Clone Detection in Secure Messaging: Improving Post-Compromise Security in Practice

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

Jaiden Fairoze
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
jfairoze@student.unimelb.edu.au

Benjamin Kiesl
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
benjamin.kiesl@cispa.saarland

Aurora Naska
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
s8aunask@stud.uni-saarland.de

ABSTRACT

We investigate whether modern messaging apps achieve the strong post-compromise security guarantees offered by their underlying protocols. In particular, we perform a black-box experiment in which a user becomes the victim of a clone attack; in this attack, the user's full state (including identity keys) is compromised by an attacker who clones their device and then later attempts to impersonate them, using the app through its user interface.

Our attack should be prevented by protocols that offer post-compromise security, and thus, by all apps that are based on Signal's double-ratchet algorithm (for instance, the Signal app, WhatsApp, and Facebook Secret Conversations). Our experiments reveal that this is not the case: most deployed messaging apps fall far short of the security that their underlying mechanisms suggest.

We conjecture that this security gap is a result of many apps trading security for usability, by tolerating certain forms of desynchronization. We show that the tolerance of desynchronization necessarily leads to loss of post-compromise security in the strict sense, but we also show that more security can be retained than is currently offered in practice. Concretely, we present a modified version of the double-ratchet algorithm that tolerates forms of desynchronization while still being able to detect cloning activity. Moreover, we formally analyze our algorithm using the Tamarin prover to show that it achieves the desired security properties.

CCS CONCEPTS

• **Security and privacy → Formal security models**; **Logic and verification**; **Privacy-preserving protocols**.

KEYWORDS

security protocols, secure messaging, clone detection, formal verification, double ratchet, post-compromise security, forward secrecy

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11…$15.00
https://doi.org/10.1145/3372297.3423354

ACM Reference Format:
Cas Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. 2020. Clone Detection in Secure Messaging: Improving Post-Compromise Security in Practice . In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3372297.3423354

1 INTRODUCTION

The advent of modern secure messaging, and the widespread deployment of the Signal protocol library in particular, has brought modern security mechanisms to millions of users. Specifically, this includes the *double-ratchet* algorithm [23] and the security properties it can provide, such as *post-compromise security* [6], which provides security guarantees even after a communicating party's secrets have been leaked.

As a result, the Signal protocol and many variations have been studied extensively in the literature [1, 2, 4, 5, 10, 19]. These studies have shown that (a) the security properties are subtle, and can be strengthened in various ways, but more importantly, (b) the underlying design of the Signal protocol provably achieves a strong form of post-compromise security: if Blake's complete state (including encryption and signing keys) is compromised by an attacker at some point, but Alex and Blake have a successful "healing" exchange afterwards, the attacker is locked out of the communication again.

The Signal protocol library can achieve this strong property through an intricate mechanism called the double ratchet. Despite the additional engineering complexity stemming from the double ratchet, the Signal protocol library is successfully used in many modern messaging systems, such as WhatsApp, the Signal app, Facebook's Secret Conversations, and Skype. Moreover, other messengers—such as Viber and Wire—use custom implementations of the Signal protocol. Together, these apps represent the vast majority of secure-messaging users. This suggests that these users now enjoy strong post-compromise security guarantees.

For this work, we investigated whether this is indeed the case, i.e., whether these modern messaging apps do indeed provide the security guarantees offered by the double ratchet. To do so, we first performed a black-box study of secure-messaging apps with respect to one of the weakest possible threats against post-compromise security, namely *clone attacks*. In such an attack, Evan compromises Blake's state by cloning Blake's device. At some point in the future,

Evan then tries to attack Blake by running the messaging app from the cloned state in order to (a) learn messages exchanged by Blake and their partner Alex, or (b) inject messages into the communication. Now, if an app does indeed provide post-compromise security, and if Alex and Blake manage to heal before the attack, this attack should be impossible; in other words, post-compromise security should ensure the absence of clone attacks after healing.

We establish that surprisingly, in practice, several of the Signal-based apps offer only partial (or even no) post-compromise security against these relatively simple attacks, in multiple dimensions: some apps provide no practical guarantees for messages post-healing, whereas others provide some form of warning to their users. This suggests that some apps gain very little to no additional security from the complex mechanisms in the Signal protocol library.

We conjecture that these reduced guarantees are the result of mechanisms that trade security for usability: the double ratchet requires intricate synchronization, and real-world deployments may struggle to reliably update and maintain state. If this state is lost, or synchronization fails for some reason, a strict post-compromise security mechanism would block the conversation. While this may be acceptable for security-sensitive users, apps with a wide demographic may instead choose to tolerate desynchronization at the cost of reduced security. In fact, we know from discussions with developers of a popular app that this is the case for their app.

In the second half of this work, we explore the theoretical foundations for this apparent dilemma and provide a solution that tolerates forms of desynchronization while still preventing post-compromise security threats such as clone attacks.

In particular, we show that a messaging system that tolerates certain types of non-malicious synchronization failures cannot achieve full post-compromise security guarantees. We identify and formalize common non-malicious desynchronization behaviors, from which we infer how to design mechanisms that tolerate them without losing all post-compromise security guarantees. We provide a concrete mechanism that detects clone attacks, and formally prove its properties using an automated prover. The soundness proof for our mechanism relies on the assumption that messages arrive in order, meaning that we trade the double-ratchet's message-loss resilience for clone detection.

The main contributions of this paper:

- We conduct an experiment showing that popular messaging apps—in contrast to their underlying protocols—do not provide post-compromise security even under a simple device-cloning attack. We analyze in detail how this impacts the security of messages communicated during and after healing.
- We demonstrate that a protocol cannot achieve post-compromise security if it tolerates certain forms of desynchronization of the communicating parties.
- We propose a desynchronization-tolerant version of Signal's double-ratchet algorithm that allows users to detect when their partner was cloned. For example, for apps that prioritize usability over security, this allows a light-weight response in the specific case of a cloning attack, without introducing warnings for other common failures; for apps with security-savvy users, detecting a clone could lead to a recommendation to generate a new identity key. We analyze our proposed algorithm formally and prove that it achieves sound clone detection.

Our formal models are available from https://github.com/dr-clone-detection/model.

The rest of this paper is structured as follows. In Section 2, we present necessary background as well as related work. In Section 3, we present our black-box experiment and discuss its outcomes in detail before we then propose our approach for clone detection—a modified version of the double ratchet—and its formal analysis in Section 4. Finally, we summarize our findings in Section 5.

2 BACKGROUND AND RELATED WORK

2.1 Related Security Properties

We briefly review the notions of *post-compromise security* and *forward secrecy*, which are achieved by Signal's double-ratchet algorithm. Intuitively, the former captures security guarantees *after* a party was compromised, whereas the latter captures security guarantees *before* a party was compromised, as illustrated in Figure 1.

Post-Compromise Security. The notion of *post-compromise security*, also known as *backward secrecy*, *future secrecy*, and *channel healing*, informally says that a party A has a security guarantee about communication with another party B, even after B's secrets have been compromised [6]. Note that post-compromise security does not say that A will have a security guarantee *immediately* after the compromise; in actual protocols that achieve post-compromise security, A has a security guarantee once some kind of *healing* has occurred after the compromise of B's secrets.

Even though this might seem unintuitive at first, there are in fact protocols that achieve forms of post-compromise security. As with forward secrecy, *how* post-compromise security is realized in practice depends on the precise security guarantees a protocol aims to achieve, and in particular, on the actual secrets that are compromised. Two typical cases are *session-key compromise* and *full local-state compromise*.

In the case of session-key compromise, only ephemeral cryptographic material is leaked to the attacker. Here, post-compromise security can be provided by a *key-evolving scheme*, i.e., a mechanism that computes the session key using some secret information from the previous session. For instance, given a key-derivation function *KDF* (i.e., a one-way function that derives one or more secret keys from its input), the i-th session key sk_i could be computed from the previous session key sk_{i-1} and a token t_i. Here, the token t_i could, for example, be a shared secret established by the two honest parties via a Diffie-Hellman key exchange. If the session key is evolved in this way, the attacker cannot compute a future session key with only the knowledge of the current session key.

In the case of full local-state compromise, all local user data, including long-term keys, is leaked to the attacker. Thus, achieving post-compromise security is much harder in practice. In fact, in the context of secure messaging, it requires the use of public-key cryptography for the derivation of message keys [10, 19]. This is because any key-evolving scheme based on symmetric cryptography satisfies the following: given the previous state and all publicly exchanged information, it is possible to compute the next state.

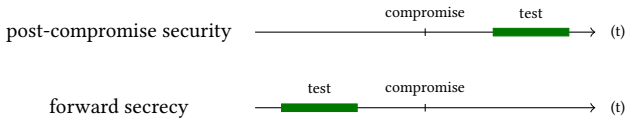

Figure 1: Post-compromise security and forward secrecy.

Forward Secrecy. The notion of *forward secrecy*, also known as *perfect forward secrecy*, informally says that ciphertexts that were sent or received *prior* to the compromise of a party remain secure after the compromise [3, 15]. For instance, imagine A and B are exchanging encrypted messages until some timepoint t at which an attacker compromises A by stealing A's secret keys—forward secrecy guarantees that the attacker will not be able to decrypt any of the messages exchanged between A and B *before* timepoint t. The exact mechanism used to provide forward secrecy depends on the level of granularity at which a protocol aims to achieve forward secrecy. Typical levels of granularity are *per message* or *per session*.

In the per-message case, forward secrecy can be achieved by evolving the encryption keys for every sent message. This is commonly achieved by deriving new keys from old keys via a one-way function. If keys are regularly updated, messages that were encrypted with previous keys stay secure even if the current key is compromised. Note that other mechanisms that achieve forward secrecy have been proposed in the literature, such as *puncturable encryption* [8, 14, 16] and *time-based methods* [9].

In the per-session case, the communicating partners exchange *ephemeral* session keys in typical protocols. Using public-key cryptography, a fresh session key is generated independently of the previous session key, and thus cannot be computed given future session keys. To reduce the expensive public-key operations, parties can derive an initial session key and, similar to the per-message case, *evolve* it throughout the conversation, e.g., by using a key-derivation function.

2.2 The Double Ratchet

We give a brief overview of Signal's double-ratchet algorithm for secure messaging; for full details, we refer to the official documentation [23] or some of the protocol's proofs [1, 5]. As the name already indicates, the double ratchet uses two separate *ratchets*—the so-called *Diffie-Hellman ratchet* and the *symmetric-key ratchet*.

Before the start of a conversation, the two parties, A and B, initialize a *session* by establishing a common secret called the *root key*. This is achieved with a separate key-agreement protocol that is not part of the double ratchet; in Signal, the so-called *Extended Triple Diffie-Hellman* (X3DH) protocol is used [20]. As part of the initialization, B also computes a private share x and a corresponding public share g^x, and then sends the public share to A. The private and public shares will be important for the Diffie-Hellman ratchet later on. Once they have finished the initialization, A and B can start exchanging messages. The idea is that every message is encrypted and decrypted with a distinct *message key*, which is derived by both parties from a so-called *symmetric chain key* that itself is derived from the root key. Thus, there is a hierarchy of keys (root key > symmetric chain key > message key) whereby the Diffie-Hellman ratchet takes care of updating the root key and computing *initial* symmetric chain keys, whereas the symmetric-key ratchet takes care of updating the symmetric chain keys and deriving new message keys as illustrated in Figure 2.

Diffie-Hellman Ratchet. At the highest level of the key hierarchy, a party updates the root key using the Diffie-Hellman ratchet every time they switch between *sending* and *receiving* messages. For example, if A has just received one or more messages and then decides to send messages, A first updates their root key; after that, A can then send multiple messages without updating the root key, until they switch to receiving messages again. Likewise, on the opposite side, B switches from sending to receiving and thus updates the root key accordingly.

For the update of the root key, a party first generates a new private share y and a corresponding public share g^y. Using their partner's current public share g^x (which they know either from the initialization or from their partner's last message), they then compute a new *shared secret* $(g^x)^y = g^{xy}$. After this, they input both the shared secret and the current root key rk_{i-1} into a key-derivation function *KDF*, which returns two values, namely the new root key rk_i and a new *initial* symmetric chain key sk_i as $\langle rk_i, sk_i \rangle = KDF(rk_{i-1}, g^{xy})$. Thus, for every update of the root key, some secret information is added (via the shared secret) to achieve post-compromise security. Importantly, the *initial* symmetric chain key, sk_i, is then used in the symmetric-key ratchet, which takes us one level lower in the key hierarchy.

Symmetric-Key Ratchet. The symmetric chain key is used to derive the actual message keys via the symmetric-key ratchet. In particular, whenever a party sends or receives a new message, they derive a distinct message key mk_i from the symmetric chain key and then also update the symmetric chain key. This is done by simply applying a *KDF* again: $\langle sk_{i+1}, mk_{i+1} \rangle = KDF(sk_i)$. For example, suppose A has just updated their root key with the Diffie-Hellman ratchet and thereby derived an initial symmetric chain key sk_0. To encrypt a new message, A now derives a message key mk_1 and a new symmetric chain key sk_1 from the initial symmetric chain key by computing $\langle sk_1, mk_1 \rangle = KDF(sk_0)$. For their next message, A then derives $\langle sk_2, mk_2 \rangle = KDF(sk_1)$. A can now repeatedly do this until they decide to switch from sending to receiving again. Likewise, on the other side, B will derive the message keys in the exact same way. By evolving the symmetric chain keys and the message keys with the *KDF*, forward secrecy is achieved because the *KDF* is a one-way function.

Additional Notes on the Shared Secrets. Note that every message must contain the current sender's public share g^y to allow the receiver to compute the shared secret g^{xy} required to update the root key. The two parties thus take turns incorporating new secret shares into the root key. To see this, suppose A (who knows B's public share g^x after initialization) starts as a sender. A derives a new root key rk_1 from the initial root key rk_0 by first generating a pair $\langle y, g^y \rangle$, then computing the shared secret g^{xy}, and finally applying the key-derivation function: $\langle rk_1, sk_1 \rangle = KDF(rk_0, g^{xy})$. When B receives A's message, B can also compute rk_1 by computing the shared secret g^{xy} from g^y and B's own private share x. If B now decides to send a message, B generates $\langle z, g^z \rangle$ and then computes

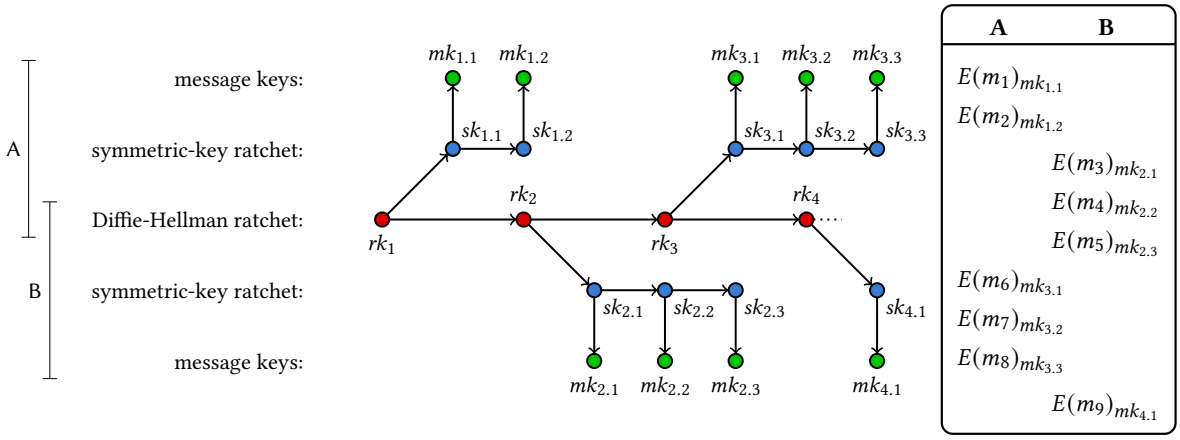

Figure 2: Key structure of the double ratchet during a conversation. Left: Evolution of the keys. Right: A's view of the conversation ($E(m)_{mk}$ denotes the encryption of message m with message key mk). All keys can be computed by *both* participants, allowing the receiver to decrypt messages encrypted by the sender. Whenever participants switch the roles of sender and receiver, they update the root key rk with the Diffie-Hellman ratchet. The symmetric-key ratchet computes a new symmetric chain key sk and a new message key mk whenever a new message is sent.

$\langle rk_2, sk_2 \rangle = KDF(rk_1, g^{yz})$. Consequently, B has replaced the x-part of the initial shared secret g^{xy} by z. In A's next turn, A would then replace the y-part by a new secret share, and so on—the shared secret is thus continuously updated with new secret information from either party.

Note that the mechanism we have described here, where exactly one public share is appended to messages, is the one used by the official double-ratchet algorithm of the Signal protocol. With this mechanism, if an attacker compromises the full state of a victim, it learns the victim's private share, meaning that at least one round of back-and-forth between the honest communication parties is required to heal the channel (since the attacker does then not know the new private share). The Wickr messenger performs more aggressive ratcheting, making the parties exchange more than one public share per message. This has a number of security implications: the messages sent by the victim will be immediately private as long as a fresh pair of shares is sampled at encryption time. However, the messages received by the victim are not immediately private, because the attacker knows the pre-shared share pairs exchanged before compromise. The time required for healing after compromise then depends on the exact protocol specification—Wickr does not explicitly specify this parameter in their technical white paper [18] or on their website (https://wickr.com/).

In theory, the double ratchet enables strong security guarantees: even if the full state with the identity key is compromised, the protocol can regain security. In practice, the security of the (identity key) storage varies widely: from secure storage on modern phones to substantially less secure storage on older hardware (as used in poorer regions) or desktop clients.

2.3 Related Work on Clone Detection

After a compromise, if the attacker becomes active and impersonates the victim, there is a chance that their actions create discord with those of the victim. This discord could be identified and then used to notify the victim that there was a clone attack. The existing proven solutions in clone detection use exactly this idea of forcing the attacker to leave traces of their actions, assuming that the attacker does not merely eavesdrop but injects their own messages.

One way to achieve this, suggested by Yu et al. [29], is with a third party that records all honest activity. The idea behind the mechanism is that both parties send certificates of their ephemeral encryption keys to an append-only log maintainer. Parties then query the log maintainer to verify the received certificates from their partner and to obtain proof that no entries of their own have been added or deleted from the log.

Suppose an attacker compromises a party and uses all existing ephemeral keys whose certificates are already published in the log. If they want to continue the impersonation they have to add their own keys to the log maintainer. This in turn leads to clone detection once the victim queries the log and detects new entries by another author. In addition, if the log maintainer is malicious, the parties can detect the maintainer's activities by "gossiping" among each other about the current status of their conversation log.

The drawback of this scheme is that the log maintainer becomes a bottleneck that requires a form of gossiping unless it is trusted. Thus, it is desirable to have a protocol that allows for clone detection without a trusted entity. Such solutions—based on *counters*, *hashing*, or *commitments*—were proposed by Milner et al. [22].

In their counter-based mechanism, the parties store the number of messages exchanged with their partner, and append the number to each message they send. Upon receiving a message, they increment their local value and compare it with the received value. If the two numbers do not match, a detection event is raised.

In the hash-based approach, the parties—instead of counting the messages exchanged—keep a hash chain and evolve it with every message they send, using the previous value of the hash and a

fresh nonce received from their partner. Again, failure to verify the received hash value with the local one will lead to detection.

Finally, in the commitment-based approach, parties tightly couple their current message with their next one by encrypting current session data with a key pair used to encrypt the next message. This encrypted session data serves as a commitment, and after successful decryption of the next message, it gets verified using the same key. If data verification fails, a clone is detected since the partner broke its commitment, thus deviating from normal behavior.

2.4 The Tamarin Prover

We use the Tamarin prover [21] for proving the critical properties of our clone-detection approach in Section 4.2. Tamarin is an automated-reasoning tool for the analysis of complex security protocols. Tamarin operates on the symbolic level, meaning that bit strings are abstracted to algebraic terms. As Tamarin is particularly well-suited for modeling complex state machines with loops and evolving state, it is a natural choice for modeling our modified version of Signal's double ratchet.

To formalize a security protocol in Tamarin, we encode the protocol as a collection of multiset-rewriting rules. Once we have encoded a protocol by a set of such rules, we can specify desired properties in a guarded fragment of many-sorted first-order logic

Tamarin tries to prove the property by refuting its negation. In case Tamarin terminates, it either outputs a proof (if the statement is true) or a counterexample (if the statement is false). A proof is provided in the form of a proof tree, whereas a counterexample is provided in the form of a trace, i.e., a sequence of steps that corresponds to a possible execution of the protocol. Proofs and counter-examples can be inspected in Tamarin's GUI.

3 BLACK-BOX EXPERIMENT

In this section, we describe our black-box experimental analysis of a range of messaging apps. Our goal is to analyze to what degree different messaging apps offer post-compromise security in the context of full-state compromise. In general, a protocol that provides post-compromise security should be resilient to a fully active attacker that controls the network, after the involved parties have healed from compromise. Since we are performing black-box analysis, we consider a strictly weaker attacker after the healing: the attacker executes a clone of the compromised state after the healing. If a protocol does not offer security with respect to this weaker attacker, it certainly does not offer post-compromise security.

In a nutshell, we consider a situation where two parties, A and B, use a messaging app to communicate with each other until a point where B's device is cloned (amounting to full-state compromise) so that there are then two seemingly identical clones B_{blake} and B_{evan}. After that, only one of the two clones, B_{blake}, continues to communicate with A, until a point where the other clone, B_{evan}, takes over. Once B_{evan} has taken over, we want to answer the following questions:

(A) Given B_{blake}'s full local state at the time of cloning, can B_{evan} read any of the messages that have been exchanged between B_{blake} and A?

(B) How does the messaging app react to the cloning situation? Does it inform the parties about unusual/suspicious behavior? Are there any errors? Or does the app just continue as if nothing happened?

We would expect that a messaging app which provides post-compromise security does not leak any messages exchanged between B_{blake} and A to B_{evan}. Moreover, while not strictly implied by post-compromise security, it would improve security if the app made the parties aware of the suspicious cloning behavior. As we will see, none of the messaging apps we considered behave that way.

In the following, we first explain the exact setup of our experiment before we analyze its outcomes.

3.1 Experimental Setup

For our evaluation, we selected messengers that explicitly claim strong security. We distinguish between

- messengers based on the Signal library (which implements the Signal protocol),
- messengers based on custom implementations of the Signal protocol, and
- messengers based on protocols different than Signal.

Table 1 gives an overview of all tested messengers, their protocols, and their version numbers. We discuss them briefly in the following.

Signal-Based Messengers (*Signal, WhatsApp, Facebook, and Skype*). Aside from Open Whisper System's Signal messenger, we considered three other messengers that directly interface with the Signal protocol. Namely, WhatsApp bases all communications through its service on the Signal protocol [28]. Moreover, Facebook Messenger and Skype have distinct features—called *Secret Conversations* [11] and *Private Conversations* [24], respectively—that support end-to-end encryption via the Signal protocol. All messengers in this group should in theory be able to achieve the same security as Signal.

Messengers with Custom Implementations of the Signal Protocol (*Viber, Wire*). Viber [27] and Wire [13] both claim that their respective protocols are in-house implementations of the standard Signal protocol. Therefore, like the messengers from the above group, Viber and Wire should be able to match Signal's security.

Messengers Based on Other Protocols (*Wickr, Olvid, Threema, and Telegram X*). Wickr [18], unlike all other assessed messengers, utilizes more aggressive ratcheting as discussed in Section 2.2. Olvid [12] is (according to a technical talk) designed to provide forward secrecy and post-compromise security through a custom algorithm, even though the official documentation does not explicitly claim a form of post-compromise security. Finally, Threema [26], and Telegram X [25] do not attempt to achieve any form of post-compromise security; we included them as a form of control.

Considered Devices. We considered Android devices as well as desktop clients. Our focus was on the Android devices since most secure-messaging applications target mobile devices. To emulate these devices, we used the Genymotion Android emulator due to its powerful cloning features. All emulated devices were modeled after Google Nexus 5X devices running Android 8.0 (API 26).

We used desktop clients to assess Wickr and Wire, which provide full support for the desktop. While these messengers had mobile versions, we used the desktop clients to circumvent compatibility issues. We ran the applications on Windows 10 virtual machines

	Signal Library				Custom Signal Implementation		Other Protocols			
	Signal[23]	WhatsApp[7]	FB SC[11]	Skype[24]	Viber[27]	Wire[13]	Wickr[18]	Olvid[12]	Threema[26]	Telegram X[25]
Protocol	`libsignal`	`libsignal`	`libsignal`	`libsignal`	Closed Source	`Proteus`	`wickr-crypto-c`	Closed Source	`threema-msgapi`	`MTProto`
App Version	V4.47.7	V2.19.274	239.1.0.17.119	V8.53.0.104	V11.6.3.4	3.10.3138.0	5.38.2	V0.7.6	V4.11k	V0.22.0.1205-x86

Table 1: Tested messengers with their underlying protocols and version numbers.

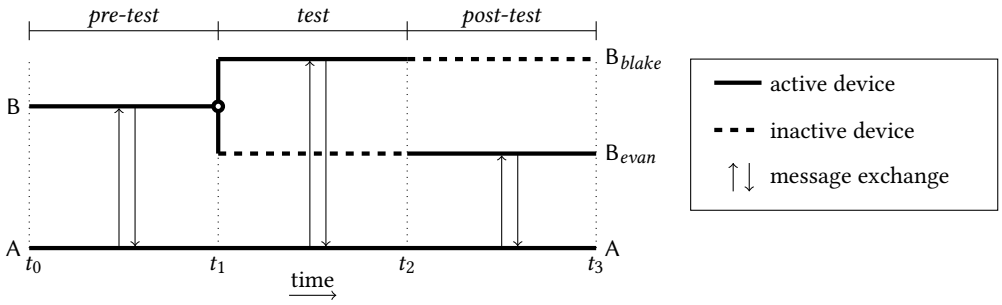

Figure 3: Phases of our experiment. In the *pre-test* phase, A and B exchange messages until t_1, where B is cloned, resulting in B_{blake} and B_{evan}. In the subsequent *test phase*, B_{blake} communicates with A until t_2, where B_{evan} takes over in the *post-test* phase. Table 2 shows which of the messages exchanged between A and B_{blake} in the test phase can be decrypted by B_{evan}. Table 3 summarizes how the devices of A and B_{evan} react to the messages exchanged in the post-test phase. Since we are interested in the healing behavior of apps with respect to confidentiality and authenticity, we interpret B_{blake} as the honest party that potentially heals during the *test* phase, and interpret B_{evan} as the clone.

created with VirtualBox. To clone a device, we cloned the entire virtual machine to duplicate the full local state. Note that for Wickr and Wire, the desktop clients can exist independently of a mobile device, as first-class devices.

3.2 Methodology

In our experiment, we consider two communicating parties, A and B. We use the notation $A \rightarrow B$ to denote that A sent a message to B. Moreover, given a sequence S of message exchanges, we write S^n to denote that S is repeated n times. For example, we write $(A \rightarrow B, B \rightarrow A)^3$ to denote that A and B sent messages back and forth three times in a row.

Our experiment consists of three phases, shown in Figure 3; we call them the *pre-test* phase, the *test* phase, and the *post-test* phase.

Pre-Test Phase. In the pre-test phase, we establish (for each messenger) a secure channel between the two parties, A and B. This involves registering a single device for each party with the service. After registration, we keep the default settings in all aspects except for so-called *read receipts* and *typing indicators*; the former indicate to the other party that a message has been read, the latter indicate that the other party is currently typing a message. We disabled these two features whenever possible, to prevent additional protocol messages from firing. Moreover, we made sure to grant the apps full device permissions to permit proper app function.

The timepoint t_0 marks the point where device configuration is completed. After that, A and B exchange a series of messages to move the conversation to an "active" state. Specifically, these pre-test messages are of the form $(A \rightarrow B, B \rightarrow A)^5$.

Test Phase. The test phase begins with the cloning event at t_1. Right before the cloning event, B is blocked from the network to ensure that states of the subsequent clones do not automatically update. We clone B's device in such a way that one clone, B_{blake}, joins the network while the other clone, B_{evan}, is kept inactive. This effectively mimics the continuation of the single device B, since from the outside it should appear as if just a single device left and rejoined the network. Once the cloning has been performed, A and B_{blake} exchange a few messages. These messages constitute the *test messages*, and are of the form $(A \rightarrow B_{blake}, B_{blake} \rightarrow A)^5$.

After the test messages have been exchanged, at timepoint t_2, the two clones switch status—B_{evan} becomes active and B_{blake} leaves the network. At this point, post-compromise security could potentially be violated if the initially-offline device were able to automatically retrieve messages from the test phase.

Post-Test Phase. In the post-test phase, A and B_{evan} conduct a final round of message exchanges. We use this phase to investigate whether or not conversation can continue after B_{blake} left. For the actual message exchange, we consider four different sequences of messages:

(1) $(A \rightarrow B_{evan}, B_{evan} \rightarrow A)^5$
(2) $(B_{evan} \rightarrow A, A \rightarrow B_{evan})^5$
(3) $(A \rightarrow B_{evan}, A \rightarrow B_{evan}, B_{evan} \rightarrow A, B_{evan} \rightarrow A)^5$
(4) $(B_{evan} \rightarrow A, B_{evan} \rightarrow A, A \rightarrow B_{evan}, A \rightarrow B_{evan})^5$

For each case, we carry out the corresponding message-exchange pattern on the devices. Specifically, when B_{blake}'s device is brought online, it is given a period of 30 seconds to allow for asynchronous update with the messaging platform. During this period, the device might receive messages from the test phase through some platform-specific mechanism whose details we are not concerned with in our black-box experiment. Note that some platforms might deliver only some (or even none) of the messages from the test phase to B_{blake}. After this period, we capture the application-level behavior of both parties.

			Message Number				
			1	2	3	4	5
Signal Library	Signal	A → B_{blake}	✓	✓	✓	✓	✓
		B_{blake} → A	✓	✓	✓	✓	✓
	WhatsApp	A → B_{blake}	✓	✓	✓	✓	✓
		B_{blake} → A	✓	✓	✓	✓	✓
	Facebook SC	A → B_{blake}	✗	✗	✗	✗	✗
		B_{blake} → A	✓	✓	✓	✓	✓
	Skype PC	A → B_{blake}	✗	✓	✓	✓	✓
		B_{blake} → A	✓	✓	✓	✓	✓
Custom Signal	Viber	A → B_{blake}	✓	✓	✓	✓	✓
		B_{blake} → A	✓	✓	✓	✓	✓
	Wire	A → B_{blake}	✗	✗	✓	✓	✓
		B_{blake} → A	✓	✓	✓	✓	✓
Other Protocols	Wickr	A → B_{blake}	✗	✗	✗	✗	✗
		B_{blake} → A	✓	✓	✓	✓	✓
	Olvid	A → B_{blake}	✓	✓	✓	✓	✓
		B_{blake} → A	✓	✓	✓	✓	✓
	Threema	A → B_{blake}	✓	✓	✓	✓	✓
		B_{blake} → A	✓	✓	✓	✓	✓
	Telegram X	A → B_{blake}	✗	✗	✗	✗	✗
		B_{blake} → A	✗	✗	✗	✗	✗

Table 2: Messages from the test phase. During the test phase, A and B_{blake} exchange messages. Afterwards, the attacker B_{evan} activates the clone. For each message from the test phase, ✓ denotes that the attacker B_{evan} does not obtain it, and ✗ denotes that the attacker learns it.

Before we discuss the results of our experiment, we want to highlight that the methodology of our experiment is guided by the double-ratchet's design, meaning that we set up conditions under which the double ratchet should provide post-compromise security and then test if this is indeed the case. The reader might want to keep this in mind when interpreting the results of the experiment, especially regarding Olvid and Wickr—two messengers that are designed to provide post-compromise security under possibly different conditions.

3.3 Results

As already mentioned, we are interested in the answers to the following two questions:

(A) Can B_{evan} read any of the messages that have been exchanged between B_{blake} and A in the test phase?

(B) How does a messaging app react to the cloning situation?

For question (A), the outcome of our experiment is summarized in Table 2. The results can be grouped into four categories:

(1) *Full Protection.* For some messengers, namely Signal, WhatsApp, Viber, Olvid, and Threema, *all* messages exchanged in the test phase remain secure. In contrast to the other messengers in that group, Threema does not use a double-ratchet-based algorithm, nor does it provide a mechanism (such as an asynchronous ratchet) to enforce post-compromise security—this shows that messengers can protect from coarse-grained adversaries with limited technical proficiency (i.e., high-level access to the platform) without complex asynchronous ratcheting.

(2) *Near-Full Protection.* Skype PC and Wire offer near-full protection—all messages except the first one (Skype PC) or the first two (Wire) from A to B_{blake} stay protected. Both messengers use double-ratchet-based cryptography, which should theoretically protect *all* messages except for the very first message from the test phase. Wire does not protect the second message from A to B_{blake}, which contradicts its theoretical level of security.

(3) *One-Sided Protection.* Facebook Secret Conversations and Wickr offer what we call *one-sided protection*: A's messages to B_{blake} are leaked to B_{evan}, but B_{blake}'s messages to A remain secure. In Wickr's case, a batch of public-key secrets is known to either party at any given time, as mentioned in Section 2.2. With this information, it is feasible that B_{evan} had all public-key shares from A and thus could decrypt one side of the test phase messages. Interestingly, while Facebook's Secret Conversations interfaces with the Signal library directly [11], it does not achieve post-compromise security.

(4) *No Protection.* Telegram X is the only messenger where *all* messages are leaked to the attacker. We expected this, because Telegram's MTProto algorithm makes no attempt to provide post-compromise security. However, as we have seen, a number of platforms without advanced mechanisms for achieving post-compromise security can still protect against an attacker restricted to UI-access. Uncovering the reasons for this security gap may positively influence messaging-protocol design.

Post-Compromise Behavior. The second question (B) of our experiment concerns the behavior of the messengers in the post-test phase. It turns out that the various messengers react in different ways to the cloning event and the respective message exchange from the test phase. A summary of their behavior is given in Table 3. We remark that the post-compromise behavior of messaging apps is unrelated to the post-compromise-security property—this section explores whether the messaging channel is usable after a cloning attack. We can distinguish three different kinds of behavior, which are the same for all four types of message exchanges:

(1) *No Protection and no Detection: Permit Communication Without Errors.* WhatsApp, Facebook Secret Conversations, Viber, Telegram X, Threema, and Wickr allow A and B_{evan} to send and receive messages in the post-test phase, without exhibiting any errors or showing any error messages. From the participants' point of view (through the user-interface) it is undetectable that any form of unusual behavior has occurred.

(2) *Partial Protection or Detection.* When using Signal, Olvid, or Wire, A and B_{evan} seem able to send and receive messages in the post-test phase, but there are errors about which the messengers alert them explicitly.

In particular, in Signal, when a sender (either A or B_{evan}) sends a message to the other party, then the receiver's app will just display the message "*Bad encrypted message*". Thus, while the message is blocked, the sender is not informed.

Wire behaves similarly to Signal: sent messages lead to corresponding receipt notifications on the sender side, but the receiver cannot decrypt the messages and instead is shown the

			Post-Compromise		
			Blocked?	Error msg?	
				Send	Recv
Signal Library	Signal	A → B_{evan}	✓	-	✓
		B_{evan} → A	✓	-	✓
	WhatsApp	A → B_{evan}	✗	-	-
		B_{evan} → A	✗	-	-
	Facebook SC	A → B_{evan}	✗	-	-
		B_{evan} → A	✗	-	-
	Skype PC	A → B_{evan}	✓	✓	✓
		B_{evan} → A	✓	✓	✓
Custom Signal	Viber	A → B_{evan}	✗	-	-
		B_{evan} → A	✗	-	-
	Wire	A → B_{evan}	✓	-	✓
		B_{evan} → A	✓	-	✓
Other Protocols	Wickr	A → B_{evan}	✗	-	-
		B_{evan} → A	✗	-	-
	Olvid	A → B_{evan}	✗	-	-
		B_{evan} → A	✓	-	✓
	Threema	A → B_{evan}	✗	-	-
		B_{evan} → A	✗	-	-
	Telegram X	A → B_{evan}	✗	-	-
		B_{evan} → A	✗	-	-

Table 3: Post-compromise behavior based on messages from the post-test phase. After a test phase during which A and B_{blake} heal, subsequent communication between A and the attacker B_{evan} should become impossible.
The "blocked?" column indicates whether communication is indeed blocked as expected. If the communication is blocked, the next two columns indicate whether the sender and recipient obtain information in this case. Here, ✓ indicates that the sender respectively receiver see an error message.

error message "*A message from [A|B] was not received*" together with an error code and a link that says "*Reset session*".

Finally, Olvid is different from the others in that its protection is one-sided. Messages sent from B_{evan} are not received by A, and B_{evan}'s device does not display corresponding receipt notifications; however, messages sent from A are still received by B_{evan}, and A's device shows corresponding receipt notifications.

(3) *Full Protection and Detection: Halt Communication with Errors.* Skype Private Conversations is unique in that it terminates communication between A and B_{evan}, and shows an error message on both devices: right after A attempts to send the first message to B_{evan}, Skype will show A the message "*Private conversation ended. Send a new invitation to resume your private conversation with* B", also B_{evan} will see the analogous message "*Private conversation ended. Send a new invitation to resume your private conversation with* A".

3.4 Analysis

Our experiment shows that messaging apps behave differently upon compromise of one of the parties. Two aspects are particularly relevant in our work—the secrecy of messages sent and received after the compromise, and the ability of the apps to detect the clone activity and to notify the parties that suspicious activity took place.

We would expect messengers using the Signal protocol library (Signal, WhatsApp, Facebook SC and Skype PC) to behave in the same way with respect to the secrecy of messages in the test phase. However, as it turns out, they span into three different groups.

Signal and WhatsApp protect all messages in the test phase, as expected from the double ratchet's post-compromise security.

Skype PC provides the protection of all messages except for the first message sent from A to B_{blake}. A possible reason for this is that no healing has happened at the point when B_{evan} takes over (i.e., the root key has not been updated via the Diffie-Hellman ratchet); therefore, the first message A sends to B_{blake} (and which is later received by B_{evan}) is exactly what B_{evan} expects.

The most surprising behavior of Signal-based apps is that of Facebook SC, where the clone, B_{evan}, can decrypt all messages sent from A to B_{blake}. For this to happen, either B_{evan} would have to compute the same ratcheting steps as B_{blake} to generate the exact same sequence of shared secrets (which happens with negligible probability), or A would have to re-encrypt the messages for B_{evan}, by using a previous state or by implicitly restarting the conversation. We conjecture that this is due to an implicit restart of the session, after which old messages are re-encrypted and sent again.

Differences can also be observed when it comes to the apps' reactions to the clone behavior. The most drastic reaction is that of Skype PC, which completely halted the communication in the post-test phase. In contrast, Facebook SC, WhatsApp, and Viber—which are also Signal-based—continue as if nothing happened. Continuing the conversation in the post-test phase without any errors would require a mechanism that resynchronizes the states of the communicating partners. This indicates that having the parties desynchronized is deemed a normal situation by these apps, which would explain why no suspicious activity is flagged. Clearly, these apps are sacrificing security for usability.

Skype's approach is the more secure solution, though it might lose on the usability end. However, it would be desirable if parties could be notified of the reason *why* the conversation is not private anymore, thus explicitly detecting possible clones.

Signal ranges somewhere between Skype PC and the other Signal-based apps: Even though the conversation seemingly continues from the perspective of the sender, none of the messages can be decrypted. The parties are therefore implicitly forced to restart the conversation without an explicit explanation what went wrong.

As we show next, security and usability can be reconciled in a better way, by equipping the double ratchet with a sound clone-detection mechanism that makes no false or vague claims.

4 PROPOSED IMPROVEMENT

As we have seen, existing messaging apps do not achieve the security guarantees offered in theory by their underlying protocols. In particular, they do not achieve post-compromise security in practice. Many of these apps rely on the Signal protocol, which has at its core the well-known *double ratchet* algorithm, which combines Diffie-Hellman secret sharing with key chains based on key-derivation functions (which in practice are hash functions).

In theory, the double ratchet evolves the key chains of both the sender and the receiver after every exchanged message, to compute for each message a distinct message key that is used for encryption on the sender side and for decryption on the receiver side. Thus, for every sender key chain of one party, there is a corresponding receiver key chain of the other party. For example, when A sends messages $m_1, \ldots, m_5$ to B, A starts out with a message key K_1 for encrypting m_1 and then successively evolves their sender key chain to compute the message keys $K_2, \ldots, K_5$ for the subsequent messages $m_2, \ldots, m_5$. On the receiver side, B evolves the receiver key chain accordingly to decrypt the messages sent by A.

In the context of our cloning experiment, the following should therefore happen in theory: In the test phase, B's first clone, B_{blake}, evolves the key chains according to the messages exchanged with A, whereas B_{evan} (who is offline during the test phase) does not evolve the key chains at all. Once the test phase is over, the key chains of B_{evan} and A are then desynchronized, meaning that an error should occur once B_{evan} and A try to exchange messages.

Since our experiment shows that most apps do not behave that way, it appears that there is a mismatch between the double ratchet's specification and its actual implementations in messaging apps. We conjecture that this mismatch is the result of an accepted *trade-off between security and error tolerance*. Unfortunately, this increased error tolerance provides an attacker with increased capabilities, opening the door for clone attacks that, as we have seen, violate post-compromise security. Whereas in theory, an app should be able to detect the suspicious behavior caused by cloning a device, this is often not the case in practice.

Possible Causes of Desynchronization Errors. In reality, errors—which manifest themselves in the form of unexpected message keys used for encryption and decryption—can have very natural reasons. Usually, they are the result of *desynchronization errors* based on so-called *state loss*, which causes a device to incorrectly update its internal state, i.e., the part of memory relevant to the correct functioning of an app. The internal state contains important information such as secret keys used in the key chains. State loss can therefore lead to the desynchronization of the key chains maintained by the sender and the receiver, resulting in unexpected behavior. From discussions with developers, we find that for global user bases, two kinds of state loss occur in practice:

- **Total-state loss** is the event where a party completely erases any previous secret values. A typical cause of total-state loss is the loss of a device, leading its owner to reinstall the messaging app on a new device. Similar situations occur when a device is wiped and reinstalled.
- **Single-state loss** is the event where a party sends a message and then fails to update its local state due to hardware malfunction, e.g., because the state update is not properly propagated to persistent memory. When the party then tries to send a new message, it uses its previous state to derive the (old) key and thus continues its key chain from there. This event is much less likely than the first, but can occur for older soft- and hardware used in impoverished regions.

Tolerating State Loss Violates Post-Compromise Security. The tolerance of state loss can increase the usability of a messaging app by allowing users to continue their communication, but this increased usability can violate post-compromise security. A simple example is an app that tolerates total-state loss: Suppose A and B are communicating with each other, and then B's device is cloned by the attacker; assume that A and B heal afterwards. This should imply the clone is locked out again. However, the attacker could then pretend to have suffered from total-state loss and thus reinitialize its state with A to communicate with A. If then at some point the original B tries to send a message to A, the message cannot be decrypted by A, which could in turn lead A to reinitialize with the original B. This means that the attacker was able to impersonate B by injecting their own messages and eavesdropping on A's messages, and they could in fact even do this repeatedly without being detected, even though A and B healed. This violates post-compromise security.

Combining Error Tolerance with Clone Detection. As we cannot achieve post-compromise security if we want to tolerate certain forms of state loss, a trade-off between error tolerance and security is inevitable. We therefore aim for a protocol that tolerates state loss while still being able to perform sound clone detection. Here, by *sound* we mean that the protocol only indicates cloning behavior if there was indeed cloning behavior; in other words, there should be no false positives. This allows a messaging app to respond to the cloning activity in a proper way, informing the users and possibly taking further measures to achieve secure communication again.

In the following section, we present a modified version of Signal's double ratchet that achieves exactly that, but before we present our protocol, we first want to highlight a few important subtleties that are involved with clone detection in the presence of state loss.

Why State Loss Breaks Existing Approaches for Clone Detection. There exist various clone-detection solutions (see Section 2.3). However, as none of these mechanisms consider the occurrence of state loss, they can falsely identify honest actions as actions of a clone.

For instance, in the *log maintainer* approach discussed in Section 2.3, a party that appends an entry to the log but fails to record it locally would detect a clone upon requesting a proof of extension from the log. The same holds for the other mechanisms, where failure to register one message exchange leads to detection. The following would happen in each protocol upon single-state loss:

(1) *counter-based:* the same counter would be reused when sending the next message,
(2) *hash-based:* the hash value would not be evolved correctly,
(3) *commitment-based:* the party would fail to remember the commitment it made.

Clone detection can get even more involved when, for instance, both parties repeatedly suffer from single-state loss. We thus require an approach that can deal with these problems.

4.1 Concrete Proposal

We propose a slight modification of Signal's double ratchet that is able to detect clones in the presence of both single-state loss and total-state loss: We add to every message a message counter and a message authentication code (MAC) computed via a special key, which we call the *epoch key*. In the original double ratchet (see Section 2), whenever two parties start a communication session, they first agree on a root key (in Signal, via the X3DH protocol),

which is then updated subsequently to evolve the key chains for sending and receiving messages. We define the epoch key *ek* to be a hash of this root key, i.e., given a root key *rk* and a hash function *h*, the epoch key is $h(rk)$.

When two parties start a communication session, they both compute the epoch key and additionally initialize two counters—the *send counter* and the *receive counter*—to the value 0. Now, whenever a party sends a message, it appends its send counter as well as a MAC of the message computed with the epoch key. Upon receipt of a message with send counter *n*, the receiver does two things:

(1) it verifies the MAC with the epoch key, and
(2) it checks if the send counter *n* of the message is greater than or equal to its own receive counter; if so, it sets its receive counter to *n*.

If the MAC verification or the counter check fails, the receiver concludes that its supposed communication partner must have been cloned. Note the receiver does not indicate whether the *current* message was sent by a clone of its supposed partner or by the partner itself; all it knows is that the partner must have been cloned. Figure 4 shows an example trace of our protocol.

As we will show, the combination of message counters and the MAC suffices not only to guarantee the soundness of our approach (lack of false positives) but also allows us to detect a large class of cloning activities. On the other hand, completeness of detection (lack of false negatives) is not achievable even in the simpler scenario with no state loss [22]. Due to space constraints, we illustrate the limitations of detection with concrete examples in Appendix A.

The use of an incrementing counter implies that the soundness of our approach depends on the loss-less, in-order delivery of messages, which could, for instance, be guaranteed by adding a buffer. It should be mentioned though that (as noted by Alwen et al. [1]) the addition of a buffer can introduce complications.

Ideas Behind the Approach. The intuitive ideas behind using the message counters and the MAC are as follows: By adding a message counter to each message, we allow the receiver to track the last counter it received from its partner—it does so by setting its receive counter to *n* for every received counter in Step 2 above. Thus, if a message with a lower message counter arrives, the receiver can conclude that something must have gone wrong, because if there were only *one* party sending messages, this party would not send an old message counter. Notice that we deliberately allow message counters that are equal to the current receive counter to cover the case where the other party has suffered from single-state loss and thus sent the same message counter twice in a row.

The message counters alone, however, are not sufficient for guaranteeing soundness, which is why we need the MAC computed with the epoch key: Suppose A has exchanged a few messages with B_{blake} until a point where A's receive counter and B_{blake}'s send counter both have the value *n*. At that point, B_{blake}'s device is cloned and the clone, B_{evan}, pretends to have suffered from total-state loss. Then, as discussed on Page 4, A might reinitialize with B_{evan}, meaning that they reset their send and receive counters to 0 and compute a new epoch key from the new root key.

The clone B_{evan} can then exchange messages with A. Now, as long as B_{blake} doesn't send any messages to A, the clone, B_{evan}, will be able to communicate with A without A noticing anything

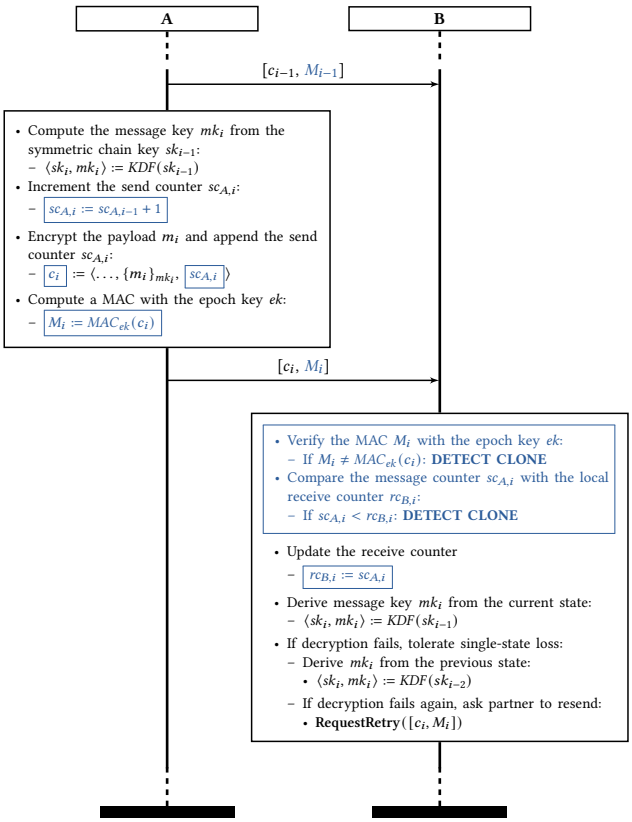

Figure 4: Example trace of our proposed protocol. Differences to the original double-ratchet algorithm are highlighted in blue. A appends to their original message their current send counter $sc_{A,i}$ and a MAC $M_{ek}(c_i)$, computed with their epoch key *ek*, of the ciphertext. When B receives the message, B detects a clone if (a) the MAC does not verify with B's epoch key *ek*, or (b) the counter is lower than B's current receive counter $rc_{B,i}$. Otherwise, B updates their receive counter and (a) decrypts the message with their current state (*normal message*), or, if this fails, (b) decrypts the message with their previous state (*single-state loss*), or, if this fails too, (c) asks A to send the message again (*potential attack* or *honest message*, see Example 4.1).

unusual. However, if B_{blake} then sends a message to A, the cloning behavior will be detected by A, because the MAC of B_{blake}'s message cannot be verified as B_{blake} is still using an old epoch key. Had A just checked if the message counter sent by B_{blake} (which still has the value $n+1$) was lower than A's current receive counter, A might have concluded that $n+1$ was greater than their current receive counter and thus A wouldn't have detected a clone. Verification of the MAC with the epoch key thus serves as an additional measure to deal with total-state loss.

Our approach adds minimal overhead to the original double ratchet: Each message is extended by just two additional fields—the message counter and the MAC. For example, a message-counter field of 32 bits (allowing to count up to more than 4 billion messages) and a 256 bit MAC add an overhead of 288 bits, or 36 bytes,

amounting to three short English words encoded in UTF-16. The space requirements for storing the two counters and the epoch key on devices are also negligible. Our approach is thus much less costly than, e.g., storing *all* previous message keys to compare newly arrived messages against them. Such an approach would have the additional drawback that it allows an attacker to decrypt all old messages once it obtained access to the keys stored on a device.

Additional Considerations. One could reasonably question the necessity of adding the message counters and the MAC instead of just detecting clones based on whether received messages can be encrypted with the current message key of the receiver chain. After all, an honest (non-cloned) party should anyhow encrypt messages with the right message key, and if it does not, there must have been a clone, right? The following example shows that this seemingly intuitive approach would lead to the unsound detection of clones. In the example, keep in mind that whenever a party sends a message right after it received a message, it evolves its root-key chain and uses the new root key to derive a message key for the new message:

Example 4.1. Suppose A evolves their root-key chain to send a new message to B, but then suffers from single-state loss and thus forgets that the root-key chain has evolved. When B then receives A's message, B uses A's new public key (which is part of their message) to also evolve the root-key chain and decrypt the encrypted portion of A's message. Usually, A and B would now both have the same root key, but since A forgot theirs, the root keys of A and B don't match. Now B decides to respond to A by sending a new message. B thus evolves the root-key chain again to derive a new message key to encrypt the message. When A then receives this message, the message cannot be decrypted, but if A concluded that B's device must have been cloned, A would be wrong.

In the above example, B's message will contain a valid MAC (computed with the epoch key) and a message counter that is greater than A's current receive counter. Our approach would thus behave as expected and not detect any cloning behavior. Example traces of how parties synchronize after single-state loss and total-state loss, as well as clone detection, can be found in Appendix A. We next present a formal analysis of our proposed algorithm.

4.2 Formal Analysis

The goal of our analysis is to create a faithful formal model of our modified double ratchet and to prove the soundness of our proposed clone-detection mechanism using the Tamarin prover. Our model consists of 21 rewriting rules as well as 20 different lemmas (stating, for instance, crucial invariants) that we proved in order to obtain our main statement, which says that whenever an honest user detects that its partner has been cloned, the partner has indeed been cloned. As it is impossible to discuss the entire model within the restricted space of this submission, we focus on a general overview as well as on critical modeling decisions. We uploaded our formal model (including all lemmas and their corresponding proofs as well as documentation to reproduce our results) to the GitHub repository https://github.com/dr-clone-detection/model.

Overview of the Model. We allow an unbounded number of *users* that can create unique bidirectional communication channels with other users. Once a user has established a communication channel with a particular partner, that channel is uniquely identified by a so-called *user thread*. This means that a single user can start an unbounded number of user threads in general, but only one user thread per communication partner. Once a pair of users has established such a channel, they can initialize (and later reinitialize) a *session* with each other.

The initialization allows them to establish secret keys—in particular the *root key* of the double ratchet—and to start exchanging messages with each other. As we focus on the double ratchet and not on the initialization protocol (which, in Signal's case is X3DH) we just modeled an abstract initialization consisting of three rewriting rules. The only assumption underlying the initialization procedure is that a pair of users cannot perform two or more initializations with each other in parallel. As part of the initialization, the users derive the *epoch key* for the clone-detection mechanism (by applying a hash function to the root key) and initialize their send counters to 1 and their receive counters to 0.

After the initialization, one of the users (the *initiator*) is ready to start sending messages while its partner (the *responder*) can receive these messages. As defined by the double ratchet, they can then take turns, switching between *sending* and *receiving* messages. Whenever a user switches from *sending* to *receiving*, it performs a ratcheting step of the Diffie-Hellman ratchet to compute a new public key and in particular a new root key. Moreover, when a user sends or receives multiple messages in a row, it evolves its sender or receiver key chain accordingly by performing ratcheting steps of the symmetric-key ratchet. All this is modeled by dedicated rewriting rules that take care of sending and receiving messages and accordingly updating a user's state.

Clone-Detection Mechanism. We incorporated our clone-detection mechanism by forcing a sending user to update its send counter and to append this send counter along with a MAC computed with the epoch key to every message. The receiver takes care of updating its receive counter with every received message. To detect clones, we added dedicated receiver rules that can be executed when either the MAC of a message is invalid or the receive counter is lower than a user's current receive counter.

State Loss. To capture single-state loss and total-state loss, we added two additional rewriting rules: In the single-state loss case, the rule defines that a user sends a message (like it would do in a normal sender rule), but instead of updating its local state (by evolving its key chains and updating its send counter), it doesn't alter the state at all. In the total-state loss case, a user can (at any point in time, and not just when sending a message) forget its whole state. From that point on, the user suffering from state loss can do nothing else than reinitialize with its partner.

Attacker Capabilities. The attacker in our model can at any point in time take the current state of a user and clone the state. We modeled this with a dedicated rule that takes all the relevant state (e.g., the root key, symmetric chain key, etc.) of a given user thread and spawns a new user thread, which can then act like a normal user and send messages to the communication partner associated with the user thread. This means that the attacker's capabilities, after cloning a user thread, are restricted to the capabilities of a normal user. This models the assumption that the attacker really

just cloned a device and then used a messaging app through its user interface. We modeled this by replacing the usual *In* and *Out* facts—offered by Tamarin to model an attacker who can control the network by intercepting or altering general messages on the network—with dedicated *Send* and *Receive* rules. We thus also used typed messages in the *Receive* facts, because the attacker is anyhow not able to create arbitrary messages.

Finally, note that once a user has been cloned, its partner is unable to distinguish between messages received from the original user and messages received from the attacker. As discussed earlier, the only restriction is that the messages sent by one user thread must arrive in the order they were sent. This applies only to user threads and not to users: It is possible that a user thread of user B_{blake} first sends a message, and then the attacker thread of B_{blake}'s clone B_{evan} sends a message, but B_{evan}'s message arrives first.

Methodology. Our main statement intuitively says that whenever an honest party (i.e., a party that itself is not a clone) claims that its communication partner was a clone, then the partner must indeed have been cloned at an earlier point in time. In guarded first-order logic, the statement reads as follows:

$$\forall\ userThread\ user\ partner\ epochKey\ rcvCtr\ key\ t_1.$$
$$(\text{DetectClone}(userThread, user, partner, rcvCtr, messageMAC)@t_1 \wedge$$
$$\neg\exists t_2.\ t_2 < t_1 \wedge \text{CloneUser}(user, partner)@t_2)$$
$$\Rightarrow \exists\ t_3.\ t_3 < t_1 \wedge \text{CloneUser}(partner, user)@t_3$$

As already mentioned, formally proving this seemingly simple statement with Tamarin required us to prove 19 additional lemmas. The main idea behind our proof strategy was to distinguish between two cases that lead a user to detect that its partner was cloned:

(1) the partner sends a message with an invalid MAC, or
(2) the partner sends a message with a message counter that is lower than the user's current receive counter.

Apart from lemmas that state several invariants and fine-grained properties useful for both cases, we then proceeded as follows.

In the case of an invalid MAC, we proved that honest users always know the current epoch key of a communication session and therefore never send an invalid MAC. Thus, if our supposed partner sends an invalid MAC, the partner must have been cloned.

In the case of the message counters, we proved (by induction) that *within a single communication session*, honest users always send messages with monotonically increasing message counters. Thus, if a user receives a message counter that is too low, the monotonicity property is violated, meaning that the partner must have been cloned. Note here that the message counters might not increase *strictly* monotonically due to single-state loss. Note also that message counters do not increase monotonically *across* sessions, as the initialization of new sessions resets the message counters .

Summary of the Analysis. All 20 lemmas of our formal model can be proved automatically by Tamarin in around 26 minutes on an 8-core machine with 30GB of memory; this, however, required extensive fine-tuning and tweaking of the heuristics for several lemmas. The lemma that took by far the most time (around 19 minutes) is the above-mentioned lemma stating that message counters increase monotonically. The reason why it takes Tamarin so long to prove this lemma is because it has to consider all possible combinations of rewriting rules that refer to message counters, which leads to a quadratic blow-up in the number of rules.

Our analysis guarantees that under the assumptions outlined in the beginning of this section, our approach is sound, meaning that it does not yield false positives. As our approach can deal with cloning events from earlier sessions (due to the MAC computed with the epoch key) as well as with cloning events from the current session (due to the message counters), it detects a large range of cloning events. Note that a completeness statement like, *if a partner was cloned, the mechanism will detect it* cannot hold, as it cannot be guaranteed in general that a clone is distinguishable from the original user. We discuss this in more detail in Appendix A.

Finally, as our protocol is a simple extension of Signal's double ratchet, we believe that our formal model can serve as an excellent foundation for other work that aims at formally analyzing the double ratchet with the Tamarin prover.

5 CONCLUSIONS

We have shown that many popular messaging apps do not provide the security guarantees offered by their underlying protocols. In particular, we demonstrated that in practice, cloning a device can lead to surprising behavior that violates post-compromise security.

A likely reason for this unexpected reduced security is that the app developers made a trade-off between security and usability, tolerating some forms of desynchronization based on state loss. As we have demonstrated, such a trade-off is inevitable, as full post-compromise security has to be given up if state loss is tolerated.

We identified two typical forms of state loss: *single-state loss* and *total-state loss*. The former usually happens when a device sends a message but then fails to update its state accordingly (e.g., failed write/flush) and thus forgets that it sent the message. The latter happens, for instance, when a user loses their device and thus reinstalls an app on a new device, which leads to a loss of all formerly established secrets.

To reconcile security and usability, we then presented a secure-messaging protocol—a modified version of Signal's double ratchet that offers a sound and simple approach for clone detection. Our approach, which is based on message counters and message authentication codes, enables users to detect cases where their supposed communication partner has been cloned, thus allowing them to react accordingly—for instance, by re-establishing secret keys with their partner via a secure channel, or for security-savvy users, generating a new identity key.

To prove the soundness of our approach, we constructed a formal model of our protocol using the automated-reasoning tool Tamarin. The main result of our analysis states that whenever our clone-detection mechanism indicates to a party that its partner has been cloned, then that partner has indeed be cloned. This formal analysis should increase the trust in our protocol, thus making it a more secure approach than simple ad-hoc mechanisms. As discussed in work on usable security [17], users might have trouble interpreting error messages. Further study is thus required to find usable ways of properly incorporating our clone-detection approach into apps' user interfaces.

REFERENCES

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

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

[3] Colin Boyd, Anish Mathuria, and Douglas Stebila. 2020. *Protocols for Authentication and Key Establishment, Second Edition*. Springer. https://doi.org/10.1007/978-3-662-58146-9

[4] Andrea Caforio, F. Betül Durak, and Serge Vaudenay. 2019. On-Demand Ratcheting with Security Awareness. *IACR Cryptology ePrint Archive* 2019 (2019), 965.

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

[6] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. 2016. On Post-compromise Security. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 164–178. https://doi.org/10.1109/CSF.2016.19

[7] Josh Constine. 2018. WhatsApp hits 1.5 billion monthly users. $19b? not so bad. https://techcrunch.com/2018/01/31/whatsapp-hits-1-5-billion-monthly-users-19b-not-so-bad. accessed: 2020-05-02.

[8] David Derler, Tibor Jager, Daniel Slamanig, and Christoph Striecks. 2018. Bloom filter encryption and applications to efficient forward-secret 0-RTT key exchange. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 425–455.

[9] Mohammad Sadeq Dousti and Rasool Jalili. 2015. Forsakes: A forward-secure authenticated key exchange protocol based on symmetric key-evolving schemes. *Advances in Mathematics of Communications* 9, 4 (2015), 471–514.

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

[11] Facebook. 2017. Messenger Secret Conversations, Technical Whitepaper. https://fbnewsroomus.files.wordpress.com/2016/07/messenger-secret-conversations-technical-whitepaper.pdf. Accessed: 2020-05-02.

[12] Matthieu Finiasz and Thomas Baignères. 2019. Security Model of Mobile Messaging Apps. https://olvid.io/assets/slides/2019-06-12%20Olvid%20-%20GDR%20S%C3%A9curit%C3%A9%20Informatique.pdf. accessed: 2019-12-03.

[13] Wire Swiss GmbH. 2018. Wire Security Whitepaper. https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf. accessed: 2020-05-02.

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

[15] Christoph G. Günther. 1989. An Identity-Based Key-Exchange Protocol. In *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings (Lecture Notes in Computer Science)*, Vol. 434. Springer, 29–37. https://doi.org/10.1007/3-540-46885-4_5

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

[17] Amir Herzberg and Hemi Leibowitz. 2016. Can Johnny finally encrypt?: evaluating E2E-encryption in popular IM applications. In *Proceedings of the 6th Workshop on Socio-Technical Aspects in Security and Trust, STAST 2016, Los Angeles, CA, USA, December 5, 2016*, Gabriele Lenzini, Giampaolo Bella, Zinaida Benenson, and Carrie E. Gates (Eds.). ACM, 17–28. https://doi.org/10.1145/3046055.3046059

[18] Chris Howell, Tom Leavy, and Joël Alwen. 2017. Wickr's Messaging Protocol. https://wickr.com/wickrs-messaging-protocol/. accessed: 2019-08-16.

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

[20] Moxie Marlinspike and Trevor Perrin. 2016. The X3DH Key Agreement Protocol. https://signal.org/docs/specifications/x3dh/. accessed: 2019-08-12.

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

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

[23] Trevor Perrin and Moxie Marlinspike. 2016. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/. accessed: 2019-08-12.

[24] Skype. 2018. Skype Private Conversation, Technical white paper. https://az705183.vo.msecnd.net/onlinesupportmedia/onlinesupport/media/skype/documents/skype-private-conversation-white-paper.pdf. Accessed: 2020-05-02.

[25] Telegram. 2019. End-to-End Encryption, Secret Chats. https://core.telegram.org/api/end-to-end. Accessed: 2020-05-02.

[26] Threema. 2019. Threema. Cryptography Whitepaper. https://threema.ch/press-files/cryptography_whitepaper.pdf. Accessed: 2020-05-02.

[27] Viber. 2019. Viber Encryption Overview. https://www.viber.com/app/uploads/viber-encryption-overview.pdf. accessed: 2020-05-02.

[28] WhatsApp. 2017. WhatsApp Encryption Overview. Technical white paper. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. Accessed: 2020-05-02.

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

A EXAMPLE TRACES

In this appendix, we provide example traces to illustrate how our proposed version of Signal's double-ratchet algorithm functions during resynchronization procedures and clone detection.

In Figure 5, we show an example trace in which a clone is detected. In Figure 6, we show an example trace for recovering from single-state loss. Similarly, we show in Figure 7 an example trace for recovering from a total-state loss.

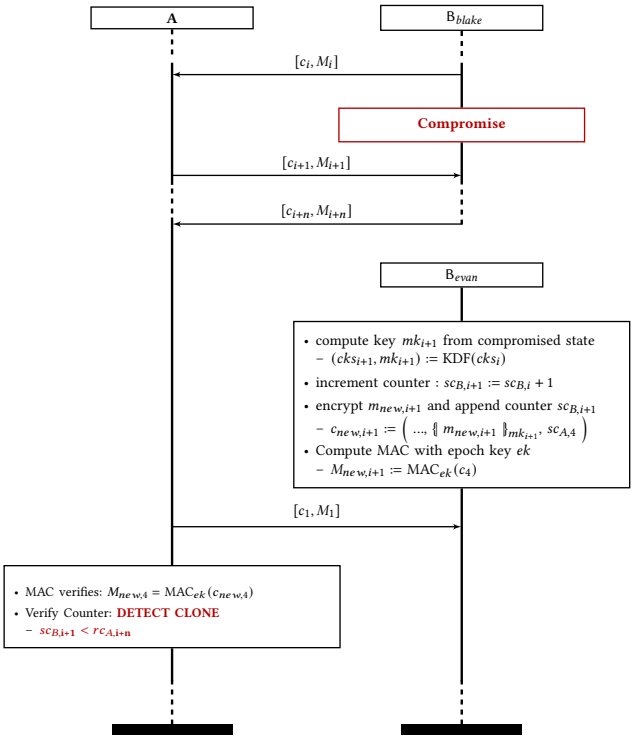

Figure 5: Example trace of clone detection. B_{blake}'s state gets compromised after sending message m_i. B_{blake} continues the conversation with A exhanging n more message until the attacker B_{evan} becomes active and injects a message of its own. A detects a clone, since the counter received $sc_{B,i+1}$ is smaller than the local counter $rc_{A,i+n}$

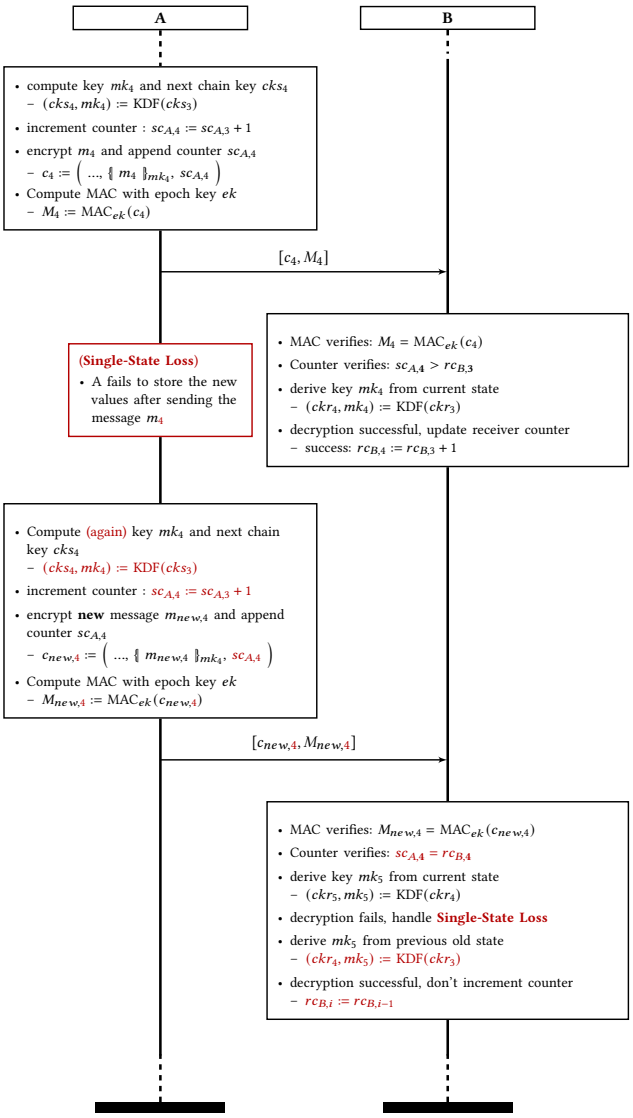

Figure 6: Example trace of the parties synchronizing after A had single-state loss. A sends message 4, B receives the message and updates their state. A has *single-state loss* after sending the message, thus it will perform the same computations to send the next message. B receives again, detects *single-state loss* and computes the message key from the previous state, synchronizing again with A.

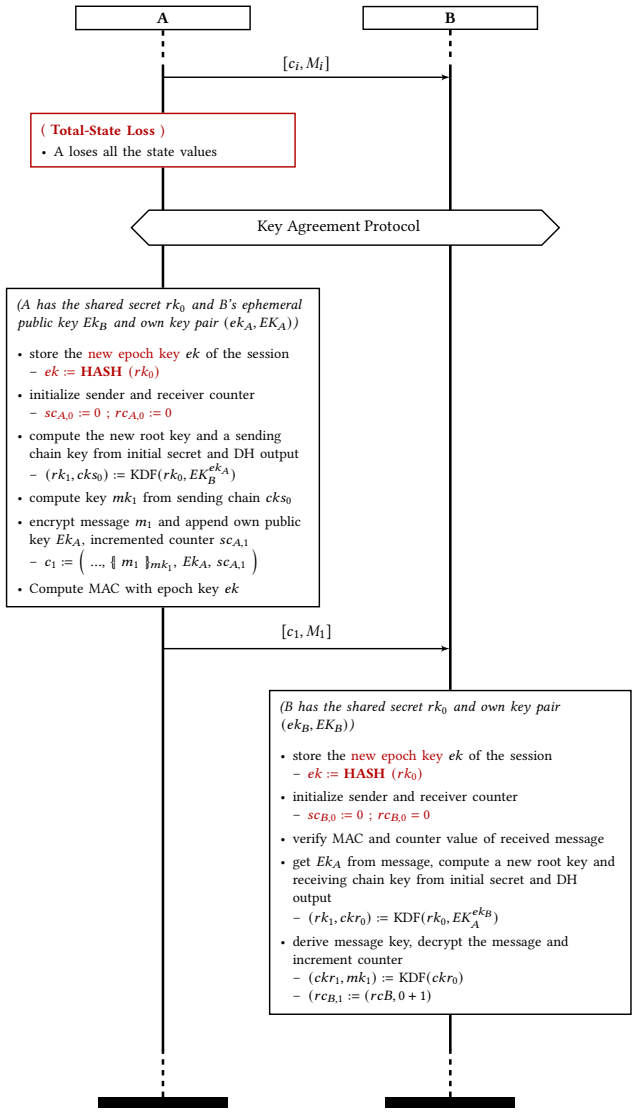

Figure 7: Example trace of the parties synchronizing after total-state loss. A sends message i, then has *total-state loss* (e.g., A loses their phone). When A wants to communicate again, first it will establish a new common secret with B, derive the new root key, initialize the counters and then compute the message key. B upon receiving the message will follow the same procedure and then the parties are once again synchronized.

B UNDETECTABLE ATTACK TRACES

In this section, we discuss scenarios in which clone detection is hard or even impossible.

Previous work [22] has shown that detection completeness—i.e., providing a clone-detection approach that can detect *all* kinds of cloning behavior—is not achievable. A trivial example to illustrate this is an attacker that, after cloning a device's state, does not use the device. Another example is an attacker whose actions do not interfere with those of the victim, as in a stateless protocol. In both cases, clone detection based solely on the messages exchanged in a protocol is provably impossible.

As we have seen, allowing state loss gives the attacker the additional capabilities of (1) resetting the state of a conversation between its victim and a third party and (2) using the same key twice to encrypt a message. This activity can be detected as described in detail in 4.1. However, even if the attacker simply follows the protocol, there are two major limitations:

First, consider a cloned party that suffers from total state loss and has yet to restart the conversation with its partner. Until that happens, the attacker can impersonate the victim using the cloned state and go undetected. This comes as no surprise, considering that state is not carried between the two sessions before and after state loss from the honest party. Thus, the protocol acts as if it was stateless and, by previous results, detection is not possible. A possible trace is the following:

(1) $B_{evan} : \texttt{Compromised}$
(2) $B_{evan} \longrightarrow A, A \longrightarrow B_{evan}$
(3) $B_{evan} : \texttt{Total-State Loss}$
(4) $B_{blake} \xrightarrow{\texttt{restart}} A, A \longrightarrow B_{blake}$
(5) $B_{evan} \xrightarrow{\texttt{restart}} A\ \texttt{(No Detection)}, A \longrightarrow B_{evan},$

Second, an attacker that clones a device and then *immediately* (before the state of the conversation is updated) sends messages while claiming single-state loss does not get detected. The reason is that to the receiver, it looks like its peer is suffering from repeated single-state loss. To recover, the honest parties need to perform an asymmetric-ratchet step, thus achieving post-compromise security and locking the attacker out.

(1) $B_{evan} \xrightarrow{\texttt{key k}} A$
(2) $B_{evan} : \texttt{Compromised \& Single-State Loss}$
(3) $B_{blake} \xrightarrow{\texttt{key k}} A, B_{blake} \xrightarrow{\texttt{key k}} A, \ldots$
(4) $A \longrightarrow B_{evan}$
(5) $B_{evan} \xrightarrow{\texttt{new key k'}} A\ \texttt{(No Detection)}, A \longrightarrow B_{evan},$

To conclude, when the cloned party loses state, detection is hard or even impossible until the key is forwarded or a new one is derived. However, this class of possible attacks is not very realistic in our current setting. In order for the attacker to go undetected, it would have to either guess exactly when state loss happens to its victim or control the victim's hardware and induce the state-loss event itself. Both cases are hard to achieve for a simple cloning attacker.