Continuous Authentication in Secure Messaging

Benjamin Dowling[1], Felix Günther[2], and Alexandre Poirrier[3]

[1] University of Sheffield, London, UK
b.dowling@sheffield.ac.uk
[2] ETH Zürich, Zürich, Switzerland
mail@felixguenther.info
[3] École polytechnique & Direction Générale de l'Armement, Paris, France
alexandre.poirrier@polytechnique.org

September 13, 2022

Abstract. Secure messaging schemes such as the Signal protocol rely on out-of-band channels to verify the authenticity of long-running communication. Such out-of-band checks however are only rarely actually performed by users in practice.
In this paper, we propose a new method for performing continuous authentication during a secure messaging session, without the need for an out-of-band channel. Leveraging the users' long-term secrets, our *Authentication Steps* extension guarantees authenticity as long as long-term secrets are not compromised, strengthening Signal's post-compromise security. Our mechanism further allows to detect a potential compromise of long-term secrets after the fact via an out-of-band channel.
Our protocol comes with a novel, formal security definition capturing continuous authentication, a general construction for Signal-like protocols, and a security proof for the proposed instantiation. We further provide a prototype implementation which seamlessly integrates on top of the official Signal Java library, together with bandwidth and storage overhead benchmarks.

Keywords: Secure messaging · Authentication · Compromise detection · Post-compromise security.

1 Introduction

The Signal end-to-end encrypted messaging protocol [20] is used by billions of people [28], in the Signal app itself and other messengers such as Facebook Messenger [11] and WhatsApp [26]. The security of encryption keys used in the Signal protocol relies on two composed cryptographic protocols. First, a Diffie–Hellman-style key exchange protocol involving long-term asymmetric keys (whose public part is distributed via a central Signal server) is used to derive a shared secret. This initial shared secret is then used by parties in Signal's Double Ratchet protocol [17] to derive symmetric keys, used to encrypt messages between the two communicating parties.

Signal's Security. There have been numerous analyses of the security of the Signal protocol, as has been recapitulated in [22], which show that security properties for messaging protocols come in a variety of flavours with different adversary powers and strengths.

For these analyses, models separate different types of secrets: *session secrets* (like ephemeral randomness or state), which are used throughout the Double Ratchet protocol, and *long-term secrets*, used only in the initial key agreement. Some [7,6] study the security of the Signal protocol in its entirety, including the X3DH key exchange. Others [1,2,18,13,10,15] focus exclusively on the ratcheting part of the protocol, thus considering only session secrets. Among other security properties, [6] and [1] confirm that against strong adversaries who control the network and can adaptively compromise session and long-term secrets, Signal offers forward secrecy (meaning that the secrecy of messages sent before a secret leakage are still secure) as well as post-compromise security [7] (meaning that after users exchange *unmodified* messages, security is restored or "healed").

A preliminary version of this paper appears in the proceedings of the *27th European Symposium on Research in Computer Security (ESORICS 2022)*, DOI: 10.1007/978-3-031-17146-8_18. This is the full version.

As pointed out by [9], the definition of post-compromise security is quite restrictive in the sense that the adversary needs to remain completely passive for security to be restored. Indeed, if the adversary remains active after a state leakage and continuously injects forged messages, authenticity is never restored. [9] instead proposes a protocol relying on an out-of-band channel (like email, SMS, or an in-person meeting) to detect such active adversaries, leveraging additional *fingerprints* computed by the protocol and compared over the out-of-band channel. However, while detection clearly is a good step towards mitigating attacks, it does not prevent the actual attack from continuing.

Locking Out Active Adversaries. The question we are interested in for this work goes one step further:

Can we, post-compromise, lock out even an active adversary from a messaging communication?

Clearly, the answer in general is: No. An active network adversary that fully compromises a user's device, including all session and long-term secrets can, by design, fully impersonate that user subsequently. Our angle to approach the above question is to distinguish (and thus better leverage), the difference between the use of session and long-term secrets in the Signal protocol.

More specifically, what if we can leverage that user long-term secrets are harder to compromise, e.g., due to stronger randomness sources or better protection in hardware? Indeed, messaging services like WhatsApp or Signal are now deployed [27] on devices which have access to secure hardware such as Trusted Platform Modules (TPM) in which long-term secrets can be stored more safely. Phones on the other hand can use smart-cards to store their long-term keys. A typical attack scenario are border searches, where a travelling user would need to give away their phone or laptop for analysis, risking the leak of their session secrets. An adversary could then remain an active Man-in-the-Middle (MitM), possibly on nation-controlled networks, yet long-term keys in smart cards or a TPM might not have been leaked. Such a breach of authenticity can have a high impact in the case of Signal as sessions are typically months or years long.

We can then ask:

Can we, post-compromise, lock out even an active adversary from a messaging communication that compromised session, but not long-term secrets?

Notably, the answer for Signal is still: No. We will show how to, generically, turn this into a Yes.

1.1 Contributions

The main contributions of this paper are

1. a formal, game-based definition of *continuous authentication*, a post-compromise security property locking out active adversaries who have not compromised long-term secrets;
2. a demonstration that protocols similar to Signal do not meet the security requirement;
3. a *generic extension* for messaging protocols to provide them with provably-secure continuous authentication;
4. a prototype implementation of our extension for Signal and an *analysis and benchmark* of the overhead it introduces;
5. exposing a discrepancy between the post-compromise security by Signal's official library and what is claimed in the literature.

Continuous Authentication. We begin in Section 2 by capturing what locking out an active adversary from a messaging communication formally means. The basis for this is a formal syntax, following [1], that captures generic *messaging schemes* that operate on an unreliable channel, derive an initial shared session secret using long-term keys and then derive further session secrets from previous state, new randomness and possibly long-term keys. This in particular encompasses the Signal Double Ratchet protocol.

We then put forward a game-based security definition for *continuous authentication*, which guarantees two core properties, the first one:

1. An active MitM adversary that compromised a user's communication state gets *locked out* of the communication *unless* it has also compromised the user's long-term secrets.
2. Users can correctly *decide whether* long-term secrets have been compromised in an active attack, via an out-of-band channel.

The first property captures the desired strengthening of messaging protocols. Let us first understand what prevents Signal's post-compromise security approach from achieving it. In the Double Ratchet protocol [17], which we recap in more detail in Section 3, all secrets are derived from prior established secret state (the so-called *root* and *chain* keys) and new randomness generated by the users (the Diffie–Hellman *ratchet* keys). The former is revealed in a full state compromise and the latter are unauthenticated and can hence be impersonated by the adversary. This issue is not specific to Signal: we show in Section 3.1 that this issue generalizes to messaging protocols which, like Signal, derive further session secrets from current state and new randomness.

The second property improves on a related issue: If long-term secrets *are* compromised, then security is not restored if users only close their current session and reopen a new one, but users will instead need to generate and distribute new long-term keys. This procedure however is cumbersome and typically involves manual effort, so ideally one would like to better know *when* it is indeed necessary. Continuous authentication offers such a checking mechanism, enabling users to only change their long-term secrets if they have indeed been leaked.

Introducing: Authentication Steps. In Section 4, we propose a generic extension to messaging protocols to achieve continuous authentication, which we call the *Authentication Steps* (sub-)protocol. It uses users' long-term keys to regularly perform authentication steps between them, locking out active adversaries post-compromise. In these authentication steps, users compare their hashed and signed ciphertext transcripts, taking unreliable transmission and message losses into account, to check for active attacks.

More concretely, an authentication step is performed in three steps to make both users learn each other views of the communication. (Recall that we consider protocols operating on an unreliable channel and some messages might not be received, so users need to rely on message numbering to determine missed messages.) In the first step, the initiator of the authentication step (Alice) sends a list of the indices of the messages she should have but did not receive. Her peer (Bob) deduces from this which ciphertexts they both have received, hashes and sends back the list of messages he missed along with a signature on the computed hash. Alice finally deduces the same list, computes the hash and checks the signature, and sends a signature of her own.

If the signatures verify, each side is ensured that no messages can have been tampered with by an adversary that did not compromise a long-term secret. Moreover, users can compare the hash computed in the last authentication step through an out-of-band channel. If the hashes differ, they have proof that an adversary managed to forge an authentication step and hence *must know* (one of) their long-term secrets, thus they must replace them as well.

Security Analysis and Benchmark. We accompany our Authentication Steps extension with a proof of security in Section 5, showing that it achieves continuous authentication, assuming the employed hash function is collision resistant and the signature scheme unforgeable. Both assumptions were already required for the core Signal protocol.

We implemented a prototype of our Authentication Steps protocol which integrates seamlessly *on top* of the official Signal Java library. We benchmark the overhead it introduces in terms of bandwidth and storage in Section 6, using simulations based on a corpus of exchanged SMS messages [5] with different conversation and channel reliability pattern. At the example of a 95%-reliable channel, our results show a mean increase of 43 bytes (+ 39%) in ciphertext size and 2.6 KB (+ 411%) in session state size compared to the unmodified Signal protocol. Overheads increase with longer communication epoch lengths and lower channel reliability.

When analysing the Signal Java library for our implementation, we also observed that a hard-coded delay in deleting previous (and outdated) state allows a state-compromising adversary to inject messages after post-compromise security—as per the cryptographic literature—should have "healed" the connection. We implemented a proof of concept to illustrate this gap between the (slightly delayed) guarantees of the official Signal Java library and the (slightly stronger) post-compromise security claimed in the literature. In our opinion, this issue is not security-critical from a very practical point of view, but we communicated our findings and a proposed fix to the Signal developers.

We conclude with Section 8.

1.2 Further Related Work

The security of real secure messaging implementations is evaluated in [8], with a focus on (de)synchronization. Somewhat similar to our setting, their analysis involves an adversary trying to break post-compromise security by impersonating a compromised user and finding discrepancies between the implementations and the formal specifications.

Similarly to our approach, [13] proposes a construction for secure messaging by signing message transcripts, focusing though on healing communication under *passive* attacks while we are interested to detect and prevent *active* attacks.

2 Continuous Authentication

This section presents what locking out an active adversary from a messaging communication formally means. The basis for this is a formal syntax, following [1], that captures generic *messaging schemes* that operate on an unreliable channel, derive an initial shared session secret using long-term keys and then derive further session secrets from previous state, new randomness and possibly long-term keys. This in particular encompasses the Signal Double Ratchet protocol.

We then put forward a game-based security definition for *continuous authentication*, which guarantees two core properties:

1. An active MitM adversary that compromised a user's communication state gets *locked out* of the communication *unless* it has also compromised the user's long-term secrets.
2. Users can correctly *decide whether* long-term secrets have been compromised in an active attack, via an out-of-band channel.

The first property captures the desired strengthening of messaging protocols. In the Double Ratchet protocol [17], all secrets are derived from prior established secret state (the so-called *root* and *chain* keys) and new randomness generated by the users (the Diffie–Hellman *ratchet* keys). The former is revealed in a full state compromise and the latter are unauthenticated and can hence be impersonated by the adversary. Therefore, an adversary can conserve its Man-in-the-Middle position indefinitely without being detected in-band.

The second property improves on a related issue: If long-term secrets *are* compromised, then security is not restored if users only close their current session and reopen a new one, but users will instead need to generate and distribute new long-term keys. This procedure however is cumbersome and typically involves manual effort, so ideally one would like to better know *when* it is indeed necessary. Continuous authentication offers such a checking mechanism, enabling users to only change their long-term secrets if they have indeed been leaked.

2.1 Messaging Schemes

A messaging scheme consists of several algorithms: The core algorithms, following [1], are used to create users (Register), initiate sessions between them (InitState) and let them send and receive messages (Send and Recv). Our definition supports an arbitrary number of users, but only two-party sessions (*i.e.*, no group chats).

In addition to the four core messaging algorithms, our formalization introduces StartAuth, a procedure which can be used to initiate an in-band authentication step[4] and DetectOOB, a procedure that compares the states of two session participants out-of-band and decides whether an adversary has used a long-term secret to avoid in-band detection.

Definition 1 (Messaging scheme). *A messaging scheme* $\mathtt{MS} = (\textsc{Register}, \textsc{InitState}, \textsc{Send}, \textsc{Recv}, \textsc{StartAuth}, \textsc{DetectOOB})$ *consists of six probabilistic algorithms:*

- Register *creates a user* U *outputting long- and medium-term information and secrets:* $(LTI_U, LTS_U, MTI_U, MTS_U) \xleftarrow{\$} \textsc{Register}()$.
- 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 public information, and creates an initial session state for* U *to communicate with* V*:* $\pi_U \xleftarrow{\$} \textsc{InitState}(LTS_U, MTS_U, LTI_V, PI)$.
- Send *takes as input the state* π_U *and long-term secrets* LTS_U *of the sender as well as a message* m *and outputs a new state, a ciphertext and a message index* $(\pi'_U, c, idx) \xleftarrow{\$} \textsc{Send}(\pi_U, LTS_U, m)$.
- Recv *takes as input the state* π_U *and long-term secrets* LTS_U *of the receiver as well as a ciphertext* c *and outputs a new state, a plaintext and an index* $(\pi'_U, m, idx) \xleftarrow{\$} \textsc{Recv}(\pi_U, LTS_U, c)$.
  Recv *may return an error* $(\perp)$ *instead of the plaintext which signals the ciphertext has not been accepted. Moreover, it may raise a* **Close** *exception which signifies the user closes the connection.*
- StartAuth *takes as input a state* π_U *and outputs a new state* $\pi'_U \xleftarrow{\$} \textsc{StartAuth}(\pi_U)$.
- DetectOOB *takes as input two states* π_U *and* π_V *and outputs a bit* $d \xleftarrow{\$} \textsc{DetectOOB}(\pi_U, \pi_V)$.

The session state contains an `auth` flag which is initially set to **None**. StartAuth is a special procedure which may set this flag to a non **None** value in the state of the party to indicate they are performing an authentication step. An authentication step is *passed* once the `auth` flags of both parties are back to **None**.

In a session between two parties, we define an *epoch* as a flow of messages, sent by one party without receiving a reply by their peer. Epochs are numbered: even epochs correspond to messages sent by the initiator of the conversation and odd epochs to messages sent by the responder. Within an epoch, messages are again numbered consecutively.

Before defining correctness for messaging schemes, the following definition introduces the notion of *matching states*. On a high-level, states are considered to be matching if either state would decrypt correctly a ciphertext sent by their matching partner.

Definition 2 (Matching states). *Let* $\mathtt{MS}$ *be a messaging scheme and* A*,* B *two users created with* $(LTI_U, LTS_U, MTI_U, MTS_U) \xleftarrow{\$} \textsc{Register}()$ *for* $U \in \{A, B\}$*. Let* π_A *(resp.* π_B*) be a state of* A *(resp.* B*) during a protocol execution.*

π_A *and* π_B *are* matching *states if, for all message* m*, ciphertext* c *and index* idx*,* $(\pi'_A, c, idx) \xleftarrow{\$} \textsc{Send}(\pi_A, LTS_A, m)$ *implies that* B *would output* $(\pi'_B, m', idx') \xleftarrow{\$} \textsc{Recv}(\pi_B, LTS_B, c)$ *with* $m' = m$ *and* $idx' = idx$.

This enables us to define correct messaging schemes.

Definition 3 (Correct messaging scheme). *Let* $\mathtt{MS}$ *be a messaging scheme. Let also* A *and* B *be two users created with the* Register *algorithm.* $\mathtt{MS}$ *is* correct *if it follows the following properties:*

1. *The index of a message created by* Send *corresponds to its epoch and number within the epoch.*
2. *The index idx returned by* Recv *is efficiently computable from the ciphertext.*
3. Recv *returns plaintext* $\perp$ *if the ciphertext corresponds to an index which has already been decrypted.*
4. *If* Recv *returns plaintext* $\perp$*, then the state remains unchanged.*

[4] This procedure can leave the state unchanged if the messaging scheme, like the original Signal protocol, does not support in-band authentication.

```
procedure createState-A(PI):
    assert(¬π_A)
    π_A ←$ InitState(LTS_A, MTS_A, LTI_B, PI)

procedure transmit-A(m):
    assert(π_A)
    (π_A, c, idx) ←$ Send(π_A, LTS_A, m)
    if c ∈ inj_B ∪ authinj ∪ passinj :
        inj_B ← inj_B \ {c}
        authinj ← authinj \ {c}
        passinj ← passinj \ {c}
    trans_B.append(c)
    return c

procedure corruptState-A():
    if π_A:
        return π_A
    return MTS_A

procedure auth-A():
    assert(lastrecv_A > 0 ∧ ¬π_A.auth ∧ ¬π_B.auth)
    π_A ←$ StartAuth(π_A)
    authinj, authidx ← inj_A, lastrecv_A
```

```
procedure corruptLTS-A():
    compromised ← True
    return LTS_A

procedure deliver-B(c):
    assert(π_B)
    try:
        (π'_B, m, idx) ←$ Recv(π_B, LTS_B, c)
        if m ≠ ⊥ ∧ idx > lastrecv_B:
            lastrecv_B ← idx
        if ¬π_B.auth ∧ π'_B.auth :
            authinj ← authinj ∪ {c ∈ inj_B | c.idx ≤ authidx}

        CheckAuthStepPassed()
        π_B ← π'_B
        if c ∉ trans_B ∧ m ≠ ⊥ :
            inj_B[idx] ← c
        return m
    except Close:
        closed ← True

procedure CheckAuthStepPassed():
    if authinj ≠ ∅ ∧ ¬π_A.auth ∧ ¬π_B.auth:
        passinj ← passinj ∪ authinj
        inj_A ← inj_A \ authinj
        inj_B ← inj_B \ authinj
        authinj ← ∅
```

Fig. 1: Oracles available to the adversary in the continuous authentication security game (cf. Definition 4). The `MS.` prefixes for functions of the messaging scheme are omitted. The `CheckAuthStepPassed` function checks if the adversary succeeded in injecting a message which passed an authentication step. Each oracle has a counterpart whose implementation is similar by swapping A and B in the implementation.

5. *If states π_A and π_B are matching, A uses* SEND *to create* $(\pi'_A, c, idx) \leftarrow \textsc{Send}(\pi_A, LTS_A, m)$ *from a plaintext m, and B inputs this ciphertext to create* $(\pi'_B, m, idx) \leftarrow \textsc{Recv}(\pi_B, LTS_B, c)$ *(the output message and index are equal because of the matching property), then* π'_A *and* π'_B *are still matching.*
6. *Given two matching states, if one of them has* $\pi.auth \neq \mathbf{None}$*, then there exists a finite number of calls to* SEND *and* RECV *such that both states get back to* $\pi.auth = \mathbf{None}$.

Property 1 ensures numbering corresponds to the notion of epochs. Property 2 makes immediate decryption [1] possible. Moreover, property 3 makes sure messages decrypted correspond to different indexes and ciphertexts cannot be replayed. Property 4 ensures bad ciphertexts do not break the scheme. Property 5 ensures the soundness property propagates to all messages in the communication. Note that this does not fully capture immediate decryption. The reader may refer to [1] to get a more precise definition of the soundness property for the Signal protocol which is compatible with the remainder of the paper. Property 6 rules out schemes where an authentication step can never end.

2.2 Security Game

We now present the formal security game capturing continuous authentication, represented in Definition 4.

The security game creates two users, Alice (A) and Bob (B), and lets the adversary interact with them using oracles to simulate a communication. As the final objective is to detect long-term secret compromise, long-term secrets are distributed honestly to parties. In contrast, medium-term secrets are generated by

```
game Detection-Game(A, MS):
    (LTI_A, LTS_A, MTI_A, MTS_A) <-$ MS.Register()
    (LTI_B, LTS_B, MTI_B, MTS_B) <-$ MS.Register()
    π_A, π_B ← None, None
    win ← False, closed ← False, compromised ← False
    trans_A, trans_B ← ∅, ∅
    inj_A, inj_B, authinj, passinj ← ∅, ∅, ∅, ∅
    A^{oracles_MS}(LTI_A, LTI_B, MTI_A, MTI_B)
    detectTrial()
    return win ∧ ¬closed
```

```
procedure detectTrial():
    assert(π_A ∧ π_B ∧ ¬π_A.auth ∧ ¬π_B.auth)
    d ← DetectOOB(π_A, π_B)
    if d ∧ ¬compromised:
        win ← True
    elif ¬d ∧ passinj ≠ ∅:
        win ← True
```

Fig. 2: Security game capturing continuous authentication.

parties, but delivered on the communication channel, allowing the adversary to tamper with them. The adversary is active on the network and can corrupt devices, leaking their current state. The adversary can also compromise long-term secrets, which sets a flag *compromised* in the game, maintaining adversary knowledge within the game. When the adversary terminates, an out-of-band detection step (`detectTrial`, see Algorithm 2) is triggered.

The adversary breaks continuous authentication (we say the adversary "wins") by (1) fooling the out-of-band detection DetectOOB to think it compromised the long-term keys when it actually did not, or (2) injecting a message and successfully passing an authentication step ($passinj \neq \emptyset$), without being detected.

Note that our model conservatively grants the adversary more power than may seem reasonable in practice. In particular, the adversary can choose when in-band and out-of-band detection steps happen (by calling StartAuth and terminating). In practice, in-band detection steps may follow a predefined schedule, and out-of-band detection steps are performed at the discretion of users.

Oracles and Security Game. The adversary has access to the following oracles, formalized in Figure 1 and made explicit for user Alice; with corresponding counterpart oracles for Bob:

- createState-A creates the initial state of Alice given some public information of Bob provided by the adversary.
- transmit-A takes a plaintext as input and simulates Alice sending it.
- deliver-A takes a ciphertext as input and simulates Alice receiving it.
- corruptState-A returns the current state of Alice.
- auth-A makes Alice request authentication.
- corruptLTS-A leaks Alice's long-term secret to the adversary.

The security game itself and resulting security notions are defined as follows.

Definition 4 (Continuous authentication). *Let $\mathcal{A}$ be a probabilistic polynomial-time adversary against a messaging scheme* MS*. It has access to oracles defined above, abbreviated as $oracles_{\texttt{MS}}$. The security game is given in Figure 2.*

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

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

The messaging scheme MS *is said to provide* continuous authentication *if for all efficient adversaries $\mathcal{A}$, $\texttt{Adv}(\mathcal{A})$ is small.*

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

- (LTI_U, LTS_U) is the long-term information and secret of user U and π_U its state.
- *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.
- 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.

Oracle Details. The transmit-A/B oracles are a wrapper around SEND, which records ciphertexts created legitimately by users. Similarly, the deliver-A/B oracles are a wrapper around RECV which add injected ciphertexts to the *inj* sets.

When Alice starts an authentication step (which happens when she receives an authentication message or when auth-A is called), *authinj* is filled with all messages that were injected to her. Authenticated messages will be those she has received in the last epoch and before.

Whenever the adversary calls deliver-A/B, a function is called to check 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.

The *win* flag can only be set to **True** in the `detectTrial` function. This happens either if parties output **True** in the out-of-band detection step but the long-term secret was not compromised (users produced a false positive) or if they output **False** but communication was successfully tampered with and the authentication step passed (the attacker was successful at avoiding detection).

3 The Signal Protocol

Signal [20,17] is an asynchronous messaging protocol using an unreliable channel which aims to provide end-to-end encryption with additional security properties such as forward secrecy and post-compromise security.

On a high-level, the protocol consists of three phases:

1. A *registration phase* happening when users join the platform. Users then generate key material and upload their public key material.
2. A *session establishment* happening when one user wants to contact another. This phase uses long-term key material to establish a shared session secret.
3. The *data exchange phase* is the remainder of the communication. We emphasize that in this phase long-term secrets are not used.

Signal involves several cryptographic components:

- an elliptic curve group to perform Diffie–Hellman key exchange operations,
- a key derivation function (KDF) do derive new key material from established secrets,
- a signature scheme for signing uploaded key material, and
- an authenticated encryption (AEAD) scheme for message encryption.

Registration Upon registration, a user U creates a *long-term key pair* (ik^U, ipk^U). It then generates a *pre-key pair* $(prek^U, prepk^U)$ as well as several *ephemeral key pairs* $((ek_i^U, epk_i^U))_i$. The public keys are all uploaded to the Signal server. This matches the REGISTER procedure from Definition 1, with the long-term keypair corresponding to long-term information and secrets (LTS_U, LTI_U) and the other keys to the medium-term information and secrets (MTS_U, MTI_U).

Session establishment In order for a user Alice (A) to initiate a session with another user Bob (B), Alice retrieves Bob's public information from the server. She then uses Signal's Diffie–Hellman-based initial key agreement protocol X3DH to establish an initial shared secret, the so-called *root key* sk_0, by combining her private long-term key and an ephemeral private key with Bob's public keys. This can be modelled in terms of Definition 1 as $\pi_A \leftarrow \textsc{InitState}(LTS_A, MTS_A, LTI_B, MTI_B)$.

Alice will transmit to Bob in the associated data AD of her first message the ephemeral public keys she has used in X3DH. Thus, upon receiving this initial message, Bob will also be able to derive the exact same shared secret, which can be modelled as $\pi_B \leftarrow \textsc{InitState}(LTS_B, MTS_B, LTI_A, AD)$.

Data exchange Recall that an epoch i is a sequence of messages sent by a user without having received any reply from their peer. When a user (for instance Alice) wants to send the first message in an epoch, she first chooses a new *ratchet key pair* (rk_i^A, rpk_i^A). Using her new ratchet key-pair and Bob's last ratchet key pair[5], she derives a shared secret as the Diffie–Hellman secret $DH_i = \mathtt{DH}(rk_i^A,\ rpk_{i-1}^B)$. From this she can derive a new *sending chain key*: $(sk_{i+1}, ck_{i,0}) \leftarrow \mathtt{KDF}(sk_i, DH_i)$.

For all messages in the epoch, Alice creates message keys using the KDF with the chain key and a constant input: $(ck_{i,j+1}, mk_{i,j}) = \mathtt{KDF}(ck_{i,j}, 1)$. Those message keys are used to encrypt messages using the AEAD scheme. The public ratchet key for this epoch is included in the associated data of all messages, so Bob can derive the same message keys to decrypt messages. Because messages may be lost or reordered, the message epoch and index are included in the associated data, so Bob can reconstruct the correct key to decrypt the message.

All those operations can be encapsulated in the Send procedure, with root key, chain keys and message keys being held in the local state. Keys are deleted from the local state once they are no longer needed.

When Bob receives one message from Alice, knowing the root key sk_i shared with Alice and receiving in the associated data her new ratchet key, he can derive the same initial chain key $ck_{i,0}$, and from it the message keys to decrypt messages. Receiving is modelled through the Recv algorithm.

This shows succinctly that Signal fits the definition of a messaging scheme given in Section 2. We will see next that it does not achieve continuous authentication.

3.1 Signal Does Not Provide Continuous Authentication

For the original Signal protocol, StartAuth is a procedure that does nothing as no authentication steps are implemented. We will now show that no choice for implementing the DetectOOB algorithm would yield a protocol which ensures continuous authentication. Formally, the following adversary against Signal succeeds in the security game for continuous authentication from Definition 4 with probability 1, no matter how DetectOOB is defined.

Proposition 1. *Let $\mathcal{A}$ be the adversary in Algorithm 1. If the messaging scheme* $\mathtt{MS}$ *of Definition 4 is the Signal protocol, then adversary $\mathcal{A}$ wins the game with probability 1.*

Proof. The proof is quite straightforward. First the adversary 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 adversary makes Bob create message 0 of epoch 1 for plaintext m_1.

However, instead of delivering the created ciphertext, the adversary drops it and forges her own ciphertext instead. To do so, she corrupts the local state of Alice using corruptState-A and performs the same computations as what Bob would have done to transmit m'_1 instead of m_1. We refer the reader to the Signal specification [17] to see that Line 12 leads to a correct ciphertext for m'_1. c_1 and c'_1 correspond to different plaintexts thus $c_1 \neq c'_1$, which means $(c'_1, AD_1) \notin trans_A$.

When the adversary calls auth-A, as (c'_1, AD_1) has been successfully injected, $(c'_1, AD_1) \in inj_A \subset$ *authinj*. Then the adversary transmits one message from Alice to Bob. When Bob receives the message,

[5] For the very first epoch of the conversation, Alice uses Bob's pre-key pair as his last ratchet key-pair.

adversary $\mathcal{A}(LTI_A, LTI_B, MTI_A, MTI_B)$**:**

$m_0, m_1 \neq m_1', m_2 \in_R \{0,1\}^*$
createState-A(MTI_B)
$c_0 \xleftarrow{\$}$ transmit-A(m_0)
createState-B($c_0.AD$)
deliver-B(c_0)
$c_1 \xleftarrow{\$}$ transmit-B(m_1)
$(c_1, AD_1) \leftarrow c_1$
$\pi_A \leftarrow$ corruptState-A()
$(\_, ck) \leftarrow \mathtt{KDF}(\pi_A.sk, \mathtt{DH}(AD_1.rpk^B,\ \pi_A.rk^A))$
$(\_, mk) \leftarrow \mathtt{KDF}(ck, 1)$
$c_1' \xleftarrow{\$} \mathtt{Enc}(mk, m_1', AD_1)$
deliver-A((c_1', AD_1))
auth-A()
$c_2 \xleftarrow{\$}$ transmit-A(m_2)
deliver-B(c_2)

Algorithm 1: A description of a successful continuous authentication adversary against the Signal protocol.

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

Thus, the adversary wins with probability 1.

The above proof solely uses the fact that the adversary can forge valid messages by only knowing the session secrets of a party. The attack hence extends to any messaging scheme that, like Signal, only relies on session secrets and randomness to create new session secrets.

4 Introducing Authentication Steps

This section presents our proposed *Authentication Steps* protocol that generically extends messaging schemes to achieve continuous authentication.

Our extension introduces authentication steps that may happen regularly at defined epochs in a session or could be user-triggered. These authentication steps leverage long-term secrets. In Signal, the long-term secret of a user consists of their private identity key, a Diffie–Hellman exponent. The Authentication Steps protocol introduces a new type of long-term secret, which is a signing key $sigk^U$.[6]

The objectives of an authentication step are twofold:

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

To that end, each party sends on the in-band channel their own view of the communication since the last authentication step. These messages are included alongside regular messages exchanged between users; as we will see, this allows the authentication steps to seamlessly be integrated on top of the existing Signal protocol.

[6] In practice, Signal already re-uses the identity key to sign a user's medium-term public key using the XEdDSA [16] signature scheme; we therefore emphasize that an implementation may similarly reuse that identity key as the signing long-term key for our authentication steps extension. In practice, this means only maintaining a single long-term secret for both Signal and our Authentication Steps protocol.

In order to maintain forward secrecy, the additional information is derived from the (public) ciphertexts sent. To save space, intermediate computations compress those ciphertexts as they are sent or received. Those intermediate computations and an authentication step are illustrated in Figure 3.

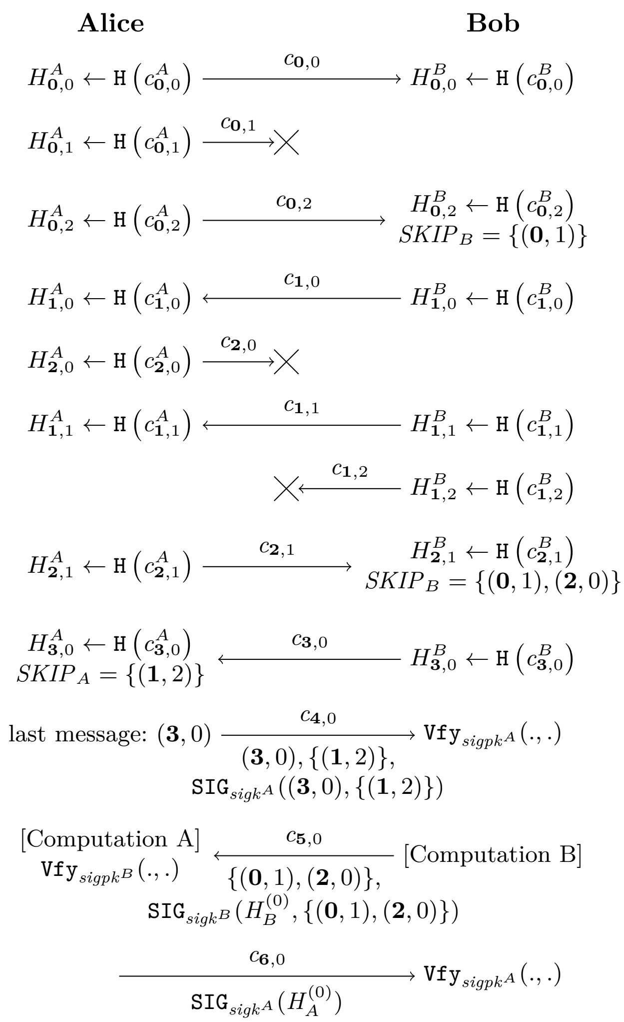


Fig. 3: An example execution of an authentication step. The actual authentication step is performed during epochs **4** to **6**, with authenticated messages from epoch **0** (epoch numbers in **boldface**). Additional data sent for those messages is included below arrows. For epochs **4** to **6**, $H_{\mathbf{i},j}$ hashes are still computed by both parties, but they are omitted in this figure as they concern the next authentication step. [Computation U] for $U \in \{A, B\}$ corresponds to the computation $H_U^{(0)} \leftarrow 0||\mathtt{H}\left(\varepsilon||H^U_{\mathbf{0},0}||H^U_{\mathbf{0},2}||H^U_{\mathbf{1},0}||H^U_{\mathbf{1},1}||H^U_{\mathbf{2},1}||H^U_{\mathbf{3},0}\right)$.

4.1 Recording Ciphertexts

In order to perform authentication steps, parties need to store the transcript of ciphertexts sent and received. The order in which messages are received is not relevant as reordering may be caused by the unreliable channel.

Instead of storing ciphertexts as sent, each party computes digests of those ciphertexts using a hash function and stores those in a dictionary. Concretely, for every sent or received message, each user U computes and stores $H^U_{i,j} = \mathtt{H}\left(c^U_{i,j}\right)$, where $\mathtt{H}$ is a cryptographic hash function and $c^U_{i,j}$ is the ciphertext corresponding to message j sent or received in epoch i by user U.

4.2 Authentication Steps

The stored (and hashed) ciphertexts are then used in the actual authentication step. An authentication step is a 3-pass message exchange and therefore requires three epochs to complete. In the following, an authentication step is described wlog. with Alice sending the first authentication message. Figure 3 illustrates an authentication step, performed in epochs 4 to 6.

The authentication step information is included in every message of the epoch. That way, the peer receives the authentication information at least once, as if they do not receive it, the epoch number will not increase. If authentication information is missing from a message where it should have been included, then the receiving party should dismiss the message.

In the first epoch, Alice sends the following additional authentication information (encrypted along with actual plaintext):

- the indexes of messages that she should have received from Bob, but did not, denoted SKIP_A,
- the index of the most recent message she has received from Bob, denoted *authidx*, and
- a signature $\mathtt{SIG}_{sigk^A}(\mathit{authidx}, \mathit{SKIP}_A)$ over both values.

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

$$H_B^{(n_B)} = n_B || \mathtt{H}\left(H_B^{(n_B-1)} ||\ ||_{(i,j)\in I_B^{(n_B)}} H^B_{i,j}\right),$$

where n_B is the number of authentication steps Bob has completed and $H_B^{(n_B-1)}$ the hash computed in the previous authentication step (with the convention $H_B^{(-1)} = \varepsilon$ the empty string). The concatenation happens in lexicographic order over $I_B^{(n_B)}$, the set of all messages sent and received by Bob since last authentication step and until message *authidx*, and excluding messages with an index contained in SKIP_A.

In the second epoch (with Bob sending messages), Bob sends the following information (along with the regular message plaintexts):

- the indexes of messages that he should have received from Alice, denoted SKIP_B, and
- a signature $\mathtt{SIG}_{sigk^B}(H_B^{(n_B)}, \mathit{SKIP}_B)$ over the hash computed and the indexes of missed messages.

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

$$H_A^{(n_A)} = n_A || \mathtt{H}\left(H_A^{(n_A-1)} ||\ ||_{(i,j)\in I_A^{(n_A)}} H^A_{i,j}\right),$$

where, like for Bob, n_A is her number of completed authentication steps and $H_A^{(n_A-1)}$ is the previous hash (or $H_A^{(-1)} = \varepsilon$). Alice then checks the signature received from Bob, using Bob's public signing key, on data $(H_A^{(n_A)}, \mathit{SKIP}_B)$.

In the third epoch, Alice sends a signature $\mathtt{SIG}_{sigk^A}(H_A^{(n_A)})$ over her hashed collection of seen messages. When Bob receives it, he verifies the signature's validity on $H_B^{(n_B)}$ using Alice's signing public key.

If at some point a signature verification fails, the verifier closes the connection. Otherwise, Alice and Bob have *passed the authentication step.*

Deniable Signing. We emphasize that any unforgeable signature scheme can be used in the authentication step. In particular, to maintain Signal's deniability of the initial key agreement (cf. [25]), signatures can be generated using designated-verifier or 2-user ring signatures [14,19], similarly to their deployment in recent proposals for Signal-like deniable key exchanges [23,24,12,3].

4.3 Detecting Compromised Long-term Secrets

In this setting, we assume that Alice and Bob have passed at least one authentication step. At each authentication step, parties derive a hash $H_A^{(n_A)}$ or $H_B^{(n_B)}$. Authentication steps succeed if the signatures over those hashes match.

On a high-level, users execute the following protocol: Using the out-of-band channel, parties compare the last hash they have computed (which they store in their state until the next authentication step) as well as the number of authentication steps performed. If the hash values and authentication steps counters match, the users output **False**, indicating that they do not detect long-term key compromise, otherwise they output **True**.

If no adversary tampers with the communication, then exchanged hashes would match. Conversely, hashes not matching means that an adversary is present. Moreover, as at least one authentication step has been successful, the adversary must have been able to forge a signature to avoid in-band detection, which indicates they know at least one long-term secret. We will formally prove these two properties of the Authentication Steps protocol in Section 5.

4.4 Handling Out-of-Order Messages

Recall that we consider messaging schemes operating on an unreliable channel. Therefore, messages may be dropped or arrive out-of-order.

The above construction does not yet 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. We now discuss 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 original protocol, as it is less resilient to delayed messages, thus less palatable to real-world implementation.

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 *SKIP* messages. The implementation we provide in Section 6 matches this behaviour.

4.5 Protocol Soundness

The described Authentication Steps protocol is correct, *i.e.*, it matches Definition 3, as shown by the following proposition.

Proposition 2 (Protocol soundness). *If no adversary tampers with the communication, i.e., modifies or injects a ciphertext, then parties pass authentication steps.*

The idea of the proof is to show that in the absence of an adversary, users always receive authentication information when performing authentication steps. This leads them to compute the same sets $I_A^{(n_A)} = I_B^{(n_B)}$ and therefore they agree on the hash computations. The full proof can be found in Appendix A.

5 Security of the Authentication Steps Protocol

We now formally establish the continuous authentication security (as per Definition 4) of our Authentication Steps protocol extension given in Section 4.

Theorem 1. *Assuming a collision resistant hash function* $\mathtt{H}$ *and an existentially unforgeable signature scheme* $\mathcal{S}$*, the Authentication Steps protocol presented in Section 4 provides continuous authentication as per Definition 4.*

Formally, the advantage of any adversary $\mathcal{A}$ *in the detection game against the Authentication Steps protocol is bounded as follows:*

$$\mathtt{Adv}(\mathcal{A}) \leq \mathtt{Adv}^{coll}_{\mathcal{B}_1}(H) + 2 \cdot \mathtt{Adv}^{EUF\text{-}CMA}_{\mathcal{B}_2}(\mathcal{S}),$$

for reduction adversaries $\mathcal{B}_1$ *and* $\mathcal{B}_2$ *given in the proof.*

The proof is separated into two cases:

1. users decide one of their long-term secrets is compromised when that is not the case; and
2. the adversary manages to inject a message and remain undetected.

We defer the detailed proof to Appendix B, and only give a proof sketch here.

In Case 1, the adversary never corrupts the long-term secret, yet the parties decide that their long-term secret is compromised. Thus, the hashes that Alice and Bob exchanged at the end of the game must be different, but both Alice and Bob verified signatures hashes in the last authentication step. It follows then that either Alice or Bob received a signature that was not produced by their peer, and that the adversary must have successfully forged a message under one of their (non-compromised) signing key. This would violate EUF-CMA security of the signature scheme, leading to the $2 \cdot \mathtt{Adv}^{EUF\text{-}CMA}_{\mathcal{B}_2}(\mathcal{S})$ term in the theorem bound.

In Case 2, the adversary must have injected a message between Alice and Bob, but when Alice and Bob exchanged their hashes at the end of the game, the hash outputs matched. It follows that between Alice's or Bob's computations there must be a hash collision, leading to the $\mathtt{Adv}^{coll}_{\mathcal{B}_1}(H)$ term in the theorem bound.

6 Implementation and Benchmarks

We implemented a prototype of our Authentication Steps protocol which integrates seamlessly *on top* of the official Signal Java library. Our full implementation can be found on GitHub[7], along with build instructions and our benchmarking tests.

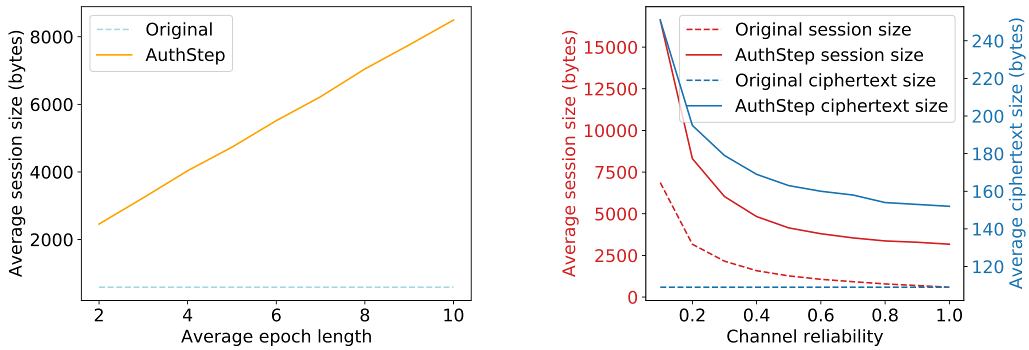
Space and Computation Overhead. Authentication steps require additional data to be computed and stored, such as the ciphertexts hashes between authentication steps.

The storage and bandwidth overhead is a function of the channel reliability and the average number of messages per authentication step, the latter being the more influential parameter. Indeed, the sender cannot know in advance which messages the peer has received, thus there is no alternative but storing every ciphertext hash individually.

As for computational overhead, computing the ciphertext hashes involves one hash invocation; additionally, at most one signature and one verification operation is performed per epoch. The signature scheme employed by Signal is XEdDSA [16]. Signing and verifying data typically requires the same amount of computation as the Diffie-Hellman key computation happening in asymmetric ratchet steps. Thus, the computational overhead is at most the same magnitude as the original computations in Signal.

Benchmarking the space overhead In order to give an estimate on the space overhead induced by authentication steps, we performed simulations of communications to compare the ciphertext sizes as well as the states sizes. The message inputs for our simulations are taken from an SMS dataset [5] composed of English text messages published by Singaporean students [4]. This dataset only provides message samples but not entire conversations with relations between messages. Conversation communications have been simulated

[7] https://github.com/apoirrier/libsignal-java-authsteps



(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.

Fig. 4: Overhead introduced by the authentication steps depending on the channel reliability and average epoch length. For this experiment, 10,000 messages were exchanged.

by randomly choosing the number of messages in each epoch following a Poisson distribution. The reliability of the channel and the average number of messages per epoch are a parameter of the experiment and their impact is studied in the following.

For the benchmarking, we used SHA-512 as the hash function (as in the original Signal implementation) and led authentication steps happen regularly every 7 epochs. For every experiment 10,000 messages are sent, deriving randomness from a random, but then fixed seed.[8]

Figure 4 studies the impact of the channel reliability and the length of epochs on the size of the state and ciphertexts. As stated above, 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 (we hence omit a graph for those here).

The less reliable the channel, the more state the parties need to store in order to 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, indexes of skipped messages need to be sent to the other party during authentication steps, which makes ciphertexts bigger on average.

6.1 Compressing the Storage Overhead

As we have seen in Section 6, storing the transcript hashes to be authenticated comes with a notable storage overhead. One may consider various ways for optimizing that storage.

Complete compression into a single hash value (e.g., through iterative hashing) may seem like an immediate optimization. However, as messages may arrive out-of-order, such compression is unreliable and would fail entirely when, e.g., the first message in an epoch does not arrive.

An alternative would be to use trees (for instance Merkle trees) to store hashes, and compress them if consecutive sequences of messages are received. This optimization would be interesting to implement and benchmark, at the same time it would make the underlying analysis and notions more complex. Furthermore,

[8] To verify the seed is not biased, 95% confidence intervals have been computed to assess the mean ciphertexts and sessions sizes. They are not displayed in the figure as they are too small (around 20 bytes for session sizes and 5 bytes for ciphertext sizes). 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.

this optimization can only be performed on the receiver's side, as the sender has no way to know which sent messages will eventually be received. Thus, the compression can only happen on at most half of the conversation, and the space optimization is bound by a factor 2.

7 Observations on the Official Implementation

While implementing the proposed protocol, we found that the state deletion strategy in Signal's official Java implementation [21] is different from the strategy described in the formal analyses in the literature, such as [1] or [6], even if the latter claim to be 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 kicks in after two epochs, which means that after two epochs of untampered communication after a state compromise, security is restored. This happens by the deletion of no longer necessary state once an epoch ends. However, the Signal implementation deletes this state only 5 epochs later, which is a hardcoded value[9].

Based on this, we observe that the following attack is possible which demonstrates that the official Signal implementation achieves only slightly weaker post-compromise security than claimed in the literature. In the middle of a communication between Alice and Bob, an adversary leaks the state of Alice. Assume that during this epoch i, Alice sent n_i messages to Bob. The adversary can, by using the leaked state, create a valid ciphertext for message $(i, n_i + 1)$ (and even more messages).

Given the literature definition of post-compromise security, as the adversary remained passive security should be restored at epoch $i + 3$. However, with the Signal implementation, as Bob's state for epoch i is not yet deleted at epoch $i + 3$, the adversary can successfully inject messages (for epoch i) to Bob.

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

An Explanation of this Weaker Property, and Fixing it. The Signal implementation disregards the total number of messages sent in the previous epoch, which is included alongside messages, and instead keeps 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 those messages never arrive while the immediate decryption property is still valid as the chain key is kept and message keys can be derived if needed.

To fix this, when a new receiving epoch begins, the value of the total number of messages can be used to derive all message keys for this epoch and then delete the chain key from the state. This recovers the strong post-compromise security as claimed in the literature.

8 Conclusion

Messaging protocols such as Signal that only use their long-term secrets for session initiation allow for state-compromising adversaries to permanently take over a connection as a Man-in-the-Middle. This paper offers a strengthened security notion, continuous authentication, which locks out an active adversary post-compromise who has not compromised long-term keys, and enables detection of long-term secret compromises using an out-of-band channel. Our Authentication Steps protocol extension generically enables this security in a provably-secure way, adding regular authentication steps in the protocol that leverages long-term keys to authenticate users and ensure no tampering has occurred. Moreover, an out-of-band protocol can be used on top of that to detect adversaries having used long-term secrets to avoid in-band detection.

We analysed the overhead introduced by authentication steps, benchmarking our prototype implementation which seamlessly integrates on top of the official Signal library. While implementing those benchmarks, we remarked that the official implementation has a weaker post-compromise security property than claimed in the literature.

[9] Cf. line 210 in https://github.com/signalapp/libsignal-protocol-java/blob/fde96d22004f32a391554e4991e4e1f0a14c2d50/java/src/main/java/org/whispersystems/libsignal/state/SessionState.java#L210.

While this paper focuses mainly on the Signal protocol, the concept of continuous authentication as well as the Authentication Steps protocol is generic. We envision that it can be adapted to other messaging protocols or protocols with long-lived connections, like TLS 1.3 resumption sessions, to provide stronger authenticity guarantees.

The Authentication Steps protocol strongly authenticates the entire transcript, even if the underlying channel is unreliable. Another interesting direction for future work is a conceivable reduced-overhead variant that only authenticates the key material (e.g., the ratchet keys in Signal) in every epoch. Such variant would still strengthen post-compromise security, yet in a weaker sense as it would not necessarily allow to detect the injection of messages at the end of compromised epochs, a property the Authentication Steps protocol provides.

References

1. Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 129–158. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17653-2_5
2. Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: The security of messaging. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part III. LNCS, vol. 10403, pp. 619–650. Springer, Heidelberg (Aug 2017). https://doi.org/10.1007/978-3-319-63697-9_21
3. Brendel, J., Fiedler, R., Günther, F., Janson, C., Stebila, D.: Post-quantum asynchronous deniable key exchange and the Signal handshake. In: PKC 2022, Part II. LNCS, vol. 13178, pp. 3–34. Springer (2022). https://doi.org/10.1007/978-3-030-97131-1_1
4. Chen, T., Kan, M.Y.: Creating a live, public short message service corpus: the NUS SMS corpus. Language Resources and Evaluation **47**(2), 299–335 (2013). https://doi.org/10.1007/s10579-012-9197-9
5. Chen, T., Kan, M.Y.: The National University of Singapore SMS Corpus. [Dataset] https://doi.org/10.25540/WVM0-4RNX (2015)
6. Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. Journal of Cryptology **33**(4), 1914–1983 (Oct 2020). https://doi.org/10.1007/s00145-020-09360-1
7. Cohn-Gordon, K., Cremers, C.J.F., Garratt, L.: On post-compromise security. In: Hicks, M., Köpf, B. (eds.) CSF 2016 Computer Security Foundations Symposium. pp. 164–178. IEEE Computer Society Press (2016). https://doi.org/10.1109/CSF.2016.19
8. Cremers, C., Fairoze, J., Kiesl, B., Naska, A.: Clone detection in secure messaging: Improving post-compromise security in practice. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 1481–1495. ACM Press (Nov 2020). https://doi.org/10.1145/3372297.3423354
9. Dowling, B., Hale, B.: Secure messaging authentication against active man-in-the-middle attacks. In: 2021 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 54–70 (2021). https://doi.org/10.1109/EuroSP51992.2021.00015
10. Durak, F.B., Vaudenay, S.: Bidirectional asynchronous ratcheted key agreement with linear complexity. In: Attrapadung, N., Yagi, T. (eds.) IWSEC 19. LNCS, vol. 11689, pp. 343–362. Springer, Heidelberg (Aug 2019). https://doi.org/10.1007/978-3-030-26834-3_20
11. Facebook: Messenger Secret Conversation, Technical Whitepaper (2016), https://about.fb.com/wp-content/uploads/2016/07/messenger-secret-conversations-technical-whitepaper.pdf
12. Hashimoto, K., Katsumata, S., Kwiatkowski, K., Prest, T.: An efficient and generic construction for signal's handshake (X3DH): Post-quantum, state leakage secure, and deniable. In: Garay, J. (ed.) PKC 2021, Part II. LNCS, vol. 12711, pp. 410–440. Springer, Heidelberg (May 2021). https://doi.org/10.1007/978-3-030-75248-4_15
13. Jaeger, J., Stepanovs, I.: Optimal channel security against fine-grained state compromise: The safety of messaging. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 33–62. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1_2
14. Jakobsson, M., Sako, K., Impagliazzo, R.: Designated verifier proofs and their applications. In: Maurer, U.M. (ed.) EUROCRYPT'96. LNCS, vol. 1070, pp. 143–154. Springer, Heidelberg (May 1996). https://doi.org/10.1007/3-540-68339-9_13
15. Jost, D., Maurer, U., Mularczyk, M.: Efficient ratcheting: Almost-optimal guarantees for secure messaging. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 159–188. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17653-2_6

16. Perrin, T.: The XEdDSA and VXEdDSA signature schemes. Tech. rep., Signal (2016), https://whispersystems.org/docs/specifications/xeddsa/
17. Perrin, T., Marlinspike, M.: The Double Ratchet algorithm. https://whispersystems.org/docs/specifications/doubleratchet/ (2016)
18. Poettering, B., Rösler, P.: Towards bidirectional ratcheted key exchange. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 3–32. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1_1
19. Rivest, R.L., Shamir, A., Tauman, Y.: How to leak a secret. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 552–565. Springer, Heidelberg (Dec 2001). https://doi.org/10.1007/3-540-45682-1_32
20. Signal: Technical information. https://signal.org/docs/
21. Systems, O.W.: libsignal-protocol-java. https://github.com/signalapp/libsignal-protocol-java (2021)
22. Unger, N., Dechand, S., Bonneau, J., Fahl, S., Perl, H., Goldberg, I., Smith, M.: SoK: Secure messaging. In: 2015 IEEE Symposium on Security and Privacy. pp. 232–249 (2015). https://doi.org/10.1109/SP.2015.22
23. Unger, N., Goldberg, I.: Deniable key exchanges for secure messaging. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015. pp. 1211–1223. ACM Press (Oct 2015). https://doi.org/10.1145/2810103.2813616
24. Unger, N., Goldberg, I.: Improved strongly deniable authenticated key exchanges for secure messaging. PoPETs **2018**(1), 21–66 (Jan 2018). https://doi.org/10.1515/popets-2018-0003
25. Vatandas, N., Gennaro, R., Ithurburn, B., Krawczyk, H.: On the cryptographic deniability of the Signal protocol. In: Conti, M., Zhou, J., Casalicchio, E., Spognardi, A. (eds.) ACNS 20, Part II. LNCS, vol. 12147, pp. 188–209. Springer, Heidelberg (Oct 2020). https://doi.org/10.1007/978-3-030-57878-7_10
26. WhatsApp Security, https://www.whatsapp.com/security/
27. How WhatsApp enables multi-device capability (2021), https://engineering.fb.com/2021/07/14/security/whatsapp-multi-device/
28. WhatsApp Security Advisories (2021), https://www.whatsapp.com/security/advisories

A Soundness of the Authentication Steps Protocol

This section shows the correctness of the Authentication Steps protocol presented in Section 4, *i.e.*, the protocol matches Definition 3. It proves Proposition 2.

Properties 1 to 4 of the correctness definition are easy to prove and come directly from the properties of the underlying 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 ($I_A^{(n_A)} = I_B^{(n_B)}$ with the notations of Section 4). We finally conclude by showing the hashes computed are the same and each signature verification succeeds.

Lemma 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. However, 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. As ciphertexts are not modified or injected in this context, the information received has been honestly generated by their communicating partner.

Lemma 2. *If no adversary tampers with the communication, then computed sets $I_A^{(n_A)}$ and $I_B^{(n_B)}$ by each party are equal: i.e., $I_A^{(n_A)} = I_B^{(n_B)}$.*

Proof. We begin the proof by proving by induction that for every authentication step, Alice and Bob agree on the lower and upper bounds on the chosen set of messages, which we denote the *period* of messages.

For the first authentication step, Alice and Bob must agree on the lower bound as it is message $(0, 0)$. Then the initiator (for instance Alice) chooses the last message she wants to authenticate (one of Bob's messages, since the authentication step starts at the beginning of an epoch) and sends this index to Bob. Bob receives this index correctly thanks to Lemma 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. Let's call (i, j) the index of the last message authenticated. As authentication steps last for three epochs, epoch i is finished for both users. They thus know which message follows (i, j) (either $(i, j + 1)$ or $(i + 1, 0)$), and they agree on this message as the first one to authenticate.

Once again, the initiator chooses the last message to authenticate and transmits this choice to the peer, 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, $I_U^{(n_U)}$ is the set of message indexes that U has sent or received in this period, except for messages that their peer did not receive (whose indices are sent in $SKIP_V$).

In both cases, $I_A^{(n_A)}$ and $I_B^{(n_B)}$ are the union of the set of messages received by Alice and the set of messages received by Bob. This proves that $I_A^{(n_A)} = I_B^{(n_B)}$.

With those lemmas, we can now prove the soundness of the Authentication Steps protocol.

Proof (Proof of Proposition 2). By definition, authentication steps pass if the connection does not close. Given the implementation, **Close** exceptions occur only when a signature verification fails.

As no adversary actively injects or modifies messages, the first signature verification performed by Bob always succeeds as the signed data is sent alongside the signature. Similarly, if no active adversary is present, then ciphertexts received will be the same as ciphertexts sent: $\forall (i, j) \in \mathcal{R},\ c_{i,j}^A = c_{i,j}^B$ where $\mathcal{R}$ is the set of indices of received messages.

The hash function is deterministic, therefore:

$$\forall (i, j) \in \mathcal{R},\ H_{i,j}^A = \mathtt{H}\left(c_{i,j}^A\right) = \mathtt{H}\left(c_{i,j}^B\right) = H_{i,j}^B.$$

Let's assume that at the beginning of an authentication step, $n_A = n_B$ and $H_A^{(n_A)} = H_A^{(n_B)}$. Recall the next $I_U^{(n_U+1)}$ for user U is computed as: $I_U^{(n_U+1)} \leftarrow (n_U+1) || \mathtt{H}\left(I_U^{(n_U)} || \ ||_{(i,j)\in I_U^{(n_U+1)}} H_{i,j}^U\right)$. Following Lemma 2, Alice and Bob agree on sets $I_A^{(n_A+1)} = I_B^{(n_B+1)} = I \subset \mathcal{R}$. Moreover, ciphertexts are ordered correctly as the identification information is included in the ciphertext and was stored accordingly. We have seen above that for all $(i, j) \in I$, $H_{i,j}^A = H_{i,j}^B$. Therefore, after the authentication step, we still have $I_A^{(n_A+1)} = I_A^{(n_A+1)}$. As at the beginning of the communication, $n_A = n_B = 0$ and $I_A^{(-1)} = I_B^{(-1)} = \varepsilon$, by induction we conclude that for every authentication step, $I_A^{(n_A)} = I_B^{(n_B)}$.

In the second epoch, Bob signs $(I_B^{(n_B)},\ SKIP_B)$ and transmits $SKIP_B$. Because we have shown $I_A^{(n_A)} = I_B^{(n_B)}$, Alice succeeds in verifying the signature on the data $(I_A^{(n_A)},\ SKIP_B)$ she computes. In the third epoch, considering $I_A^{(n_A)} = I_B^{(n_B)}$, Bob also succeeds in verifying the signature. As all signature verifications succeed, Alice and Bob pass the authentication step.

B Security of the Authentication Steps Protocol

This section proves Theorem 1, which states that the Authentication Steps protocol is secure given the assumption that the underlying cryptographic primitives are secure, namely the hash function and the signature scheme. We denote $\mathtt{Adv}_{\mathcal{A}}^{coll}(H)$ the advantage of an adversary trying to find a collision for a hash function H and $\mathtt{Adv}_{\mathcal{A}}^{EUF\text{-}CMA}(\mathcal{S})$ the advantage of an adversary in the EUF-CMA (Existential UnForgeability in the Chosen Message Attack setting) game against a signature scheme $\mathcal{S}$.

False Positives and False Negatives. Before proving security of the Authentication Steps protocol from Section 4, we introduce several useful definitions.

Recall that from the specification, no authentication steps can overlap. Therefore, users will reject messages that start a new authentication step if they are currently performing one. We can thus number authentication steps from a user U's point of view from 1 to n_U.

For the actual theorem proof, we split the winning condition into two events, which we denote the false positive case and the false negative case, and use results from Propositions 3 and 4 to give an upper bound on their probability.

Definition 5. *Given an adversary $\mathcal{A}$ playing the security game of Definition 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, $\neg closed \wedge \neg compromise \wedge d$ is true,*
- FN *is the event that at the end of the game, $\neg closed \wedge passinj \neq \emptyset \wedge \neg d$ is true.*

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

Proof (Proof of Theorem 1). Let $\mathcal{A}$ be an adversary in the game of Definition 4. 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.

Therefore:

$$\mathtt{Adv}(\mathcal{A}) = \Pr[\mathrm{W}] = \Pr[\mathrm{FP}] + \Pr[\mathrm{FN}] .$$

Moreover, Proposition 4 states that $\Pr[\mathrm{FP}] \leq 2 \cdot \mathtt{Adv}_{\mathcal{B}_2}^{EUF\text{-}CMA}(\mathcal{S})$ and Proposition 3 states that $\Pr[\mathrm{FN}] \leq \mathtt{Adv}_{\mathcal{B}_1}^{coll}(H)$, which proves the inequality.

B.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. We first introduce Lemma 3 which is used to prove Proposition 4.

Lemma 3. *Let $\mathcal{A}$ be an adversary playing the security game of Definition 4 against the Authentication Steps protocol from Section 4.*

If $passinj \neq \emptyset$ at the end of the game, it means that there exists some user $U \in \{A, B\}$, an authentication step j for U and a message index $i \in I_U^{(j)}$ such that $c_i^A \neq c_i^B$ (where one of the ciphertext could be $\perp$ if the corresponding user has sent no ciphertext for index i).

Proof. 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 Figure 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 1). For both cases, this happens when a user U enters an authentication step (wlog. we choose authentication step j), and message i comes from inj_U.

Message i has already been received because it is in inj_U when added to *authinj*, *i.e.*, when the authentication step begins, so it is not a skipped message. Moreover, $i \leq auth.authidx$ given the implementation of StartAuth.

$\pi_U.lastauth$ 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, 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 $i \in I_U^{(j)}$.

Because $i \in inj_U$, it means U received and accepted ciphertext c_i^U (when it was added to inj_U). If V is the peer of U, then c_i^V (if it exists) cannot be equal to c_i^U because otherwise it would have been generated honestly by V and therefore removed from injected sets in lines 5 to 7 of transmit-A/B in Figure 1, or never added to inj_U because in $trans_U$ (see Line 8 of transmit-A and Line 11 of deliver-B in Figure 1). Therefore, $c_i^A \neq c_i^B$.

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 3 (False negatives). *Let $\mathcal{A}$ be an adversary in the detection game of Definition 4 playing against the Authentication Steps protocol presented in Section 4.*

Then $\Pr[\mathrm{FN}] \leq \mathtt{Adv}^{coll}_{\mathcal{B}_1}(H)$*, for a reduction adversary $\mathcal{B}_1$ constructed in the proof.*

Proof. Let $\mathcal{A}$ be an adversary producing event FN. Recall from Definition 5 that $\mathrm{FN} = \neg \mathit{closed} \wedge \mathit{passinj} \neq \emptyset \wedge \neg d$.

In particular, *passinj* is not empty at the end of the game. According to Lemma 3, this implies the existence of an authentication step j_0 for user $V \in \{A, B\}$ and some index i such that $i \in I_V^{(j_0)}$ and $c_i^A \neq c_i^B$.

However, d is **False**. Given the computation of d in the DetectOOB procedure this means that $\pi_A.H_A^{(n_A)} = \pi_B.H_B^{(n_B)}$.

Recall that for any user U, $H_U^{(n_U)} = n_U || H_U^{(n_U-1)}$. In particular $n_A = n_B$ and Alice and Bob have seen the same number of authentication steps.

Hashes $H_U^{(j)}$ are computed as follows: $H_U^{(j)} \leftarrow \mathtt{H}\left(H_U^{(j-1)} \mid\mid ||_{k \in \mathtt{sorted}(I_U^{(j)})} \pi_U.H_k^U\right)$ for any $j \geq 0$ and with $H_U^{(-1)} = \varepsilon$.

For any $j \geq 0$, if $H_A^{(j)} = H_B^{(j)}$, then there are only two possibilities:

1. either $H_A^{(j-1)} \mid\mid ||_{k \in \mathtt{sorted}(I_A^{(j)})} \pi_A.H_k^A \neq H_B^{(j-1)} \mid\mid ||_{k \in \mathtt{sorted}(I_B^{(j)})} \pi_B.H_k^B$;
2. 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:

$$||_{k \in \mathtt{sorted}(I_A^{(j)})} \pi_A.H_k^A = ||_{k \in \mathtt{sorted}(I_B^{(j)})} \pi_B.H_k^B.$$

This is true in particular for $j = j_0$. Recall that elements of $\pi_U.H^U$ are hashes of ciphertexts computed on sending and receiving.

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 $H_W = \mathtt{H}(c^W)$ the corresponding hashes (where H_W is at the same position in the concatenation that H_V but for the other user).

Because the concatenations are equal, $H_V = H_W$. However, $c^W \neq c_i^V$. Indeed, if we had $c^W = c_i^V$, then both would correspond to the same index i, but c^W is the version of ciphertext i sent by W and c_i^V is the version received by V. However, by definition of i, we necessarily have $c_i^A \neq c_i^B$ and therefore the equality is impossible. As $H_V = H_W$ but $c^W \neq c_i^V$, we have 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 $\mathcal{B}_1$ from the detection game to the hash collision game is immediate.

This shows that $\Pr[\mathrm{FN}] \leq \mathtt{Adv}^{coll}_{\mathcal{B}_1}(H)$.

B.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 4 (False positives). *Let $\mathcal{A}$ be an adversary in the detection game of Definition 4 playing against the Authentication Steps protocol of Section 4.*

Then $\Pr[\mathrm{FP}] \leq 2 \cdot \mathtt{Adv}_{\mathcal{B}_2}^{EUF-CMA}(\mathcal{S})$*, for a reduction adversary $\mathcal{B}_2$ constructed in the proof.*

Proof. Let $\mathcal{A}$ be an adversary producing event FP. Having $\neg$*compromise* means that $\mathcal{A}$ never calls the corruptLTS-A/B oracles. Moreover, $\neg$*closed* means the communication never closes, which means that signature verifications always succeed. We will build an adversary $\mathcal{B}_2$ for 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 $\mathcal{A}$.

$\mathcal{B}_2$ creates two users Alice and Bob, but will embed a public key provided by the EUF-CMA challenger into one party's signing key-pair and use the signing oracle to generate signatures.

As user U_1 is entirely generated by $\mathcal{B}_2$, the adversary can simulate the oracles concerning U_1, and therefore they are similar to the oracles defined in Figure 1.

$\mathcal{B}_2$ keeps track of signature forgeries. Every time $\mathcal{B}_2$ signs a message using the oracle provided by his challenger, $\mathcal{B}_2$ stores it. Moreover, every time a signature on the signing public key pk given by the EUF-CMA game is verified, $\mathcal{B}_2$ checks if the signature was produced by the signing oracle. If that is not the case, but the verification is successful, $\mathcal{B}_2$ stops and outputs the corresponding pair m^*, σ^*.

To simulate user U_0 whose private key is unknown, $\mathcal{B}_2$ can also use the original 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 $\mathcal{B}_2$ does not need to simulate those oracles when the event FP happens. In order to create the signature, $\mathcal{B}_2$ can query their own challenger with message $\pi_{U_0}.auth$ to get the signature using U_0's private key. Therefore, $\mathcal{B}_2$ is correctly defined and can act as a challenger for $\mathcal{A}$.

Let's now prove that if $\mathcal{A}$ triggers the event FP, then $\mathcal{B}_2$ wins the EUF-CMA game with probability at least $\frac{1}{2}$.

During his last authentication step n, U_1 verified successfully a signature σ on $\pi_{U_1}.auth$ by using U_0's public signing key $sigpk^{U_0}$. $\pi_{U_1}.auth$ contains in particular $n_U = n$ and $H = H_1$ computed by U_1. Because U_0 and U_1 can number their authentication steps, they will produce at most one signature on an *auth* set having $n_U = n$. At the end of the game, parties output $d = \mathbf{True}$. From the implementation of DetectOOB, this means that $\pi_A.H_A^{(n_A)} \neq \pi_B.H_B^{(n_B)}$. Given the definition of $\pi_U.H_U^{(n_U)}$ this means that during the last authentication step of each party:

$$\pi_A.n_A || \pi_A.auth.H \neq \pi_B.n_B || \pi_B.auth.H.$$

There are two disjoint possibilities:

1. either $\pi_A.n_A = \pi_B.n_B$ but $\pi_A.auth.H \neq \pi_B.auth.H$;
2. either $\pi_A.n_A \neq \pi_B.n_B$;

In Case 1, $\pi_A.n_A = \pi_B.n_B = n$ and $\pi_A.auth.H \neq \pi_B.auth.H$. Yet U_1's verification of σ succeeded on the data $\pi_{U_1}.auth$ which contains $n_U = n$ and $H = H_1$. However, as stated above U_0 can produce and sign at most one set *auth* with $n_U = n$, and this set has $H = \pi_{U_0}.auth.H \neq \pi_{U_1}.auth.H = H_1$. Therefore, $\pi_{U_1}.auth$ was not submitted to the signing oracle, and yet σ verifies over $\pi_{U_1}.auth$, so $\mathcal{B}_2$ can output this forgery.

In Case 2, $\pi_A.n_A \neq \pi_B.n_B$. 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 $\frac{1}{2}$, $\pi_{U_0}.n_{U_0} < \pi_{U_1}.n_{U_1}$.

In that case, U_0 cannot have signed a set $\pi_{U_0}.auth$ with $n_{U_0} = \pi_{U_1}.n_{U_1}$ 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 $\mathcal{A}$ triggers FP then $\mathcal{B}_2$ wins the EUF-CMA game. This leads to the upper bound $\Pr[\mathrm{FP}] \leq 2 \cdot \mathtt{Adv}_{\mathcal{B}_2}^{EUF\text{-}CMA}(\mathcal{S})$.