Impossibility Results for Post-Compromise Security in Real-World Communication Systems

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

Niklas Medinger
CISPA Helmholtz Center for Information Security, Saarland University
niklas.medinger@cispa.de

Aurora Naska
CISPA Helmholtz Center for Information Security, Saarland University
aurora.naska@cispa.de

November 18, 2024 – Version 1.0

Abstract

Modern secure communication systems, such as iMessage, WhatsApp, and Signal include intricate mechanisms that aim to achieve very strong security properties. These mechanisms typically involve continuously merging in new fresh secrets into the keying material, which is used to encrypt messages during communications. In the literature, these mechanisms have been proven to achieve forms of Post-Compromise Security (PCS): the ability to provide communication security even if the full state of a party was compromised some time in the past. However, recent work has shown these proofs do not transfer to the end-user level, possibly because of usability concerns. This has raised the question of whether end-users can actually obtain PCS or not, and under which conditions.

Here we show and formally prove that communication systems that need to be resilient against certain types of state loss (which can occur in practice) fundamentally cannot achieve full PCS for end-users. Whereas previous work showed that the Signal messenger did not achieve this with its current session-management layer, we isolate the exact conditions that cause this failure, and why this cannot be simply solved in communication systems by implementing a different session-management layer or an entirely different protocol. Moreover, we clarify the trade-off of the maximum number of sessions between two users (40 in Signal) in terms of failure-resilience versus security.

Our results have direct consequences for the design of future secure communication systems, and could motivate either the simplification of redundant mechanisms, or the improvement of session-management designs to provide better security trade-offs with respect to state loss/failure tolerance.

1 Introduction

Modern communication systems include intricate mechanisms that aim to achieve very strong security properties. One of these properties is *Post-Compromise Security (PCS)* [16]: even after the entire state of a device is compromised at some point by an adversary, future communications can still obtain meaningful security properties (esp. against active adversaries). The high-level intuition is that this can be achieved by continuously merging fresh secrets into the keys used for message encryption, in such a way that if the adversary is passive for a short time, it becomes locked out of the communications again, effectively *healing* the communication.

Many globally used communication systems, like Apple iMessage, WhatsApp [43], Signal messenger [37], Google Messages [24], and Element [39] use intricate mechanisms to implement such a continuous merging in of fresh secrets. The canonical mechanism to implement this was introduced by Signal and is called the Double Ratchet [31, 35] (and notably its sub-component the asymmetric ratchet). In the literature, the Double Ratchet and related PCS mechanisms have been studied extensively, e.g., [3, 9, 11, 12, 15, 17, 18, 25, 26, 29, 35, 38]. The majority of these works prove that for some protocol, if a healing step occurs after a compromise – a key refresh where the adversary is temporarily passive – then security is recovered, thus satisfying PCS.

However, recent work [20] showed that this PCS guarantee does not necessarily lift to the application level. For example, for the Signal messenger, the underlying Double Ratchet can be proven to provide PCS. However, end users do not obtain a similar PCS guarantee, because there is a mismatch between the single stream of messages that users see, and the underlying set of up to 40 Double-Ratchet sessions that underlies this stream. In reality, the Signal messenger allows any adversary that has the long-term secrets of A to start a new Double Ratchet session with B, even if A and B previously healed all their ongoing sessions. From a user's perspective, this means that their single stream of messages can never heal, and they thus do not obtain PCS. This user-level PCS guarantee was coined conversation-based PCS in [20], which we refer to as *conversation PCS* . The PCS failure is not caused by a bug in Signal, but is a design choice by its designers: for a number of reasons, an end user might lose (parts of) their latest local state: to enable such a user to still communicate with its partners, the Signal messenger allows creating a new Double Ratchet session. As a side-effect, an adversary may also exploit this possibility to start a new session without knowing the latest local state, thereby undoing any positive effects from healing.

This has led to the following open question: *can real-world communication systems actually achieve full conversation PCS or is this fundamentally impossible, and under which conditions?*

In this work we solve this question and show that communication systems that need to be resilient against certain types of state loss, fundamentally cannot achieve full conversation PCS. Our results imply that real-world messengers like Signal and others, which are deployed on hardware platforms where memory might be slightly unreliable (e.g., bit flips) or operating-system level backups might be used, cannot hope to achieve the full PCS guarantees.

However, this does not mean that we cannot do better in practice. Already in [20] it was shown that the Signal messenger could obtain better guarantees after compromise by slightly modifying its session-management layer, which determines when and how different (possibly concurrent) Double Ratchet sessions are initiated between two parties A and B. In this work, we go beyond this protocol-specific result, and explore several parts of the design space for generic communicating systems that aim for PCS-like guarantees. Notably, we show which session-management policies do not help towards better PCS guarantees, and which session handling policy offers stronger PCS guarantees than currently deployed protocols. Moreover, we demystify the effects of the maximum number of concurrent sessions in such protocols, enabling protocol designers to make more informed choices.

Contributions. Our main contributions are:

1. We show, and formally prove, that full PCS for an end-user is fundamentally unachievable in communication systems that need to be resilient against certain forms of state loss, which includes all real-world messaging apps.
2. We explore the design space of these communication systems, and show how real-world requirements lead to violation of conversation PCS for both deployed and proposed solutions.
3. We generalize and extend previous Signal-specific findings to arbitrary communication systems, and show how to instantiate a session-management layer with a stronger PCS guarantee than what is currently achieved in deployed systems. We clarify the impact of the choice of the parameter that controls the maximum number of concurrent sessions between two users.

We prove our formal statements using the Tamarin prover: Our formal models and results are available for reproducibility, and extensions in [21]

Outline. We provide background on the PCS property and communication systems in Section 2. We explore the design space, define the problem and provide intuition on our statements in Section 3. Then, in Section 4 we present our formal models and analysis on the impossibility results. In Section 5, we describe our proposed solution and its formal analysis. Lastly, we discuss interesting insights and future work in Section 6, and we end with our conclusions in Section 7.

2 Background

2.1 Post-Compromise Security (PCS)

Post-Compromise Security (PCS) is a guarantee that ensures the security of future communication following the compromise of all secrets of a device. The messages exchanged between the honest parties become confidential again, after the so-called "healing", a period during which the attacker remains passive and does not interfere with the honest parties exchanging messages [16]. Typically, the healing serves to refresh their secrets by incorporating untempered entropy, e.g., Diffie-Hellman ephemeral secrets, and remove the attacker from the conversation.

PCS was initially defined in two versions: a) Weak-PCS, where the attacker has access to the long-term secrets only during the compromise period, and b) Full-PCS, where the attacker learns the long-term secrets of the victim. In later work [20], PCS was defined in the context of secure messaging in terms of the level of security it achieves: a) session-level PCS, where messages exchanged in a session are secure after the session secrets are compromised, and b) conversation-level PCS, where the messages exchanged in any session of the conversation are secure, after a single session is compromised. [13] defines PCS with a single-session specific metric to enable the comparison of the achieved security on different protocols and adversaries. The property has seen an extensive line of research, most prominently in secure messaging [3, 9, 11, 12, 15, 17–19, 25, 26].

2.2 Communication Systems

In this section, we give an overview on deployed communication systems, the properties they aim to provide, and how they manage concurrent sessions.

Underlying Protocols. The Signal protocol, composed by the X3DH protocol [31] and the Double Ratchet [35], is one of the most widely deployed messaging protocols. It is used by apps like the Signal messenger [37], WhatsApp [43], Google Messages [24], and many more. The protocol is constructed such that after an authenticated key exchange to bootstrap the

communication, the users derive a shared symmetric key. From the shared key, they derive the messaging keys using the Double Ratchet [35]. The main idea of the algorithm, it to maintain a chain of keys and update it by introducing new entropy–ephemeral Diffie-Hellman keys–every time the users exchange messages going back and forth. The intricate mechanism aims to achieve for the users forward secrecy and post-compromise security [3, 15].

Matrix [39] is another messaging protocol constructed by two algorithms: Olm and Megolm. Olm [41] is constructed by the Triple Diffie-Hellman protocol [32] and the Double Ratchet [35]. Olm is used to send relevant information for establishing a Megolm channel. Then, to actually encrypt messages and move forward the state of the users the protocol uses the Megolm Ratchet [40]. Users maintain two uni-directional channel (effectively sessions), one for sending and one for receiving. The protocol provides weak forward secrecy, and it aims to achieve some version of PCS by rotating its sessions.

Other protocols include Apples's iMessage PQ3 protocol, IETF's Messaging Layer Security (MLS) protocol [7], and Wire's Proteus Protocol [44] which is an implementation of the Signal protocol.

Properties. Some basic security guarantees of communication protocols include *authentication*, *confidentiality*, and *integrity*. Stronger guarantees talk about how the communication protocols react in case of a compromise. *Forward secrecy* states that messages exchanged before a compromise are secure. *Post-compromise security* states that messages exchanged after a compromise are secure after healing. Other properties, related to how to protocol reacts in case of lost or out-of-order messages are a) *immediate decryption*, messages are decrypted upon being received and placed in their correct order in the conversation, and b) *message-loss resilience*, future messages can still be decrypted even if some messages are lost by the network.

Managing Concurrent Sessions. As we will see in this work, communication systems need to maintain a management layer that handles the concurrent underlying connections of the protocols. Such concurrent connections are typically called sessions. The session-management layer then is in charge of the creation, storage, update, and deletion of these session. The choice on how to handle each one of these operations throughout the usage of the communication system by the end-user, is defined by the system.

A prominent session-management layer is the Sesame algorithm [30], which is deployed for the Signal app. In Sesame, each user maintains in their local database information about their potential communication partner's in the *UserRecords* and their linked devices in the *DeviceRecords*. For each one of the device records, Sesame stores a list of sessions shared between the local device and the partner's device. In this list, there is one *active* session and the remainder are marked as *inactive* sessions.

The main idea being that the two devices always communicate using the active session, and when a message is received from an inactive one, the latter is promoted to being the current active. This means that receiving out-of-order messages, makes the parties switch their communication to encrypt using the old session from where the message originated. A formal analysis of the management layer paired with the Signal protocol and experiments with the deployed system of the Signal app showed that PCS is violated this setting [20]. A similar work [42] showed that the session-management layers of Wire, called Proteus Dialogue (which resembles the Sesame algorithm) also suffers from the same vulnerabilities discovered in Signal.

3 Resilient Communication Protocols

In this section, we explore the design space of systems that aim to prove PCS and the mapping between usability constraints and their implication to security. We define a class of protocols,

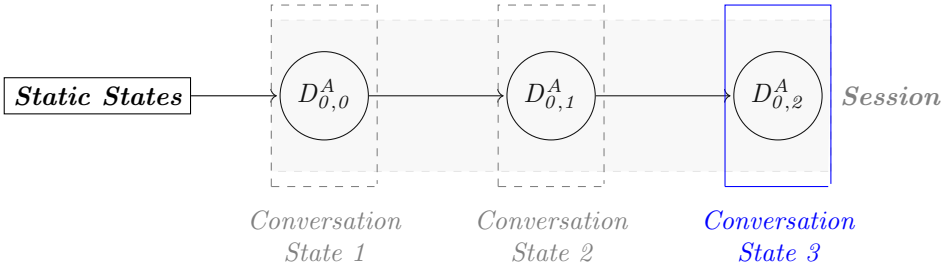

Figure 1: **Evolution of the conversation state of A in the ideal execution.** End-user A bootstraps the conversation using both parties' static states to derive the dynamic state, $D^A_{0,0}$, and subsequently updates it on every message exchange as a long-running session. The ideal environment ensures that both parties can derive and agree on the next dynamic states, $D^A_{0,1}$, $D^A_{0,2}$, and so on.

named *resilient communication protocols*, that abstractly models any two-party protocol that interacts with an unreliable environment. Then, we recall different levels of PCS properties for the end-users and the threat models against which they should hold.

We reason in this abstract class of protocols and draw our main conclusion: for any protocol within this class, i.e. deployed and resilient against real-world failures, the highest level of PCS is fundamentally unachievable. We provide an intuition, through informal arguments and examples, on why these results hold, before setting out to formally prove them in Section 4. We also state our second conclusion on a positive result: resilient messaging protocols can achieve a stronger PCS guarantee for the end-user than what is currently achieved in practice.

3.1 Abstract Communication System

We consider two end-users, A and B, that wish to communicate with each other using a communication protocol that aims to provide PCS. Each end-user has a single device or client that they use to communicate, which we refer to simply as A and B. In order to bootstrap the end-to-end communication, each end-user needs a public identifier. Typically, this is implemented in practice as a long-term private-public key pair, potentially with a certificate to validate the identity depending on the system used. However, to be as general as possible, we do not assume any specific implementation of the identity, but instead refer to the persistent identity as the end-user's *static state*. We assume that the *static state* is not updated, and we will later return to explaining this choice and the implications of updating or rotating the static state in Section 3.3.

Using the static state, the two end-users authenticate one another and derive secrets to encrypt messages end-to-end. For example, often these secrets are instantiated as symmetric keys, but it could also be an agreement on ephemeral asymmetric keys. Modern communication protocols wish to achieve strong security properties for their users, like post-compromise security, and therefore do not encrypt all messages using the same initial secret. In particular, to achieve PCS, the secrets need to evolve by incorporating new entropy, such that an attacker can be locked out of the communication if it misses the updates. We define the *dynamic state* of A, respectively of B, to contain all state that changes as messages are exchanged, and the secrets to encrypt and decrypt messages between A and B. As the name suggests, the dynamic state is updated with the progression of the communication. Lastly, A and B can evolve their dynamic states asynchronously, and as messages are being exchanged, they should both be able agree on the current dynamic state, e.g., A evolves their state to send, and B will evolve their state one they

Layer	Concept	Definition
Communication Protocol	Static State	The fixed identity of the end-user and their device, typically in the form of public-private key pairs.
	Dynamic State	The state of a session, that is updated through time or protocols steps by adding new entropy, e.g., the Double Ratchet state.
	Conversation State	The set of all dynamic states in memory at a particular timepoint.
Environment	Static State Loss	The event of losing all the memory of an end-user, e.g., because lost physical device.
	Dynamic State Loss	The event of losing or not having the correct dynamic state, e.g., because of hardware failures, and restored backups.
	Dynamic State Compromise	The event of an attacker compromising and learning the secrets of a dynamic state.
	Static State Compromise	The event of an attacker compromising and learning the identity of a user.
Session-Handling	Policy	Restrictions on how a communication system manages the multiple dynamic states within a conversation, i.e., their creation, storage, deletion and lifespan.
	Dynamic State Loss Resilience	The ability of the protocol to enable the end-users to recover after losing the dynamic state, to synchronize their state and continue the conversation.
	Static State Loss Resilience	The ability of the protocol to enable the end-users to recover after losing their identity, to synchronize their state and continue the conversation.

Table 1: **Summary of concepts.** We categorize the presented concepts according to the communication architecture layer they belong, namely the communication protocol, the real-world environment, and session management layer.

receive the message.

The linear view of the messages exchanged from one end-user to the other is defined as a *conversation*. The *conversation state* contains at any time point the snapshot of the underlying dynamic states in this linear view. A communication protocol might have to maintain several concurrent dynamic states, as we will see later, but for now it only needs to contain a single one.

We use the term *ideal execution* to denote an execution of the communication system in the absence of failures, i.e., users can update their state and maintain it synchronized with their partner's. We illustrate such as execution in Figure 1. Next, we provide examples from the real-world on how to instantiate the dynamic and the static state.

Example 1: Static and Dynamic State. In WhatsApp [43], Google Messages [24], the Signal app [37], the static state includes the user's identity key pair, and the device's identity key pair. The dynamic state is the state of a session: the execution the X3DH [31] and the subsequent update of the state using the Double Ratchet algorithm [37]. In messengers that adopt the Matrix protocol [39], e.g., Element, the static state includes the device fingerprint and identity key pairs, and the dynamic state is the execution of Olm [41] to establish a session and Megolm [40] to send messages. In Apple's iMessage, the static state includes the long-term user identity key pair, and the dynamic state keeps track of the PQ3 protocol [29, 38] to establish the session and exchange messages.

We refer the reader to Table 1, for a summary of the concepts we presented and will further introduce throughout this section. Now, we explore components that communication systems consider to enable a seamless conversation, despite any desynchronization, or errors that may happen during the communication.

3.2 Real-World Failure Modes

In the ideal execution, A and B start a conversation with a shared dynamic state and update it in lockstep–they can always synchronize. However, during the lifespan of a conversation, end-users can become desynchronized. In general, events that disturb or destroy state can be very fine-grained (a single bitflip). To establish general results, we over-approximate all possible types of state loss into two categories: loss of dynamic state and loss of static state. Loss of all state is then modeled as a combination of the two.

Dynamic state loss is the event where an end-user loses all the values generated and stored in a dynamic state. The end-user can lose part of the state, but we model the worst case scenario: all stored secrets are no longer accessible in the local memory. Dynamic state loss can happen due to normal operations or can be induced by errors at the hardware layer. We provide below two examples.

Example 2: Restore Backup. The end-user restores in their device an old backup, which overwrites the current dynamic state with the dynamic state of the backup. From the device's perspective, a dynamic state loss event took place, and they jumped back to a state in the past.

Example 3: Hardware Failure. The memory hardware can suffer from failures, e.g., a bit/multi-bit error, or losing part of the state [28]. In the worst case, an unrecoverable error renders the entire dynamic state unusable. This is manifested in the communication system as inability to encrypt and decrypt messages, amounting to dynamic state loss.

Static state loss is the event where an end-user loses their static state and all the dynamic state related to it. The identity can be lost, because they reinstalled the communication system's client, or due to an error in the physical device.

Example 4: Lost device. The end-user loses their device, and along with it the static state of the system. In this case, the end-user suffered from a static state loss and to recover they need to reinstall the client in a new device.

3.3 Functional Requirements

Now, that in the real-world environment it is possible for the end-users to lose their dynamic and static state, they can become desynchronized. Lets assume A loses their state. A and B can no longer communicate with one another since: a) the dynamic state's secrets are lost for A and they cannot encrypt, b) messages sent by B cannot be decrypted, and c) the app maintains a single dynamic state per conversation, therefore no resynchronization is possible.

Were an app to implement the described protocol, the end-user would be locked out of the conversation anytime there is a desynchronization. The app is rendered unusable even during expected operations such as dynamic state loss. The same arguments can be applied for static state loss.

A protocol deployed in the real-world should be resilient against events such as dynamic and static loss. We define this class of protocols as *resilient communication protocols*, which enable the participants to continue sending and receiving messages after every state loss event. We define the following two properties for the resilient messaging protocol:

Definition 1. *Dynamic State Loss Resilience*: After every dynamic state loss, a honest end-user is able to continue the communication, i.e., to send and receive messages.

Definition 2. *Static State Loss Resilience*: After every static state loss, a honest end-user is able to continue the communication, i.e., to send and receive messages.

Dynamic State Loss Resilience. Once an end-user loses their dynamic state, the only way to re-synchronize with their partner is to go back to a common point where they both agree on,

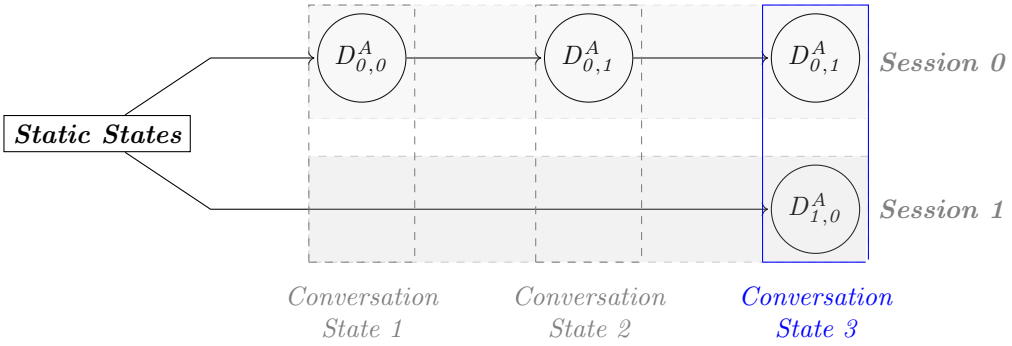

Figure 2: **Evolution of the conversation state of A in the real-world environment.** End-user A starts a new dynamic state $D^A_{1,0}$ and progresses in the *resilient communication protocol* after the parties are desynchronized and no longer agree on the current dynamic state $D^A_{0,1}$.

i.e., the static state. User's know each other's static state, and from that point they can negotiate new secrets and continue the conversation. In practice, this amounts to A and B executing a new key exchange to authenticate their identities and compute a new shared dynamic state. To achieve this, the conversation allows for multiple dynamic states as shown in Figure 2. In a conversation, A and B have one or multiple running different dynamic states at the same time, and overall *multiple* during the lifetime of that conversation.

Static State Loss Resilience. After losing the static state, the end-user needs to recover their identity, and re-authenticate, or log-in into their account. The identity could be stored in a local backup or using a third-party, where the user has access to or using a recovery mechanism. Re-authentication, typically requires an out-of-band channel, for example verifying one's email address or phone number.

Rotating the static state. The static state could be "restored" by rotating the identity, effectively generating new secrets such as long-term identity keys. Notice that in this case, the end-user would depend on another secret or state to authorize the update of the identity, such as an user secret or account password. Since the static state is no longer static, it would be considered part of the dynamic state, and this new secret becomes the static state of the end-user. As a result, all of our statements over the static and dynamic states would simply shift to talk about different instantiations. I.e., the problem of losing and recovering state would manifest itself at another layer of the secrets. Therefore, we fix the static state to be the unique identity of the end-user, which is recoverable using a backup or a recovery mechanism, but otherwise it is not changed.

3.4 Security Requirements

Now, we define the threat model of the communication system and the different post-compromise security properties.

Threat Model. In our work, we consider an attacker that can compromise dynamically any end-user at any time. We define two types of compromise:

1. *Dynamic State Compromise:* The attacker compromises a dynamic state, and learns all the stored secrets.
2. *Static State Compromise:* The attacker compromises the static state, and learns all the stored identity secrets.

When the attacker compromises both the static and dynamic state, we refer to it as a *full compromise*. Additionally, we consider the attacker to be *active* and interfere in the protocol, e.g., by injecting their own messages in the protocol. For completeness, we will also briefly mention the *passive* attacker, which does not interact with the system, instead it records and tries to decrypt the traffic later (harvest-now-decrypt-later threat model).

PCS properties. From the threat models and the level in which the property should hold, we can categorize four PCS properties for the messaging system.

Initially, we define the PCS from the secure messaging space. Messengers, being an instantiation of communication system described so far, claim to achieve PCS against a passive eavesdropper.

Definition 3. *Session PCS - Passive Attacker*: Security of the session is restored after healing, even when a passive attacker compromises the dynamic state.

Next, we define PCS studied in the literature, which holds over a single session, and the attacker compromises all user secrets.

Definition 4. *Session PCS - Full Compromise*: Security of the session is restored after healing, even when an active attacker compromises the static and dynamic state.

Considering that the conversation of two end-users can have multiple dynamic states, [20] defines the conversation-level PCS property against a dynamic state compromise.

Definition 5. *Conversation PCS - Dynamic State Compromise*: Security of the conversation is restored after healing, even when an active attacker compromises the dynamic state.

Lastly, we define the conversation level property against the stronger threat model from the literature, where both the dynamic and static state are compromised, i.e., full PCS achieved at the end-user level. The property guarantees that the healing event can lock the attacker out of the entire conversation of the end-user.

Definition 6. *Conversation PCS - Full Compromise*: Security of the conversation is restored after healing, even when an active attacker compromises the static and dynamic state, i.e., all the secrets of a conversation.

3.5 Overview of Main Results and Consequences

We explored the design space of an abstract communication system: the real-world failures and the functionalities that the app needs to account for, as well as varying levels of PCS guarantees and threat models. Now, we will provide intuition for our two main results, which we will prove in Section 4 and Section 5.2, respectively.

Our first main result captures a trade-off between resilience and security requirements:

A resilient communication protocol cannot achieve conversation PCS against a full compromise (Definition 6).

The underlying intuition is that a resilient protocol tolerates dynamic state loss by allowing users to continue the protocol purely based on their static state. Since this is exactly the state compromised in the full PCS threat model, the latter can continue communicating by creating a new dynamic state with the partner. Similarly, the protocol tolerates static state loss by allowing the user to recover the *same* static state, making the problem persist across static state recovery. If a protocol has no truly static state, for example because it also rotates long-term keys, it

cannot be resilient because after losing dynamic state, it has no shared state to bootstrap from, as discussed previously in Section 3.3.

Our second main result is a positive result:

A resilient communication protocol can achieve conversation PCS against a dynamic state compromise (Definition 5).

The intuition is that the resilient protocol allows the users to create new dynamic states (because it tolerates dynamic state loss). Dynamic states can be managed such that state updates ensure that previous dynamic states are deprecated. This way, the protocol can clean the system from a compromised state, and with it remove the attacker. Thus, if healing occurs after a compromise, the attacker cannot rejoin the protocol, because it does not know the static state nor the current dynamic state.

Relating our results to existing PCS solutions At first glance, our results seem to contradict known results from the literature. The difference is that existing work on PCS considers only the single session setting, and at the same time does not considering resilient communications: when two parties desynchronize, there is no mechanism or security guarantee for any later communications. Indeed, existing solutions do not provide PCS after desynchronization [3, 11, 15, 23, 26, 29, 38]: their results only hold for single sessions that do not desynchronize and do not carry over to the real-world multi-session deployments.

The authors of [22] propose an authenticated continuous key agreement (ACKA) protocol that can detect and prevent attacks from an active MitM. By treating the Signal protocol as a continuous key agreement, they can provide an add-on to achieve the strong property in the protocol. However, when the protocol is deployed, e.g., in the Signal app, at the user-level there is no longer a single long-lived running session. In reality, parties are allowed to start multiple sessions, and as a result the active MitM attacker is not detected or prevented at the user-level, because of the following behavior: the attacker compromises the identity key, the honest parties heal in a session, the attacker starts a parallel session with the partner.

Our general observations also have implications for group-key based protocols such as IETF's Messaging Layer Security (MLS) protocol [7], where desynchronized users can rejoin the group. The rejoining requires the static state of the user. The full PCS property would suffer the obvious attack of the adversary rejoining instead of the compromised party, and since the latter can suffer from dynamic state loss, they cannot tell if they have been impersonated.

In a similar way, work on group messaging [2, 4–6, 10, 14, 36] also consider single session guarantees. The initial work on interoperability of secure messaging in [27] also does not consider PCS for the end user, but rather for a single session.

Matrix does not achieve full PCS [1], and their recommendation on how to enable it is to "periodically start a new session" [40] with a "maximum number of at least 4 sessions per device" [39]. Unfortunately, their recommendation leads to worse PCS guarantees, because it introduces more opportunities for the adversary to violate PCS. Essentially, the apps from Example 1, including Matrix, allow the following behavior: the static state of A is compromised, A heals, the attacker starts a new session using the compromised state and injects new messages. In particular for the Signal app, this was experimentally shown previously in [20].

4 Impossibility Results

In this section, we develop our formal model of a *communication protocol* that aims to achieve PCS with the goal to prove our impossibility result. We do not reason about a specific instantiation (or

implementation) of such a protocol, but create an abstract model that allows us to reason about *any* instantiation that fits into the design space we explored in Section 3. We show step by step the construction of the *resilient communication protocol* model, and then prove the impossibility result.

First, we explain our methodology in 4.1, and then introduce the tool used for the analysis, the Tamarin Prover [33] in 4.2. We develop our formal models in 4.3, while highlighting some of the design choices. Finally, we present the results of our analysis in 4.4. We publish our models and results to enable reproducibility, and extensions in [21].

4.1 Methodology

Our methodology is to incrementally develop abstract formal models that cover the failure modes, and show, step by step, the consequences of enabling different behaviors leading to the loss of PCS.

We start with a Base model of the communication system defined in Section 3.1. This model captures the bare-bones functionalities that are necessarily present in any communication system that aims to provide PCS guarantees. To begin with, we only consider *ideal* executions of the protocol. Building on top of this model, we add the real-world modes of failures, namely a) dynamic state loss, and b) static state loss. In the resulting two non-resilient models, $\mathsf{Non\text{-}Resilient}_{\mathsf{Dynamic}}$ and $\mathsf{Non\text{-}Resilient}_{\mathsf{Static}}$, we prove that indeed communication is stopped after these events: the user is locked out of the protocol.

Motivated by our observation in Section 3.3, that a real-world protocols needs to offer its users a way to recover from state loss, we create the $\mathsf{Resilient}_{\mathsf{Dynamic}}$ and $\mathsf{Resilient}_{\mathsf{Static}}$ models. In $\mathsf{Resilient}_{\mathsf{Dynamic}}$, we enable multiple dynamic states, which allows the users to continue communication from another dynamic state, and in $\mathsf{Resilient}_{\mathsf{Static}}$, we enable multiple dynamic states and a static recovery mechanism allowing the users to recover from both failures. For both models, we show that they overcome the usability issues of their predecessor model, and formally prove that they are indeed resilient against the respective failures. However, this resilience comes at a cost: both models no longer provide PCS (Definition 6), which we showcase via attack traces found in each of the models.

Finally, we use the $\mathsf{Resilient}_{\mathsf{Static}}$ model, the most expressive model, to show through attack traces, that different classes of policies that manage the multiple dynamic states fail to recover conversation PCS. We are systematic enough to cover different classes of policies:

1. **No policy**: The attacker can compromise old sessions to inject messages [20], and start new sessions.
2. **Sequential Sessions:** To mitigate the first attack, creating new session now deprecates the previous session making sessions sequential. Deprecated sessions are still kept around to receive messages.
3. **Token-Passing between sessions:** Following the logic of how PCS can be achieved in [16], the active session now passes a token to the next active session when it gets deprecated. This is done to create a chain of sessions, where missed updates remove the attacker from the chain. However, this policy introduces a new point-of-failure: If the active session containing the token is lost, the user is locked out of any further communication. As a result, the protocol is no longer resilient against dynamic state loss.
4. **Third-party providing Token:** To improve on the shortcomings of storing the token locally, the protocol can store it with a trusted third-party which does not lose state; making this class of protocols resilient against state loss. During protocol execution, the end-users can query the trusted third-party with information stored in their static state to retrieve the token when necessary.

4.2 The Tamarin Prover

The Tamarin Prover is an open-source symbolic analysis tool for the analysis and verification of large scale cryptographic protocols [33]. However, in this work, we use it as a tool to analyze a generic unbounded concurrent system. For instance, we do not make use of its equational theories to model cryptographic primitives, or its inbuilt Dolev-Yao adversary. We provide below a high-level overview of the tool and some of the key concepts on how to model systems and properties.

A user provides the tool with a model of their system, a property they want to verify, and the threat model against which the property should hold.

The input of the tool is modelled using a Tamarin specific language based on multiset rewriting. In particular, the state of the system is recorded in so-called facts, e.g., $\mathsf{Fr}(id_A)$ denotes a special fact, the so-called fresh fact, that guarantess that id_A is a unique, freshly generated value. To specify how the system can transition from one state to another, the user defines rules. A rule is constructed by a left-hand size (**L**)–the preconditions to execute the transition, a right-hand side (**R**)–the updated system after taking the transition, and the observable logged action (**A**). Rules can also be used to create threat model by, e.g., modeling the compromise of a party, or the attacker executing a protocol step. We provide below an example of a rule that creates a user device. After taking this transition, a new fact $!\mathsf{UserDevice}(id_A)$ is added to the system whose freshly generated identifier is logged in the action $!\mathsf{CreateDevice}(id_A)$. The ! here denotes a persistent fact, i.e., a fact that is never removed from the system even when present on the left-hand side of a rule.

$$[\ \mathsf{Fr}(id_A)\ ] -[\ \mathsf{CreateDevice}(id_A)\ ]\rightarrow [\ !\mathsf{UserDevice}(idA)\ ]$$

To express properties in Tamarin, the user specifies trace properties using a guarded first-order logic fragment over the observable logged actions. From the previous example, the observable action is $\mathsf{CreateDevice}\ (id_A)$, logging the creation of the device with identifier id_A. A simple property over this action could state that there exists a trace in the system where two different devices are created.

$$\exists\ id_A\ id_B\ \sharp i\ \sharp j\ .\ not(id_A\ =\ id_B) \wedge$$
$$\mathsf{CreateDevice}(id_A)@\sharp i\ \wedge\ \mathsf{CreateDevice}(id_B)@\sharp j$$

To prove a property, Tamarin negates it and tries to turn the counterexample into a valid trace of the protocol by reasoning backwards using the user-defined protocol rules, as well as some inbuilt ones. If this backwards reasoning fails in *all* cases, Tamarin concludes that there exists no counterexample, and the property is proven. If a counterexample can be constructed, Tamarin will output an attack trace. Due to the undecidability of the problem, running out-of-memory (or time) is unavoidable and always possible. However, in this case, Tamarin offers a graphical user-interface where the user can inspect partial proofs and manually guide the tool. Additionally, users can create custom heuristics for better proof search, and state intermediate lemmas, often called *invariants*, which Tamarin can use to prove more complex statements.

4.3 Modeling the Communication System

In this section, we describe our incremental Tamarin models of a communication protocol that aims to achieve PCS. We start out with the Base model and then explain how each subsequent model builds on the its predecessors. Finally, we model different policies for dynamic state handling in the *resilient communication protocol* model.

Base Model

The Base model captures the abstract communication system; two end-users can bootstrap a conversation and update their dynamic state as described in Section 3.1. The model is minimal in the sense that it does not allow for more than a single dynamic state between two users and every execution of the protocol is *ideal*, i.e., there is no dynamic or static state loss.

Setup We model an unbounded number of end-users that communicate with one another in pairs of two. Each of these end-users possesses a single device. Once the parties bootstrap their communication, they can exchange an unbounded number of messages going back and forth.

Bootstrapping the protocol To model the end-users and their static state, we use the persistent fact $!\mathsf{UserDevice}(id)$, which carries a unique identifier id. When two devices start the communication, they create from both parties' static state their respective dynamic state. We show below a simplified rule modelling the creation of a dynamic state.

$$
\begin{aligned}
&[\quad !\mathsf{UserDevice}(id_A), !\mathsf{UserDevice}(id_B),\\
&\quad\ \mathsf{Fr}(s), \mathsf{Fr}(sid)\quad]\\
&-[\ \mathsf{CreateDynamicState}(sid, id_A, id_B, s)\ ]\rightarrow\\
&[\quad \mathsf{DynamicStateA}(sid, id_A, id_B, s),\\
&\quad\ \mathsf{DynamicStateB}(sid, id_A, id_B, s), \ldots\quad]
\end{aligned}
$$

Here, $\mathsf{DynamicStateA}(sid, id_A, id_B, s)$ models $D^A_{0,0}$. We use the identifier sid to refer to the chain of dynamic states which starts with this one, and the value s to model the secret that is continuously updated as the state evolves. The action CreateDynamicState logs the creation of the dynamic state.

Restricting to Single Session To allow only a single session of dynamic states between a pair of users, we make use of Tamarin's restriction feature, and introduce the following restriction:

$$
\begin{aligned}
&\forall\ sid\ sid'\ id_A\ id_A\ s\ s'\ \sharp i\ \sharp j\ .\\
&\mathsf{CreateDynamicState}(sid, id_A, id_B, s)@\sharp i\\
&\ \wedge\ \mathsf{CreateDynamicState}(sid', id_A, id_B, s')@\sharp j\\
&\ \Rightarrow\ \sharp i = \sharp j
\end{aligned}
$$

This restriction states that if there is a pair of CreateDynamicState actions that agree on the users id_A and id_B, they are the same one, i.e., only a single dynamic state can be created for an end-user pair.

Progressing in the protocol Now that the parties have established their conversation, they can act as *Sender* and *Receiver* in an update of their respective dynamic state. We illustrate the rule of A progressing as a sender. The rule for B is analogous:

$$
\begin{aligned}
&[\quad \mathsf{DynamicStateA}(sid, id_A, id_B, s), \mathsf{Fr}(s')\quad]\\
&-[\ \mathsf{HonestStep}(sid, id_A, id_B, s),\\
&\quad\ \mathsf{HonestSendStep}(sid, id_A, id_B, s), \ldots\ ]\rightarrow\\
&[\quad \mathsf{DynamicStateA}(sid, id_A, id_B, \langle s, s'\rangle),\\
&\quad\ !\mathsf{UpdateDynStateB}(sid, id_A, id_B, s, \langle s, s'\rangle)\quad]
\end{aligned}
$$

This rule models A sending a message to B. Notice that we are not using TAMARIN's inbuilt network, but use the fact !UpdateDynStateB to model a message that B can receive. In this rule, A samples new entropy, Fr (s') and updates their dynamic state to include it. We use the action HonestStep to log the fact that an honest user, in this case A, executed a dynamic state update, and the action HonestSendStep to more precisely record that the update was caused by sending a message.

Threat Model The attacker can compromise both the dynamic state and the static state of the end-users at any point in time. We model the attacker's knowledge after the compromise in fact, a) !CompromisedDevice(id) models the attacker compromising and leaning the static state of device id, and b) CompromisedDynStateB(sid, id_A, id_B, s) models the compromise of the dynamic state.

Then, to model the attacker's capabilities, we create *attacker controlled* rules. Below, we showcase the attacker's variant of the rule for being able to receive, decrypt and learn the secret of a dynamic state:

$$\begin{aligned}
&[\ \ \mathsf{CompromisedDynStateB}(sid, id_A, id_B, s),\\
&\ \ \ \mathsf{!UpdateDynStateB}(sid, id_A, id_B, s, \langle s, s'\rangle)\ \ ]\\
&-[\ \mathsf{AttackerKnows}(\langle s, s'\rangle)\ ]\!\rightarrow\\
&[\ \ \mathsf{CompromisedDynStateB}(sid, id_A, id_B, \langle s, s'\rangle)\ \ ]
\end{aligned}$$

We use the action AttackerKnows to record that the attacker knows a particular secret. Later on, we will use this to formally state our PCS properties within TAMARIN.

Modelling State Loss

Now, we model the real-world failure on the communication protocol described in Section 3.1. From the Base model, we add dynamic and static state loss, resulting in two new models, $\mathsf{Non\text{-}Resilient}_{\mathsf{Dynamic}}$ and $\mathsf{Non\text{-}Resilient}_{\mathsf{Static}}$ respectively.

To model dynamic state loss, we add a rule that consumes an existing DynamicState fact, which can then no longer be used to progress in the protocol. We show the dynamic state loss of A, analogous rule for B, in the following:

$$\begin{aligned}
&[\ \ \mathsf{DynamicStateA}(sid, id_A, id_B, s)\ \ ]\\
&-[\ \ \mathsf{DynamicStateLossA}(\dots)\ \ ]\!\rightarrow\\
&[\ \ ]
\end{aligned}$$

We model the static state loss using similar rules. Additionally, this also implies that any previous dynamic states are also unusable. To model the latter, we add restriction that enforces that after a static state loss no more progress from any dynamic states of this static state is allowed.

$$\begin{aligned}
&\forall\ sid\ id_A\ id_B\ s\ \sharp i\ \sharp j\ .\\
&\mathsf{StaticStateLoss}(id_A)@\sharp i\\
&\wedge\ \mathsf{HonestStep}(sid, id_A, id_B, s)@\sharp j\\
&\wedge \sharp i < \sharp j \Rightarrow\ \bot
\end{aligned}$$

Recovering From State Loss

To adapt to non-ideal executions of the protocol, we add mechanisms to recover from dynamic state loss and static state loss to the previous model, and create the $\mathsf{Resilient}_{\mathsf{Dynamic}}$ and $\mathsf{Resilient}_{\mathsf{Static}}$ models.

In the $\mathsf{Resilient}_{\mathsf{Dynamic}}$ model, we recover from dynamic state loss by removing the restriction that limits dynamic state creation per user pair. This allows now for an *unbounded* number of session within a conversation. With more than one session between users, communication can now continue in case of a dynamic state loss by simply creating a new dynamic state for further communication.

In the $\mathsf{Resilient}_{\mathsf{Static}}$ model, we recover from static state loss by adding a new rule that abstractly models an event of recovering the same static state from a trusted-third party or backup. The rule raises the $\mathsf{StaticStateRecovery}(id)$ action. Then, we modify the state loss restriction to allow the parties progressing in the protocol, strictly if they have recovered from the loss:

$$\begin{aligned}
&\forall\ sid\ id_A\ id_B\ s\ \sharp i\ \sharp j\ .\ \sharp i < \sharp j\ \wedge \\
&\mathsf{StaticStateLoss}(id_A)@\sharp i \\
&\wedge\ \mathsf{HonestStep}(sid, id_A, id_B, s)@\sharp j \\
&\Rightarrow\ (\exists\ \sharp k\ .\mathsf{StaticStateRecovery}(id_A)@\sharp k \\
&\wedge \sharp i < \sharp k \wedge \sharp k < \sharp j)
\end{aligned}$$

Policies

We will now describe how we model the family of policies we have identified in Section 4.1.

No Policy In the absence of any policy, our $\mathsf{Resilient}_{\mathsf{Static}}$ suffices since it does not put any restrictions on the dynamic state management.

Sequential Sessions The *sequential sessions* policy enforces that only the latest session can encrypt and send new messages, while older sessions are only allowed to receive messages which might still be in-flight. To implement the first part of the policy, we add the following restriction to our $\mathsf{Resilient}_{\mathsf{Static}}$ model:

$$\begin{aligned}
&\forall\ sid\ sid'\ id_A\ id_B\ s\ s'\ s''\ \sharp i\ \sharp j\ \sharp k\ . \\
&\mathsf{CreateDynamicState}(sid, id_A, id_B, s)@\sharp i \\
&\wedge\ \mathsf{CreateDynamicState}(sid', id_A, id_B, s')@\sharp j \\
&\wedge\ \mathsf{HonestSendStep}(sid, id_A, id_B, s'')@\sharp k \\
&\wedge\ \sharp i < \sharp j \wedge \sharp j < \sharp k\ \Rightarrow\ \bot.
\end{aligned}$$

Formally, the restriction enforces that only the latest dynamic state can execute HonestSendStep actions, i.e., send updates to the peer. The resulting model, we call the Sequential model.

To model that older session can still receive in-flight messages, we copy the rules for receiving messages of the honest parties, and add actions that distinguish them from the original rules. Then, we add a restriction that prevents older sessions from using the original receive rules–allowing them to only receive in-flight messages.

Token-Passing between sessions This policy builds on the *sequential sessions* policy and additionally introduces a secret *token* which is passed from the currently active session to the next one to achieve PCS at the conversation-level. Consequently, there exists only a single dynamic state per user that can create new dynamic states because it alone posses the token. As we have already shown in our $\mathsf{Non\text{-}Resilient}_{\mathsf{Static}}$ model, this design fails to achieve resilience against dynamic state loss: It is impossible to recover from the loss of this special dynamic state; no other state can create new ones.

Third-party providing Token This policy builds on the last policy by outsourcing the storage of the token to a trusted third-party, which does not lose state. The benefit of this is that the protocol again becomes resistant against dynamic state loss because the affected party can query the third-party with their static state to create a new dynamic state.

To model this policy, we add a unique fact, $\mathsf{TTP}(t, t')$, for each pair of end-users to our model, which acts as a storage for the current token t and next token t'. Additionally, the dynamic state is initialized with the current token. Whenever, a new dynamic state is created, the trusted third-party provides the next token only when the current token matches the current dynamic state's token. To recover from dynamic state loss, we also allow creating a dynamic state from the static state and a token provided by the trusted third-party. We call the resulting model TokenPassing, and refer to our models [21] for more details.

4.4 Formal Analysis Results

In this section, we summarize the results of our formal analysis with Tamarin.

1) Non-Resilience We first show two system that achieve PCS, but are not resilient against state loss, i.e., the loss of *dynamic* and *static* state. In the $\mathsf{Non\text{-}Resilient}_{\mathsf{Dynamic}}$ model, we show the following property for end-user A:

$$\begin{aligned}
&\forall\ sid_1\ uid_A\ id_A\ uid_B\ id_B\ rk1\ rk2\ \sharp i\ .\\
&\mathsf{DynamicStateLossA}(sid_1,\ uid_A,\ id_A,\ uid_B,\ id_B,\ rk1)@\sharp i\\
&\Rightarrow\ not\ (\exists\ sid_2\ rk2\ \sharp j\ .\ \sharp i < \sharp j\ \wedge\\
&\mathsf{HonestStepA}(sid_2,\ uid_A,\ id_A,\ uid_B,\ rk2)\ @\sharp j\ )
\end{aligned}$$

Formally, the property encodes that after the $\mathsf{DynamicStateLossA}(sid_1,\ uid_A,\ id_A,\ uid_B,\ id_B,\ rk1)$ action, the user cannot do a communication step $\mathsf{HonestStepA}(sid_2,\ uid_A,\ id_A,\ uid_B,\ rk2)$. This holds across *any* session that they share with the user uid_B. The intuition behind is straightforward: Losing the *single* dynamic state shared between uid_A and uid_B without a recovery mechanism locks the user out of any further communication.

In the $\mathsf{Non\text{-}Resilient}_{\mathsf{Static}}$ model we show that a after static state loss the user can no longer participate in the protocol. In Tamarin, the property is the same to the previous property but it reasons about the StaticStateLoss fact instead. Since static state is lost, so are all the conversation's concurrent dynamic states, leaving the user locked out of the conversation.

2) Resilience In the $\mathsf{Resilient}_{\mathsf{Dynamic}}$ model, we establish that the users can recover from dynamic state loss and continue communicating with their partner. This is captured by the following property in Tamarin:

$$\begin{aligned}
&\exists\ sid_1\ sid_2\ uid_A\ id_A\ uid_B\ id_B\ rk1\ rk2\ \sharp i\ .\sharp i < \sharp j\ \wedge\\
&\mathsf{DynamicStateLossA}(sid_1,\ uid_A,\ id_A,\ uid_B,\ id_B,\ rk1)@\sharp i\\
&\wedge\ \mathsf{HonestStepA}(sid_2,\ uid_A,\ id_A,\ uid_B,\ idb,\ rk2)\ @\sharp j
\end{aligned}$$

The property states that it is possible for end-user A to continue executing HonestStepA at timepoint $\sharp j$, potentially in a different session, after they previously suffered from a DynamicStateLossA at timepoint $\sharp i$ ($\sharp i < \sharp j$) event. Intuitively, this is now possible because the model is no longer restricted to a single session between end-user pairs, i.e., A and B can create a new dynamic state and continue in a different session of the conversation.

Analogously, we establish in the $\mathsf{Resilient}_{\mathsf{Static}}$ model that the static state loss recovery mechanism is also effective. We prove the existence of a trace characterized by the following statement:

$$\begin{aligned}&\exists\ sid\ uid_A\ id_A\ uid_B\ id_B\ rk\ .\sharp i < \sharp j \wedge\\ &\mathsf{StaticStateLoss}(\ uid_A,\ id_A\ )@\sharp i\ \wedge\\ &\mathsf{HonestStepA}(sid,\ uid_A,\ id_A,\ uid_B,\ rk)\ @\sharp j\end{aligned}$$

Again, the property is analogous to the one for dynamic state loss. Proving this property, we show that a party that suffered a static state loss can continue participating in the protocol afterwards. This property is now achieved for the following technical reasons: Recall from Section 4 that we add a rule raising the StaticStateRecovery action to the $\mathsf{Resilient}_{\mathsf{Static}}$ model, and changed the existing restriction modeling static state loss to be ineffective if the event is raised for a party. As a consequence, static state recovery is now possible.

3) Impossibility Results Now, we formally establish our impossibility result:

In our model, a resilient communication protocol cannot achieve conversation PCS against a full compromise attacker (Definition 6)

Definition 6. Conversation PCS - Full The property states that after the healing of A in session sid_1, if there is a protocol step such as send or receive in *any* session, and the attacker knows this secret, the only way this is possible is if they compromised the victim again after healing, or they additionally compromised the partner.

$$\begin{aligned}&\forall\ sid_1\ sid_2\ uid_A\ id_A\ uid_B\ id_B\ rk\ \sharp i1\ \sharp i2\ \sharp i3\ .\ \sharp i1 < \sharp i2\ \wedge\\ &\ \textit{// After a party heals session } sid_1,\\ &\mathsf{HealingA}(\ sid_1,\ uid_A,\ id_A,\ uid_B,\ id_B,\ \ldots)@\sharp i1\ \wedge\\ &\ \textit{// and receives/sends in any session with secret rk,}\\ &\mathsf{HonestStep}(\ sid_2,\ uid_A,\ id_A,\ uid_B,\ id_B,\ rk\ )@\sharp i2\ \wedge\\ &\ \textit{// if attacker knows this secret rk,}\\ &\mathsf{AttackerKnows}(\ rk\ )@\sharp i3\ \wedge\\ &\Rightarrow\ \ \textit{// is because the attacker either:}\\ &\ \textit{// compromised the state again after healing,}\\ &(\exists\ \sharp j\ .\ \sharp i1 < \sharp j\ \wedge\\ &\mathsf{CompromiseA}(uid_A,\ id_A,\ uid_B,\ id_B,\ \ldots)\ @\sharp j\ )\ )\\ &\|(\exists\ \sharp j\ .\sharp i1 < \sharp j\ \wedge\ \mathsf{CompromiseDevice}(uid_A,\ id_A,\ )\ @\sharp j\ )\ )\\ &\ \textit{// or compromised the state of the partner.}\\ &\|(\exists\ \sharp j\ .\mathsf{CompromiseB}(uid_A,\ id_A,\ uid_B,\ id_B,\ \ldots)\ @\sharp j\ )\ )\\ &\|(\exists\ \sharp j\ .\mathsf{CompromiseDevice}(uid_B,\ id_B,\ \ldots)\ @\sharp j\ )\ )\end{aligned}$$

To prove our impossibility result, we find counterexamples to Definition 6 in the $\mathsf{Resilient}_{\mathsf{Static}}$, the Sequential, and the TokenPassing model. The impossibility results correspond to the informal intuition built in Section 3, independently how the sessions are managed, they are generated by the static state, which is compromised by the attacker.

Analysis Times For all of our models, we specify 14 intermediate helper lemmas to aid Tamarin in the verification. These lemmas help Tamarin during its proof search by identifying invariants of the system, e.g., the ordering of specific actions, and allow Tamarin to discard some traces earlier.

Model	PCS (Definition 6)	Run Time	Helper Lemmas
No Policy ($\mathsf{Non\text{-}Resilient}_{\mathsf{Static}}$)	✗	330s	14
Sequential	✗	1h	14
TokenPassing	✗	6s*	14

Table 2: Summary of Tamarin Impossiblity Results.
Our proofs are obtained by using Tamarin's *Tactic* feature to guide its proof search, as well as some manual stored proofs (marked by *). The runtimes consist of automatic verification via Tamarin's CLI and automatic verification using its GUI. All models were run on a ThinkPad X1 Carbon Gen 9 with 32Gb of RAM.

With these lemmas, we are able to fully automatically verify the properties related to the resilience of our *non-resilient* models and our *resilient* models in around 9 seconds per model using Tamarin's command-line interface.

To establish our impossibility result, we can find the counterexample in the $\mathsf{Resilient}_{\mathsf{Static}}$ model automatically with Tamarin's CLI in around 360 seconds. Including the time it takes Tamarin to verify our intermediate lemmas, the verification time for this model is roughly 390 seconds.

Finding the counterexample in both the Sequential model and the TokenPassing model required more human interaction. For the former, we use Tamarin's GUI and initially guide it by hand, but have the tool then finish the proof automatically. For the latter, we resort to manually specifying a trace that violates the PCS property and then constructing that trace manually in the GUI. We provide all models and files containing both counterexamples in [21].

5 Improved Generalized Session-Handling

From our results in the previous section, we know that conversation PCS cannot be achieved if the attacker compromises the static state (Definition 6). However, messengers can still achieve a stronger PCS than what is being currently offered (from Definition 3 to Definition 5). We will first describe the recommended solution of the improved session-handling layer. Then, we model the policy in the resilient protocol model, and formally prove PCS, thus reducing the attack space in case of a dynamic state compromise.

5.1 Proposal

We recommend a generalized policy for all resilient communication systems that aim to improve their security guarantees post-compromise, based on policies from both the literature and deployed in practice.

Basic functionalities The basic functionalities of our solution are based on the Sesame algorithm [30]. Each device locally keeps a list of the partners and their devices, as well as the sessions that they create with each one of those devices.

Update policy Following [20], we also recommend that sessions are created and used sequentially when sending. This means that once a session is replaced, that old session must no longer

be used afterwards to encrypt new messages.

Number of stored sessions (N) Each device maintains $\mathsf{N} \geq 2$ sessions: the latest one to send and receive messages, and the previous $\mathsf{N}-1$ sessions to strictly only receive older messages. To optimize security and usability, we recommend $\mathsf{N} = 2$. The previous session is needed in two scenarios: a) the session was just updated and there can still be in-flight messages that arrive later, and b) the devices started simultaneous sessions and they need one session to send and the other to receive. In case of simultaneously started sessions, devices could perform an additional check to abandon one of the two, when creation times are relatively close to one another, or can agree to start a new one. We expand in Section 6.1 on the impact of the parameter N on security versus resilience.

Update policy - User Interface For user-facing communication systems, the user interface should display received messages in an order that respects the order of the session it was sent from. I.e., a message received from an older session should appear in the conversation, in the correct place in the past where the session was active, and the message was decrypted. However, this is not how apps currently handle these messages: instead, they will appear at the bottom (latest part of the conversation) despite being sent from an older session. During normal usage of the messaging app, this behavior occurs rarely, however, if an attacker actively injects messages in the conversation, the end-user is currently not notified of it. To ensure that the end-user is aware of this happening, we additionally recommend highlighting these messages in the user interface.

Session deletion Since the update policy specifies to maintain sequential sessions, the sessions list implements a First-In-First-Out order, i.e., older sessions are removed first. This was already suggested in [30], but not yet implemented in practice in messengers such as Signal. The previous $\mathsf{N}-1$ sessions can be removed after a predefined time T, and converge to keeping only the current session. The choice of T specifies the time window during which apps still expect any in-flight out-of-order messages, and do not want to re-encrypt them. Choosing a small time window will result in the same issues discussed in Section 6.1.

Security Comparing our recommended solution to Sesame, we do not add any additional overhead, while strictly offering a better security guarantee. Adding the update policy of [20], enables a resilient communication protocol to achieve conversion PCS against *dynamic* state compromise[1], which is otherwise not achieved in a session management layer without any policies, e.g., Sesame. Our recommendations remove the attack vector of compromising a session and using it to inject *new* messages indefinitely.

However, an attacker that has compromised an old session, can still use them to inject *legacy* messages, essentially messages that appear to have been sent from the past. The apps can restrict the attack vector by reducing the number of sessions. From our recommendation to restrict the number of stored sessions $\mathsf{N} = 2$, we create a threshold on the time window in which an attacker can misuse an old compromise, i.e., N session updates, renders previous sessions unusable in the system. We show an attack trace in 5.2 to illustrate the subtle security differences from choosing N in our formal analysis, and discuss the wider usability-security trade-off in Section 6.1. Essentially, a lower N has better security (no previous messages are accepted, they need to be re-encrypted), a higher N offers better resilience (reducing the need for re-encrypting past messages in case of failures).

[1]This does not violate our impossibility result, which involved the compromise of the static state.

5.2 Formal Analysis

To formally verify our proposed solution, we develop and analyze a Tamarin model, which we call the Proposal model. We build this model starting from the $\mathsf{Resilient}_{\mathsf{Static}}$ model from Section 4. We add the *sequential session* update policy, and a new restriction which models that only the latest two sessions can receive *legacy* messages. This models our recommendation regarding the number of stored sessions ($\mathsf{N} = 2$) and update policy. The exact technical details of the Tamarin model can be found in [21].

Definition 5. Conversation PCS We formally modelled conversation PCS against dynamic state compromise in Tamarin with the following statement:

$$
\begin{aligned}
&\forall\ sid_1\ sid_2\ uid_A\ id_A\ uid_B\ id_B\ rk\ \sharp i1\ \sharp i2\ \sharp i3\ .\ \sharp i1 < \sharp i2\ \wedge \\
&\quad \textit{// After a party heals session } sid_1\textit{,} \\
&\mathsf{HealingA}(\ sid_1,\ uid_A,\ id_A,\ uid_B,\ id_B,\ \ldots)@\sharp i1\ \wedge \\
&\quad \textit{// and receives/sends in any session with secret } rk\textit{,} \\
&\mathsf{HonestStep}(\ sid_2,\ uid_A,\ id_A,\ uid_B,\ id_B,\ rk\ )@\sharp i2\ \wedge \\
&\quad \textit{// if attacker knows this secret } rk\textit{,} \\
&\mathsf{AttackerKnows}(\ rk\ )@\sharp i3\ \wedge \\
&\Rightarrow\ \ \textit{// is because the attacker either:} \\
&\quad \textit{// compromised the dynamic state again after healing,} \\
&(\exists\ \sharp j\ .\ \sharp i1 < \sharp j\ \wedge \\
&\mathsf{CompromiseA}(uid_A,\ id_A,\ uid_B,\ id_B,\ \ldots)\ @\sharp j\ )\ ) \\
&\quad \textit{// compromised the static state,} \\
&\|(\exists\ \sharp j\ .\ \mathsf{CompromiseDevice}(uid_A,\ id_A,\ )\ @\sharp j\ )\ ) \\
&\quad \textit{// or compromised the state of the partner.} \\
&\|(\exists\ \sharp j\ .\mathsf{CompromiseB}(uid_A,\ id_A,\ uid_B,\ id_B,\ \ldots)\ @\sharp j\ )\ ) \\
&\|(\exists\ \sharp j\ .\mathsf{CompromiseDevice}(uid_B,\ id_B,\ \ldots)\ @\sharp j\ )\ )
\end{aligned}
$$

For simplicity of presentation we have omitted some details, e.g., HealingA is a shorthand for the role switching that is required to heal A. During role switching A introduces new entropy in their dynamic state. The formal property specifies that after A has healed inside of a session, the attacker can only learn the secret used in a HonestStep that occurred in *any* protocol session after the heal, if they compromise the dynamic or static state of the victim again after the heal (not protected by PCS), or they compromised the partner.

Proving Strategy. To prove the property, we used 14 intermediate lemmas, and Tamarin's *tactics* feature to prioritize state facts related to the respective user a lemma is concerned with. When proving the PCS property, we prioritized the actions of the attacker, e.g., compromising devices and dynamic states.

We prove the conversation PCS property automatically in Tamarin's graphical user interface (GUI) in around 1 hour of time on a ThinkPad X1 Carbon Gen 9 with 32Gb of RAM. Using the GUI was necessary as we struggled with memory limitations using Tamarin's command-line interface. The GUI remedies this by allowing us to *cut* the proof into disjoint case-distinctions which we can then prove separately; needing only a fraction of the memory.

Reduced legacy messages attack vector. Finally, we also prove that the Proposal model does not contain traces where an end-user receives *legacy* messages from older sessions ($\mathsf{N} > 2$).

Formally, these traces are characterized by the following statement:

$$
\begin{aligned}
&\exists\ sid_1\ sid_2\ sid_3\ uid_A\ id_A\ uid_B\ id_B\ \sharp i\ \sharp j\ \sharp k\ \sharp l\ \sharp m\ .\\
&\ \sharp i < \sharp k\ \wedge \sharp k < \sharp l\ \wedge\\
&\mathsf{CreateDynState}(sid1,\ uid_A,\ id_A,\ uid_B,\ id_B,\ \ldots)@\sharp i\ \wedge\\
&\mathsf{AttackerSendMsg}(sid1,\ uid_A,\ id_A,\ uid_B,\ id_B,\ \ldots)@\sharp j\ \wedge\\
&\mathsf{CreateDynState}(sid2,\ uid_A,\ id_A,\ uid_B,\ id_B,\ \ldots)@\sharp k\ \wedge\\
&\mathsf{CreateDynState}(sid3,\ uid_A,\ id_A,\ uid_B,\ id_B,\ \ldots)@\sharp l\ \wedge\\
&\mathsf{ReceiveLegacyMsg}(sid1,\ uid_A,\ id_A,\ uid_B,\ idb,\ \ldots\ @\sharp m
\end{aligned}
$$

The property encodes the existence of a particular attack, and states that is it possible that: if there are two new sessions being created, *sid2* at timepoint $\sharp k$, and *sid3* at timepoint $\sharp l$, then the user can still accept messages sent by an attacker from a previous session *sid1*. For the model with with our policy restrictions, Tamarin automatically disproves this property almost instantly (5s), i.e., proving no such trace exists. In contrast, when removing the $\mathsf{N} = 2$ policy restriction from the model, we can prove the property true and display a corresponding attack trace in the GUI in 5s. The absence of these traces in the Proposal model shows that our proposal reduces the attack surface.

6 Discussion

6.1 The design space of the number of stored sessions: Demystifying Signal's 40 sessions

One interesting question is related to the number of sessions the systems need to store and what its implications are for resilience and security.

Consider a system that immediately discards the older session when creating a new one, $\mathsf{N} = 1$. The following scenario would take place: A and B independently start new sessions at the same time, and upon receiving the message from their partner, they discard the local session and accept the incoming one. Now, A sends messages using the session started by B, $D^B_{0,0}$, however B has deleted that session and cannot decrypt. All the messages received will trigger a retry request to be re-encrypted in a new session. Figure 3 depicts the described scenario. The same logic applies when A, respectively B, refreshes their session, and all in-flight messages from the older one, trigger a retry request.

Since A and B communicate asynchronously, it is possible that messages keep crossing. In addition, if A loses the dynamic state M times, a message sent from a session before the loss, $\mathsf{N}_{\mathsf{sender}} < \mathsf{N}_{\mathsf{current}} - \mathsf{M}$, will trigger a retry request upon being received. When apps evaluate that end-users refresh their sessions very frequently, e.g., faulty hardware, and do not want the retransmitting overhead, the number of stored session can be modified to their needs, $\mathsf{N} = \mathsf{M}$. However, storing a large N leads to the attacker being able to misuse any of those sessions to inject messages as shown in previous works [20, 42]. The parameter N can be seen as a trade-off between the resilience against an unreliable environment, and the attack vector of using older sessions to circumvent the healing of a specific session. The Signal messenger (current version: Android app v7.24.2, November 2024, and any previous versions) stores 40 session at one time. Following our previous argument, the number of sessions equals the window of tolerance of the app to receive out-of-order messages from previous sessions and decrypt without retransmitting messages. Previously, Signal (Android app v5.40.4, May 2022) restarted sessions every one hour, which translates to accepting messages without retransmitting from the last 40 hours, or last 40 active sessions. Any other messages, would trigger a retry request and therefore a new session.

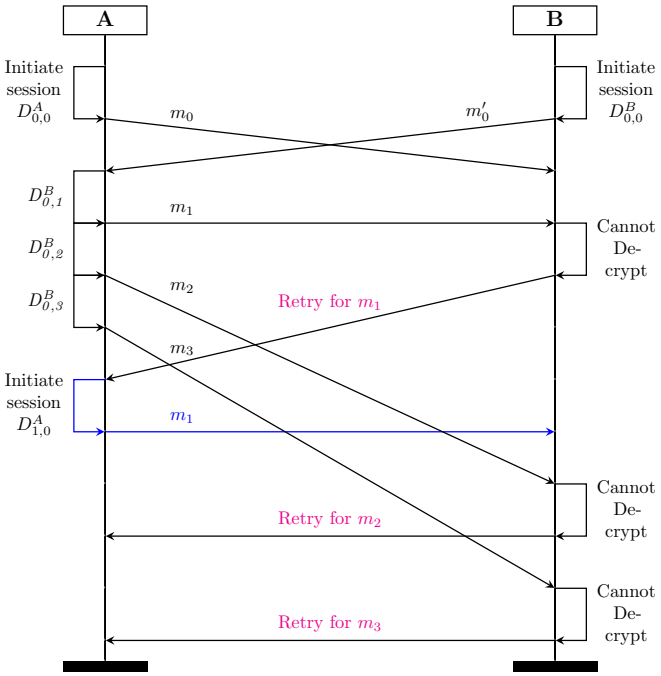

Figure 3: **Round-trips overhead on storing $\mathsf{N} = 1$ session.** A and B storing a single session triggers retry requests on out-of-order messages from the previous session. If parties stored $\mathsf{N} = 2$ sessions, they are decrypted upon being received.

At the current version, Signal does not refresh their session every hour, so this parameter has remained unchanged because of legacy code, and it translates to accepting messages in the latest 40 updates from which they received or sent a message.

6.2 Towards compromise detection

From our results that PCS cannot be achieved at the conversation layer, it becomes clear that the problem repeats itself in detecting the compromise. The intuition behind it is similar, allowed operations due to failure modes enable the attacker to mimic these behaviors and become indistinguishable from the victim. Any deployable detection mechanisms would need to rely on an out-of-band channel or require the explicit end-user's involvement, to detect at least a *subset* of the more trivial attacks.

We can observe this already in the literature, however substantial work is needed in the future to formally prove the claim. Principles and foundations for compromise detection were proposed in [34], but this work did not consider potential failure modes such as state loss. More recently, [8] develops a detection mechanism against an active attacker using ordinals (the epoch and message numbers of the Double Ratchet) and also a hash of the message transcript. Under the realistic assumptions of real-world implementations, their construction yields *false positives*, i.e., an attacker will be flagged over normal operations of the app. Consider the simple example, where

A restores an old backup, starts a new session with partner B and sends a message. This will incorrectly result in a compromise detection, since the ordinals and the hashed transcript do not match. Additionally, if the scheme is deployed strictly per session, the attacker can circumvent it by spawning a new session, while remaining undetected on the session between A and B. Similarly in [18, 20, 34], the mechanisms either do not consider multi-sessions or desynchronisation. If these mechanisms are used with multiple sessions or in the presence of failure modes, this typically either introduces false positives or possibility circumvented, thus rendering them ineffective in practice.

7 Conclusions

We have shown that real-world communication systems cannot at the same time provide full conversation PCS as well as resilience against specific forms of state loss, such as loss of dynamic state. However, these forms of state loss are unavoidable in real world deployment for a variety of reasons, including memory glitches or OS-level backups. For designs like Signal and Matrix that use concurrent sessions, we have shown a trade-off between resilience and security.

It is tempting to think that rotating long-term keys can circumvent these problems, but such solutions essentially suffer from the exact same impossibility result under slightly modified conditions on state loss. In the end, it seems that achieving stronger PCS guarantees requires to prevent, or more likely, assume that state loss occurs extremely infrequently, in which case one might be able to rely on external assumptions such as parallel out-of-band (OOB) solutions. However, most real-world communication systems cannot realistically assume that OOB mechanisms can be arbitrarily and immediately invoked, thereby leading to a trade-off with availability (and in case of human end-users, usability).

If full PCS cannot be achieved, another option is to focus instead on compromise detection. While we argued in Section 6.2 that detection suffers from similar problems, it may still be possible to achieve in some cases. However, before designers consider this direction, they must first consider the question: if a potential compromise were to be detected, what would be the possible consequences, given that it is usually impossible to differentiate the honest party from the adversary? In high-end security settings or with high-risk users, aborting communications entirely and relying on OOB mechanisms might be possible. However, where high availability or usability is the priority, this may not be a realistic option.

References

[1] Martin R. Albrecht, Benjamin Dowling, and Daniel Jones. Device-Oriented Group Messaging: A Formal Cryptographic Analysis of Matrix' Core. In *SP*, pages 2666–1685. IEEE, 2024.

[2] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, and Krzyzstof Pietrzak. DeCAF: Decentralizable CGKA with Fast Healing. In *SCN (2)*, volume 14974 of *Lecture Notes in Computer Science*, pages 294–313. Springer, 2024.

[3] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. In *EUROCRYPT (1)*, volume 11476 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2019.

[4] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous Group Key Agreement with Active Security. In *TCC (2)*, volume 12551 of *Lecture Notes in Computer Science*, pages 261–290. Springer, 2020.

[5] David Balbás, Daniel Collins, and Phillip Gajland. WhatsUpp with Sender Keys? Analysis, Improvements and Security Proofs. 14442:307–341, 2023.

[6] David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic Administration for Secure Group Messaging. In *USENIX Security Symposium*, pages 1253–1270. USENIX Association, 2023.

[7] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol, 2023.

[8] Khashayar Barooti, Daniel Collins, Simone Colombo, Loïs Huguenin-Dumittan, and Serge Vaudenay. On active attack detection in messaging with immediate decryption. In *CRYPTO (4)*, volume 14084 of *Lecture Notes in Computer Science*, pages 362–395. Springer, 2023.

[9] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted Encryption and Key Exchange: The Security of Messaging. In *CRYPTO (3)*, volume 10403 of *Lecture Notes in Computer Science*, pages 619–650. Springer, 2017.

[10] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the Price of Concurrency in Group Ratcheting Protocols. In *TCC (2)*, volume 12551 of *Lecture Notes in Computer Science*, pages 198–228. Springer, 2020.

[11] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A More Complete Analysis of the Signal Double Ratchet Algorithm. In *CRYPTO (1)*, volume 13507 of *Lecture Notes in Computer Science*, pages 784–813. Springer, 2022.

[12] Olivier Blazy, Angèle Bossuat, Xavier Bultel, Pierre-Alain Fouque, Cristina Onete, and Elena Pagnin. SAID: Reshaping Signal into an Identity-Based Asynchronous Messaging Protocol with Authenticated Ratcheting. In *EuroS&P*, pages 294–309. IEEE, 2019.

[13] Olivier Blazy, Ioana Boureanu, Pascal Lafourcade, Cristina Onete, and Léo Robert. How fast do you heal? A taxonomy for post-compromise security in secure-channel establishment. In *USENIX Security Symposium*, pages 5917–5934. USENIX Association, 2023.

[14] Sébastien Campion, Julien Devigne, Céline Duguey, and Pierre-Alain Fouque. Multi-Device for Signal. In *ACNS (2)*, volume 12147 of *Lecture Notes in Computer Science*, pages 167–187. Springer, 2020.

[15] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A Formal Security Analysis of the Signal Messaging Protocol. *J. Cryptol.*, 33(4):1914–1983, 2020.

[16] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On Post-compromise Security. In *CSF*, pages 164–178. IEEE Computer Society, 2016.

[17] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees. In *CCS*, pages 1802–1819. ACM, 2018.

[18] Cas Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. Clone Detection in Secure Messaging: Improving Post-Compromise Security in Practice. In *CCS*, pages 1481–1495. ACM, 2020.

[19] Cas Cremers, Britta Hale, and Konrad Kohbrok. The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter. In *USENIX Security Symposium*, pages 1847–1864. USENIX Association, 2021.

[20] Cas Cremers, Charlie Jacomme, and Aurora Naska. Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations. In *USENIX Security Symposium*, pages 1235–1252. USENIX Association, 2023.

[21] Cas Cremers, Niklas Medinger, and Aurora Naska. PCS Analysis Models. https://github.com/pcsanalysiseprint/pcsanalysiseprint, November 2024.

[22] Benjamin Dowling and Britta Hale. Authenticated Continuous Key Agreement: Active MitM Detection and Prevention. 2023. URL: https://eprint.iacr.org/2023/228.

[23] F. Betül Durak and Serge Vaudenay. Bidirectional Asynchronous Ratcheted Key Agreement with Linear Complexity. 2018. URL: https://eprint.iacr.org/2018/889, doi:10.1007/978-3-030-26834-3_20.

[24] Google Messages. Technical Paper. online, Accessed: 26.09.2024. URL: https://www.gstatic.com/messages/papers/messages_e2ee.pdf.

[25] Joseph Jaeger and Igors Stepanovs. Optimal Channel Security Against Fine-Grained State Compromise: The Safety of Messaging. In *CRYPTO (1)*, volume 10991 of *Lecture Notes in Computer Science*, pages 33–62. Springer, 2018.

[26] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient Ratcheting: Almost-Optimal Guarantees for Secure Messaging. In *EUROCRYPT (1)*, volume 11476 of *Lecture Notes in Computer Science*, pages 159–188. Springer, 2019.

[27] Julia Len, Esha Ghosh, Paul Grubbs, and Paul Rösler. Interoperability in End-to-End Encrypted Messaging. 2023. URL: https://eprint.iacr.org/2023/386.

[28] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility. In *USENIX ATC*. USENIX Association, 2010.

[29] Felix Linker, Ralf Sasse, and David Basin. A Formal Analysis of Apple's iMessage PQ3 Protocol. Cryptology ePrint Archive, Paper 2024/1395, 2024. URL: https://eprint.iacr.org/2024/1395.

[30] Moxie Marlinspike and Trevor Perrin. The Sesame Algorithm: Session Management for Asynchronous Message Encryption. 2017. URL: https://signal.org/docs/specifications/sesame/.

[31] Moxie Marlinspike and Trevor Perrin. The X3DH Key Agreement Protocol. Accessed: 26.09.2024. URL: https://signal.org/docs/specifications/x3dh/.

[32] Marlinspike, Moxie. Simplifying OTR deniability. online, Accessed: 26.09.2024. URL: https://signal.org/blog/simplifying-otr-deniability/.

[33] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

[34] Kevin Milner, Cas Cremers, Jiangshan Yu, and Mark Ryan. Automatically Detecting the Misuse of Secrets: Foundations, Design Principles, and Applications. In *CSF*, pages 203–216. IEEE Computer Society, 2017.

[35] Trevor Perrin and Moxie Marlinspike. The Double Ratchet Algorithm. Accessed: 26.09.2024. URL: https://signal.org/docs/specifications/doubleratchet/.

[36] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema. In *EuroS&P*, pages 415–429. IEEE, 2018.

[37] Signal Messenger. Technical Specification. online, Accessed: 26.09.2024. URL: https://signal.org/docs/.

[38] Douglas Stebila. Security analysis of the iMessage PQ3 protocol. Cryptology ePrint Archive, Paper 2024/357, 2024. URL: https://eprint.iacr.org/2024/357.

[39] The Matrix.org Foundation. Matrix specification: Client-Server API. online, Accessed: 26.09.2024. URL: https://spec.matrix.org/v1.11/client-server-api/.

[40] The Matrix.org Foundation. Megolm. online, Accessed: 26.09.2024. URL: https://gitlab.matrix.org/matrix-org/olm/-/raw/master/docs/megolm.md.

[41] The Matrix.org Foundation. Olm. online, Accessed: 26.09.2024. URL: https://gitlab.matrix.org/matrix-org/olm/-/raw/master/docs/olm.md.

[42] Andreas Tsouloupas. Breaking Cryptography in the Wild: The Loose Ends of the Wire. Master's thesis, ETH Zurich, 2023. URL: https://doi.org/10.3929/ethz-b-000673362.

[43] WhatsApp. Technical White Paper. online, Accessed: 26.09.2024. URL: https://www.whatsapp.com/security/.

[44] Wire. Proteus. online, Accessed: 26.09.2024. URL: https://github.com/wireapp/proteus.