce the adversary knows a long-term secret.

Signal is an asynchronous protocol running on a potentially lossy channel. This means that sent messages may not be received. Therefore only messages modified or injected by the adversary are relevant for detection. Such an action is called *tampering with the communication* in this paper.

Adversaries which can modify the state of a party, even in case of compromise, are excluded.

4.1 Modification overview

The new protocol extends the Double Ratchet protocol to add some *authentication steps* in the asymmetric ratchet. Authentication steps may happen regularly at defined epochs in a session. Users might also initiate authentication steps.

In those authentication steps, long-term secrets are involved. In the existing Signal protocol, the long-term secret of a user consists of their private identity key. The modified protocol introduces a new type of long-term secret, which is a signing key $sigk^U$. In practice, the identity key is already reused to sign the medium-term public key, and therefore an implementation may reuse the identity key as a signing long-term key. This additional key is introduced to formally prove security. Signing keys have the same setting

as identity keys: it is assumed that they define the identity of the generator and are distributed honestly.

The objective of an authentication steps is dual:

1. to convince parties that they are communicating with the holder of their peer's private key,
2. to detect tampering with messages since the last authentication step.

To that end, each party sends their own view of the communication since the last authentication step. This additional information is transmitted on the usual channel used by the Signal protocol. It is included alongside regular messages exchanged between users.

In order to maintain forward secrecy, the additional information derives from ciphertexts, as ciphertexts are public and already known to a potential adversary. To save space, intermediate computations compress those ciphertexts as they are sent or received. Those intermediate computations and an authentication step are illustrated on Figure 4.1.

Adversaries having disclosed long-term secrets can avoid detection by authentication steps. However they require to use the disclosed long-term secrets to avoid detection and therefore prove their knowledge of a long-term secret, which may be detected with an out-of-band channel later.

4.1.1 Recording ciphertexts

In an authentication step, parties wish to compare their view on messages they have received. They want to know if they agree on the transcript of ciphertexts received. The order in which messages are received is not relevant as reordering may be caused by the unreliable channel. Ciphertext indices make sure both parties have the final same view of the conversation.

Plaintexts are the relevant information for parties. However, sending plaintexts in an authentication step would breach forward secrecy, as an attacker compromising the state during the authentication step would be able to decrypt older messages.

Therefore, authentication steps transmit a transcript of ciphertexts received and sent since the last authentication step. Ciphertexts are public knowledge, therefore sending them would not break forward secrecy. Moreover, a ciphertext includes the message epoch and number in cleartext, which makes it easy to order them. If every ciphertext received corresponds to a sent ciphertext, then there has been no tampering.

Sending during an authentication step the transcript of ciphertexts as such would introduce a huge overhead, as a certain number of ciphertexts would

need to be transmitted alongside the regular ciphertext. I also requires storage space to save every ciphertext between authentication steps.

Thus, instead of storing ciphertext as such, parties compute hashes of those ciphertexts and store them in a dictionary, with the index corresponding to the message epoch and number. Those hashes are then combined during the authentication step to be sent to the other party.

More specifically, at every step of the symmetric ratchet, each user U computes and stores the following hash:

$$H_{i,j}^{U} = \mathtt{H}\left(c_{i,j}^{U}\right)$$

where $\mathtt{H}$ is a hash function and $c_{i,j}^{U}$ is the ciphertext corresponding to message j sent or received in epoch i by user U.

Hash functions are defined more precisely in Definition 5.1.

This corresponds to the computations $H_{i,j}^{U}$ performed by each party on Figure 4.1.

4.1.2 Authentication steps

The computations from the previous section are used in authentication steps which are described in this part.

To guarantee security, authentication steps should happen regularly in the Signal protocol. The scheduling of authentication steps is left to developers. An authentication step is a 3-way protocol and therefore requires 3 epochs to be completed. In the following, an authentication step is described with Alice sending the first authentication message.

If authentication information is missing from a message while it should be included, then parties should dismiss the message. For instance, if an authentication step starts at epoch i, then all messages from epoch i to $i+2$ must hold authentication information. If that is not the case, parties need to discard those ciphertexts. If developers define a specific schedule for authentication steps, both parties can be aware of when an authentication step is supposed to happen.

During an epoch of an authentication step, the sender includes the information described below in every message they send to their peer. That way, the peer receives at least once the authentication information. If they do not receive it, it means no message from this epoch has been received and thus the epoch does not increase.

This additional information can be included as a header of messages. This header can be encrypted as described by section 4 of [17]. Header encryption

uses additional *header keys* which are beyond the scope of this paper. In this paper, the additional information is displayed as an addition to the plaintext and is therefore encrypted by the message keys.

In the first epoch, Alice sends the following additional information:

- the indexes of messages present in $MKSKIPPED_A$, denoted $SKIPPED_A$,
- the index of the most recent message she has received from Bob, denoted *authidx*,
- a signature $\mathtt{SIG}_{sigk^A}((authidx, SKIPPED_A))$

This allows Bob to know which messages Alice wants to authenticate. When Bob receives this message, he first checks the signature using Alice's signing public key. If it fails, Bob closes the connection. In case of success, Bob computes the following hash:

$$H_B = \mathtt{H}\left(N_B || H_B || \; ||_{(i,j) \in I_B} H^B_{i,j}\right)$$

with $\mathtt{H}$ a hash function, H_B the previous hash he computed in his last authentication step (or ε if this is the first authentication step). The concatenation happens in lexicographic order over I_B, the set of all messages sent and received by Bob since last authentication step until message *authidx*, excluding messages with index in $SKIPPED_A$. N_B is a counter used to number the authentication steps Bob has completed in this session.

The first message potentially in I_B is the message following *authidx* of the previous authentication step.

The only requirement is that authentication steps should not overlap, and parties must wait for the current authentication step to finish before starting the next one. This ensures at least two epochs pass before a new authentication step is triggered, thus letting all parties know which one is the first message to authenticate, even if it has been dropped.

When the next epoch arrives (with Bob sending messages), Bob sends along the following information:

- $SKIPPED_B$,
- $\mathtt{SIG}_{sigk^B}((H_B, SKIPPED_B))$.

When Alice receives Bob's message, she extracts the list $SKIPPED_B$ and computes the following hash:

$$H_A = \mathtt{H}\left(N_A || H_A || \; ||_{(i,j) \in I_A} H^A_{i,j}\right)$$

Once again, the concatenation happens in lexicographic order and I_A is the set of messages sent and received by Alice between the last message authenticated and message *authidx*, excluding messages in $SKIPPED_B$.

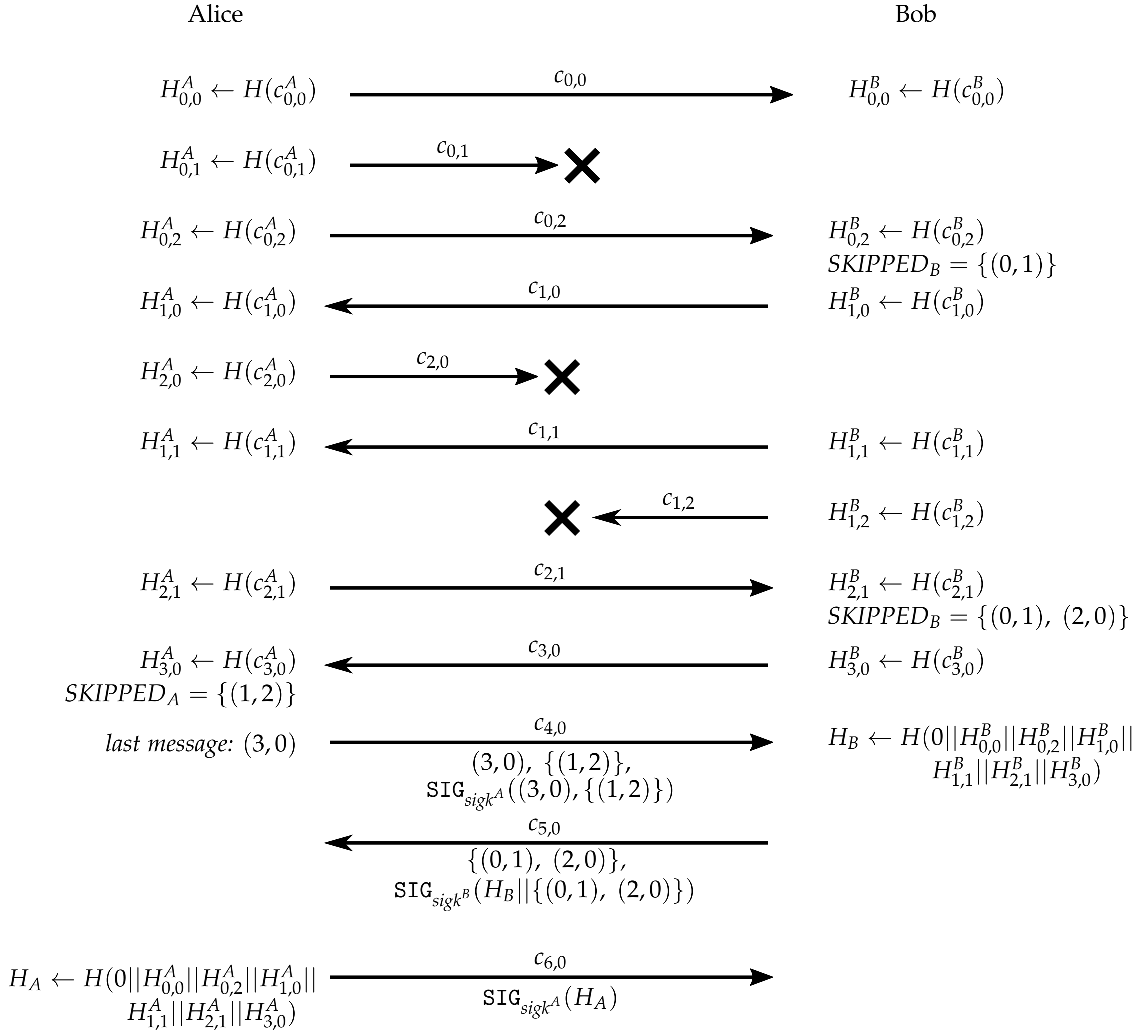


Figure 4.1: An example authentication step. The actual authentication step is performed during epochs 4 to 6, with authenticated messages from epoch 0. Additional data sent for those messages is included below arrows. For epochs 4 to 6, $H_{i,j}$ hashes are still computed by both parties, but they are omitted in this graph as they concern the next authentication step.

Alice then checks the signature received from Bob, using Bob's public signing key, on data $(H_A, SKIPPED_B)$. If it is not valid, she closes the connection. Otherwise, in the third epoch she sends a signature $\texttt{SIG}_{sigk^A}(H_A)$.

When Bob receives it, he checks the signature's validity on H_B using Alice's signing public key. If it is invalid, he closes the connection.

If both parties reach this point without closing the connection, then they have *passed the authentication step*.

On Figure 4.1, authentication steps correspond to epochs 4 to 6.

4.2 Out-of-order messages

Signal operates on an unreliable channel. Therefore messages can be dropped or arrive out-of-order.

The above construction does not deal with *late messages*. A *late message* is a skipped message such that an authentication step passed between the moment it was sent and the moment it is received.

This section presents two potential solutions for dealing with such messages.

The first solution, which is easier, is to simply discard keys for late messages when an authentication step happens. In that case, such messages will never be decrypted even if they do arrive later on. This diminishes the immediate decryption property of the Signal protocol, as it is less resilient to delayed messages.

The second solution makes the protocol more complex. Alice and Bob need to maintain another dictionary, composed of messages that were skipped in a previous authentication step, but arrived since then. They would hash the corresponding ciphertexts and add them in the final hash computation. Both Alice and Bob would send this dictionary to the other party, next to *SKIPPED* messages.

More details on immediate decryption are provided in Section 6.1.

4.3 Protocol soundness

This section shows the correctness of the protocol, *i.e.* the protocol matches Definition 3.3.

Properties 1 to 5 are easy to prove and come directly from the properties of the original Signal protocol.

To prove the remaining properties, we first demonstrate that parties always receive authentication information if they continue to communicate. Then we prove that they agree on the same set of messages to authenticate ($I_A = I_B$ with the notations of the previous section). We finally conclude by showing the hashes computed are the same and each signature verification succeeds.

Lemma 4.1 *If no adversary tampers with the communication, then parties either stay in the same epoch or receive the authentication information from their peer.*

Proof The authentication information is sent along all messages in an epoch. To advance to the next epoch, a party needs to receive at least one message from their peer's epoch, which includes the authentication information. Indeed, if the authentication information is absent from a message when it

is supposed to be present, the message is discarded and the state remains unchanged. Therefore parties cannot advance to the next epoch without receiving the authentication information. As ciphertexts are not modified or injected in this context, the information received is legitimate. □

Lemma 4.2 *If no adversary tampers with the communication, then computed sets I_A and I_B by each party are equal: $I_A = I_B$.*

Proof We begin the proof by proving by induction that for every authentication step, Alice and Bob agree on the same period of messages to authenticate, where a period of messages is defined as the lower and upper bounds on the chosen set of messages.

For the first authentication step, Alice and Bob agree on the lower bound: it is message $(0, 0)$. Then the initiator (for instance Alice) chooses the last message she wants to authenticate (one of Bob's message) and sends this index to Bob. Bob receives this index correctly thanks to Lemma 4.1. So for the first authentication step, Alice and Bob agree on the period of messages to authenticate.

Let's assume Alice and Bob agreed on the period of messages to authenticate for the last authentication step. Then the first message to authenticate in this authentication step is the message following the upper bound chosen in the previous authentication step. As at least two epochs have passed between this message and the authentication, both parties know which message it is as the total number of messages in that epoch has been transmitted. As by induction both Alice and Bob agreed on the upper bound for the last authentication step, they now agree on the lower bound for this authentication step.

Once again, the initiator (for instance Alice) chooses the last message to authenticate and transmits her choice to Bob, so both agree on the upper bound.

By induction, for every authentication step, Alice and Bob agree on the same period of messages to authenticate.

Then for a given authentication step (with Alice as initiator), I_B is the set of message indexes that Bob has sent or received in this period, except for messages that Alice did not receive (whose indices she sends in $SKIPPED_A$). Therefore I_B is the set of all messages in the agreed period that Bob sent and which arrived to Alice, and all messages that Bob received.

On the other hand, I_A is the set of message indexes that Alice has sent or received, except for messages that Bob did not receive (sent in $SKIPPED_B$). I_A is the set of messages that Alice sent and Bob received and messages that Alice received.

In both cases, I_A and I_B are the union of the set of messages received by Alice and the set of messages received by Bob. This proves that $I_A = I_B = I$. □

Lemma 4.3 *If no adversary tampers with the communication, then Alice and Bob see the same number of authentication steps.*

Given the notations of the previous section, this means that $N_A = N_B$ when they are used to compute hashes.

Proof As stated in the specification, authentication steps cannot overlap, therefore Alice and Bob can clearly number their authentication steps.

As no adversary is present, they perform authentication steps with one another and thus see the same number of authentication steps.

Therefore during authentication steps, $N_A = N_B$. □

Proposition 4.4 (Protocol soundness) *If no adversary tampers with the communication, i.e. modifies or injects a ciphertext, then parties pass authentication steps.*

Proof Parties pass authentication steps if the connection does not close during the authentication step.

As no adversary is present, the first signature verification performed by Bob always succeeds as the signed data is sent alongside the signature.

If no adversary is present, then ciphertexts received will be the same as ciphertexts sent:

$$\forall (i,j) \in \mathcal{R},\ c^A_{i,j} = c^B_{i,j}$$

(where $\mathcal{R}$ is the set of indices of received messages).

The hash function is deterministic, therefore:

$$\forall (i,j) \in \mathcal{R},\ H^A_{i,j} = \mathtt{H}\left(c^A_{i,j}\right) = \mathtt{H}\left(c^B_{i,j}\right) = H^B_{i,j}$$

Because of Lemma 4.2, Alice and Bob agree on sets $I_A = I_B = I \subset \mathcal{R}$. Moreover ciphertexts are ordered correctly as the identification information is included in the ciphertext and was stored accordingly.

If $H_A = H_B$, then by concatenating hashes the next computed H_A and H_B also verify $H_A = H_B$. Indeed Lemma 4.3 proves that $N_A = N_B$, the induction gives the previous $H_A = H_B$.

As $H_A = H_B = \varepsilon$ at the beginning of the communication, by induction hashes H_A and H_B computed in an authentication step are equal.

In the second epoch, Bob signs H_B, $SKIPPED_B$. Because we have shown $H_A = H_B$, Alice succeeds in verifying the signature on the data H_A, $SKIPPED_B$

she computes. In the third epoch, considering $H_A = H_B$, Bob also succeeds in verifying the signature.

Finally, Alice and Bob have passed the authentication step. □

The above proposition confirms that messages are decrypted as expected and that there is a way to end authentication steps, which proves properties 6 and 7.

4.4 Detecting long-term secret compromise

This section presents the protocol Alice and Bob use to detect if one long-term key is compromised. In this setting, Alice and Bob have communicated and passed at least one authentication step. The following protocol enables them to decide if there has been a tampering with their communication.

At each authentication step, parties derive a hash H_A or H_B. Authentication steps succeed if the signatures over those hashes match. Users also store a counter for each session to count the number of authentication steps performed. When parties meet, they compare the last hash they have computed, which they store in their state until next authentication step as well as the number of authentication steps performed.

The core idea is that if the hash or the number of authentication steps performed differs, it means that the communication has been tampered with, and thus parties can deduce that an attacker is present. As all authentication steps were successful, the attacker probably knows a long-term secret and used it to avoid in-band detection (*i.e.* to pass authentication steps).

4.5 Authentication steps protocol as a messaging scheme

This section shows that the authentication step protocol described above matches the messaging scheme definition from Definition 3.1. It is based on the Signal implementation from Section 3.3, where procedures are modified to include the changes described above.

There are four main modifications compared to the implementation from Section 3.3:

- there is a new long-term key, which is the signing key,
- ciphertexts are hashed and hashes are stored in the local state,
- the StartAuth procedure now modifies the state, and if an authentication step is happening, Send and Recv procedure behave accordingly,
- the out-of-band detection procedure DetectOOB is implemented as presented in Section 4.4 .

Those modifications require more state to be maintained locally by devices. Therefore, in `Connect-Send` and `Connect-Recv` procedures, the line $\pi \leftarrow \{...\}$ adds the following elements to the state:

- *ctxthashes* : $\emptyset$ is a set used to store ciphertexts hashes;
- H_{auth} : ε is the hash computed during the last authentication step;
- *lastauth* : $(0, 0)$ is the index of the last message authenticated;
- *changedEpoch* : **True** is a flag used to track if an authentication message is the first one received or not;
- n_{auth} : 0 is the authentication step counter.

The StartAuth procedure changes *auth* to some relevant information about the authentication step as described on Figure 4.2. Also, Send and Recv procedures record the ciphertexts and manage the authentication steps. Those changes are described on Figure 4.2. In order to highlight only the changes for those procedures, as well as the change in Register, the implementation is depicted as a wrapper around the original Signal procedures (referred as `Signal`.Register, `Signal`.Send and `Signal`.Recv).

1 **procedure** SEND(π_U, LTS_U, m):
2 $authinfo \leftarrow \emptyset$
3 **if** $\pi_U.auth \wedge \pi_U.i \geq \pi_U.auth.startEpoch$:
4 $\pi_U.changedEpoch \leftarrow$ **True**
5 $authinfo.\sigma \xleftarrow{\$}$ $\mathtt{SIG}_{LTS_U.sigk^U}(\pi_U.auth)$
6 **if** $\pi_U.auth.step == 1$:
7 $authinfo.auth \leftarrow \pi_U.auth$
8 **elif** $\pi_U.auth.step == 2$:
9 $authinfo.auth \leftarrow \pi_U.auth.SKIPPED_U$
10 $(\pi_U, c, idx) \xleftarrow{\$}$ Signal.SEND$(\pi_U, LTS_U, (m, authinfo))$
11 $\pi_U.ctxthashes[idx] \leftarrow \mathtt{H}(c)$
12 **return** (π_U, c, idx)

1 **procedure** RECV(π_U, LTS_U, c):
2 $(\pi'_U, m, idx) \xleftarrow{\$}$ Signal.RECV(π_U, LTS_U, c)
3 **if** $m == \perp$:
4 **return** $(\pi_U, \perp, idx)$
5 $(m, authinfo) \leftarrow m$
6 **if** $\pi_U.auth \wedge (idx.i \geq \pi_U.auth.startEpoch) \wedge \neg authinfo$:
7 **return** $(\pi_U, \perp, idx)$
8 $\pi'_U.ctxthashes[idx] \leftarrow \mathtt{H}(c)$
9 **if** $(idx.i \neq \pi'_U.i) \vee (\pi_U.auth \wedge \neg \pi_U.changedEpoch) \vee \neg authinfo$:
10 **return** (π'_U, m, idx)
11 $\pi'_U.changedEpoch \leftarrow$ **False**
12 manageAuth$(\pi_U.LTI_V, \pi'_U, authinfo)$
13 **return** (π'_U, m, idx)

1 **procedure** REGISTER():
2 $(ipk^P, ik^P, MTI_P, MTS_P) \xleftarrow{\$}$ Signal.REGISTER()
3 $(sigpk^P, sigk^P) \xleftarrow{\$}$ KeyGen()
4 $LTI_P \leftarrow \left(ipk^P, sigpk^P\right)$
5 $LTS_P \leftarrow \left(ik^P, sigk^P\right)$
6 **return** $(LTI_P, LTS_P, MTI_P, MTS_P)$

1 **procedure** STARTAUTH(π_U):
2 **if** Recv-Epoch(π_U):
3 $lastIdx \leftarrow (\pi_U.i, \pi_U.j_V)$
4 $e \leftarrow \pi_U.i + 1$
5 **else:**
6 $lastIdx \leftarrow (\pi_U.i - 1, \pi_U.j_V)$
7 $e \leftarrow \pi_U.i + 2$
8 $\pi_U.auth \leftarrow \{step : 1, startEpoch : e, authidx : lastIdx, SKIPPED_U : \pi_U.SKIPPED_U\}$
9 $\pi_U.changedEpoch \leftarrow$ **False**
10 **return** π_U

1 **procedure** DETECTOOB(π_A, π_B):
2 **return** $\pi_A.H_{auth} \neq \pi_B.H_{auth}$

Figure 4.2: Procedures modified from the original Signal implementation. The helper procedure manageAuth is defined on Algorithm 3.

```
1  procedure manageAuth(LTI_V, π_U, authinfo):
2    if authinfo.auth ∧ (¬π_U.auth ∨ (π_U.i < π_U.auth.startEpoch)):
3      if ¬Vfy_{LTI_V.sigpk^V}(authinfo.auth, authinfo.σ):
4        raise Close
5      π_U.auth ← authinfo.auth
6      π_U.auth.step ← 2
7      π_U.auth.SKIPPED_U ←
         π_U.SKIPPED_U ∩ [π_U.lastauth, authinfo.authidx]
8      I ← [π_U.lastauth, authinfo.authidx]\(π_U.auth.SKIPPED_U ∪
         π_U.auth.SKIPPED_V)
9      π_U.lastauth ← authinfo.authidx
10     π_U.auth.H ← H(π_U.H_auth || ||_{k∈sorted(I)} π_U.ctxthashes[k])
11     π_U.n_auth ++
12     π_U.H_auth ← π_U.n_auth || π_U.auth.H
13   elif π_U.auth.step == 1:
14     π_U.auth.SKIPPED_V ← authinfo.SKIPPED_V
15     π_U.auth.step ← 2
16     I ← [π_U.lastauth, π_U.auth.authidx]\(π_U.auth.SKIPPED_U ∪
         π_U.auth.SKIPPED_V)
17     π_U.lastauth ← authinfo.authidx
18     π_U.auth.H ← H(π_U.H_auth || ||_{k∈sorted(I)} π_U.ctxthashes[k])
19     π_U.n_auth ++
20     π_U.H_auth ← π_U.n_auth || π_U.auth.H
21     if ¬Vfy_{LTI_V.sigpk^V}(π_U.auth, authinfo.σ):
22       raise Close
23     π_U.auth.step ← 3
24   elif π_U.auth.step == 2:
25     π_U.auth.step ← 3
26     if ¬Vfy_{LTI_V.sigpk^V}(π_U.auth, authinfo.σ):
27       raise Close
28     π_U.auth ← None
29   elif π_U.auth.step == 3:
30     π_U.auth ← None
```

Algorithm 3: Management of the state when receiving such an authentication step message. Each `if` part corresponds to one epoch of the authentication step. U is the user calling the procedure and V their peer.

Chapter 5

Security of the new Signal protocol

This section proves that the construction given in Section 4 is secure in the detection game of Definition 3.4.

Before the actual theorem and proof of security, Section 5.1 defines the security notions for the cryptographic primitives used in this paper.

5.1 Cryptographic primitives

Section 5's goal is to prove that the scheme defined in Section 4 is secure given the assumption that the underlying cryptographic primitives are secure, namely the hash function and the signature scheme. This section defines formally those primitives and their security properties.

Definition 5.1 (Hash function security) *Let $\lambda \in \mathbb{N}$ be a security parameter and $P : \mathbb{N} \to (\{0,1\}^*)^*$ a system parameterization.*

A hash function $\mathtt{H}$ *is an efficient deterministic algorithm* $\mathtt{H} : (\mathbb{N}, \{0,1\}^*, \{0,1\}^*) \to \{0,1\}^k$ *which takes as input security parameter λ, a system parameter $\Lambda \in P(\lambda)$ and a message m of arbitrary length and outputs a digest of fixed size k.*

We denote $\mathtt{H}_{\lambda,\Lambda}(m) = \mathtt{H}(\lambda, \Lambda, m)$, or even omit the λ, Λ part if the context is clear.

The collision resistance game *is as follows:*

1. *given λ, the challenger generates $\Lambda \xleftarrow{\$} P(\lambda)$;*
2. *the challenger gives (λ, Λ) to the adversary;*
3. *the adversary $\mathcal{A}$ outputs two distinct messages $m_1 \neq m_2$.*

The adversary wins the game by finding a collision for the hash function, i.e. if $\mathtt{H}(m_1) == \mathtt{H}(m_2)$. We call advantage *of $\mathcal{A}$ her probability of winning the game:*

$$\mathtt{Adv}_{\mathtt{H}}[\mathcal{A}](\lambda) = \Pr[\mathtt{H}(m_1) == \mathtt{H}(m_2)]$$

The hash function is collision resistant if for every polynomial-time adversary $\mathcal{A}$, the advantage of $\mathcal{A}$ is negligible. We define:

$$\mathtt{Adv}_{\lambda}^{coll}(H) = sup_{\mathcal{A}}\ \mathtt{Adv}_{\mathtt{H}}[\mathcal{A}](\lambda)$$

The following definition defines signature schemes.

Definition 5.2 (Signature scheme) *A* signature scheme *is a triple* $\mathcal{S} = (\mathtt{KeyGen}, \mathtt{SIG}, \mathtt{Vfy})$ *of efficient algorithms where:*

- $\mathtt{KeyGen}$*:* $\emptyset \to \mathcal{PK} \times \mathcal{SK}$ *generates key-pairs.*
- $\mathtt{SIG} : \mathcal{SK} \times \{0,1\}^* \to \mathfrak{S}$ *is the signing algorithm. It takes as input a private key and a message and outputs a signature from an output space* $\mathfrak{S}$*.*
- $\mathtt{Vfy} : \mathcal{PK} \times \{0,1\}^* \times \mathfrak{S} \to \{0,1\}$ *is the verifying algorithm. It takes as input a public key, a message and a signature and outputs a bit to signify if the verification accepts or rejects.*

A signature scheme verifies the following correctness property:

$$\forall m \in \{0,1\}^*, (pk, sk) \xleftarrow{\$} \mathtt{KeyGen()} \implies \Pr\left[\mathtt{Vfy}(pk, m, \mathtt{SIG}(sk, m)) = 1\right] = 1$$

We denote $\mathtt{SIG}_{sk}(m) = \mathtt{SIG}(sk, m)$ *and* $\mathtt{Vfy}_{pk}(m, \sigma) = \mathtt{Vfy}(pk, m, \sigma)$ *for* $(pk, sk, m, \sigma) \in \mathcal{PK} \times \mathcal{SK} \times \{0,1\}^* \times \mathfrak{S}$*.*

The following definition gives the existential forgery security property for signature schemes. For this security property, the attacker has access to an oracle to sign arbitrary messages (CMA: Chosen Message Attack) and she needs to come up with a forgery (EUF: Existential UnForgeability).

Definition 5.3 (EUF-CMA) *Let* $\mathcal{S} = (\mathtt{KeyGen}, \mathtt{SIG}, \mathtt{Vfy})$ *a signature scheme. The EUF-CMA game runs as follows between a challenger and an attacker* $\mathcal{A}$*:*

1. *the challenger generates a key-pair* $(pk, sk) \xleftarrow{\$} \mathtt{KeyGen()}$ *and gives the public key pk to* $\mathcal{A}$*;*
2. $\mathcal{A}$ *may query the challenger with messages* $m_i \in \{0,1\}^*$*; the challenger answers with* $\sigma_i \xleftarrow{\$} \mathtt{SIG}_{sk}(m_i)$*;*
3. *Eventually* $\mathcal{A}$ *outputs a pair* $(m^*, \sigma^*) \notin \{(m_1, \sigma_1), ...\}$*.*

We denote advantage of $\mathcal{A}$ *his probability of finding a valid forgery, i.e.:*

$$\mathtt{Adv}_{\mathcal{S}}(\mathcal{A}) = \Pr\left[\mathtt{Vfy}_{pk}(m^*, \sigma^*) = 1\right]$$

A signature scheme $\mathcal{S}$ *is* EUF-CMA secure *if for all polynomial-time adversary* $\mathcal{A}$*, their advantage is negligible. We define:*

$$\mathtt{Adv}^{EUF-CMA}(\mathcal{S}) = sup_{\mathcal{A}}\ \mathtt{Adv}_{\mathcal{S}}(\mathcal{A})$$

5.2 Security proof

Before proving the security of the modified Signal protocol presented in Section 4, several useful definitions and lemmas are defined.

The following definition defines events that will be used in the proof.

Definition 5.4 *Given an adversary $\mathcal{A}$ playing the detection game of Definition 3.4, we define the following events:*

- W *is the event that the $\mathcal{A}$ wins the game,*
- FP *is the event that at the end of the game, $\neg closed \land \neg compromise \land d$ is true,*
- FN *is the event that at the end of the game, $\neg closed \land passinj \neq \emptyset \land \neg d$ is true.*

FP *and* FN *respectively stand for false positive and false negative.*

The following lemma shows that from one party's point of view, their authentication steps never overlap. This enables to number their authentication steps, and to link each authentication step to the transcript and set of messages they compute.

Lemma 5.5 *Let $\mathcal{A}$ be an adversary in the detection game of Definition 3.4 playing against the modified Signal protocol of Section 4.*

Then from Alice's point of view (resp. Bob's), no authentication steps overlap. That way, we can number Alice's authentication steps from 1 to n_A (resp. n_B for Bob).

Moreover, each passed authentication step with index i_U from U's point of view (where $U \in \{A, B\}$ is one party) produces exactly one element H_U saved in the state $\pi_U.H_{auth} \leftarrow i_U||H_U$. To compute this element, exactly one set of indexes I_U is computed.

Proof By definition, an authentication step starts from a party U's point of view when $\pi_U.auth$ becomes not **None**. Given the implementation on Figure 4.2 and Algorithm 3, an authentication step can start only when StartAuth is called or by receiving a message containing authentication information if the party is not currently performing an authentication step.

More precisely, the initiator U of the authentication step starts with $\pi_U.auth.epoch = 1$. There are two possibilities:

1. either the StartAuth procedure was called during a receiving epoch (the case on line 2 of the StartAuth procedure on Figure 4.2)
2. either the StartAuth procedure was called during a sending epoch (case on line 5).

For the first case, the initiator receives messages from the current epoch i, then sends authentication messages in epoch $i+1$. For the second case, the initiator does not send authentication messages in epoch i, receives messages in epoch $i+1$ and finally sends the first authentication messages in epoch $i+2$.

That way, in any case, authentication messages are always sent for every message in an epoch: an authentication step cannot start in the middle of an epoch. In the second case, if the initiator receives a message with authentication information in epoch $i+1$, it assumes the role of responder.

After the initiator has sent the first authentication messages in epoch j (with $j = i+1$ for the first case and $j = i+2$ for the second case), the initiator will be receiving a message from epoch $j+1$, which contains authentication information. Using it, it can compute exactly one set I_U, which is used to compute the hash H_U that is set as the value of $\pi_U.H_{auth}$. This is the case line 13 of the `manageAuth` procedure in Algorithm 3.

The initiator then goes to $\pi_U.auth.epoch = 3$ and their authentication step ends when receiving a message from the next epoch.

On the other hand, the responder V of the authentication step enters the authentication step once he receives a message containing authentication information and if he has no authentication step happening (this is described by the case line 2 of `manageAuth` in Algorithm 3). In that case, the responder computes exactly one set I_V and stores one hash H_V in $\pi_V.H_{auth}$, then goes to $\pi_V.auth.epoch = 2$. The authentication step ends once receiving a message from the next epoch.

Therefore authentication steps do not overlap, and for each one exactly one hash is added to H_{auth} and exactly one set is computed per authentication step. □

Given the previous lemma, we can number authentication steps from a user U's point of view. For $j \in [\![1, n_U]\!]$, we denote $H_U^{(j)}$ the element such that $\pi_U.H_{auth} \leftarrow j||H_U^{(j)}$ and $I_U^{(j)}$ the set computed during the jth authentication step of party U.

The following theorem states the security property for the modified Signal protocol.

Theorem 5.6 *Given some collision resistant hash function* H *and some EUF-CMA signature scheme $\mathcal{I}$, the protocol presented in Section 4 is secure for the detection game of Definition 3.4.*

More precisely, if $\mathcal{A}$ is an adversary playing the detection game against the modified Signal protocol of Section 4, its advantage is bounded according to the following inequality:

$$\texttt{Adv}(\mathcal{A}) \leq \texttt{Adv}_\lambda^{coll}(H) + 2 \cdot \texttt{Adv}^{EUF-CMA}(\mathcal{S})$$

The proof uses results from Propositions 5.8 and 5.9 which are found in the next sections.

Proof Let $\mathcal{A}$ be an attacker in the detection game.

Because of the implementation of the `detectTrial` function and because the *win* flag is only set in this function, it is immediate that $\mathrm{W} = \mathrm{FP} \sqcup \mathrm{FN}$ which are the events defined in Definition 5.4.

Therefore:

$$\begin{aligned} \texttt{Adv}(\mathcal{A}) &= \Pr[\mathrm{W}] \\ &= \Pr[\mathrm{FP}] + \Pr[\mathrm{FN}] \end{aligned}$$

Moreover, Proposition 5.9 states that $\Pr[\mathrm{FP}] \leq 2 \cdot \texttt{Adv}^{EUF-CMA}(\mathcal{S})$ and Proposition 5.8 states that $\Pr[\mathrm{FN}] \leq \texttt{Adv}_\lambda^{coll}(H)$, which proves the inequality. □

5.2.1 Upper bound for false negatives

This section gives an upper bound on the probability $\Pr[\mathrm{FN}]$ that an adversary produces a false negative in the game. It first defines some lemmas which are used to prove the proposition at the end of the section.

Lemma 5.7 *Let $\mathcal{A}$ be an adversary playing the detection game of Definition 3.4 against the modified Signal protocol from Section 4.*

If passinj $\neq \emptyset$ at the end of the game, it means that there exists some message index i such that both conditions are true:

1. *there exists a user $U \in \{A, B\}$ and an authentication step j for user U such that $i \in I_U^{(j)}$,*
2. *$c_i^A \neq c_i^B$ (where one of the ciphertext could be $\perp$ if the corresponding user has received no ciphertext for index i).*

Proof Because of Lemma 5.5, it makes sense to number the authentication steps for each user and to link to each authentication step their computed set I.

In the following we consider an execution of the game which leads to *passinj* $\neq \emptyset$ at the end of the game.

Let i be the index of a ciphertext in *passinj*. *passinj* is filled only in the `CheckAuthStepPassed` function (see Algorithm 1) if *authinj* is not empty.

authinj is filled only at two places: at line 4 of the auth-A /B oracle, or at line 8 of deliver-A /B (see Figure 3.3). For both cases, this happens when a party enters an authentication step. Let U be one user entering authentication step j such that ciphertext i is added to *authinj* when entering the authentication step (and therefore ciphertext i is added to *passinj* at the end of the authentication step).

Message i has already been received because it is added to *authinj*, and therefore is in one *inj* set when added to *authinj*, *i.e.* when the authentication step begins. Because of the StartAuth implementation on Figure 4.2, $i \leq auth.authidx$. Moreover, this message is not a skipped message for U as it has already been received.

$\pi_U.lastauth$ contains the index of the last message authenticated. As *authinj* is cleared at the end of every authentication step, having the ciphertext corresponding to index i in *authinj* means that it has not been authenticated in a previous authentication step. Moreover, it is specified in Section 4.2 that messages coming before the previous authentication step are not decrypted, which means that necessarily $i > \pi_U.lastauth$.

This proves that for this authentication step j, $i \in [\pi_U.lastauth, authinfo.authidx]$ and not in U's skipped dictionary, which means $i \in I_U^{(j)}$ (see its definition line 8 and 16 in Algorithm 3).

Therefore point 1 is proved.

Moreover, $i \in passinj$ at the end of the game. As stated earlier, this means that some user V received and accepted the ciphertext c_i^V (when it was added to inj_V). If $\bar{V}$ is the peer of V, then $c_i^{\bar{V}}$ (if it exists) cannot be equal to c_i^V because otherwise it would have been generated honestly by $\bar{V}$ and therefore removed from injected sets in lines 5 to 7 of transmit-A /B in Figure 3.3, or never added to inj_V because in $trans_V$ (see line 8 of transmit-A and line 11 of deliver-B in Figure 3.3).

This proves the second point of the lemma. □

The following proposition gives an upper bound on the probability that the adversary produces a false negative.

Note that in the construction given in Section 4, hashes are used to save space for ciphertexts. However, if no hashes were used and transcripts of actual ciphertexts were stored instead, false negatives could never happen.

Proposition 5.8 (False negative) *Let $\mathcal{A}$ be an adversary in the detection game of Definition 3.4 playing against the modified Signal protocol presented in Section 4.*

Then $\Pr[\mathrm{FN}] \leq \mathtt{Adv}_{\lambda}^{coll}(H)$

Proof Let $\mathcal{A}$ be an adversary producing event FN. Recall from Definition 5.4 that $\text{FN} = \neg closed \wedge passinj \neq \emptyset \wedge \neg d$.

In particular, *passinj* is not empty at the end of the game. According to Lemma 5.7, this implies the existence of some index i such that $i \in I_V^{(j_0)}$ for some user V and authentication step j_0 of V and such that $c_i^A \neq c_i^B$.

However, d is **False**. Given the computation of d in the DetectOOB procedure of Figure 4.2, it means that $\pi_A.H_{auth} = \pi_B.H_{auth}$.

Recall that $\pi_U.H_{auth}$ corresponds to the concatenation of the authentication step index n_{auth} and the hash $H_U^{(n_U)}$ computed in U's last authentication step (for $U \in \{A, B\}$). From lines 10 and 18 of `manageAuth` in Algorithm 3, those hashes are computed as follows:

$$H_U^{(j)} \leftarrow \texttt{H}\left(j - 1 \mid\mid H_U^{(j-1)} \mid\mid \mid\mid_{k \in \texttt{sorted}(I_U^{(j)})} \pi_U.ctxthashes[k]\right)$$

with $H_U^{(0)} = \varepsilon$.

Having $\pi_A.H_{auth} = \pi_B.H_{auth}$ means that Alice and Bob have seen the same number of authentication steps.

Moreover, if for some index $j > 1$, $H_A^{(j)} = H_B^{(j)}$, then there are two possibilities:

- either $H_A^{(j-1)} \mid\mid \mid\mid_{k \in \texttt{sorted}(I_A^{(j)})} \pi_A.ctxthashes[k] \neq H_B^{(j-1)} \mid\mid \mid\mid_{k \in \texttt{sorted}(I_B^{(j)})} \pi_B.ctxthashes[k]$;
- either they are equal.

For the first case, because $H_A^{(j)} = H_B^{(j)}$ but the two inputs to the hash function are different, we have a hash collision.

The second case would induce a propagation property and yield $H_A^{(j-1)} = H_B^{(j-1)}$.

As the equality $H_A^{(j)} = H_B^{(j)}$ is true for the last authentication step, by induction we can deduce that either there is a hash collision or for all authentication step $j > 0$:

$$H_A^{(j-1)} \mid\mid \mid\mid_{k \in \texttt{sorted}(I_A^{(j)})} \pi_A.ctxthashes[k] = H_B^{(j-1)} \mid\mid \mid\mid_{k \in \texttt{sorted}(I_B^{(j)})} \pi_B.ctxthashes[k]$$

.

This is true in particular for $j = j_0$. Recall that elements of $\pi_U.ctxthashes$ are hashes of ciphertexts computed on line 11 of Send and 8 of Recv on Figure 4.2.

As the hash function produces outputs of the same length, it means that there are exactly the same number of hashes in each concatenation. Moreover, $i \in I_V^{(j_0)}$ so one hash corresponds to the ciphertext with index i. Let's

denote $H_V = \mathtt{H}(c_i^V)$ and $H_{\bar{V}} = \mathtt{H}(c^{\bar{V}})$ the corresponding hashes (where $H_{\bar{V}}$ is at the same position in the concatenation that H_V but in the other concatenation).

Because the concatenations are equal, $H_V = H_{\bar{V}}$. However, either $c^{\bar{V}}$ does not correspond to index i, and in that case $c^{\bar{V}} \neq c_i^V$ because the index is efficiently computable from the ciphertext. Or $c^{\bar{V}}$ does correspond to index i, but we showed that $c_i^A \neq c_i^B$. In both cases, there is a hash collision.

Therefore, any case leading the adversary to a false negative shows that the adversary could produce an explicit hash collision, and therefore the reduction from the compromise game to the hash collision game is immediate.

This shows that $\Pr[\mathrm{FN}] \leq \mathtt{Adv}_\lambda^{coll}(H)$. □

5.2.2 Upper bound for false positives

This section gives an upper bound on the probability $\Pr[\mathrm{FP}]$ that the adversary produces a false positive in the game.

Proposition 5.9 (False positive) *Let $\mathcal{A}$ be an adversary in the detection game of Definition 3.4 playing against the modified Signal protocol of Section 4.*

Then $\Pr[\mathrm{FP}] \leq 2 \cdot \mathtt{Adv}^{EUF-CMA}(\mathcal{S})$.

Proof Let $\mathcal{A}$ be an adversary producing event FP.

Recall from Definition 5.4 that $\mathrm{FP} = \neg closed \wedge \neg compromise \wedge d$.

Having $\neg$*compromise* means that $\mathcal{A}$ never calls the corruptLTS oracle. Moreover $\neg$*closed* means the communication never closes. Given the implementation on figure 4.2, this means that signature checks always succeed. We will therefore build an adversary $\mathcal{B}$ in the EUF-CMA game against signature scheme $\mathcal{S}$ as a wrapper around $\mathcal{A}$, which acts as a challenger in the detection game for $\mathcal{A}$.

$\mathcal{B}$ creates two users Alice and Bob, but will not generate one of both party's signing key-pair and use the public key given by his own challenger instead, using his signing oracle to get signatures. This is performed on line 6 of the

following:

```
1  attacker B(pk):
2      b ∈_R {0,1}
3      (U_b, U_{1-b}) ← (A, B)
4      (LTI_{U_0}, LTS_{U_0}, MTI_{U_0}, MTS_{U_0}) <-$ MS.Register()
5      (LTI_{U_1}, LTS_{U_1}, MTI_{U_1}, MTS_{U_1}) <-$ MS.Register()
6      (LTI_{U_0}.sigpk^{U_0}, LTS_{U_0}.sigk^{U_0}) ← (pk, ε)
7      π_{U_0}, π_{U_1} ← None, None
8      m*, σ* ← ⊥, ⊥
9      A^{oracles_MS}(LTI_A, LTI_B, MTI_A, MTI_B)
10     return m*, σ*
```

As user U_1 is entirely generated by $\mathcal{B}$, the attacker can simulate the oracles concerning U_1 and therefore they are similar to the oracles defined in Figure 3.3. The attacker does not need to keep track of injected messages or to check if $\mathcal{A}$ wins at the end of the game.

$\mathcal{B}$ keeps track of signature forgeries. Every time $\mathcal{B}$ signs a message using the oracle provided by his challenger, he stores it. Moreover, every time a signature on the given signing public key pk is verified in the `manageAuth` function, $\mathcal{B}$ checks is the message is forged. If that is the case and the verification is successful, $\mathcal{B}$ stops and outputs the corresponding pair m^*, σ^*.

To simulate user U_0 whose private key is unknown, $\mathcal{B}$ can also use the same oracles, except for transmit$-U_0$ which is the only oracle using U_0's private signing key in the SEND procedure. Recall that the corruptLTS-A /B oracles are not called by adversary $\mathcal{A}$ and therefore $\mathcal{B}$ does not need to simulate those oracles when the event FP happens (if such an oracle is called, $\mathcal{B}$ stops and returns random values, but this case is of no interest for the reduction that follows).

U_0's signing private key is only used on line 5 of the SEND procedure defined on figure 4.2. In order to create the signature, $\mathcal{B}$ can query their own challenger with message $\pi_{U_0}.auth$ to get the signature using U_0's private key.

Therefore $\mathcal{B}$ is correctly defined and can act as a challenger for $\mathcal{A}$.

Let's now prove that if $\mathcal{A}$ produces the event FP, then $\mathcal{B}$ wins the EUF-CMA game with probability at least $\frac{1}{2}$.

When FP happens, at the end of the game parties output $d = \textbf{True}$. From the implementation of DETECTOOB on Figure 4.2, this means that $\pi_A.H_{auth} \neq \pi_B.H_{auth}$.

Given the definition of $\pi_U.H_{auth}$ on lines 12 and 20 of `manageAuth` on Algorithm 3, it means that during the last authentication step of each party:

$$\pi_A.n_{auth}||\pi_A.auth.H \neq \pi_B.n_{auth}||\pi_B.auth.H$$

There are two disjoint possibilities:

1. either $\pi_A.n_{auth} \neq \pi_B.n_{auth}$;
2. either $\pi_A.n_{auth} = \pi_B.n_{auth}$ but $\pi_A.auth.H \neq \pi_B.auth.H$.

During his last authentication step, U_1 verified a signature σ on $\pi_{U_1}.auth$ by using U_0's public signing key $sigpk^{U_0}$. This happens either on line 21 or 26 of `manageAuth` in Algorithm 3. $\pi_{U_1}.auth$ contains in particular:

- the n_{auth} computed by U_1;
- H computed by U_1;

If case number 2 happens, then $\pi_A.n_{auth} = \pi_B.n_{auth} = n$. As parties will only compute at most one hash per authentication step, there is only one signature of a set $\pi_{U_0}.auth$ produced by U_0 which has $n_{auth} = n$. If case 2 happens, then $\pi_A.auth.H \neq \pi_B.auth.H$. Recall however that the connection does not close, which means that all signatures verifications are successful. This is especially true for signature σ checked by U_1 on $\pi_{U_1}.auth$, which contains $\pi_{U_1}.auth.H$.

$\pi_{U_1}.auth$ contains $n_{auth} = n$ and $\pi_{U_1}.auth.H \neq \pi_{U_0}.auth.H$. As stated earlier, parties can produce at most one signature on sets with the same n_{auth}, therefore $\pi_{U_1}.auth$ has not been produced by U_0, *i.e.* by the signing oracle of the signature game.

Therefore if case 2 happens, σ is a valid forged signature on $\pi_{U_1}.auth$.

If case number 1 happens, then $\pi_A.n_{auth} \neq \pi_B.n_{auth}$. Recall that U_0 and U_1 are chosen uniformly at random at the beginning of the game. Because the signing key-pair and signatures are sampled and created in the same way in the detection game and in the reduction when using the signing oracle, $\mathcal{A}$ cannot distinguish which key-pair is used in the signing game. Therefore with probability $\frac{1}{2}$, $\pi_{U_0}.n_{auth} < \pi_{U_1}.n_{auth}$.

In that case, U_0 cannot have signed a set $\pi_{U_0}.auth$ with $n_{auth} = \pi_{U_1}.n_{auth}$ as it has not yet reached the correct number of authentication steps. This once again yields a valid signature forgery.

Therefore, with probability at least $\frac{1}{2}$, if $\mathcal{A}$ produces the event FP then $\mathcal{B}$ wins the EUF-CMA game.

This leads to the upper bound:

$$\Pr[\text{FP}] \leq 2 \cdot \texttt{Adv}^{EUF-CMA}(\mathcal{S}) \qquad \square$$

Chapter 6

Extensions and limitations

In order to simplify the construction from Section 4 and the security proof, some simplifications have been made to highlight the main contributions by limiting the amount of easy but cumbersome technical details. This section highlights those simplifications and provides some answers to the issues deriving from those.

6.1 Immediate decryption

As stated in Section 4.2, messages that were sent before an authentication step but arrive after it has begun are not decrypted.

This weakens the Signal protocol's immediate decryption property as every unreceived message when an authentication step starts are discarded.

This section offers two solutions. The first solution undermines the security guarantees of the delayed messages. Those messages can still be decrypted when they are received but are considered tainted and their authenticity is no longer guaranteed.

The second solution modifies the protocol from Section 4 in order to account for these delayed messages and include them in the next authentication step.

6.1.1 Weaker security game

The following modifications weaken the security game from Section 3, which enables the protocol presented in Section 4 to be secure even if messages sent before an authentication step are allowed to be decrypted afterwards.

This removes the limitation imposed in Section 4 on the immediate decryption property about late messages. Recall that we refer to a late message as a skipped message sent before some authentication step and arriving after it.

To achieve this notion, another condition is added to line 11 of the deliver-B oracle defined on Figure 3.3:

$$\textbf{if } c \notin trans_B \wedge m \neq \perp \wedge idx \geq authidx:$$

As a result of this change, only messages whose index is greater than the last message authenticated are added to the *inj* sets.

Thus, thee attacker cannot win the game by injecting a late message, which weakens the security properties achieved by our proposed protocol. However, this allows the protocol from Section 4 to decrypt those messages and still be secure given this security game.

In addition, the security proof only requires a small modification in the proof of Lemma 5.7. Recall that the proof finds a message with index i such that $i \in I_U^{(j)}$. This message is selected by taking a message in *passinj*. One step to prove this belonging is to prove that $i \in [\![\pi_U.lastauth, authinfo.authidx]\!]$.

In the original proof, the reason why $i \geq \pi_U.lastauth$ comes from the fact that older messages are not decrypted. For this new version of the security game, the reason is that those older messages are not added to the *inj* set and therefore cannot be present in *authinj* nor in *passinj*.

6.1.2 Extending the Signal protocol

This section presents another solution to the immediate decryption issue. The protocol presented in Section 4 is extended to deal with older messages.

We introduce a new array, *LATE*, which is used to store the indices of late messages when they are received. Those messages are decrypted immediately as they are received and will be authenticated in the next authentication step.

In practice, this new array is added to the state in `Connect-Send` and `Connect-Recv`. Moreover, in the Recv procedure on Figure 4.2, the following is added between lines 8 and 9:

1 **if** $idx < \pi_V.lastauth$**:**
2 │ $\pi'_V.LATE.\texttt{append}(idx)$

This adds to the *LATE* array the indexes of every late message.

Moreover, those messages are later represented in the authenticated messages. This can be done by replacing lines 8 and 16 of `manageAuth` on Algorithm 3 by the following two lines:

1 $I \leftarrow \pi_U.LATE \cup [\pi_U.lastauth, authinfo.authidx] \backslash (\pi_U.auth.SKIPPED_U \cup \pi_U.auth.SKIPPED_V)$

2 $\pi_U.LATE \leftarrow \emptyset$

Moreover, during authentication steps, when the *SKIPPED* dictionary is sent to the other party, the *LATE* array is also sent. Upon reception, it is merged to the local *LATE* dictionary.

The security proof remains unchanged apart from the proof of Lemma 5.7. There are two cases: either the chosen message from *authinj* is a not-late message, and the original proof applies, or it is a late message. Because late messages are always inserted into a *LATE* array and that those arrays are transmitted during authentication steps, then late messages are also included in I sets during authentication steps.

The main idea remains that every received message is authenticated in the authentication step following the reception.

6.2 Multi-party setting

The security definition given in Section 3 only involves two parties. However, in practice Signal conversations involve several (billions) participants who may have conversations with more than a unique peer[1].

This section gives an insight on how our proposed security game could be modified to fit the multi-party setting and gives some intuition on how the two-party setting proof can be extended to cover the multi-party setting.

Given the notion of proving long-term secret knowledge, a multi-party security definition would assume a finite number of known users whose long-term public keys are correctly distributed. The difference with the two-party setting is that the attacker can open several distinct sessions between participants.

A straightforward modification to the security game could be done as follows: given an established session between two defined users Alice and Bob, the attacker's objective remains the same as in the two-party setting, except that it can now open more sessions.

With respect to changes to the proof in the two-party setting presented in Section 5, the attacker will still need to either forge a signature on a hash computed during an authentication step or produce a hash collision.

Ratchet keys are sampled at random by parties and can therefore act as nonces. This makes it highly unlikely that all ratchet keys would be sampled in the same way, and therefore the attacker has a very small chance of being able to make one party sign the same hash across sessions. Therefore, the adversary can win the security game as in the two-party setting, but can also win if one party signs the same value in two different sessions. The

[1] We still exclude group chats in this paper, sessions can only be opened with two people but more than one session can be opened

advantage of the adversary in the multi-party setting is thus the advantage of the adversary in the two-party setting plus the probability of this event happening. The probability of having a honest party sign the same data twice is lower or equal than the probability of generating the same ratchet keys in two sessions (as public ratchet keys are directly included in the ciphertexts). Thus, if the attacker does not control the source of randomness of the parties, the probability of this collision happening is quite low.

On the other hand, opening more sessions does not help the adversary to find a hash collision.

Therefore, intuitively the protocol would still be secure in a multi-party setting, with a security bound a little bit looser than in the two party setting.

6.3 Simultaneous session initiation

Either because of chance or because of the channel reliability, two parties may decide to initiate a session simultaneously, which means that both parties send their first message before the other party receives a message.

That edge case is not considered in the main part of this paper because it makes the agreement on the order of messages complicated for the hash computation during authentication steps. Note that in the implementation of the modified Signal protocol described in Section 4, this situation cannot happen. Indeed, if both Alice and Bob send a message with epoch 0, when Alice receives a message from Bob with epoch 0, she will discard it as it has an invalid epoch number corresponding to her state. In the security game, this situation can occur but then no message will be accepted and the attacker cannot win the game.

The creators of Signal deal with this case by using the Sesame algorithm, which manages sessions from their asymmetric creation to their symmetric usage [20].

If this particular event occurs, when a party detects the simultaneous initiation, both parties choose deterministically one of both created sessions and discard the other (for instance by choosing the session initiated by the party with the smaller public identity key) [5].

The issue can be resolved in the same spirit as Section 6.1, either by defining a weaker security game in which the attacker cannot initiate two sessions at the same time but authorizing the protocol to open two sessions, or by extending the protocol to deal with this situation. In the latter case, one could perform a final authentication step on the discarded session to assess the presence of an attacker, and add the resulting hash to the session kept alive. Normally participants should be aware of this situation from the very first epoch, which means only one flow of messages will be discarded. This

flow could also be included as the first epoch of the communication for authentication step hash computation.

6.4 Deniability

One security property at the core of the Signal protocol which has not been considered so far in this report is deniability [6].

Deniability, and more specifically *offline deniability*, is the ability to deny having participating in a communication given a transcript of the conversation. The original Signal protocol is deniable because it is composed of a deniable key exchange followed by a protocol based solely on the shared key derived from this key exchange [25].

More specifically, offline deniability has the following set-up: two users, Alice and Bob, have a conversation using a messaging scheme. Alice is a honest participant, while Bob may deviate from the protocol to force Alice to prove her participation in the the conversation. Offline deniability is achieved if there exists a simulator which doesn't have access to Alice's long-term private keys and can produce transcripts of communication with the same distribution as transcripts generated between Alice and Bob.

The modified Signal protocol presented in Section 4 introduces signatures performed on the ciphertexts exchanged using the long-term secrets of parties. This very likely breaks the deniability property.

One possible mitigation to get the deniability property back would be to used Designated Verifier Signatures [12]. For this kind of security primitives, only the designated verifier is convinced of the validity of the signature, even if the verifier shares secret information.

Informally, as the original Signal protocol offers deniability, then there exists a simulator creating transcripts with the same distribution as transcripts produced between Alice and Bob. Moreover, Designated Verifier Signatures can also be produced by Bob, as no third party can verify their validity. Therefore a simulator could assign random values to signatures and a third party could not distinguish between real transcripts and generated transcripts, which would mean the Authentication Steps protocol still has offline deniability.

6.5 Application to other settings

The research presented in this paper could be extended to a more broader class of protocols where long-term keys are used to initiate a session, and the authenticity of a session relies only on the first derived shared secret.

For those protocols, an adapted version of authentication steps might provide stronger authenticity guarantees.

For instance, TLS 1.3 implements a session resumption mechanism [21]. With this mechanism, when a session is opened with the traditional handshake, a resumption shared key is created for both the client and the server. Using this resumption key, sessions can be resumed without having to authenticate the parties again with certificates. This leads to long-lived sessions and may potentially lead to authentication issues in case of certificate revocation for instance [11].

Reintroducing some type of certificate verification in the same spirit of authentication steps could lead to better authenticity guarantees while keeping the advantages of session resumption for 0-RTT session establishment.

Chapter 7

Implementation

This section describes a Java implementation for our new protocol. The implementation is based on the official Java implementation for the Signal application [24].

In the following, Section 7.1 presents how the original implementation works. Section 7.2 presents how tests have been designed to verify that modifications do not impact the existing functionalities and that they match the security specifications of Section 3. Section 7.3 presents first the modifications performed then the tests results. Section 7.4 presents some benchmarking on the overhead introduced by the changes made to the protocol. Finally Section 7.5 presents some observations made on the official implementation.

7.1 Original implementation

For the purpose of this paper, only interactions between two parties and a potential attacker are considered. The existing unit test `SessionCipherTest` implements an interaction between two parties Alice and Bob. After the initialization of both sessions, the object retrieved is a `SessionCipher`.

`SessionCipher` exposes two public methods, whose signature are as follows:

```
/**
 * Encrypt a message.
 *
 * @param  paddedMessage The plaintext message bytes, optionally padded to a constant multiple.
 * @return A ciphertext message encrypted to the recipient+device tuple.
 */
```

```
7   public CiphertextMessage encrypt(byte[]
       paddedMessage) throws UntrustedIdentityException
8
9   /**
10   * Decrypt a message.
11   *
12   * @param  ciphertext The {@link SignalMessage} to
       decrypt.
13   *
14   * @return The plaintext.
15   * @throws InvalidMessageException if the input is
       not valid ciphertext.
16   * @throws DuplicateMessageException if the input
       is a message that has already been received.
17   * @throws LegacyMessageException if the input is a
        message formatted by a protocol version that
18   *                                 is no longer
       supported.
19   * @throws NoSessionException if there is no
       established session for this contact.
20   */
21  public byte[] decrypt(SignalMessage ciphertext)
22      throws InvalidMessageException,
           DuplicateMessageException,
           LegacyMessageException,
23      NoSessionException, UntrustedIdentityException
```

Listing 7.1: Methods of `SessionCipher`

This class can be used in the following way, where `aliceCipher` and `bobCipher` both are `SessionCipher`:

```
1     byte[] alicePlaintext = "This is a plaintext
         message.".getBytes();
2     CiphertextMessage message = aliceCipher.encrypt(
         alicePlaintext);
3     byte[] bobPlaintext = bobCipher.decrypt(new
         SignalMessage(message.serialize()));
4     assertTrue(Arrays.equals(alicePlaintext,
         bobPlaintext));
```

Listing 7.2: Example of usage of the libsignal library

A `SessionCipher` object is thus used to encrypt or decrypt messages. `decrypt` can throw several exceptions if the decryption fails. Plaintexts are described

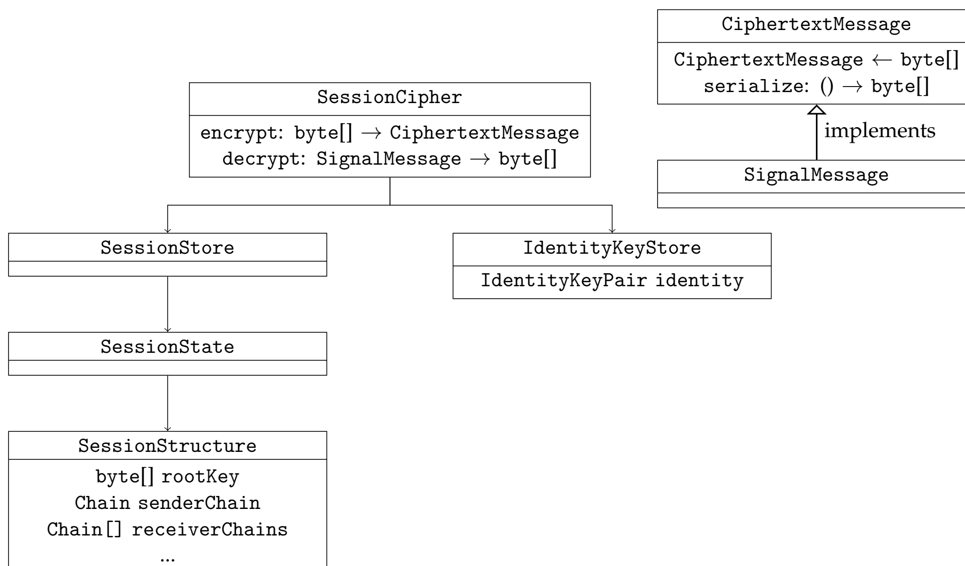


Figure 7.1: Simplified class diagram for the original Signal implementation.

as byte arrays and ciphertexts are `SignalMessage` which implements the `CiphertextMessage` interface. The interface offers a `serialize` function to transform the ciphertext to a byte array to send it on the network and can be initialized by providing a byte array.

In order to implement the `encrypt` and `decrypt` functions, a `SessionCipher` has access to different storage spaces. In particular, it has access to an `IdentityKeyStore` which holds the identity key-pair of the party. It also has access to a `SessionStore`, which holds a `SessionState` object. This existing separation between storage of long-term secrets and session secrets implicitly justifies the security model where the adversary could compromise the local state but not long-term secrets.

The `SessionState` class represents the state of a device. It is a wrapper around a `SessionStructure` variable, which is a protocol buffer [7] holding the state such as the different KDF chains and message keys.

The diagram on Figure 7.1 gives a simplified vision of the different classes and their interaction.

7.2 Tests

This section presents the different tests implemented to check if the modifications performed do not break the messaging scheme and provide the necessary security guarantees. The original Signal tests can be used to check if all previous functionalities are still working.

Test name	Authentication steps	Out-of-order	Attacker
`testNoAdvOneAuth`	1	Simple	None
`testReplaceOneAuth`	1	Simple	Replace
`testInjectOneAuth`	1	Simple	Inject
`testNoAdvTwoAuth`	2	Simple	None
`testReplaceTwoAuth`	2	Simple	Replace
`testInjectTwoAuth`	2	Simple	Inject
`testManyEpochs`	150	None	None
`testInjectAcrossAuth`	2	Late	Inject

Table 7.1: List of the different kinds of tests implemented.

Newly created tests vary according to the following parameters:

- the number of authentication steps;
- the presence of out-of-order or dropped messages;
- the presence of an attacker.

There are several types of out-of-order messages. The type denoted as *simple* describes any message dropped or out-of-order, even messages during authentication steps. They can arrive in any possible order, but messages sent before an authentication step never arrive after it. This is in contrast to *late* messages, which are out-of-order messages that can arrive even after an authentication step arrives. This is to distinguish the different cases of immediate decryption as described in Section 6.1.

We also capture two types of attackers: replace and inject. Attacker *replace* leaks the state of one party. When the party sends the next message, the attacker drops the message and replaces the ciphertext with a forged ciphertext, encrypting a different plaintext message. On the other hand, attacker *inject* leaks the state of one party at the end of an epoch, and adds a new message at the end of the epoch.

The different tests are summarized in Table 7.1. The different types of attackers and out-of-order messages are also illustrated on Figure 7.2.

There are four different types of scenarios. For all those scenarios, the authentication steps scheduling is predefined: one authentication step begins every 7 epochs. The first kind of tests, which end with `OneAuth`, have only one authentication step. The main objective of those tests is to assess the reliability of the protocol in case of out-of-order or dropped messages, even in the event of authentication messages being delayed or dropped. The second kind, ending with `TwoAuth`, introduces a second authentication step. If there is an attacker, it injects a message between the first and the second authentication step. The goal of those tests is to ensure that having one

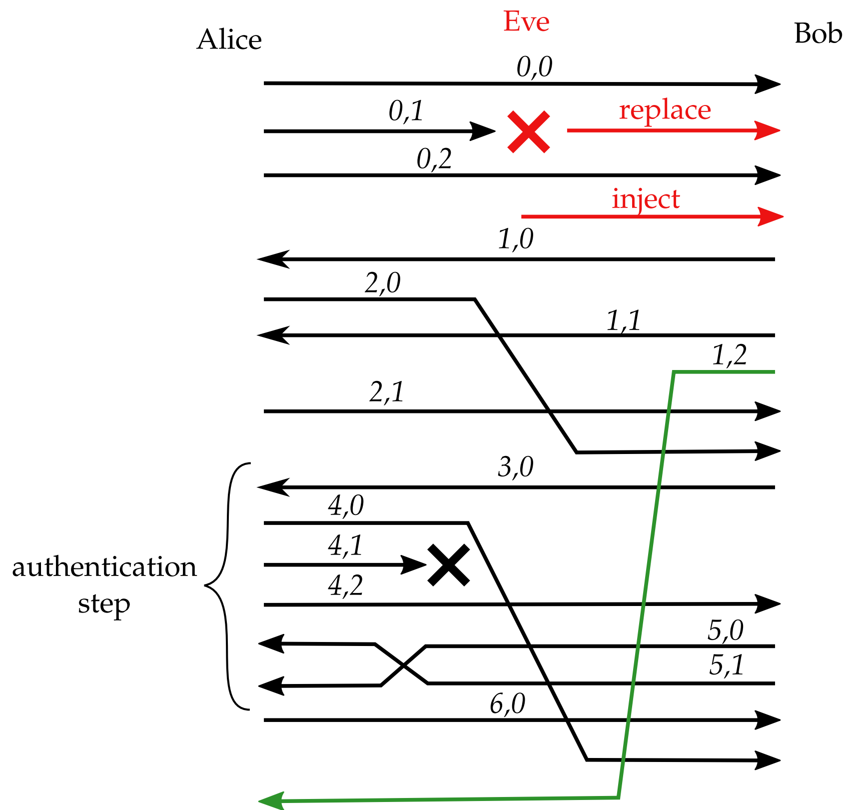


Figure 7.2: Illustration of a test. In red are both types of attackers, one injecting message (0,3) and the other replacing message (0,1). In the communication messages may arrive out-of-order (such as (2,0)) or be dropped (such as (4,1)). This is an illustration of the *simple* version for out-of-order messages. However message (1,2) in green is a *late* message as it was sent before the authentication step but arrives after it.

authentication step does not compromise the reliability of the protocol for the next authentication step. The third kind of test, `testManyEpochs`, verifies if the protocol is reliable for long-lived sessions with a lot of epochs. This test is solely a soundness test, where no adversaries are present and no messages are dropped or reordered. Finally, the fourth kind of tests, `testInjectAcrossAuth`, checks the immediate decryption property across authentication steps, verifying that such messages are indeed decrypted and that their authenticity is still guaranteed by authentication steps.

A new exception, `AuthStepSignatureException`, is defined. It is used by the protocol to indicate some signature verification in an authentication step. It corresponds to the **Close** event from Section 3.

```
1  public void testNoAdvOneAuth() throws ...
2  {
3    Pair<SessionCipherAuthStep, SessionCipherAuthStep
```

```
      > ciphers = initializeSessionsV3();

    runOneAuthStep(ciphers.first(), ciphers.second(),
        AttackerType.NONE);
  }

  public void testReplaceOneAuth() throws ...
  {
    Pair<SessionCipherAuthStep, SessionCipherAuthStep
      > ciphers = initializeSessionsV3();

    try {
      runOneAuthStep(ciphers.first(), ciphers.second
        (), AttackerType.REPLACE);
      throw new AssertionError("Authentication step
        should not succeed when an adversary is
        present");
    } catch(AuthStepSignatureException e) {
      // With the adversary the authentication step
        should fail.
      // So if we're here this is good.
    }
  }
```

Listing 7.3: Implementation of two tests case

Tests all follow the same structure, which is illustrated on Listing 7.3. First, two `SessionCipher` objects are initialized (one for Alice and one for Bob). They actually are `SessionCipherAuthTest` objects, which inherits from `SessionCipher`. It may override the `encrypt` and `decrypt` functions, and add some procedure for leaking the state to let an attacker inject messages.

One function is called to perform the simulation. In the listing this function is `runOneAuthStep`, which corresponds to the test case with one authentication step. The third parameter determines the type of attacker. If no attacker is present, then no exceptions should occur, as in `testNoAdvOneAuth`. However if an adversary is present, `SessionCipherAuthTest` should occur, therefore the exception is caught and an error is thrown if the exception does not occur.

In addition to those above, a test has been designed to verify the correctness of the out-of-band protocol for detecting adversaries knowing long-term secrets. In this test, an adversary leaks the local state and long-term secrets of both parties and impersonate them. At the end of the communication, parties perform the out-of-band protocol. The adversary should

be detected, which is determined in the Java code by a new exception `OutOfBandCheckException`.

Moreover, every soundness test described above (in which no adversary is present) is augmented to include at the end of the communication an out-of-band check, which should not raise any exception.

7.3 Extending the protocol

This section presents how the original Signal implementation presented in Section 7.1 has been modified to integrate the changes from Section 4.

The second part of this section describes how the implementation works as expected.

7.3.1 Presentation of the modifications

The actual Java implementation is very similar to the pseudo-code used in Figure 4.2. Figure 7.3 shows the different extensions performed on the original Signal implementation. The class diagram comes from Figure 7.1 and coloured text indicates some modification performed on the original implementation.

The `SessionCipher` class is extended to `SessionCipherAuthStep` to add functionalities.

A new class `Attacker` represents both types of attackers. A `leakState` method is added to the `SessionCipherAuthStep` class to return the local state of a party. Those additions are displayed in dark blue on Figure 7.3.

Additions in dark red are used to implement the global numbering of epochs. Indeed, the original implementation identifies each epoch by the ratchet public key corresponding to that epoch and not by an epoch number. However, the modified protocol requires a monotonically increasing epoch number in order to identify which messages are authenticated. Therefore the state holds a list associating the most recent ratchet public keys to epoch numbers. This list is used in the `getEpochNumber` method to convert a ratchet public key to an epoch number.

Additions in dark spring green denote modifications made to hash ciphertexts and store the hashes in the `ctxtHashes` variable of the state. This happens in the `storeCiphertext` method. In the original Signal implementation, only keys of skipped messages are stored, as the *MKSKIPPED* dictionary (see Section 2.6). The `updateSkipped` method also stores the indices of those skipped messages in the `currentSkipped` array (which corresponds to the *SKIPPED* dictionary presented in Section 4.1.2). When an authentication step begins, this vector is copied to `ourSkipped` in the `fixSkipped` method.

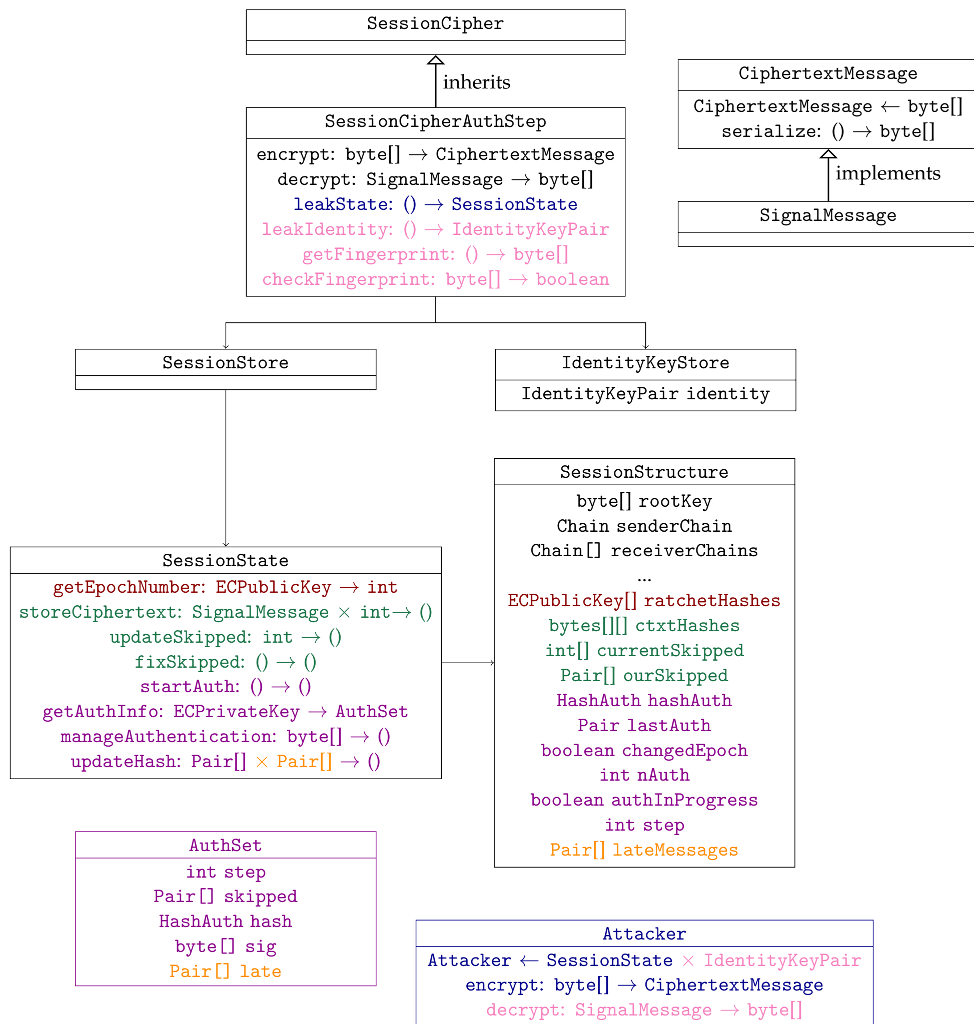


Figure 7.3: Extended implementation for authentication steps. The encrypt and decrypt functions are overridden accordingly. Blue additions enable the presence of an attacker, red to convert public ratchet keys to epoch numbers, green to hash ciphertexts and keep track of skipped messages, purple to perform authentication steps, orange to keep track of late messages and pink to implement the out-of-band protocol.

Additions in dark magenta show the additional state needed to perform authentication steps. `hashAuth` stores the index and hash value of the last authentication step and `lastAuth` is the index of the last authenticated message. There are several additional functions: `manageAuthentication` is the Java version of the `manageAuth` function of Algorithm 3, `updateHash` performs the hash computation for a given authentication step and `getAuthInfo` returns the authentication information to send.

This authentication information is of a new type `AuthSet`. It can be serialized and deserialized and is prepended to every Signal plaintext. In order to correctly decode it, the first four bytes of every plaintext are used to encode the size of this additional information.

Signatures are computed by already existing code, in the same way signatures of medium term keys are performed. The hash function used here is SHA-512.

Changes in dark orange highlight how late messages (which arrive after the authentication step following their sending) are dealt with. It adds a new dictionary to keep track of late messages. This dictionary is added to the authentication information transmitted and is used to compute authentication step hashes.

Finally, changes in persian pink implement the out-of-band detection protocol, with a method for creating a fingerprint and another method for checking the validity of the peer's fingerprint. In case of failure, the `OutOfBandCheckException` is raised. Moreover, the adversary is extended to be able to leak the identity key of each party and to entirely simulate a honest party.

7.3.2 Test results

The whole implementation can be found on the ETH Gitlab[1].

Tests were performed on three different implementations:

1. the original implementation; tests can be launched on branch `test0` (commit 83bb9964);
2. the implementation without late messages management (as described on Listing 7.3 without the orange modifications); tests can be started on branch `test1` (commit ce316071);
3. the full implementation presented on Listing 7.3; tests can be started on branch `test2` (commit c90ef00b).

The results of tests can be found on Table 7.2. Tests can be launched with the following command:

[1] `https://gitlab.ethz.ch/bdowling/continuous-auth-in-sec-msg`

	Original Signal tests	testNoAdvOneAuth	testReplaceOneAuth	testInjectOneAuth	testNoAdvTwoAuth	testReplaceTwoAuth	testInjectTwoAuth	testManyEpochs	testInjectAcrossAuth
Original	✔	✔	✘	✘	✔	✘	✘	✔	✘
Without Late	✔	✔	✔	✔	✔	✔	✔	✔	✘
With Late	✔	✔	✔	✔	✔	✔	✔	✔	✔

Table 7.2: Tests results

```
./gradlew clean test --tests TestName
```

with `TestName` the name of the group of tests to launch. There are 3 original Signal tests, `SessionBuilderTest`, `SessionCipherTest` and `SimultaneousInitiateTests`. All tests described in Section 7.2 are in the group `AuthStepTest`.

The original version fails all tests where an adversary is present: it does not detect the described adversarial behaviour as there are no authentication steps.

For the modified versions, every test without an adversary is working as expected, which tells that the protocol is sound. Moreover, tests with adversary but without late messages succeed. This shows the desired security properties are met.

The implementation without management of late messages fails the test where a late message is injected, but the implementation with management of late messages succeeds.

Every successive implementation pass the original Signal tests.

In addition to the tests in Table 7.2, the soundness tests have been extended to check the soundness of the out-of-band detection protocol.

Moreover, the new test `testOutOfBand` checking that an adversary with long-term secret knowledge is detected by the out-of-band procedure succeeds.

Those tests concerning the out-of-band protocol can be run on the `testOOB` branch (commit 6574c367).

7.4 Overhead introduced by authentication steps

This section gives some remarks about the implementation. The additional security properties come at the cost of some overhead, in term of bytes transmitted, of additional space taken on the device and on computational overhead.

7.4.1 Space and computation overhead

Authentication steps require more data to be computed and stored. This data is kept in the state and corresponds to:

- ciphertext hashes;
- public ratchet keys for epoch numbering;
- lists of skipped and late messages.

This data is purged from the state when it has been used after each authentication step.

Therefore the storage overhead is a function of the following parameters:

- the size of a hash;
- the channel reliability: if the channel is less reliable, the state will hold more indices for keeping track of dropped messages;
- the Signal parameters on immediate decryption: data for messages which will no longer be decrypted can be safely purged;
- the frequency of authentication steps: the more frequent authentication steps are, the less data needs to be stored;
- the average number of messages per epoch: the more messages there are in an epoch, the more messages there will be between authentication steps and therefore the more ciphertext hashes need to be stored.

The average number of messages per epoch is the most crucial parameter, as it can induce an unbounded overhead as compared to the original Signal state's size. Indeed, the sender cannot know in advance which messages the peer has received. Thus there is no alternative possibility than storing every ciphertext hash individually.

On the other hand, keeping the skipped messages list and the public ratchet keys do not overly impact the memory as those are already stored in the original Signal state. The only additional data stored is the conversion from keys to actual integer indexes which is of the same magnitude.

Ciphertexts overhead is impacted by the same parameters. Indeed, a hash is included in ciphertexts in authentication steps as well as the list of skipped

messages. The less authentication steps there are, the lower the ciphertext overhead will be on average, as only messages during authentication steps introduce a significant overhead. However if an authentication step epoch is long, this can introduce an unbounded overhead.

As for computational overhead, computing the ciphertext hashes involves one hash invocation. During authentication steps, some signature operations are also required.

The signature scheme employed by Signal and during authentication steps is XEdDSA [16]. Signing data is typically the same amount of work as a Diffie-Hellman key computation. As there is at most one additional signature generation per epoch introduced by authentication steps, the computational overhead is at most the same magnitude as the original protocol.

7.4.2 Benchmarking the space overhead

In order to give an estimation of the space overhead induced by the authentication steps, a simulation of a communication has been performed to compare the ciphertext sizes as well as the states sizes.

Input texts come from a SMS dataset [3] composed of English text messages provided by Singaporean students [4]. This dataset only provides messages samples but not entire conversations with relations between messages.

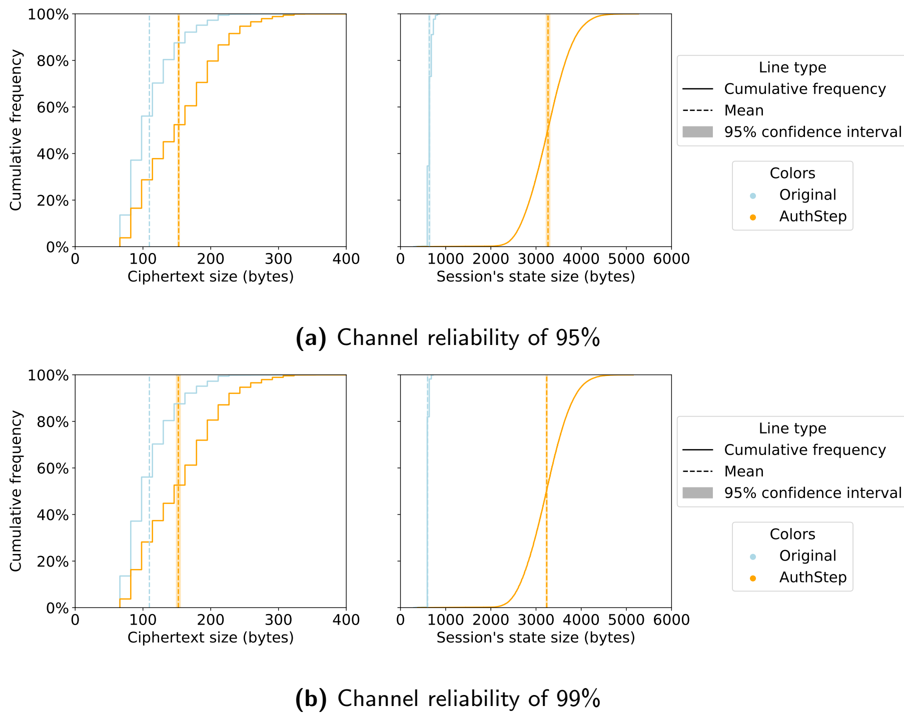
Therefore the communication has been simulated as follows. First an average number of messages for each epoch is chosen. Then for each epoch, the actual number of messages in this epoch is chosen randomly using a Poisson law.

Moreover, the channel is unreliable, meaning that some messages are dropped randomly. The reliability of the channel is a parameter of the experiment and its impact is studied in the following.

For the benchmarking performed in this section, the hash function is SHA-512 as presented earlier. Authentication steps happen regularly every 7 epochs.

Figure 7.4 compares the ciphertext and state sizes every time a ciphertext is sent. 10000 messages were sent, with on average 3 messages per epoch.

Randomness here depends on a chosen seed. To verify the seed is not biased, 95% confidence intervals have been computed to assess the mean ciphertexts and sessions sizes. The confidence intervals are displayed on Figure 7.4, sometimes not appearing as they are really small (the length of the confidence intervals are around 5 bytes for ciphertext sizes and around 20 bytes for session sizes). For computing those confidence intervals, only two different seeds were required and the intervals are already sufficiently small to



(a) Channel reliability of 95%

(b) Channel reliability of 99%

Figure 7.4: Comparison of ciphertexts and states sizes between the modified and the original Signal protocol. For this experiment, 10000 messages were exchanged with an average of 3 messages per epoch.

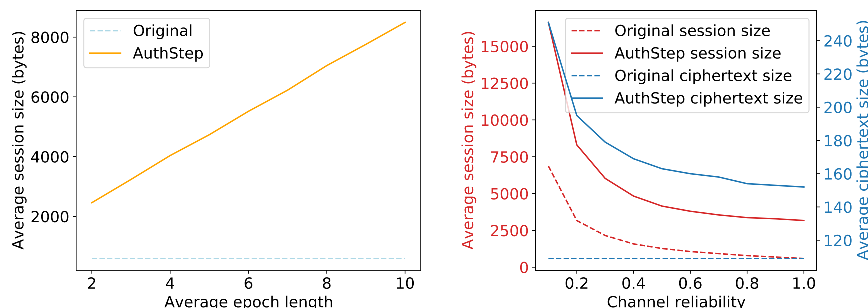
be confident in the results obtained (the diameter of the confidence interval is more than 50 times smaller than the mean). Those confidence intervals assume that each seed leads to independent experiments, where the ciphertexts and sessions sizes means are distributed according to a normal law of unknowns mean and deviation.

As discussed in the previous section, the experiment shows that even if sessions take around 4 times the size of an original session, the session size is still bounded throughout the experiment and does not grow indefinitely. As for ciphertexts, there is a bigger proportion of large ciphertexts in comparison to the original Signal protocol, which comes from the additional data.

Figure 7.5 studies the impact of the channel reliability and the length of epochs on the size of the state.

As stated in the previous section, while the average epoch length does not impact the session size in the original protocol, it does increase linearly for the authentication steps, as every ciphertext hash needs to be stored. However this does not impact the ciphertext sizes (graph not shown here).

The less reliable the channel, the more state parties need to store in order to



(a) Evolution of the state's size with longer epochs. The channel is reliable in this experiment.

(b) Evolution of the state and ciphertext average size with the channel reliability. On average there are 3 messages per epoch.

Figure 7.5: Overhead introduced by the authentication steps depending on the channel reliability and average epoch length. For this experiment, 10000 messages were exchanged.

keep immediate decryption. This explains the monotony of both curves in the channel reliability graph. Moreover, while the channel reliability does not impact the ciphertext sizes in the original protocol, having a more unreliable channel makes the ciphertexts bigger in the authentication steps messages. Indeed, indices of skipped messages need to be sent to the other party during authentication steps, which makes ciphertexts bigger on average.

7.5 Observations on the official implementation

While implementing the proposed protocol, I found out that the state deletion strategy in the code [24] is different from the strategy described in analysis such as [1] or [6], even if the latter is based on the implementation.

The official Signal specification [17] itself is unclear, and the strategy used in the implementation is implied but not made explicit.

In [1] or [6], post-compromise security happens after two epochs, which means that two epochs after a state compromise, security is restored. This happens by the deletion of no longer necessary state once an epoch is ended. However, the implementation deletes this state only 5 epochs later, which is a hardcoded value.

Thus, we come to the conclusion that the official implementation has weaker post-compromise security properties than claimed in the literature.

7.5.1 Design of an attack breaking post-compromise security

We use the previous observation to introduce the following attack breaking post-compromise security in the Signal implementation.

In the middle of a communication between Alice and Bob, an attacker leaks the state of Alice. Let's assume we are at epoch i and Alice sends n_i messages in this epoch. Then the attacker can, by using the leaked state, create a valid ciphertext for message $(i, n_i + 1)$ (and even more messages).

In epoch $i + 3$, according to the usual post-compromise security definition, as the adversary has remained passive on the network, it should not be able to decrypt or inject new messages.

However, with the Signal implementation, as the state for epoch i is not yet deleted in epoch $i + 3$, the attacker can inject messages (for epoch i) to Bob, which breaks post-compromise security.

Note however that security is restored 5 epochs after compromise, therefore the implementation still guarantees a weaker post-compromise security property.

This attack has been implemented both for the official Java implementation (see branch *attackjava*, commit b0310d3e)and for the official C implementation [23] (see branch *attackc*, commit c4ccf066). For both implementations, a test has been designed to check if injected messages are rejected upon reception. Both tests fail as the post-compromise security is not strong enough and injected messages are decrypted by the victim.

7.5.2 Explanation behind this weaker property

In the implementation, the state of parties holds one `Chain` object per receiving epoch. This object contains both the chain key for this epoch and potentially computed message keys for missed messages. When the next receiving epoch arrives, the receiving party knows the total number of messages sent by their peer as it is included in the associated data of the ciphertext. The chain key should then be used to compute all message keys for this epoch and then be discarded.

However, in the implementation, developers made the choice to disregard the total number of messages sent in the previous epoch, and instead keep the chain key without computing in advance message keys for missed messages.

This saves computation time and space as the keys are not computed if messages never arrive (as they were already missed). The immediate decryption property is still valid as the chain key is kept and message keys can be derived if needed.

Note also that in the theoretical definitions of post-compromise security, an attacker is allowed to inject messages from previous epochs which would have been missed, which is what happens here. However in this case, parties can detect that the injected message is not legitimate.

7.5.3 Fixing post-compromise security

On the *fix* branch (commit c2123110), I have implemented a fix for the Java library. When a new receiving epoch begins, the value of the total number of messages is used to derive all message keys for this epoch, then the chain key is deleted from the state. This brings back a strong post-compromise security, as shown by the test which no longer fails.

Chapter 8

Conclusion

Authenticity of messages exchanged using a protocol like Signal rely on out-of-band protocols to verify that users in a session have matching states.

This paper offers a solution to detect MitM attackers using the in-band channel. This solution relies on long-term keys which still need to be authenticated with an out-of-band channel. It adds authentication steps to the Signal protocol, which may be triggered regularly. Those authentication steps certify that a user is communicating with a holder of their peer's long-term secret.

At the expanse of computational and space overhead, MitM adversaries who do not know long-term secrets are detected by the protocol proposed in this paper without using an out-of-band channel.

Moreover, if out-of-band checks are still required to verify the authenticity of long-term secrets, they can also be used to detect adversaries having used long-term secrets to avoid detection on the in-band channel. This enables users to refresh their long-term secrets only if attackers have demonstrated a compromise of their long-term secrets.

While this paper presents authentication steps for the Signal protocol, the core ideas of the new protocol may be adapted to other settings, such as the TLS 1.3 sessions resumption mechanism, to provide additional authenticity guarantees.

Bibliography

[1] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 129–158. Springer, 2019.

[2] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In *Annual International Cryptology Conference*, pages 619–650. Springer, 2017.

[3] T. Chen and Kan Min-Yen. The National University of Singapore SMS Corpus. [Dataset] https://doi.org/10.25540/WVM0-4RNX, 2015.

[4] Tao Chen and Min-Yen Kan. Creating a live, public short message service corpus: the nus sms corpus. *Language Resources and Evaluation*, 47(2):299–335, 2013.

[5] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. Cryptology ePrint Archive, Report 2016/1013, 2016. https://eprint.iacr.org/2016/1013.

[6] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017*, pages 451–466, 2017.

[7] Google Developers. Protocol buffers. https://developers.google.com/protocol-buffers, 2021.

[8] Benjamin Dowling and Britta Hale. There can be no compromise: The necessity of ratcheted authentication in secure messaging. *IACR Cryptol. ePrint Arch.*, 2020:541, 2020.

[9] Simon Eberz, Martin Strohmeier, Matthias Wilhelm, and Ivan Martinovic. A practical man-in-the-middle attack on signal-based key generation protocols. In *European symposium on research in computer security*, pages 235–252. Springer, 2012.

[10] Facebook. Messenger secret conversation, technical whitepaper. Technical report, Facebook, 2016. https://about.fb.com/wp-content/uploads/2016/07/messenger-secret-conversations-technical-whitepaper.pdf.

[11] Steffen Fries. Question to TLS 1.3 and certificate revocation checks in long lasting connections. IETF Mail Archive, 2021. https://mailarchive.ietf.org/arch/msg/tls/vTxwj2iShME6c7AHg_Ub-_eS_fM/.

[12] Markus Jakobsson, Kazue Sako, and Russell Impagliazzo. Designated verifier proofs and their applications. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 143–154, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[13] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 435–450. IEEE, 2017.

[14] Hugo Krawczyk and Pasi Eronen. Hmac-based extract-and-expand key derivation function (hkdf). RFC 5869, RFC Editor, May 2010. https://www.ietf.org/rfc/rfc5869.txt.

[15] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic curves for security. RFC 7748, RFC Editor, January 2016. http://www.ietf.org/rfc/rfc7748.txt.

[16] T. Perrin. The xeddsa and vxeddsa signature schemes. Technical report, 2016. https://whispersystems.org/docs/specifications/xeddsa/.

[17] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. *https://whispersystems.org/docs/specifications/doubleratchet/*, 2016.

[18] Trevor Perrin and Moxie Marlinspike. Recovery from compromise. *https://whispersystems.org/docs/specifications/doubleratchet/#recovery-from-compromise*, 2016.

[19] Trevor Perrin and Moxie Marlinspike. The x3dh key agreement protocol. *https://whispersystems.org/docs/specifications/x3dh/*, 2016.

[20] Trevor Perrin and Moxie Marlinspike. The sesame algorithm: Session management for asynchronous message encryption. *https://signal.org/docs/specifications/sesame/*, 2017.

[21] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, RFC Editor, August 2018.

[22] Gustavus J Simmons. Symmetric and asymmetric encryption. *ACM Computing Surveys (CSUR)*, 11(4):305–330, 1979.

[23] Open Whisper Systems. libsignal-protocol-c. https://github.com/signalapp/libsignal-protocol-c, 2021.

[24] Open Whisper Systems. libsignal-protocol-java. https://github.com/signalapp/libsignal-protocol-java, 2021.

[25] Nihal Vatandas, Rosario Gennaro, Bertrand Ithurburn, and Hugo Krawczyk. On the cryptographic deniability of the signal protocol. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security*, pages 188–209, Cham, 2020. Springer International Publishing.

[26] Whatsapp. Whatsapp security advisories, 2021. https://www.whatsapp.com/security/advisories.

[27] WhatsApp. Whatsapp security page. Technical report, Facebook, 2021. https://www.whatsapp.com/security/.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

CONTINUOUS AUTHENTICATION IN SECURE MESSAGING

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	First name(s):
POIRRIER	Alexandre, Phuoc, Jean-Marc, Basile

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	Signature(s)
Zürich, on 04/03/2021	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.