Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations

February 20, 2023 - v2.0

Cas Cremers
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
cremers@cispa.de

Charlie Jacomme[*]
Inria Paris, France
charlie.jacomme@inria.fr

Aurora Naska
CISPA Helmholtz Center for Information Security
Universität des Saarlandes
Saarbrücken, Germany
aurora.naska@cispa.de

Abstract

The building blocks for secure messaging apps, such as Signal's X3DH and Double Ratchet (DR) protocols, have received a lot of attention from the research community. They have notably been proved to meet strong security properties even in the case of compromise such as Forward Secrecy (FS) and Post-Compromise Security (PCS). However, there is a lack of formal study of these properties at the application level. Whereas the research works have studied such properties in the context of a single ratcheting chain, a conversation between two persons in a messaging application can in fact be the result of merging multiple ratcheting chains.

In this work, we initiate the formal analysis of secure messaging taking the session-handling layer into account, and apply our approach to Sesame, Signal's session management. We first experimentally show practical scenarios in which PCS can be violated in Signal by a clone attacker, despite its use of the Double Ratchet. We identify how this is enabled by Signal's session-handling layer. We then design a formal model of the session-handling layer of Signal that is tractable for automated verification with the Tamarin prover, and use this model to rediscover the PCS violation and propose two provably secure mechanisms to offer stronger guarantees.

1 Introduction

Modern secure messaging apps use intricate cryptographic building blocks to achieve stronger security guarantees. For instance, the Double Ratchet (DR) protocol is a core component of the Signal protocol library, and is used by many secure messaging apps including WhatsApp, the Signal App, and Facebook Secret Conversations.

There is a rich line of research formally analyzing and proving the strong security properties of the Double Ratchet and its variants, e.g., [1, 2, 4, 9–11, 19, 25]. This includes, for example, proving that the DR achieves Forward Secrecy (FS) and modern properties like Post-Compromise Security (PCS): even after a full compromise of a party's device state, attackers are locked out of the conversation again if the victim can exchange a few messages with their partner without the attacker interfering [12]. Over time, these models have been improved in terms of precision, granularity, and threat models, getting closer to real-world DR implementations.

While these are important results, they focus on a specific building block only, and not on the security guarantees that users actually get when using a messaging app. In particular, while the Extended Triple-Diffie-Hellman (X3DH) handshake and the Double Ratchet (DR) have been extensively formally studied, there has been no formal analysis that includes other layers, such as the session-handling. In fact, several application-level behaviors of messaging apps were discovered that seem to offer to their end users lower security guarantees compared to what might be expected from using X3DH and DR protocols, such as [13, 31].

We initiate the formal study of the session-handling layer of secure messaging apps, picking Signal's Sesame protocol as a first case study. This is technically challenging, as we would ideally like to model the

[*] This work received funding from the France 2030 program managed by the French National Research Agency under grant agreement No. ANR-22-PECY-0006.

entirety of the session-handling layer with a detailed model of the X3DH and DR protocols. However, the fact is that state-of-the-art formal approaches struggle to deal with X3DH and DR accurately even without any additional mechanisms. To make any analysis tractable, we have to devise a reasonable abstraction of the session-handling layer and its underlying building blocks.

We motivate our work by experimentally showing two scenarios in which Signal Android app v5.40.4 (May 2022) does not achieve PCS despite using the DR protocol. Our core observation that we strive to formalize is that while a single DR session preserves PCS, this does not propagate to the higher level of the conversation between two end users as conversations are invisibly constructed from possibly multiple concurrent sessions that have their own ratcheting chains.

Interestingly, our experiments show that PCS is broken even though our attacker is more restricted in capabilities than the classical *active* attacker from the literature $\mathcal{A}_{\text{Ideal}}$. However, our attacker model for the experiments, $\mathcal{A}_{\text{Experiment}}$ that can compromise at some point the user phone but otherwise follows the protocol, is stronger than the *passive* threat model of PCS from Signal's specification $\mathcal{A}_{\text{Signal}}$. Following this, we systematize different threat models for PCS based on attacker capabilities and level of compromise. In particular, we consider two meaningful weakened versions of the literature: a) $\mathcal{A}_{\text{Experiment}}$, from our experiments, and b) $\mathcal{A}_{\text{Formal}}$, which is the standard model used in state-of-the-art verification tools. It is on $\mathcal{A}_{\text{Formal}}$ that we perform our security analysis of the session-handling layer and of some proposed improvements.

Thus, our work ranges from the applied (real-world experiments on the current Signal app with a clone attacker) to the foundational (first formal modeling of session-handling layer for secure messaging, and automated analysis).

Contributions Our main contributions are the following:

- We first show the real-world impact of the session-handling layer: we showcase two practical scenarios in which the current Signal app does not achieve PCS, despite using the DR protocol, against a clone attacker.
  Our experiments show how guarantees proven for the underlying building blocks (e.g., in [1,9,24]) do not transfer to the application level. We analyze the root causes and determine they derive from design decisions in Signal's session-handling layer: the Sesame protocol.
- We initiate the formal study of security properties of secure messaging that take the session-handling layer into account. We create the first formal model of Signal that includes an abstraction of its session-handling layer Sesame and the ratcheting mechanisms. Notably, we introduce the notion of conversation-based Post-Compromise Security (PCS), which lifts the classical PCS property to the conversation level. Automated analysis of this model using the TAMARIN prover automatically discovers the scenario in which PCS is violated at the session-handling layer despite using the DR protocol.
- We show that protection against a clone attacker at Signal's session-handling layer can be improved by a simple fix, and use our model to formally prove that the fix restores conversation-based PCS. We also propose a clone detection mechanism for Signal that supports the first fix and which we prove sound in our models.

We provide our models publicly for inspection and reproduction [14]: they are notably intended for developing refined Sesame models or as a starting point to model the session-handling of other apps.

Outline We introduce the required background on Signal and its core mechanisms in Section 2. We then detail two scenarios in which PCS is violated, and their experimental set-up in Section 3. We follow up by proposing our mechanisms in Section 4. In Section 5 we develop our formal model of Signal's session-handling layer. We then use this model to automatically detect property violations, and to prove that our PCS fix and clone detection mechanism indeed work. In Section 6 we discuss the practicalities of implementing our proposed solutions. We finally summarize our findings and provide an outlook in Section 7. In addition, we provide a third scenario that violates PCS in Appendix A.1 (independently reported in [18]), and discuss some additional security parameters of the implementation in Appendix A.2.

Related work The notion of PCS was introduced and formalized in [12]. Since then, the Double Ratchet (DR) and Extended Triple Diffie-Hellman (X3DH) protocols have received multiple increasingly fine-grained analyses [1,2,4,9,11,19], and are still subject to active study. In particular, recent studies have proposed post-quantum secure versions of the X3DH key exchange protocol [6,7,16,22]. Similar to our approach, some of the previous analysis relied on formal analysis tools [3, 25]. However, these focused on the two underlying DR and X3DH protocols, not Sesame. To the best of our knowledge, we are the first to formally analyze the final security provided by the combination of Sesame and the underlying layers.

In [31], the authors show an attack on Sesame that allows the registration of an attacker's device capable of sending and receiving messages. In another work [8] is proposed an alternative to Sesame that opens only a single channel between two users, while hiding the number of partner's devices. From the experimental point of view, the concrete security of Signal under cloning was studied in [13]. In contrast, we show that compared to their experiment on Signal Android v4.47.7 (August 2019), some security guarantees of Signal were lost compared to v5.40.4 (May 2022). Previous work on clone detection propose mechanisms that use *third*-parties [32] or *counters*, *hashing*, and *commitments* [29] to flag the attacker.

A related work is [18] that proposes a new mechanism to improve Signal's PCS guarantees. They consider a PCS property against a fully active attacker while our attacker is inactive during the small healing time-frame and otherwise active. More importantly, they also only consider a single DR session, while we put the focus on multiple parallel DR sessions and lift security to conversations. The two core PCS violations we uncover are due to parallel sessions and are out of scope of [18]. During our experiments, we also independently rediscovered one of their PCS violations, but since this is orthogonal to our main points, we only describe it in Appendix A.1. A related work [**?**] classifies attacker models for PCS and provide PCS variants that depend on the speed of the recovery. Compared to us, they do not consider out-of-order messages in their properties nor multiple parallel sessions; this leads them to theoretically claim several variants for PCS for Signal (see [**?**, Fig 2, Local Active attacker]) which we illustrate do not hold in practice for users. It is however natural to link their threat models to ours. They consider three dimensions for the attacker model, the *Reach* dimension to specify which sort of keys the attacker can compromise, the *Power* dimension for whether the attacker is active or passive, and the *Access* dimension, for whether the attacker sits on the network or is even on the server side. Looking forward to Fig. 4, we also consider the *Power* dimension, but in a more fine-grained way, as well as the *Reach* dimension. We do not however consider the *Access* dimension.

2 Background

2.1 Security properties of messaging protocols

In the following, we give an overview of two security properties achieved by the core Signal protocol DR, namely *forward secrecy* and *post-compromise security*. While both capture session security guarantees with respect to a compromise, the former talks about previous sessions and the latter about future ones, as shown in Fig. 1.

Forward Secrecy (FS) FS is the guarantee that compromise of a session does not impact the security of previous sessions of the protocol [5]. In other words, despite revealing the current session's state (e.g. identity key and session key) to an attacker, previous message keys cannot be computed.

In reality, there are subtle variants of FS. One main form relates to the compromise of long-term keys such as identity keys and an attacker that can at least observe all network traffic. To achieve this variant of FS, parties can use asymmetric cryptography to derive the session keys from ephemeral asymmetric keys such that each key is independent of the previous, while the identity keys are used for authentication. In practice this is usually realized using ephemeral Diffie-Hellman keys or KEMs, but needs a back-and-forth roundtrip to offer protection. In contrast, if keys were decrypted from network traffic using identity keys, this form of FS would not be met, as they would for example in basic key transport protocols.

A second form of FS involves attackers that can only reveal session/message keys, and can be achieved without roundtrips. This can be achieved by encrypting each with a different key, i.e., an attacker knowing the key of message i, cannot compute the previous message keys $[0, i-1]$. Instead of encrypting the messages with the session key, the latter can be used as an input to a key derivation function KDF (i.e., a one-way function), which outputs a new message key and the forwarded session secret. In the literature, FS can also be achieved using time-based methods [17] or puncturable encryption [15, 20, 21].

Post-Compromise Security (PCS) PCS is the guarantee a party A has, that security of their conversation with partner B can be recovered (healed) after the compromise of the latter [12]. In other words, leaking the partner's keys does not mean that all future communication can be decrypted. To restore security, the parties need a *healing* phase, during which the attacker does not interfere with the honest communication. Depending on the type of compromise, a protocol offers two levels of PCS a) via *weak* or partial compromise of ephemeral secrets (*session* keys) or b) via *full* compromise of the state (both *long-term* and *session* keys).

Weak-PCS is achieved when the attacker leaks session-specific keys, but has only temporary access to the long-term secrets e.g., if the identity keys are stored encrypted in an external device. This means that the protocol can use the long-term secret (or a token derived from it) to evolve old corrupted session keys and lock the attacker out. More specifically, the parties can compute the session key sk for run i of the protocol as $sk_i = KDF(sk_{i-1}, token)$. If the session key is forwarded this way, the attacker cannot compute future keys without knowing the token, hence security is restored.

In the full compromise case, the attacker knows all the keys at the time of compromise, therefore the parties need to incorporate new ephemeral secrets in the derivation of the session key. This can be done using asymmetric cryptography [19, 23, 24], such that enough entropy is introduced in the session key during the healing phase, e.g., using the Diffie-Hellman output as the token. Having been passive in the healing phase, the attacker cannot compute the new key.

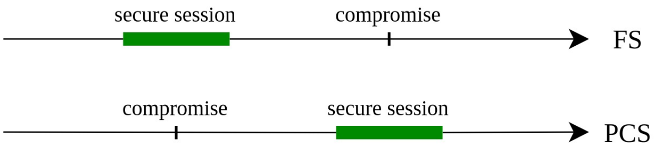

Figure 1: Forward Secrecy (FS) and Post-Compromise Security (PCS).

2.2 Double Ratchet

Signal's Double Ratchet (DR) is a widely used protocol to exchange messages between two parties which guarantees strong security properties such as forward secrecy and post-compromise security. The protocol is constructed by two parts: a symmetric and an asymmetric ratchet, the latter also referred as the Diffie-Hellman (DH) ratchet. As a prerequisite, DR expects the parties to authenticate one another and establish a shared key before the start of the protocol, e.g., Signal uses the Extended Triple Diffie-Hellman (X3DH) protocol [26].

DR is constructed from a hierarchy of three types of keys: a root key which is the shared key between the parties at the start of the protocol, chain keys derived from the root key, and message keys derived from the chain keys. When the parties switch their roles, say from sender to receiver they perform the asymmetric step, and when they maintain the same role they perform the symmetric step. We will shortly summarize both ratchets below, and refer the reader to the documentation [30] for more details.

Asymmetric Ratchet The core idea of this step is to introduce ephemeral DH keys into the root key, thus achieving PCS. Consider a party A switching roles, say from receiver to sender. They generate a DH key pair (a, g^a), and get their partners public key g^b. Then, A computes the current root key rk_i and sending chain key ck_0 using a KDF with input the old key and the DH output, i.e., $rk_i, ck_0 = \mathtt{KDF}(rk_{i-1}, g^{ab})$. The initial sending chain ck_0 serves as input for the symmetric ratchet to derive the actual message keys. A will keep performing the symmetric ratchet, i.e., send messages sequentially, until interrupted by an incoming message, signaling role switch with the partner.

Symmetric Ratchet The symmetric step forwards the chain key, using a KDF function, to derive new keys for each message, thus providing forward secrecy, Resuming the previous example, A can now compute any message key mk_i from the chain key, specifically, $ck_i, mk_i = \mathtt{KDF}(ck_{i-1})$. Then, A uses mk_i to encrypt a message and along with the ciphertext sends their public DH key g^a. Upon receiving this message, the partner will compute the same steps and be able to decrypt.

2.3 Sesame: Signal's session management

Sesame is a session management protocol responsible for managing sessions and multiple devices for the libsignal library. The protocol enables a user to link several devices to their account and handles the synchronization of messages sent and received with the devices of their partners. The core idea is to create a session between the sending device and every other device of both parties, i.e., a message sent from user A to B is encrypted for every device of B and all other linked devices of A. This means that Sesame has to manage a local database containing the records of each partner's device and their respective open sessions. To maintain the device list, Sesame depends on the Signal server to inform it of any new or removed devices. In addition, Sesame uses a mailbox server to store the messages sent to devices until they can retrieve them, enabling asynchronous communication.

Sesame is also designed to handle multiple sessions between two devices. According to the specification, this is done to ease the convergence to a single session in cases where two devices are desynchronized. These cases include: a) parties starting a session simultaneously and hence resulting in different derived keys, b) one of the parties restoring an old backup, or c) losing their local state. Sesame keeps a list of multiple sessions per device, and upon receiving a message encrypted from any of them, promotes it as the new *active* session, thus making both parties agree on the current state.

Initialize Session Sesame uses a key agreement protocol (i.e., X3DH in Signal) to initialize the sessions between the sending device and all receiving devices. For this, X3DH needs at least two types of keys: an identity key and an ephemeral key, called a pre-key bundle. The idea is to use the keys of both parties in an intricate key derivation algorithm, in which only the two involved participants can derive a common secret [26]. In Sesame's implementation, the identity key pair is assigned per user account, and the ephemeral keys are generated per device and signed by the identity key. This allows for user authentication and uniqueness of session keys among the devices of the same account. When a device wants to start a new conversation, they retrieve from the server they prekey bundle and compute a shared key that serves as the initial root key to the DR algorithm described in Section 2.2.

Multiple Devices End users in Signal can have multiple devices linked to their accounts. The devices share the user's identity key but have per-device prekey bundles. The latter is then used during the X3DH agreement to set up pairwise Signal sessions between them. For example, assume A has m devices and wants to send a message to their partner B's n devices. A device of A will start independent sessions with the n recipient devices of B and their own $m - 1$ linked devices and encrypt the payload using the state of each session. The states are then stored locally on the device, indexed by the user and device identifiers for which they should be used. To maintain an updated device list, the users depend on the server to inform them of new and deleted devices of their partner.

Multiple Sessions In reality, between two devices there are multiple sessions, which are managed by the Sesame algorithm. From these sessions, one is the current *active* session used to encrypt messages and the others are stale sessions kept for synchronization or out-of-order messages. However, Sesame allows the older sessions to be promoted to active ones. In a nutshell the mechanisms states that, any session which can decrypt an incoming message becomes the current active session between the two devices. The mechanism of reactivating older sessions, enables a) the decryption of messages without loss, and b) agreement on a common session even when the parties are momentarily desynchronized.

Retry Message Another feature of Sesame is to keep a record of sent messages until the sender receives a valid receipt message. This means that if the partner cannot decrypt any message, they can send an *unencrypted* retry request indexed by the message identifier. Upon receiving the request, the sender encrypts the message with a new key derived from either: a) the current active session (in case it is different from the one used to encrypt the message previously), or b) establish a new session with the device.

Session Reset and Expiration Sesame also suggests a session expiration policy, however the implemented mechanism differs from the specification. In the design of Sesame, it is suggested to delete old sessions depending on their creation timestamp. In practice the app limits the of number sessions (40 sessions) that are maintained at any time in a FIFO fashion, i.e., anytime it creates a new session it removes the last one in the pile from the list. In addition, previous versions of the Signal app used to offer a session reset which is not included in the specification. The idea is that the users can trigger the creation of new sessions themselves directly from the user interface. However, this was deprecated in Signalapp version v5.25.0[1]. Instead, the

[1] Signal's Session Reset Deprecation: link.

conversation is automatically reset every one hour when the two parties need to establish a new session. This means the keys used in the conversation are replaced every hour by freshly computed ones. In theory this could yield increased security for the protocol, however as we will see later, this is not the case in practice.

3 Concrete Signal scenarios violating PCS

In this section, we show two real-world scenarios in which a clone attacker can violate PCS for Signal, which are consequences of the current design of Signal's session-handling layer Sesame. In our scenarios, we consider a so-called *clone* attacker. The clone compromises the full state of a party, including session specific keys and long-term identity keys. However, it is limited in using the user interface of the app, instead of performing complex operations with the compromised secrets. In practical terms, this can be an attacker gaining temporary access to a device and duplicating its contents to run a parallel app instance. Notably, the clone does not need to have any knowledge of intricate attack vectors or the inner workings of the protocols. Our experiments were performed on the Signal Android app version v5.40.4 (May 2022).

Intuitively, we consider the following scenario in attacking the conversation between A and B: At some point during the message exchange between the honest parties, we compromise A's device, effectively cloning it. We then have A and B continue their conversation without the attacker being online, i.e., the *healing* phase. From the analyses of the DR protocol, we expect that this heals the conversation. We then re-activate the clone. The expected behavior now is that the clone should be locked out of the conversation of A and B.

From a high level point of view, we found the following scenarios that violate this expectation:

1. if an old clone and the honest partner are online, decryption failures on the clone side trigger retry messages, resulting in a new session agreement and the honest partner resending previous messages to the clone (Section 3.1), and
2. after a time-triggered reset (creating a new session), a clone can still re-enable an old session (Section 3.2).

We did find a third scenario in which PCS is violated: old receiver chains that are still stored can be prolongated by a clone. We describe this scenario in Appendix A.1, since (a) it exploits the DR implementation, and is not linked to Sesame's design, and (b) it was independently reported in [18].

Interestingly, a clone that simply uses the unmodified Signal app is enough to perform these scenarios. In the following, we show how we were able to exploit emulated clones to produce the attacks on the deployed Signal app, and explain the attacks using Signal's log as well as the Sesame/Signal specifications and its open source code.

Experimental setup We conducted a series of experiments on the Signal application to find potential weaknesses introduced by Sesame. First, we created two legitimate accounts in the deployed messaging app (version 5.40.4). To register the users, we modeled two android devices (Android Api 10.0) using the Genymotion emulator (version 3.2.1). After the two honest parties started a conversation together, we investigate the consequences of a compromise. Our threat model allows for an attacker to fully compromise the state of a party and run another instance of the app with the cloned state. To mimic this, we used the cloning feature of the emulator, which allows to fully duplicate an entire virtual machine. We then explore the behaviors of the multiple parties by exchanging messages between them, triggering different behaviors (such as message resend, session reset, etc) potentially leading to violations of properties such as PCS.

To precisely identify the causes of the observed behaviors we downloaded the debug logs provided by the app.[2] From these logs, we can extract the following information:

1. Which chain keys are used for encryption (unique public identifier of the chain key for logging purpose)?
2. When do parties send retry requests?
3. When did the parties initialize a new session?

This allowed us to deduce which actions were triggered to allow messages to be received without any error in the user interface, which should have otherwise been protected by PCS. In addition, since Signal's library is open source, we cross-checked with the implementation and pinpointed the exact parts of the code that enabled the scenarios.

[2] Logs are found in the user interface under Settings/Help/Debug Log.

Notation The experiments are conducted between 3 agents: two honest parties A and B, and A's clone (A_{clone}). When a message is sent from A to B we use the notation $A \rightarrow B$ and vice versa. To denote that the parties are exchanging roles between sending and receiving we use $A \leftrightarrow B$, which means $A \rightarrow B$ and $B \rightarrow A$. Repetition of the message exchange n times is expressed as $(A \leftrightarrow B)^n$.

3.1 Abusing the Retry feature

In this scenario, we investigate whether a compromised party can recover the security of the conversation once all chain keys known by the attacker are deleted from the app's local memory. The experiment considers two parties A and B having a conversation, until A is cloned (i.e., the attacker has an identical copy of the messaging app). The honest parties continue the conversation without interference; in terms of the underlying DR, this should enable healing. Once the clone's machine goes online, they check if the clone receives past messages or can inject new ones in the conversation.

Concretely, we have A and B exchange more than 5 messages in the conversation after the clone to represent healing, and we check if the clone can still inject their own messages in the conversation. We need 5 or more messages, because, as shown in Appendix A.1, the conversation does not heal due to Signal storing at any time the latest 5 receiver chains, which can be extended by the clone.

In this experiment, we noticed that B was able to receive all the messages sent from the clone, as long as the two are online at the same time. This means that PCS is not guaranteed, since the clone can continue the conversation even after 5 consecutive healings. We will now see how A_{Clone} and B resynchronize, abusing the retry feature of Sesame.

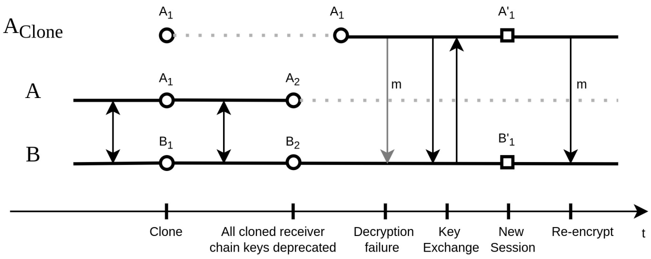

Figure 2: Abusing the retry option upon decryption error on Signal. After parties A and B exchange messages, A at state ○A_1 is cloned. The clone goes offline (denoted by the dotted line) and the parties exchange messages to heal their session to ○A_2. A goes offline, while the clone encrypts a message m with the old session state ○A_1. B cannot decrypt the message, so they initiate a new session marked, respectively, as □ B'_1 and □ A'_1. B can decrypt the message received in the last step.

Concrete steps The scenario can be reproduced as follows:

1. A and B exchange 4 messages: $(A \leftrightarrow B)^4$.
2. Duplicate A's machine and send clone offline.
3. A and B exchange 6 more messages: $(A \leftrightarrow B)^6$.
4. A goes offline, and clone goes online.
5. Clone sends 4 consecutive messages to B $(A_{Clone} \rightarrow B)^4$
6. B can decrypt the messages sent by the clone.

Fig. 2 illustrates the scenario described above.

Analysis The experiment shows that the clone can insert their own messages despite the honest parties having deleted all the compromised chain keys from memory. This is due to Sesame's feature of retrying to send a message. Basically, once B receives the clone's messages sent in step 5), it realizes that it cannot decrypt them and sends a retry receipt. Once the clone processes the message, it will initiate a new X3DH key exchange with B and re-encrypt the messages using the new session's keys. Now, B can derive the new session keys and decrypt the messages.

We verified with the app's logs that the clone received the retry receipt from B, followed by fetching keys from the server and starting a new session. Also, on the receiver's side in B's logs, we can observe the successful decryption in the newly created session. The retry mechanism is described in the Signal documentation[3] and implementation[4]. We thus showed how the retry option of the session management breaks the PCS guarantee offered by the underlying protocol DR, by allowing a clone to continue the conversation with A's partner despite the healing phase. In fact, not only can the clone inject messages, but also have a conversation in parallel to A, as we will show in the next scenario.

3.2 Abusing session reactivation

As the conversation evolves, Signal automatically resets sessions every hour, i.e., the parties generate new secrets and deprecate the old ones. One would assume that this increases the security of the conversation, and potentially prevents the scenario from Section 3.1. However, Sesame allows for older sessions to be reinstated as the main sending session, i.e., the parties will encrypt messages using the latest state of these sessions. The following example gives an intuition on why this seemingly small feature can break the PCS security.

Example: Consider the clone of a compromised party as described in Appendix A.1 and Section 3.1. The two honest parties resetting the session means that all compromised session secrets are replaced by new ones. From the DR design, this implies that the attacker has no knowledge of the current session keys and should be locked out of the conversation. However, due to the time-based reset, once the clone goes online they will start a new session with the partner as well and can continue the conversation. In addition, Sesame stores previous sessions and promotes any old session as the current active, upon receiving a message from it. Assume now that the honest party also sends a message from their session. Upon receiving the message, the partner will switch to the session that it can be decrypted with and reply to the honest party using the keys stored there. As a result, the victim can also continue the conversation, despite the clone doing the same in parallel in another session.

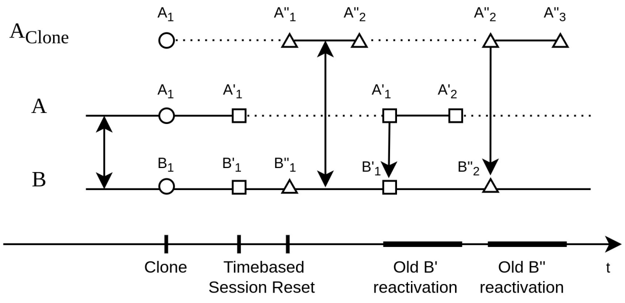

Figure 3: Abusing session reactivation, making it possible for A and its clone to have parallel sessions with B. A and B exchange messages, until A at state $\circ A_1$ is cloned. The clone goes offline (denoted by the dotted line) and the parties exchange messages until session reset at $\square A_1'$. A goes offline, and when the clone sends a message, the automatic reset starts a new session $\triangle A_1''$ and respectively for $\triangle B_1''$ for B. Now, when A sends a message, B reactivates $\square B_1'$, and $\triangle B_2''$ for when the clone does.

Concrete steps The steps for this scenario are:

1. A and B exchange 4 messages: $(A \leftrightarrow B)^4$.
2. Duplicate A's machine and send clone offline.
3. Signal resets the session after 1 hour.
4. A and B exchange 6 more messages: $(A \leftrightarrow B)^6$.
5. A goes offline, and clone goes online.
6. Clone sends 4 consecutive messages to B $(A_{Clone} \rightarrow B)^4$
7. B can decrypt the messages sent by the clone and continue the conversation.
8. Clone goes offline, A online. A encrypts 4 messages to B $(A \rightarrow B)^4$.
9. B can decrypt the messages sent by A.

Fig. 3 shows the scenario described above.

[3] Sesame's retry option documentation: link.

[4] Sesame's retry option implementation: link.

Analysis The experiment shows that indeed A and their clone can have parallel conversations with B, without the conversation ever healing or any of the agents receiving any notification or sign of compromise in the user interface. The parties exchange 6 messages in step 4) in order to delete any old stored keys. The previous scenario is excluded, since the time-based reset makes it impossible for the clone to use the old compromised session. Instead, they are forced by the app to start a new session with B and encrypt their messages with correct keys. This means that B has no reason to issue a retry request. Note that the clone interacts only with the user interface, therefore it strictly follows the Signal protocol. The same logic follows for the real A at step 8), where they send messages with keys B is able to compute, hence no decryption error is thrown in the app (no need for retry option).

There are two mechanisms intertwined here:

- The reset session between the attacker and A allows for the compromise to continuity.
- The session switching allows for a compromise and honest session to exist in parallel without any detection.

We could confirm from the logs that a new session was indeed established between the clone and B at step 6. The logs also confirmed that when receiving a message from an old session, B would log that they "decrypted with previous session state", which is also visible in the implementation [5]. We thus saw the old session reactivation at step 8). This could go on like this with B alternating between the compromised and the honest session, where the latest one used to receive is promoted to be the current active one [6]. The victim can never heal in this case, since on every automatic reset both of them are forced by the app to start two new parallel sessions with B.

Existing mitigation There is currently a bound on the number of sessions stored (40 sessions). We would expect this bound to lock the attacker out at some point. However, this is not the case for two reasons.

First, consider the case of a full compromise of both the identity key and the session states, which corresponds to our previous experiment. Let us assume that the clone stays offline for more than 40 hours, which is the lifetime of a session before it is deleted[7]. A and B can then delete enough sessions so that old ones cannot be reactivated. However, because the clone knows the identity key, it will automatically start a new session with B when going back online.

Second, consider a weaker threat model where the bound would intuitively be enough: the case of a partial compromise, where only one session was compromised and the attacker cannot initiate new sessions with B. Then, it seems that the bound would still not be enough, as an attacker could regularly send messages to B, so that even if no reset goes through, the compromised session is regularly promoted and stays at the top of the stored session list, and is thus never dropped. This could be circumvented if the implementation followed the session's expiration policy (see Section 2.3). Note that to carry out this experiment, we would however deviate from an attacker that simply uses the official Signal app.

4 Improving Sesame's PCS guarantees

Experimental summary Currently, Signal's session-handling layer allows the following scenarios in which PCS is violated:

1. After a compromise, the cloned device can initiate new sessions with the partner. The creation of this new session may either follow a session reset from a timeout as in Section 3.2, or from a retry receipt Section 3.1.
2. When there are both a malicious session and a valid session running in parallel, the compromise will continue undetected, as we showed in Section 3.2.

Threat model systematization Our aim is to propose provable fixes in the later sections. We thus need to clarify the set of relevant threat models, as summarized in Fig. 4.

The threat model considered in Sesame's specs, which we will refer to as threat model $\mathcal{A}_{\text{Signal}}$, is defined as a passive attacker that has fully compromised the device's state. Its claims of security against a passive attacker are derived from the underlying protocols it manages the sessions for, namely DR and X3DH. A passive attacker is an eavesdropper that does not interfere with the protocol flow or the agents.

[5] Sesame's Previous Session Decryption: link.

[6] Sesame's Session Reactivation: link.

[7] Number of stored sessions in Signal: link.

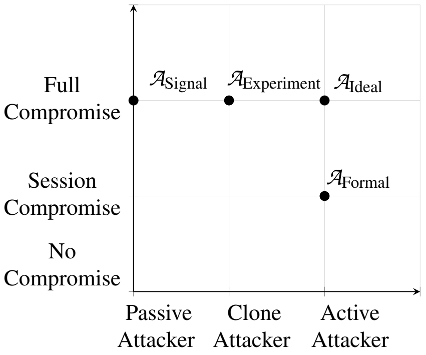

Figure 4: Signal's four threat models categorized by attacker capabilities and level of compromise, which are: the threat model from Signal's specification ($\mathcal{A}_{\text{Signal}}$), the DR's literature threat model ($\mathcal{A}_{\text{Ideal}}$), the experiment model in this work ($\mathcal{A}_{\text{Experiment}}$), and the formal analysis model ($\mathcal{A}_{\text{Formal}}$). Full compromise means session and long-term keys compromise.

In contrast, in the literature it is shown that DR is resilient to a stronger adversary that not only has total control over the network but can drop, inject and manipulate any message actively. In an ideal world, we would be able to lift the guarantees from the literature over the DR to Signal, even in this stronger active attacker threat model. We will refer to an active attacker that can fully compromised the state of an agent as the threat model $\mathcal{A}_{\text{Ideal}}$, the ideal goal. However, Signal has no mechanism to achieve this, since it would require to refresh the identity keys of compromised agents, which is currently treated as a new registered user. Achieving PCS for $\mathcal{A}_{\text{Ideal}}$ would require significant new developments of the protocol.

However, there are two interesting weakenings of $\mathcal{A}_{\text{Ideal}}$:

1. Our experiment threat model $\mathcal{A}_{\text{Experiment}}$: an attacker that can compromise fully the user, but then strictly follows the protocol and the features from the app.
2. Our formal analysis threat model $\mathcal{A}_{\text{Formal}}$: an active attacker that can however only compromise a single session and not the identity keys.

$\mathcal{A}_{\text{Experiment}}$ has been considered in previous work, which proved that Signal could achieve PCS for this threat model. This is not the case anymore we show in Section 3. Furthermore, we also demonstrate in our formal analysis that PCS is in fact lost due to design choices in Sesame for $\mathcal{A}_{\text{Formal}}$. This in turn makes it impossible to achieve PCS for $\mathcal{A}_{\text{Ideal}}$ even with new mechanisms for identity key revocation. Further, achieving PCS at $\mathcal{A}_{\text{Ideal}}$ with a key revocation implies to be able to know when we need to revoke keys: there is the need for clone detection at the lower levels.

Threat models in real-life The threat models discussed in this paper can be tied in multiple ways to real life scenarios. They all rely on the possibility of compromise of some keys, either long-term or short-term. There are several scenarios in which this can happen, e.g., malware installed on the phone may be able to extract the keys from memory, or when a third person has physical access to a device, such as when crossing a border control point. It is relevant to distinguish long-term and short-term keys, as long-term keys can be stored inside a safer storage, with a TPM or an enclave, whereas short-term keys are typically easier to compromise. With this is mind, concrete instantiations of the previous threat models can be:

- $\mathcal{A}_{\text{Signal}}$: an attacker that can compromise short-term keys (long-term keys are irrelevant in the passive case as they must be used to produce signatures), but can only observe traffic and not manipulate it, or obtain traffic after it happened. While it is reasonable to assume that the attacker cannot interfere with direct traffic between the phone and the server, it is more surprising to not consider that the attacker can use the compromised keys to establish its own connections with the server.
- $\mathcal{A}_{\text{Experiment}}$: generalizing the previous case, this is the case where the attacker compromises both short-term and long-term keys and can use them to create new connections with the server, simply using Signal's app, but cannot interfere otherwise with later exchanges between the server and the honest phone. In practice, this is the scenario where an attacker can have physical access to the phone, completely clone it, and then use the clone to connect to internet. This requires physical access to the phone, but no particular resources afterwards except an internet connection.
- $\mathcal{A}_{\text{Ideal}}$: here we make the attacker fully active, and it can now also interfere with the connections between the honest phone and the server. In addition to the compromise, this may require in practice active

surveillance of the global network, or more simply the deployment of malicious wifi hotspots.

- $\mathcal{A}_{\text{Formal}}$: This is the transposition of $\mathcal{A}_{\text{Ideal}}$, where we assume that long-term keys are stored in a TPM, and only short term keys can be compromised. This is a natural model for formal verification tools, with a fully active attacker and where we can precisely define the possible compromises. Developments in this model pave the way for a full model in $\mathcal{A}_{\text{Ideal}}$.

An attack in one of these models may in fact require weaker attacker capabilities, as is the case of our attacks that are in $\mathcal{A}_{\text{Experiment}}$, but do not rely on the long-term keys.

Fixes overview To restore the Sesame PCS in $\mathcal{A}_{\text{Formal}}$, our first fix specifies that an outdated session must only be allowed to receive older messages, and never re-enabled to send new ones. Moreover, our second fix specifies that receiving a message on an outdated session should trigger a warning message, sent to both the current and the outdated session, thus enabling for clone detection on the other end. Our first solution is a minimal fix to restore PCS that does not break the current functionalities and ensures that no messages are lost. Our second fix requires adding messages to the protocol flow, and is thus slightly more complex to implement.

In the following, we first precisely describe our fixes, and then show how we prove their security using the TAMARIN prover. From a high level point of view, we provide four distinct security results:

- We prove that when restricted to a single session between two parties, a model of the Double Ratchet with idealized cryptography does provide PCS. This also leads to a clear formalization of the best PCS guarantees that the Double Ratchet can provide.
- We show that when multiple sessions are allowed in parallel, our model is expressive enough that the scenario of Section 3 is automatically found in TAMARIN.
- We prove that when implementing our first fix, we restore PCS between two parties.
- And finally, we prove that the clone detection fix is sound: it does not raise false positives and cannot be triggered without an actual compromise.

4.1 Description of fixes

PCS fix As we have seen, PCS is lost because healing one session does not heal others. Subsequently, we should make sure that after the current active sessions have been healed from a compromise, older sessions cannot be used again. A first idea would be to instantly fully deprecate older sessions and keys whenever we create a new one. However, this results in messages being lost because the parties are unable to decrypt out of order messages. Thus, we instead suggest that:

- sessions must never become active again and used to send messages after they became inactive;
- inactive sessions can still be used to receive messages.

The solution is simple, practical, and preserves the current level of usability with minimal changes to the code. Notice that in the case of multiple devices in a conversation, all pairwise devices will have their own sessions. This means that our fix does not affect the normal communication when for example a device is offline and the conversation's state has moved forward because either a) the other devices send the messages using the symmetric ratchet of the same session, or) start new sessions using a new prekey bundle of the offline device. In both cases, the offline device can compute the new secrets and decrypt all messages.

As we will prove later, our fix allows lifting the precise PCS guarantee provided by a single session to multiple sessions.

With respect to the specification, our fix is obtained by removing item 4 of section 3.4 from [27]), which says that upon decryption of a message "if the relevant session is not active it is activated". This step of the algorithm is responsible for promoting an old session as the current active one.

The issue of Clone detection With our previous fix implemented, we restore PCS in $\mathcal{A}_{\text{Formal}}$, but a clone in $\mathcal{A}_{\text{Experiment}}$ may never be detected. Indeed, whenever either the clone or the honest party go online, they can trigger the creation of a fresh session and become the current active session with their partner. This results in the inactive party to simply not receive any message as long as it does not go online. We thus propose a mechanism that allows the detection of such behaviors. In particular, assuming that A is talking to B, and A_{Clone} is a clone of A, we need to detect the following two situations:

- A and B share an active session, and B receives a message from A_{Clone} in a deprecated session.
- A_{Clone} and B share an active session, and B receives a message from A in a deprecated session.

It is impossible for B to distinguish whether the received message is from a clone or the honest party. However, B can notify their partner that they received a message from a deprecated session and include in this warning message some information about that session. This way, the responsibility falls on to the victim A to decide whether this warning message implies the existence of a clone or not. To cover the two previously described situations, we need to make sure that A receives the warning message in either case, therefore B needs to encrypt it for both the outdated and the active session.

There is a core difficulty for the clone detection mechanism: the mechanism should not be activated during the normal operations of the protocol, i.e., only a cloning behavior should result in a detection. Without this property, an attacker could trigger false positives, leading to denial of service attacks. This property is called the *soundness* of clone detection in [29].

Clone detection mechanism We make three additions to the current flow. First, we specify that when a new root key rk is computed, both parties derive a root key identifier $\mathtt{pk}(\mathsf{rk})$ and a warning key $\mathsf{wk} = \mathtt{KDF}(\textit{"warning"}, \mathsf{rk})$ and store them in memory. Then, when B with a current active session root key rk_a receives a message inside an inactive session with root key rk_i, B sends to the other party the pair of ciphertexts $(\mathtt{enc}(\mathtt{pk}(\mathsf{rk}_a), \mathsf{wk}_i), \mathtt{enc}(\mathtt{pk}(\mathsf{rk}_i), \mathsf{wk}_a))$. Finally, whenever a party receives a warning message that decrypts to $\mathtt{pk}(\mathsf{rk}_x)$, if this identifier is stored in the memory, the warning is ignored, and if this identifier corresponds to a root key never computed by the other party, the compromise error is raised. Notice that the detection happens at the pairwise communication level between two devices and uses the causality of starting a session to detect the clone. This means the mechanism is unrestricted by the number of devices in the conversation.

Attack examples Going back to the initial example of a conversation between two honest clients A and B, we assumed that a device of A falls temporarily under the control of the attacker e.g., at a border control, or from malware. The attacker that cloned the state of the device can impersonate A to B. When A goes back online, they are notified by our clone detection mechanisms that an attacker has reused an old session key to encrypt messages to B or if they have started a new session on their behalf, leading the victim to decide on whether to trigger a healing mechanism (see Section 6).

In the cloning attack from [13], the attacker compromises the victim's state, the honest devices exchange 5 messages to heal their conversation, and then the clone goes back online and encrypts new messages. In our clone detection mechanism, A needs to go online one more time in order to detect the clone's activity. The following two scenarios are possible:

1. The clone started a new session when it was active, so when A goes online it receives a warning message and detects the new session started by the clone.
2. The clone used the same session state as the honest parties. Here the session is healed thanks to the PCS guarantee and the clone cannot impersonate A, hence there is no need for detection.

In comparison, the attack found in [**?**], and rediscovered during our experiments (see Appendix A.1) considers only a single session, and differently from our work and [13], the honest parties do not exchange enough messages to heal the conversation. The attacker can send new messages using the stored keys in the compromised state, because of the slow deletion of keys from state in the implementation. This means that because of the mismatch between what is considered a healed session in the specification and the actual implementation, the parties have not healed and the session does not have PCS. As a result, we cannot lift the security to conversation-PCS or clone detection.

5 Formal Analysis of Secure Messaging with Session-Handling

In this section, we describe our multiple formal models and give the intuition behind their security proofs. To perform the analysis we rely on the TAMARIN prover, which we introduce in the following along with its way to model security properties. We summarize our security analysis in Table 1 and show the results obtained using the TAMARIN prover in Table 2. We only provide here high-level presentations of our models, and refer the reader to Appendix A.3 and [14] for more details.

5.1 Tamarin

The TAMARIN prover is a tool for formally modeling and analyzing complex cryptographic protocols [28]. TAMARIN works within the symbolic model, meaning that messages are expressed as terms built from variables, constants and function symbols. Rules allow modeling the possible protocol steps, or protocol actions, that can be executed by the multiple parties in parallel, and thus model network inputs, outputs and computations.

Rules can be labeled with so-called events. The semantics of the rules together with the labels yield the set of possible executions, i.e., possible behaviors of the protocol sessions in the presence of the adversary. The security properties are then modeled as guarded first-order logic formulas, and can refer to the events occurring in a specific trace, e.g.,

$$\begin{aligned} \forall\, userId, key, i.\ \mathtt{ReceivedPayload}(userId, key)\ @i \\ \Rightarrow \neg\exists\, j.\ \mathtt{K}(key)\ @j \end{aligned}$$

Here `ReceivedPayload` is the event raised whenever the user identified by *userId* receives a payload encrypted with *key*, and *K* is a builtin TAMARIN event that models that the attacker knows or can compute the argument. Intuitively, this property states that if a user has received a payload at some timepoint *i* inside a trace, there is no point *j* in the trace where the attacker can compute the decryption key. In other terms, this is the secrecy of the keys.

The tool has a built-in attacker, the so-called Dolev-Yao attacker, which has control over the network and can inject, drop and manipulate any traffic. Furthermore, TAMARIN allows the user to add custom equational theories to model additional attacker capabilities or data structures.

Given a protocol model, a specification of the attacker and a security property, the tool returns either a proof that the property holds for all traces of the protocol, a counterexample, or it does not terminate (the underlying problem is undecidable). In case of termination, the user can inspect the proof steps or the graphical representation of the counterexample in the interactive mode. Moreover, the user can guide the tool in proof finding, by writing and proving helper lemmas such as invariants of the protocol.

5.2 Formal model with Session-Handling

We first systematize here the points that should ideally be taken into account to perform a formal analysis of Signal, or any similar application. This corresponds to an ideal goal, that we found is currently out of reach for existing automated analysis tools such as Proverif or Tamarin. We thus introduce several simplifications and abstractions in order to perform a first formal analysis of the core mechanism that we aim to improve here, the multiple session management.

Ideal Goal and limitations In an ideal world, to study the PCS guarantees of Signal a formal model should capture:

- a model of the X3DH and the DR protocols, including publication of Diffie-Hellman share bundles signed with identity keys on the server;
- capture the multiple session managements as well as new device registration;
- allow for compromises of all different materials, chain keys, root keys and identity keys.

Verifying such a model is currently out of our reach and its analysis would probably need to rely on modular result. For instance, the latest Signal analysis using Proverif only considers the X3DH protocol followed by only three message exchange in the DR [25]. To make this first study of the multiple session management tractable, we thus performed several simplifications and abstractions. As we will see when discussing the proofs, they required significant human intervention, even with the following abstractions. We estimate the human effort to be in the order of a month.

Atomic operations To tame the complexity, we abstracted away the initialization of the sessions, replacing the X3DH protocol by a single step creating a new session between two given parties. The intuition is that at our higher level, if X3DH is secure and was successfully completed, then it should be equivalent to instantiating a fresh shared secret between the two parties. We also simplified the asymmetric ratchet step of the double ratchet, by collapsing the sending and the receiving steps into a single rule, which instantly provide the two parties with a new shared secret value to rely on. This is in fact a simplification needed to capture the PCS property: in our context, healing means that an asymmetric ratchet was fully completed and that the attacker did not interfere with its execution. As such, the PCS property considers the asymmetric ratchet, which is the healing step, as an atomic step that the attacker cannot interfere with. If the attacker can interfere and for instance block the execution of a healing, PCS cannot even be expressed in a meaningful fashion. The difficulty of formally verifying PCS against an active adversary is also discussed in [18]: instead of seeing the asynchronous ratchet as an atomic step that the attacker cannot interfere with, they choose to base their security definition on an additional recurring authentication mechanism running in parallel that the attacker must try to cheat.

As a consequence of the two previous simplifications, notice that our model does not need to explicitly capture Diffie-Hellman exponentiations, as we directly model the resulting fresh secret. This significantly reduces the complexity of the symbolic analysis, and may be one of the core points that makes the analysis tractable.

Multiple devices We considered that device registration is a separate operation and that it was reasonable to not include it in the model. With respect to Sesame and our observed PCS violations, we showed in the experiments that the attacks were indeed possible even with a single device registered.

Possible compromise scenarios Each possible compromise adds a layer of complexity to the PCS proof, as it introduces new cases in which the attacker may break the security. In our models, we do not capture the fact that the attacker can compromise the long term identity keys of a party, similarly to [18]. This is meaningful because long term identity keys can be stored in a more secure way, and one first needs to consider PCS against only chain and root key compromise before hoping to achieve it against stronger attackers.

Our formal model and fixes in fact pave the way to aiming for PCS against such strongest forms of compromise: if we have PCS when everything but the long term keys are compromised, we force the attacker to use the identity keys to sign new pre-key bundles in order to maintain the compromise, and simple mechanisms could be set into motion on the honest devices to detect those malicious bundles.

Final Protocol Model Our formal model allows performing the following actions in an atomic fashion:

Initialization - Instantiates a new session with a fresh shared root key between two participants. This corresponds to reaching the final state of the X3DH in a single step.

Send Message - Symmetric ratchet performed on the sender side.

Receive Message - Symmetric ratchet on the receiver side.

Skip Message - Skip a message and ratchet forward to receive a later one.

Asymmetric Ratchet - In a single step swap sender and receiver roles, giving them a new fresh shared root key.

Every initialization leads to the creation of a fresh session between the two parties, each party creating a fresh identifier sid_A and sid_B to identify this new session. In our core model, the number of parties, the number of possible sessions between two parties, as well as the number of ratchets (symmetric and asymmetric) within a given session are all unbounded. By adding restrictions on how parallel sessions are managed, this first model can then be used to instantiate multiple variants of Sesame.

Threat Model The attacker has full control over the network, can manipulate and build new messages, but the cryptography is assumed to be perfect. That is, the attacker can see the encrypted sent messages, but can only decrypt them if it knows the secret key.

In addition, we also consider that the attacker can at any time compromise the currently stored keys of a party, e.g. learn their sending/receiving keys or root keys. An important point is that due to skipped messages, a receiving key can be compromised at any time in the future even after asymmetric ratchets. This corresponds to the behavior of storing old receiving chain keys.

As a consequence of this threat model, remark that if the attacker knows the current key, it can then decide to trigger an asymmetric step either with the receiver or the sender, which locks out the other party.

Session-based PCS We first define here the classical PCS property, which considers the security of each session individually, and thus effectively reasons only about a single ratchet chain. Intuitively, whenever inside a session there was a heal and a message sent afterwards, the attacker cannot learn this message unless it compromises one of the party again. Conversely, the only way that the attacker can decrypt a message is to perform a compromise after the latest heal.

We raise the following events to express the session-based PCS property:

- the event $\mathtt{Sent}(\mathsf{sid}_a, A, B, \mathsf{sck})$ whenever the agent A over its session sid_A sends to agent B a message encrypted with the encryption key sck;

- the events $\texttt{Heal}(\mathsf{sid}_A, A, B)$ and $\texttt{Heal}(sid_B, B, A)$ when the two parties perform an asymmetric ratchet, and thus heal their session;
- and raise $\texttt{Compromise}(A, B)$ whenever the attacker chooses to compromise the cryptographic material of a session of A talking to B.

Capturing this PCS notion formally using TAMARIN's first-order logic language, is as follows:

$$\begin{array}{l} \forall\, A, B, \mathsf{sid}_A, \mathsf{sck}, i, j, k. \\ \quad \texttt{Sent}(\mathsf{sid}_A, A, B, \mathsf{sck})@i \ \&\ \texttt{K}(\mathsf{sck})@j\ \& \\ \quad \texttt{Heal}(\mathsf{sid}_A, A, B)@k\ \&\ k < i \\ \quad\quad \Rightarrow \left(\begin{array}{l} \quad \exists\, l.\ \texttt{Compromise}(A,B)@l\ \&\ k < l \\ \|\ \exists\, l.\ \texttt{Compromise}(B,A)@l\ \&\ k < l \end{array} \right) \end{array}$$

This means that if we are in a situation where A sends a message to B using sck key, $\texttt{Sent}(\mathsf{sid}_A, A, B, \mathsf{sck})@i$, and A healed before sending this message, $\texttt{Heal}(\mathsf{sid}_A, A, B)@k\ \&\ k < i$, and the attacker knows the key $\texttt{K}(\mathsf{sck})@j$, then the attacker must have compromised the keys of A talking to B after the heal $\texttt{Compromise}(A, B)@l\ \&\ k < l$, or the keys of B talking to A.

Conversation-based PCS One of the core insights in our formalization is that the app user does not (cannot) observe PCS per session: messaging apps invisibly merge different sessions with the same peer into a single displayed conversation. This is also ultimately the reason why the PCS properties proven for the DR do not seem to hold at the session-handling layer. Instead, at the session-handling layer or higher, we need to use a different technical definition of PCS that considers the conversation between two agents instead of sessions. This effectively encodes that from the user's point of view, multiple sessions are invisibly merged into a single conversation. Indeed, in all our experiments, the classical per session PCS holds, but the intuitive notion of PCS over the full conversation shown to the user at the higher level does not. To express this new goal, we drop the session identifier from the send and heal event. If there is any healing phase during the conversation, that is, over any of the involved sessions, then any message sent afterwards should be secret unless there is a new compromise. This yields the following instantiation of conversation-based PCS:

$$\begin{array}{l} \forall\, A, B, \mathsf{sck}, i, j, k. \\ \quad \texttt{Sent}(A, B, \mathsf{sck})@i\ \&\ \texttt{K}(\mathsf{sck})@j\ \& \\ \quad \texttt{Heal}(A, B)@k\ \&\ k < i \\ \quad\quad \Rightarrow \left(\begin{array}{l} \quad \exists\, l.\ \texttt{Compromise}(A,B)@l\ \&\ k < l \\ \|\ \exists\, l.\ \texttt{Compromise}(B,A)@l\ \&\ k < l \end{array} \right) \end{array}$$

While at first glance this property may seem simpler than the session-based property, its proof obligation is in fact more complex: because the sending and healing are no longer bound to the same session, this property depends on the scope of healing in relation to messages sent in *any* session of the sender. As we will see later, our proof of conversation-based PCS requires us to first prove that session-based PCS holds.

In addition to these two PCS properties, we also model the forward secrecy (FS) properties, which essentially state that the attacker can only learn messages after it compromised a session. FS is in fact often an intermediate step towards proving PCS. We only discuss the more interesting PCS proofs in the following discussions but note that we in all cases also proved the corresponding FS property.

5.3 Single session PCS

We first prove that if we enforce that two parties always use the same session, PCS is obtained. While this is not a realistic assumption, this is an intermediate step over which we build the proofs for the real use cases. In this case, session-based PCS and conversation-based PCS collapse.

Proof To perform the PCS proof, it is necessary to answer the question: how could the attacker learn some key sck. From an intuitive level, the following scenarios are possible:

- the current state of sender or receiver was compromised,
- the sender or the receiver was compromised previously, but still on the same chain, and the attacker just has to ratchet forward the key to compute sck,
- the sender or the receiver was compromised even before the last asymmetric ratchet, the attacker then became active, performed an asymmetric ratchet with the sender, and since then the sender is in fact talking to the attacker.

Mechanism	Source	Threat Model	PCS	Clone Detection
Double Ratchet	[1]	$\mathcal{A}_{\text{Formal}}$	✓ [1, 4, 9, 11]	Implicit
Old Sesame	v4.47.0	$\mathcal{A}_{\text{Experiment}}$	✓ [13]	✗
Sesame (currently deployed)	v5.40.4	$\mathcal{A}_{\text{Signal}}$	✓	✗
		$\mathcal{A}_{\text{Experiment}}$	✗ [Section 3, Appendix A.1]	✗
		$\mathcal{A}_{\text{Formal}}$	✗ [Section 5, TAMARIN]	✗
Sesame with sequential sessions	This work	$\mathcal{A}_{\text{Formal}}$	✓	Implicit
Sesame, sequential sessions + warning message	This work	$\mathcal{A}_{\text{Formal}}$	✓	✓

Table 1: Security Analysis Summary
Summary of results from our experiments and TAMARIN analysis. Our results cover a single session Double Ratchet, the older version Sesame studied in a previous work, the currently deployed Sesame mechanism, and our two proposed incremental fixes.

Mechanism	Source	Property	Result	Run Time (s)	Helper Lemmas
Double Ratchet (Sesame with single session)	[1]	PCS	✓	25	24
Sesame (currently deployed)	v5.40.4	PCS	✗	2	0
Sesame with sequential sessions	This work	PCS	✓	36	27
Sesame, sequential sessions + warning message	This work	Clone Detection	✓	31	23

Table 2: TAMARIN Formal Analysis Summary
Results obtained with the TAMARIN prover. Our proofs are obtained by using a so-called oracle to guide TAMARIN's proof search, as well as some manually stored proof. The run times are given for TAMARIN to find the proof automatically and to verify the manual proofs for each lemma. When running the models we did not include the supplementary proofs, such as forward secrecy. The models were run on a Intel(R) Xeon(R) CPU E5-4650L 2.60GHz server with 756GB of RAM, and 4 threads per TAMARIN call. We also provide the number of helper lemmas needed to prove the property.

A core difficulty of the proof is that we model an unbounded number of asymmetric or symmetric ratchet steps for each session. An attacker may know a sending key because it knew a previous sending key, or a previous root key, and so on. While the PCS property concerns the sent messages and the sender chain keys, we see here that the property is in fact interdependent between receiver and sender keys. We must then consider all the possible keys the attacker can learn or compute instead of only considering the sender chain key, and reason by induction. Namely, with respect to PCS, we must consider that the attacker may know a sending key (case 1), a receiver key (case 2), or a root key (case 3), each of them being possible either because another case happened previously, or because there was a compromise.

This is in fact a generic *observation* relative to Tamarin proofs. While we want to prove by induction a formula $P_1 \Rightarrow Q$, we see that we in fact also need to prove $P_2 \Rightarrow Q$ and $P_3 \Rightarrow Q$. However, we crucially cannot prove each of those formulas separately due to their interdependence, and we must prove instead in one big induction $\bigwedge_{1 \leq i \leq 3}(P_i \Rightarrow Q)$, or equivalently $(P_1 \vee P_2 \vee P_3) \Rightarrow Q$.

In our case, we have to prove the following property, corresponding to a strengthening of the original PCS property:

$$\begin{array}{ll}
\forall A, B, \mathsf{rk}, \mathsf{rck}, \mathsf{sck}, i, j, k. & \\
\quad \big(\big(\mathtt{Sent}(A,B,\mathsf{sck})@i \;\&\; \mathtt{K}(\mathsf{sck})@j \;\& & \\
\quad \;\mathtt{Heal}(A,B)@k \;\&\; k < i\big) & \textit{case}1 \\
\quad \vee & \\
\quad \big(\,\mathtt{Root}(A,B,\mathsf{rk})@i \;\&\; \mathtt{K}(\mathsf{rk})@j \;\& & \\
\quad \;\mathtt{Heal}(A,B)@k \;\&\; k < i\big) & \textit{case}2 \\
\quad \vee & \\
\quad \big(\,\mathtt{Received}(A,B,\mathsf{rck})@i \;\&\; \mathtt{K}(\mathsf{rck})@j \;\& & \\
\quad \;\mathtt{Heal}(A,B)@k \;\&\; k < i\big)\,\big) & \textit{case}3 \\
\Rightarrow \left(\begin{array}{l} \;\;\exists\, l.\ \mathtt{Compromise}(A,B)@l \;\&\; k < l \\ \vee \exists\, l.\ \mathtt{Compromise}(B,A)@l \;\&\; k < l \end{array}\right) & \textit{implied}
\end{array}$$

Proving this strengthened induction proved to be too difficult for TAMARIN's current automated heuristics, and we had to perform it in the interactive mode. However, we relied on TAMARIN automation to prove many structural properties over the protocol, without which we would have not been able to complete the proof. To give an example of such a structural property, consider the following formula:

$$\begin{array}{l} \forall A_1, B_1, A_2, B_2, \mathsf{rck}, i, j, k. \\ \quad \mathtt{Received}(A_1, B_1, \mathsf{rck})@i \ \&\ \mathtt{Sent}(B_2, A_2, \mathsf{rck})@j \\ \quad\quad \Rightarrow A_1 = A_2 \ \&\ B_1 = B_2 \end{array}$$

This formula states that if a receiver and a sender are both agreeing on some secret key rck , then they are necessarily talking to each other, i.e., $A_1 = A_2$ and $B_1 = B_2$. This is for instance needed when trying to prove the previous induction, in the reasoning how the attacker can obtain the receiving chain rck (case 3). Proving this lemma helps TAMARIN in deducing that compromise of any other party, other than A or B, does not result in knowing rck, since the receiving chain key is unique per pair of users.

Overall, the proof required 24 such structural helper lemmas, which are proved automatically by TAMARIN with an oracle. The strengthened PCS with 718 proof steps, had to be selected manually out of the available proof steps.

5.4 Conversation-based PCS

We model the current design of Sesame by allowing in the previous model any number of sessions between A and B to run in parallel. In this model, we first describe how TAMARIN rediscovers the scenario trace corresponding to Section 3.2. Then, we show how we model our fix and prove it secure.

Conversation-based PCS violation We observe on this model that if we consider the full conversation between two parties, then PCS is lost. This suggests that our model is general enough to capture the experimentally found scenario. However, we proved previously that for a single session, we have PCS between two agents. The session-based PCS property still holds for this new model with multiple sessions, and is essentially a proof that multiple sessions of the DR do not share any cryptographic material and are independent. Interestingly, we could in this setting mostly reuse the proof made in the single session case to prove the session-based PCS.

Modeling the fix To introduce a notion of active and inactive session, we specify that whenever a new session is created for a party, all sending actions and asymmetric ratcheting that would use a previous session are forbidden. To do so, we rely on so-called restrictions: they are expressed in TAMARIN in the same language as the security properties, but are used to forbid certain executions.

For instance, we can model that we have an event $\mathtt{NewSession}(A, \mathsf{sid})$ raised by party A whenever they create a new session with identifier sid, and that in addition to the $\mathtt{Sent}$ event there is also a $\mathtt{SentSID}(\mathsf{sid})$ event raised whenever A uses sid to send a message. Then, a core feature of our fix is modeled by adding the following restriction:

$$\begin{array}{l} \neg(\exists A, \mathsf{sid}_1, \mathsf{sid}_2, i, j, k. \mathtt{NewSession}(A, \mathsf{sid}_1)@i \ \& \\ \mathtt{NewSession}(A, \mathsf{sid}_2)@j \ \&\ \mathtt{SentSID}(\mathsf{sid}_1)@k \ \&\ i < j < k) \end{array}$$

This restriction expresses that for any execution, A cannot send a message from session sid_1, once it has created a new session sid_2. This ensures the deprecation of an old session sid_1, once it is replaced by newer one sid_2.

Proof To prove the conversation-based PCS between A and B, we then rely on both the previous per session proof of PCS and the restrictions modeling our fix. Essentially, if our fix is correct, the restrictions will forbid all attacks arising from lifting the per session PCS to the conversation-based PCS property. We were indeed able to carry out this proof in TAMARIN. This proof required 4 additional structural helper lemmas, and the conversation-based PCS took 146 steps by reusing all the previous proofs.

5.5 Proof for the clone detection mechanism

With the first fix, we restore the conversation PCS against the $\mathcal{A}_{\text{Formal}}$ threat model. However, to also add some security in the case of $\mathcal{A}_{\text{Experiment}}$, we also model a clone detection mechanism that would detect whenever an attacker is initiating new sessions with a compromise identity key.

Modeling the mechanism To model clone detection, we enrich the state of the sessions with the warning and identifier root keys, as well as the current session identifier, and modify the receive message action so that whenever a message is received over a deprecated session, it sends out the two corresponding root key warnings.

We then added the **warning process** action to the protocol, where a party can receive over its current active session a warning. This rule executed by agent A decrypts a warning message of the usage of a session with root key identifier $\mathtt{pk}_{ro}$ and then raises the event $\mathtt{Report}(A, \mathtt{pk}_{ro})$. By having each root key creation also raise the event $\mathtt{Rk}(A, \mathtt{rk})$, we express the actual processing of the report by adding the restriction:

$$\begin{aligned}&\forall A, rk, i.\ \mathtt{Report}(A, \mathtt{pk}(rk))@i\\ &\qquad \Rightarrow\ \neg(\exists\ j.\ \mathtt{Rk}(A, rk)@j\ \&\ j < i)\end{aligned}$$

This enforces the soundness of clone detection: A reports an error for some root key identifier only if the corresponding root key was never computed by A.

Proof Our goal is to prove that the attacker cannot trigger a false positive of our mechanisms, that is, a report can only be raised if there was a compromise of some agent. An interesting point of the PCS proof is that the attacker is able to compute a root key of A only if it compromised B or A. So, any warning message received by A is either:

- from a honest conversation, in which case A will recognize the root key,
- from the attacker, in which case there must have been a compromise.

This mean that proving the clone detection mechanism is equivalent to proving the following two cases: a) honest warning messages do not trigger the error message, and b) the attacker needs to compromise one of the agent to compute a valid warning message. The first case is straightforward, and the second has mostly been done in the previous PCS proofs.

Following this idea, we were able to prove that our proposed mechanism is sound. The proof effort involved the previous PCS proof, two new helper lemmas proved in 429 steps and 14 steps respectively, and the final proof done in 115 steps.

6 Practicality of fixes

Memory usage A downside of our clone detection mechanism is that it requires devices to store a long list of warning root keys in order to correctly report compromises without false positives (soundness of the clone detection). However, in the full compromise setting, it can be argued that our clone detection mechanism is stronger than needed: our PCS fix quickly heals all honestly established sessions, and we may not care about detecting low impact compromise of those sessions. If we only care about detecting the case where the attacker is using the compromised identity key to initiate a fresh session, our clone detection mechanism can be adapted so that instead of deriving warning keys from each root key, we only derive warning keys from the first root key of the session. This solution would then be such that the memory usage is strongly reduced and cloned sessions are detected.

Device Reset There are cases where we reset the state of a device to a clean blank state or to a backup, for instance after the original device was lost or there was a state loss. In such cases, our clone detection mechanism could yield false reports, either after restoring a backup, where the user may receive a report over a root key computed after the point of the backup, or after a full reset, where the user may receive a report over a root key corresponding to the older identity key.

To mitigate those false reports, a simple solution is to specify that a device that performed a backup restore or a reset should ignore any clone detection reports it obtains from a partner that just came online for the first time since the reset. However, this may lead to an attacker exploiting this small time-frame to avoid detection, and this mechanism may still let some false positive report be produced in some edge cases.

Following up on clone detection We do not tackle in our work what should happen after a clone detection warning. However, because our clone detection is sound, this only happens of the attacker compromised a long term identity key. To restore trust, the user needs to trigger a full reset on the server side, erasing all identity keys and bundles, and uploading new identity keys. Since our detection mechanism is sound, i.e., lack of false detection, apps can automatically trigger a mechanism to refresh the secrets and notify the victim appropriately or ask for their input in edge cases. Another idea would be to rely on an Out-Of-Band channel, and leave it to the user to activate the recovery. However, further security and usability studies are needed on the best mechanism to ensure a secure key replacement and clear communication to the end user.

Implementability of our suggestions An open question is whether our proposed solutions may introduce unexpected behaviors that could break the correct execution of Signal. Our first improvement only restricts the set of possible behaviors from the protocols. In theory, this could lead to desynchronization if the head sessions between two devices do not match. This would never block communication, but a different session would be used for sending and one for receiving, and each would not be able to ratchet asymmetrically. If such a situation were to occur, it would be automatically resolved by the session refresh currently happening every hour. Our second fix adds new message exchanges to the usual flow. According to our Tamarin models, the warning can only be triggered if a compromise did happen, and thus should not lead to any additional threat, and cannot be used, e.g., to trigger a denial-of-service. Importantly, while our models did not raise any unexpected behaviors, issues could come from unexpected places from the complex Signal ecosystem: testing deployments in real life conditions could help decide this.

7 Conclusion

Our experimental result based on multiple PCS violations on the application level illustrated the need to consider PCS not at the per-session level but at the conversation level.

We introduced the first formal model that includes the multiple session management layer of a secure messaging protocol, taking Signal as a first case study. We used this model to show how at the conversation level PCS is violated by design choices in Sesame, and use our model to prove the correctness of our two proposed improvements.

As a future direction w.r.t. Signal, our models could be more precise, covering more high-level features like the addition of new devices, or more low level features like a more precise DR. It is however likely that this can only be achieved by improving existing verification tools to make such more detailed models tractable. It would also be valuable to obtain computational guarantees that do consider the conversation-based PCS. Furthermore, it would be interesting to apply the conversation-based PCS approach to other end-to-end messaging apps. The closest one is Whatsapp. It would however require significant reverse-engineering efforts: it relies on libsignal but with code that is not open source, making it difficult to analyze and notably extract how sessions are managed.

Responsible Disclosure Since Signal's documentation only considers an attacker that is passive after a compromise (which could be satisfied with a much simpler protocol), our PCS violations are outside of its stated threat model. Nevertheless, we contacted the original author of the Sesame specification with our observations and fix, and they contacted the Signal development team.

References

[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] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. *DY**: A modular symbolic verification framework for executable crypto-

graphic protocol code. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 523–542. IEEE, 2021.

[4] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A more complete analysis of the Signal Double Ratchet algorithm. In *Annual International Cryptology Conference*. Springer, 2022.

[5] Colin Boyd, Anish Mathuria, and Douglas Stebila. *Protocols for authentication and key establishment*, volume 1. Springer, 2003.

[6] Jacqueline Brendel, Rune Fiedler, Felix Günther, Christian Janson, and Douglas Stebila. Post-quantum asynchronous deniable key exchange and the Signal handshake. In *IACR International Conference on Public-Key Cryptography*, pages 3–34. Springer, 2022.

[7] Jacqueline Brendel, Marc Fischlin, Felix Günther, Christian Janson, and Douglas Stebila. Towards post-quantum security for Signal's X3DH handshake. In *International Conference on Selected Areas in Cryptography*, pages 404–430. Springer, 2020.

[8] Sébastien Campion, Julien Devigne, Céline Duguey, and Pierre-Alain Fouque. Multi-device for Signal. In *International Conference on Applied Cryptography and Network Security*, pages 167–187. Springer, 2020.

[9] Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. Universally composable End-to-End secure messaging. In *Annual International Cryptology Conference*. Springer, 2022.

[10] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A Formal Security Analysis of the Signal Messaging Protocol. In *EuroS&P*, pages 451–466. IEEE, 2017.

[11] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, 2020.

[12] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178. IEEE, 2016.

[13] Cas Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. Clone detection in secure messaging: improving Post-Compromise Security in practice. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1481–1495, 2020.

[14] Cas Cremers, Charlie Jacomme, and Aurora Naska. Models and proofs. `https://github.com/sesame-symbolic-model/sesame-model`, 2022.

[15] David Derler, Kai Gellert, Tibor Jager, Daniel Slamanig, and Christoph Striecks. Bloom filter encryption and applications to efficient forward-secret 0-rtt key exchange. *Journal of Cryptology*, 34(2):1–59, 2021.

[16] Samuel Dobson and Steven D Galbraith. Post-quantum Signal key agreement with SIDH. *Cryptology ePrint Archive*, 2021.

[17] Mohammad Sadeq Dousti and Rasool Jalili. Forsakes: A forward-secure authenticated key exchange protocol based on symmetric key-evolving schemes. *Cryptology ePrint Archive*, 2014.

[18] Benjamin Dowling, Felix Günther, and Alexandre Poirrier. Continuous authentication in secure messaging. In *European Symposium on Research in Computer Security*, pages 361–381. Springer, 2022.

[19] F Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In *International Workshop on Security*, pages 343–362. Springer, 2019.

[20] Matthew D Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *2015 IEEE Symposium on Security and Privacy*, pages 305–320. IEEE, 2015.

[21] Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 0-RTT key exchange with full forward secrecy. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 519–548. Springer, 2017.

[22] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. An efficient and generic construction for Signal's handshake (X3DH): post-quantum, state leakage secure, and deniable. *Journal of Cryptology*, 35(3):1–78, 2022.

[23] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: the safety of messaging. In *Annual International Cryptology Conference*, pages 33–62. Springer, 2018.

[24] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: almost-optimal guarantees for secure messaging. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 159–188. Springer, 2019.

[25] 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.

[26] Moxie Marlinspike and Trevor Perrin. The X3DH key agreement protocol. *Open Whisper Systems*, 2016.

[27] Moxie Marlinspike and Trevor Perrin. The Sesame algorithm: session management for asynchronous message encryption. *Revision*, 2:2017–04, 2017.

[28] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *International conference on computer aided verification*, pages 696–701. Springer, 2013.

[29] Kevin Milner, Cas Cremers, Jiangshan Yu, and Mark Ryan. Automatically detecting the misuse of secrets: Foundations, design principles, and applications. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 203–216. IEEE, 2017.

[30] Trevor Perrin and Moxie Marlinspike. The Double Ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/, 2016. accessed: 2022-08-18.

[31] Jan Wichelmann, Sebastian Berndt, Claudius Pott, and Thomas Eisenbarth. Help, my Signal has bad device! In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–105. Springer, 2021.

[32] Jiangshan Yu, Mark Ryan, and Cas Cremers. Decim: Detecting endpoint compromise in messaging. *IEEE Transactions on Information Forensics and Security*, 13(1):106–118, 2017.

A Appendix

A.1 Additional PCS violation

We describe here the PCS violation we found that is independent of the multiple sessions layer, and was independently discovered by [18].

This scenario also considers if PCS is achieved in practice by the Signal app. In previous work [13], its authors showed that Signal (app version v4.47.7) was able to protect past messages and lock an attacker out of the conversation. The experiment considers A and B having a conversation, until A is cloned. The honest parties continue the conversation, and once the clone goes online, the experiment checks if the clone receives past messages or can inject new ones in the conversation. We performed a slight variation of their experiment, where in addition the parties go offline in order to prevent any other mechanisms from being triggered. Thus, we focus solely on PCS, and disregard any features that may also come into play, such as message re-encryption.

The main goal of the experiment is to check whether a fully compromised user can recover security of a session and prevents its clone from injecting their own messages. This should be achieved by the DR protocol, since exchange of messages with the partner introduces new entropy in the key derivation. In particular, once

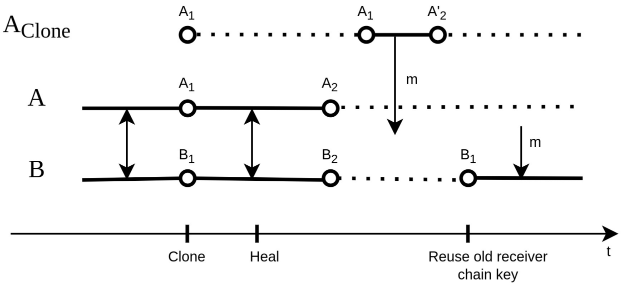

Figure 5: Abusing stored receiver chain keys on Signal. In the first phase, honest parties A and B exchange messages until A is cloned. The clone goes offline (denoted by the dotted line) and the parties exchange messages to heal their session. A and B go offline, and the clone encrypts a message m with the old session state A_1. When, B goes online they can decrypt the message by using the old receiver chain stored at state B_1.

the session keys (root, sender, and receiver chain) are ratcheted forward, the attacker should not be able to recompute them.

Although we expected the same experiment results as in [13], we noticed an unexpected decrease in the security of Signal. On the positive side, the clone is not able to decrypt past messages exchanged between the honest parties during the healing phase i.e., when the clone is offline. However, future communication between the clone and the honest party are not blocked anymore: the clone can send messages that the partner is able to decrypt without any error or notification shown in the user interface.

Concrete steps The attack of Fig. 5 is obtained when:

1. A and B exchange 4 messages: $(A \leftrightarrow B)^4$.
2. Duplicate A's machine and send clone offline.
3. A and B exchange 4 more messages: $(A \leftrightarrow B)^4$.
4. A and B go offline, and clone goes online.
5. Clone sends 4 consecutive messages to B $(A_{Clone} \rightarrow B)^4$
6. Clone goes offline, and B online. B can decrypt the messages sent by the clone.

Analysis The intuition behind the scenario is that B can forward a key from a past state, which they should have deleted. Notice that the clone follows the protocol acting as an honest user, therefore using only the local state to compute the next message key. Moreover, since A and B's devices are offline at the time, the clone has no external input on the progress of the conversation. On the receiver side, the fact that B is able to compute the session key means they can go back to a previous receiver chain and compute the next message key.

Checking the debug logs, we found that the chain key being used to encrypt the clone's messages was the same as the one A had used to encrypt their messages at the time of cloning. In addition, we cross-checked with the implementation of Signal and found that the scenario is indeed possible. In fact, Signal allows for the parties to store up to 5 receiver chain keys at any time [8]. In theory, it should be impossible to extend those receiver chains, as only the skipped messages before the corresponding asymmetric ratchet should have been stored. However, the compromised key is still stored in the receiver's state, and the clone can extend older chains and use this to send any number of messages impersonating A. This behavior contradicts PCS, as after the asymmetric ratchet the old compromised material should be useless except for skipped messages.

A.2 Security Parameters in Signal

Signal is deployed with a specific set of parameters that may directly affect the security of the protocol:

1. 25000 forward jumps - Signal allows for the message keys to be forwarded 25000 steps ahead and stored locally. A message with any counter smaller than the parameter results in the partner storing all keys up to the message counter. This means that even if the honest parties heal from the compromise, the attacker can still encrypt messages using the skipped message keys.
2. Reset every 1 hour - The parties perform a new handshake every one hour.

[8] Signal's stored receiver chains: link.

3. 5 stored chains - The 5 latest receiver and sender chains are stored. This allows an attacker to extend any of these chains with an arbitrary number of new messages.
4. 40 sessions stored - Signal stores 40 sessions per device. A compromise in any of them allows the attacker to continue undetected the conversation in that session.

A.3 Formal details of the model

We provide here more details on our formal models and refer to [14] for the actual models. TAMARIN uses multi-set rewriting rules (MSR) to model protocols. Such rules, given a set of currently available facts, consume them to produce new facts. Each fact can be seen as a local state, and the global state of a protocol can then be modelled through a set of facts. A protocol step is modelled with one MSR, and TAMARIN relies on some builtins facts to correctly model protocols:

- the $\mathtt{Fr}(n)$ fact must only appear on the left-hand-side of a MSR, and is used to load a new fresh nonce n;
- the $\mathtt{In}(x)$ fact can be used on the left-hand-side to model an input from the attacker-controlled network;
- the $\mathtt{Out}(t)$ fact on the right-hand-side models an output to the attacker-controlled network.

Our protocol models rely on two main fact, one to model the state of a current sending chain, and one to model the state of a receiving chain. The fact $\mathtt{SndCK}(rk, sidA, A, B, ck)$ tells that agent A currently has a session with agent B with a local thread identifier $sidA$ on A's side, the current root key is rk and the chain key is ck. In a correct execution, it is expected that on B side there is a corresponding receiver chain, stored in the fact $!\mathtt{RcvCK}(sidB, rk, B, A, ck)$. Remark that the receiving fact is prefixed by the ! symbol: this denotes that this fact is persistent, and an MSR rule will not consume it.

If two agent identities $\mathtt{Id}(A)$ and $\mathtt{Id}(B)$ were created in an initialisation, this rule creates a new session for two users:

$$\begin{gathered}[\mathtt{Id}(A), \mathtt{Id}(B), \mathtt{Fr}(rk), \mathtt{Fr}(sidA), \mathtt{Fr}(sidB)] \\ -[\mathtt{NewSession}(A, sidA), \mathtt{NewSession}(B, sidB)]\rightarrow \\ [\mathtt{SndCK}(rk, sidA, A, B, h(rk)), !\mathtt{RcvCK}(sidB, rk, B, A, h(rk))]\end{gathered}$$

Here, the first line is the left-hand-side of the MSR that samples the fresh values needed to instantiate the session, the middle parts corresponds to raising the events used in Section 5 to specify security properties, and the last line is the right-hand-side effectively creating the two facts corresponding to the state of A and B. Such a rule is an abstraction of the full X3DH protocol, where we simply say that two identities can suddenly share a valid fresh root key.

We then use a rule to model the sending of a message, which will only rely on a single agent here as the communication is over the network.

$$\begin{gathered}[\mathtt{SndCK}(rk, sidA, A, B, ck), \mathtt{Fr}(m)]-[\mathtt{Sent}(A, B, ck)]\rightarrow \\ [\mathtt{SndCK}(rk, sidA, A, B, h(ck)), \mathtt{Out}(senc(m, ck))]\end{gathered}$$

This rule given the current state of a sending chain sends a fresh payload m encrypted with the current chain key ck. In parallel, it raises an event about the sending of the message, and in addition produce the new state of the chain where the chain key has been ratcheted forward.

On the receive side, the situation is a bit more complex due to skipped messages. A rule directly allows the receiver to move forward in the chain, and another one to receive a message (this is why we prefix receiving facts with !):

$$[!\mathtt{RcvCK}(sidB, rk, B, A, ck)] \rightarrow [!\mathtt{RcvCK}(sidB, rk, B, A, h(ck))]$$

$$\begin{gathered}[!\mathtt{RcvCK}(sidB, rk, B, A, ck), \mathtt{In}(enc(m, ck))] \\ -[\mathtt{Received}(B, A, ck)]\rightarrow []\end{gathered}$$

We need to add in our models a restriction forbidding that the second rule can be triggered twice:

$$\begin{aligned}\forall A, B, \mathsf{rck}, i, j.\ &\mathtt{Received}(A, B, \mathsf{rck})@i \\ &\& \ \mathtt{Received}(A, B, \mathsf{rck})@j \Rightarrow i = j\end{aligned}$$

Possible compromises must be specified explicitly through a rule, for instance with:

$$[\mathtt{SndCK}(rk, sidA, A, B, ck)]-[\mathtt{Compromise}(A, B)]\rightarrow[\mathtt{Out}(ck)]$$

A similar rule on the receiver side models its own compromises. The last rule to model a high-level double-ratchet process is the asymmetric step. In a single rule, two agents can perform an asymmetric step and obtain a new fresh root key *nk*. The sending and receiver roles are then swapped:

$$\begin{gathered}[\mathtt{SndCK}(rk, sidA, A, B, ck), \mathtt{Fr}(nk), !\mathtt{RcvCK}(sidB, rk, B, A, ck)] \\ -[\mathtt{Heal}(A, B, sidA), \mathtt{Heal}(B, A, sidB)]\!\!\rightarrow \\ [\mathtt{SndCK}(nk, sidB, B, A, h(nk)), !\mathtt{RcvCK}(sidA, nk, A, B, h(nk))]\end{gathered}$$

Together, these rules abstract a double-ratchet with skipped messages and multiple sessions. This high-level presentation omits several details of our models, e.g.,:

- many events are added to allow writing helping lemmas;
- bookkeeping facts are also added to ease reasoning;
- more rules are needed to model that an attacker can, e.g., perform an asymmetric ratchet with a user if it knows the corresponding chain-key;
- some additional restrictions to remove undesirable behaviors, such as an execution where an agent tries to initiate a session with themselves.