Automated formal analysis of Signal's Double Ratchet: attacks, fixes and security proofs

April 13th, 2026 – v1[⋆]

Vincent Cheval
University of Oxford
United Kingdom

Charlie Jacomme
Université de Lorraine, CNRS, Inria, LORIA
France

Jessica Richards
University of Oxford
United Kingdom

Abstract

The Double Ratchet (DR) protocol is a core security component of several end-to-end encrypted communications services, primarily Signal Messenger, WhatsApp, and Facebook Messenger, servicing billions of users. In this work, we provide the first formal analysis of the DR covering all of its features, including out-of-order message arrivals. This analysis is highly automated, allows for all possible key compromises and notably proves Post-Compromise Security (PCS). We also provide partial results for the security of more complex protocol variants, these being the extension of the DR with encrypted headers, and composition with PQXDH as the initial key-exchange.

Our analysis uncovered three attacks on the protocol, two of which we confirmed to be present in the main implementation, and a third which exists in the specification. Each of these attacks weakened or broke Forward Secrecy, and are to the best of our knowledge the first such known attacks. In each case, the issues were reported to the Signal developers and subsequently fixed. Overall, our analysis provides new guarantees of the security of Signal Messenger, and demonstrates the high level of security provided by the DR under a variety of strong threat models.

1. Introduction

Signal Messenger is an end-to-end encrypted messaging application based on three core components: PQXDH, the initial key-exchange; the Double Ratchet (DR), a ratcheting mechanism to generate distinct keys for each encrypted message; and Sesame, an application layer protocol that handles multiple devices and sessions between users. It aims to be widely deployable and usable, supporting asynchronous communications, multiple devices, and out-of-order delivery. It also promises strong security guarantees, even under strong compromise scenarios. The underlying security protocol is thus a complex, highly stateful protocol with many different levels of key derivations. Similarly, the target security properties are complicated to express, having both Forward Secrecy (FS), where a compromise does not enable the attacker to decrypt past messages, and Post-Compromise Security (PCS), where even after a compromise, if the attacker does not interfere during a time-frame, the conversation heals and messages become secure again.

Due to the widespread usage of the core library implementing the corresponding protocol, it has garnered a lot of attention from the research community, which delivered several pen-and-paper security analyses [1], [2], [3], as well as several computer-aided analyses [4], [5], [6], [7] based on formal methods and automated verification tools such as TAMARIN [8], CRYPTOVERIF [9], and PROVERIF [10]. These analyses have built a strong understanding of the security properties that can be expected from Signal, and proved its security in a variety of threat models and scenarios. Through these analyses, several weaknesses in both the PCS of Signal and in the post-quantum resistance of the first specification of PQXDH were uncovered. However, none of these works identified weaknesses in the forward secrecy of Signal.

Despite those analyses, several blind spots remained due to the complexity of the protocols putting comprehensive proofs out of the reach of automated analysis tools at the time. For this reason, the full specification of the Double Ratchet was never analysed, and neither was the interplay between PQXDH, the DR, and Sesame. Due to recent improvements in PROVERIF, notably better support for lemmas and stateful protocols, we are able to present in this paper the first full analysis of the DR using PROVERIF, yielding several security proofs and weaknesses.

We note that the bulk of our analysis was carried out before October 2025, when Signal introduced the Triple Ratchet, an extension where the Diffie-Hellman based DR is accompanied by a post-quantum secure ratchet dubbed SPQR. Following the classical post-quantum hybridization pattern of communication protocols, Signal's security should not be negatively impacted even if SPQR is completely insecure. In our analysis, we explicitly model an insecure SPQR, and demonstrate this does not affect the security guarantees.

[⋆]: An extended abstract of this paper appears at IEEE S&P'26; this is the full version.

Contributions. Our core contributions are twofold:

- We provide the first complete and automated analysis of the Double Ratchet protocol, leading to the first automated proof of PCS and FS with fine-grained specification of compromises.
- We uncovered three attacks on the forward secrecy of Signal, one over the specification only, which was found fully automatically by PROVERIF, and two over the implementation that PROVERIF hinted at, but which were found in a more manual fashion. To the best of our knowledge, those are the first weaknesses over the forward secrecy of Signal to be reported.

Our PROVERIF model includes features such as skipped messages and header encryption, and we express all security properties in the most precise way possible, modelling and tracking precisely which compromises of which material and intermediate keys enables an attacker to decrypt messages.

The model was as much as possible developed to be a straightforward translation of the specification, and the proofs required several person-months of work by PROVERIF experts to successfully complete. Overall, our analysis increases the understanding of the precise security guarantees provided by Signal, and we join a body of works that provides proofs of the security of Signal's underlying components and advocates for its security.

All our PROVERIF models can be found at [11], along with reproducibility instructions.

Related Work on the Double Ratchet. We first explore the related works which analysed the DR, and summarise the main differences between such works and our work in Table 1. Some of the main points where models differ are: in which attacker model the analysis took place; whether the analysis was mechanized through a tool or pen-and-paper; and whether the DR was analysed for a bounded or unbounded number of asymmetric ratchet steps. Additionally, to consider faithfulness of a model to the specification, we specify whether the analysis supported out-of-order message arrivals. Supporting this feature results in a more complex analysis, as skipped message keys must be stored in long term memory until they are used. In addition, the state for a sending participant must still keep track of the previous receiving chain-key, further increasing state complexity.

The pen-and-paper analyses of Signal such as [1], [2], [3], [12], [13], [14] consider the DR (sometimes composed with X3DH), and define several variants of security, either in the game-based setting or the ideal functionality setting. Those analyses often perform simplifications, for instance by not modelling the storage of message keys enabling out-of-order decryption (see [1, Section 6]), or by omitting some options of the specification such as header encryption. Finally, compared to computer-aided cryptography, it is very difficult to ascertain the correctness of pen-and-paper proofs. Given the strong high-level differences between the pen-and-paper and computer-aided approach, we do not describe in detail the subtler differences between these pen-and-paper analyses, and only provide in Table 1 the most salient points we mentioned. We note that the level of detail provided by [14], which covers the DR among other examples and contributions, does not allow us to clearly ascertain the exact protocol model compared to the other mentioned works, but, as it claims to follow [1], it does not cover out-of-order decryption nor header encryption.

Among the computer-aided analyses of the DR, [5] studied (a small variant of) X3DH in CRYPTOVERIF, and the composition of X3DH and up to 3 messages of DR in PROVERIF, thus yielding an execution model which only allows bounded executions. Their analysis proves forward secrecy, as well as future secrecy. The latter ensures that, once an agent derives a key at step n, compromising its state at any earlier step $n - i$ does not enable an attacker to recover the current key. In the real world, past states cannot be leaked as they no longer exist. However, this provides a simple way to model a realistic scenario in which the state is leaked to an active attacker at step $n - i$, but the attacker subsequently becomes passive and is unable to exploit this state until after step n. In this way, giving the compromised state to the attacker earlier and making the attacker passive is equivalent to only giving the compromised state later on. This property is a variant of PCS with a very strong requirement for the healing: the attacker must be fully passive between the compromise state and the later key we want to prove secret. The more general notion of PCS typically considered, in particular in our work, is that to heal, the adversary just needs to be passive for one step somewhere in between the compromise and the later key, rather than throughout the entire interval.

The DR was also studied in the DY* framework [6], thus yielding an executable implementation with security guarantees attached. However, this analysis did not consider some features like out-of-order decryption or header encryption. Furthermore, we note that they refer to PCS, but in fact prove something akin to the previously mentioned future secrecy. Indeed, they state that for a root key RK_n: "If an active attacker compromises RK_n before we compute RK_{n+1}, we get no guarantees for RK_{n+1}". For clarity, as both [5] and [6] prove a similar weaker notion of PCS, we refer to this property as weak PCS (wPCS) in Table 1.

Finally, in very recent work [15], the DR was studied with TAMARIN when composed with X3DH without one time prekeys, and for the DR without our-of-order decryption nor header encryption support. Furthermore, they prove a property close to forward secrecy, and only enable compromise of DH private keys and not of the states of the agents, which contains the root key and chain keys.

Papers	Protocol	Model	Mechanization	Unbounded Execution	Out-of-order Decryption	Header Encryption	PCS	Compromises
[1], [13] [14]	X3DH+DR	Comp.	✘	✓	✘	✘	✓	All
[2], [3], [12]	DR	Comp.	✘	✓	✘	✘	✓	All
[5]	X3DH+DR	Symb.	PROVERIF	✘(≤ 3 ratchets)	✘	✘	✘(FS&wPCS)	All
[6]	X3DH+DR	Symb.	DY*	✓	✘	✘	✘(FS&wPCS)	All
[15]	X3DH$^-$+DR	Symb.	TAMARIN	✓	✘	✘	✘(FS)	DH keys only
Our models								
DR	DR	Symb.	PROVERIF	✓	✓	✘	✓	All
DR-HE	DR	Symb.	PROVERIF	✓	✓	✓	✘(FS)	All
DR-HE*	DR	Symb.	PROVERIF	✓	✓	✓	✓	All but DHs
DR-PQXDH*	PQXDH+DR	Symb.	PROVERIF	✘(single session but unbounded ratchet)	✓	✘	✓	All, but DHs or PQXDH

TABLE 1. COMPARISON BETWEEN OUR MODELS AND RELATED-WORK ANALYSIS ON THE DOUBLE RATCHET. WE DENOTE COMP. AND SYMB., FOR THE COMPUTATIONAL AND SYMBOLIC MODEL RESPECTIVELY. X3DH$^-$ IS X3DH WITHOUT ONE-TIME PREKEYS. IF THE MODEL DOES NOT ALLOW UNBOUNDED EXECUTION, WE MENTION THE NUMBER OF RATCHETING STEPS ALLOWED. THE "ALL" COMPROMISE MEANS THAT ANY VALUE STORED IN THE CURRENT STATE OF AN AGENT MUST BE COMPROMISABLE, WHICH INCLUDES THE CURRENT ROOT KEY, THE CURRENT CHAIN KEY, THE CURRENT PRIVATE DH KEY, AND ADDITIONALLY, WHEN OUT-OF-ORDER DECRYPTION IS MODELLED, THE PREVIOUS CHAIN KEY AND ANY SKIPPED AND UNUSED MESSAGE KEYS. WE REFER TO SECTIONS 4 AND 5 FOR MORE DETAILS ON OUR MODELS.

Additional related work. Related to Signal's analysis, PQXDH was analysed in isolation in [4] with CRYPTOVERIF and PROVERIF, but without any consideration for the DR. We build upon their PROVERIF models to work towards an analysis of the composition of PQXDH and the DR. Finally, the interactions between Sesame and the DR were verified using TAMARIN in [7], but to enable the analysis of Sesame they had to rely on a very high level and abstracted version of the DR.

Beyond the DR in particular, PCS has also been verified in the context of computer-aided cryptography on other protocols in a few works, namely on the iMessagePQ3 protocol with TAMARIN [16], on TokenWeaver [17] also with TAMARIN, and on a toy example of a bounded key derivation in [18] with both TAMARIN and PROVERIF.

2. Presentation of Double Ratchet

We briefly explain the Double Ratchet protocol, touching on additional aspects as relevant to our analysis. A full specification is available from Signal [19], [20]. For simplicity, we present here a single Double Ratchet session between an initiator I and a responder R. The initiator sends the first message, but as the protocol evolves after the initial messages, either one may be the current person sending or receiving messages.

2.1. Double Ratchet

The DR protocol consists of two parts, the symmetric ratchet, which updates each time a new message is sent and the asymmetric ratchet, which updates each time the sending and receiving roles are swapped. The latter relies on public Diffie-Hellman (DH) keys sent alongside messages. We summarize in Fig. 1 the full key derivation of the DR, and present separately below each ratchet.

2.1.1. Asymmetric ratchet. The (simplified) internal state of an agent X running the DR at step $i \geq 1$ depends on:

- DHs^i, the current sender DH ephemeral private key;
- DHr^{i-1}, the current receiver DH ephemeral public key;
- RK^i, the current root key.

Each RK^i is used to create a chain key CK_0^{i+1}, which derives encryption keys for messages from I to R when i is even, and in the reverse direction if i is odd. In the case $i = 0$, RK^0 is set to an initial shared secret key SK, with the DHs set to R's signed pre-key and DHr and CK_0^0 set to the value None. Whenever an agent receives DHr^{i+1} over the network, it computes the next two root keys as follows:

1) $\mathsf{RK}^{i+1} \parallel \mathsf{CK}_0^{i+1} = \mathtt{KDF}_{\mathsf{rk}}(\mathsf{RK}^i, (\mathsf{DHr}^{i+1})^{\mathsf{DHs}^i})$;
2) generate a fresh DHs^{i+2} and send its public part;
3) $\mathsf{RK}^{i+2} \| \mathsf{CK}_0^{i+2} = \mathtt{KDF}_{\mathsf{rk}}(\mathsf{RK}^{i+1}, (\mathsf{DHr}^{i+1})^{\mathsf{DHs}^{i+2}})$.

If $\mathtt{KDF}_{\mathsf{rk}}$ is a one way function, then compromising RK^i does not affect the secrecy of the previous keys, which intuitively provides forward secrecy. In addition, as soon as we receive an honestly generated DHr^{i+1} value, RK^{i+2} is secret even if the previous keys were compromised, which intuitively provides post-compromise security. Note that at each ratchet step, we increase our ratchet index by two. This reflects the two steps involved in the ratchet, effectively performing two half ratchets. A consequence of this is that I will always be on an odd ratchet step, whilst R will always be even.

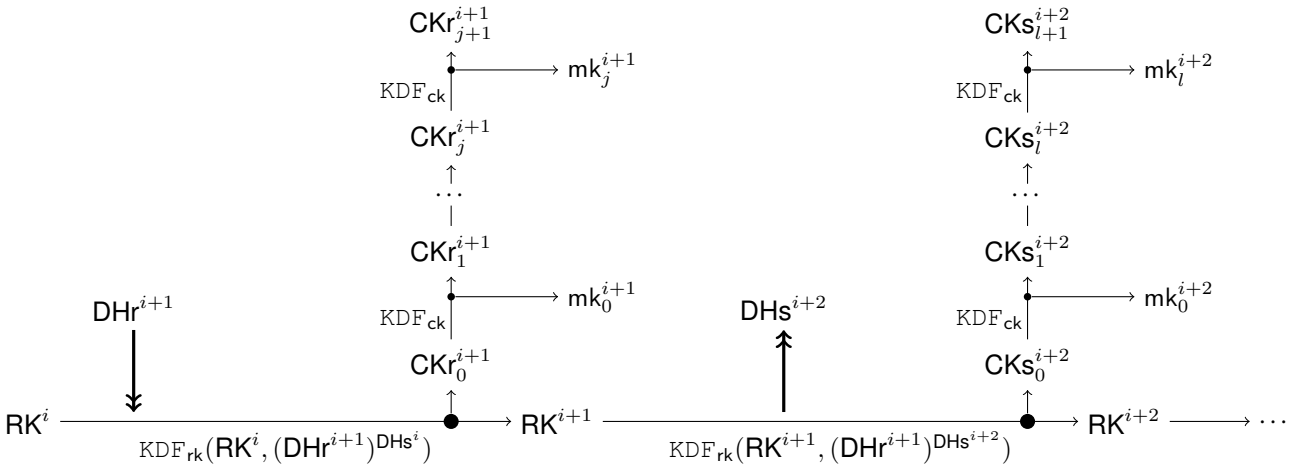

Figure 1. Key derivation underlying the Double Ratchet. A double-headed arrow represents a communication over the network, either receiving a new DHr value or sending out a freshly generated public DHs value. A single-headed arrow represents an internal computation, describing how one value is computed based on previous ones. The $\mathtt{KDF}_{\mathsf{rk}}$ takes a root key and Diffie-Hellman output and returns a new root key and a chain key. The $\mathtt{KDF}_{\mathsf{ck}}$ takes a chain key and returns a new chain key and a message key. Whenever a RK, CKr or CKs is input to a key derivation function, it is deleted from the internal state, and thus, only the rightmost or highest element of each type in this drawing would be stored in the internal state of a device.

2.1.2. Symmetric ratchet. The initial chain keys output by $\mathtt{KDF}_{\mathsf{rk}}$ form the base of the symmetric ratchet. Similarly to the root keys, a chain key may be considered *sending* (CKs) or *receiving* (CKr) depending on which agent we are considering. For I, odd i are for sending and even i are for receiving, and this is reversed for R.

From each CK_j^i we derive a new chain key and a message key $\mathsf{CK}_{j+1}^i \parallel \mathsf{mk}_j^i = \mathtt{KDF}_{\mathsf{ck}}(\mathsf{CK}_j^i)$. The message key is used to symmetrically encrypt one message. In particular, assuming all sent messages have been received, the current sender's CKs should be the same as the current receiver's CKr. In the same fashion as the asymmetric ratchet, a one way $\mathtt{KDF}_{\mathsf{ck}}$ intuitively provides forward secrecy by keeping previous chain and message keys secret even if the current chain key is compromised.

2.1.3. All together. The (simplified) internal state of an agent X running the DR at step i can be summarized as:
$\mathtt{st}_X = (\mathsf{DHs}^i, \mathsf{DHr}^{i-1}, \mathsf{RK}^i, \mathsf{CKr}_j^{i-1}, \mathsf{CKs}_j^i)$

2.2. Extension with Header encryption

In order to communicate ratchet public keys and deal with out-of-order message arrivals, encrypted Signal messages have a plaintext header sent alongside them. This header contains the current public receiving DH key DHr and the number of messages sent in the current and previous ratchet step by the sender (n, pn), so the receiver can skip messages as appropriate.

It may be desirable for a user to hide this information, to obfuscate the number of ratchets that have taken place and the ordering of messages. So Signal has documentation for how their protocol could be extended with header encryption. Note that this is *not* currently implemented in Signal. Header encryption adds an additional four keys to the state, HKs, NHKs, HKr, NHKr. These are respectively the current / next header keys for sending / receiving. Whenever a message is sent, the header is symmetrically encrypted with the sender's HKs and decrypted by the receiver's HKr or NHKr. The header keys are updated on each asymmetric ratchet. If a participant is unable to decrypt a header with HKr then they try NHKr and if the latter is successful this indicates the receiver should perform a ratchet step. During this ratchet step the next header keys become the current header keys and two new next header keys are derived.

Initially, I and R have two header keys shared, forming I's HKs and NHKr and R's NHKr, NHKs. It is not specified how these values should be agreed upon in composition with PQXDH. To derive subsequent keys, $\mathtt{KDF}_{\mathsf{rk}}$ is changed to $\mathtt{KDF}_{\mathsf{rk\text{-}he}}$ which additionally outputs a header key. Thus, we simultaneously derive our sending chain key and our next sending header key and likewise for receiving keys.

2.3. Integration in Signal Messenger

The Double Ratchet protocol is only one aspect of how Signal sends messages. Two other important aspects are how the initial secret key SK is derived, and how multiple sessions are handled. The latter is achieved using the *Sesame* algorithm.

An important property is that two devices may have multiple active Double Ratchet sessions between them. When receiving a ciphertext in this case, it is tested against all possible sessions to decrypt it.

Initial secrets are derived using the Post-Quantum eXtended Diffie-Hellman (*PQXDH*) protocol. These values for a Double Ratchet session are the secret key SK and the first DR public key DHs^0. This involves Diffie-Hellman calculuations using several long and short term secrets.

Both parties have a long term identity key (IK) and a medium term signed pre-key (SPK), which is signed with their identity key. The identity keys are assumed to be authenticated by participants out of band. Finally, participants have several one-time use pre keys (OPK), as well as a one-time or last resort post-quantum signed pre-key (PQSPK). These are each Diffie-Hellman keys, except for the PQSPK which are public keys for a Key Encapsulation Mechanism (KEM). We recall that given a KEM public key, one can encapsulate against it, producing a shared secret SS as well as a ciphertext that the owner of the secret key will be able to decapsulate and get SS. the public components of which are published on a Signal server, with the medium term and one time keys being refreshed as required.

The initiator I contacts the server and retrieves the responder's $\mathsf{IK}_R^{\mathsf{pk}}$, $\mathsf{SPK}_R^{\mathsf{pk}}$, $\mathsf{PQSPK}_R^{\mathsf{pk}}$ and one of the responder's $\mathsf{OPK}^{\mathsf{pk}}$ (if there is one available). Using this, the initiator generates a fresh DH key EK_I and uses this to create the SK as follows, with the fourth argument omitted in the case no $\mathsf{OPK}_R^{\mathsf{pk}}$ was available, and SS being the result of a KEM encapsulation against one of the $\mathsf{PQSPK}_R^{\mathsf{pk}}$:

$$\mathrm{H}((\mathsf{SPK}_R^{\mathsf{pk}})^{\mathsf{IK}_I^{\mathsf{sk}}} \| (\mathsf{IK}_R^{\mathsf{pk}})^{\mathsf{EK}_I^{\mathsf{sk}}} \| (\mathsf{SPK}_R^{\mathsf{pk}})^{\mathsf{EK}_I^{\mathsf{sk}}} \| (\mathsf{OPK}_R^{\mathsf{pk}})^{\mathsf{EK}_I^{\mathsf{sk}}} \| \mathsf{SS})$$

Here X^{pk} and X^{sk} represents the public and private part of X respectively.

The initiator then sends the responder $\mathsf{EK}_I^{\mathsf{pk}}$, $\mathsf{IK}_I^{\mathsf{pk}}$, $\mathsf{IK}_R^{\mathsf{pk}}$, $\mathsf{SPK}_R^{\mathsf{pk}}$, the ciphertext corresponding to SS, and information about the OPK_R used alongside the first message, so both parties can derive SK. Finally, $\mathsf{SPK}_R^{\mathsf{sk}}$ is used as DHs^0.

3. Breaking forward secrecy

In the course of our analysis, we identified three distinct ways to break or weaken the forward secrecy of the DR, one that only concerned the specification, and two which were present in the implementation. After interaction with Signal's developers, all the attacks led to updates on the specification and implementation.

3.1. Attack on the specification

We provide below an excerpt from the specification of the decryption function (given in Python code in [19]), only keeping the most relevant lines, and expanding the call to the function RatchetReceiveKey.

```
1  def RatchetDecrypt(state, header, ciphertext, AD):
2    ...
3    if header.dh != state.DHr:
4      ...
5      DHRatchet(state, header)
6    ...
7    state.CKr, mk = KDF_CK(state.CKr)
8    ...
9    return DECRYPT(mk, ciphertext, ...)
```

Here, we see that whenever we receive a message with a new DH element in the header, we must perform a ratchet, then apply the `KDF_CK` to the receiving chain key CKr and then decrypt. According to the specification, given a shared symmetric key SK and some DH value DHs, the initial state of R is $\mathtt{st}_0 = (\mathsf{DHs}, \mathsf{None}, \mathsf{SK}, \mathsf{None}, \mathsf{None})$. We normally expect that the first message received should trigger a ratchet, so $\mathtt{st}_0$ is never used directly for decryption. However, inside Python, None is a value that might be parsed and received over the network. So, by building a header with DH element `header.dh = None`, notice how the `RatchetDecrypt` function skips the ratcheting step as the test in line 3 fails, and directly computes the message key as `state.CKr, mk = KDF_CK(None)`. This is a public value, and thus an attacker can produce valid ciphertexts for this key. We can then break authentication and secrecy of the first message decrypted by R in this setting.

Severity. As mentioned, this issue only concerned the specification. Yet, an implementer strictly following the specification could have produced an insecure instance, notably assuming that the HMAC implementation would accept None as a key. The issue here comes from using a loosely typed language to describe the specifications, and over whether None is a value with a domain fully separate from the rest of the world or not. While browsing online, we did not find any DR implementation that suffered from this issue. This is mainly because the DR is generally preceded by a key-exchange that produces a slightly different initial DR state that does not suffer from the described issue. However, we stress that an implementation strictly following the specification would have been completely insecure.

Discovery. This attack was automatically found by PROVERIF. As we tried to follow the DR specification as closely as possible in our models, the tool directly gave us an attack trace where a responder decrypted an attacker produced message using a None key. This attack was not discovered by previous analysis of the DR because the DR was never analysed by itself, but only in composition with PQXDH, where the states are initialised differently.

Fix. Solving this issue is straightforward: one only needs to make sure that `KDF_CK` fails when receiving None, or that the ratchet is always triggered at the initial step, or that None can never be received inside the header. The Signal developers acknowledged the issue, which was fixed in the revision 4 of the DR specification in November 2025, by adding to the specification of the `KDF_CK(ck)` that "If ck is None this function must fail in a way that terminates processing." (see [19, page 18, revision 4]).

3.2. Attacks on the implementation

We present here the real-world attacks on the forward secrecy of Signal that we uncovered.

3.2.1. Weak states forced storage.

Identifying weak-states. When composed with PQXDH, a DR state on R's side is initialised in six sequential steps:

A.1) Receive a PQXDH message from I, made up of two sub messages (m, ct), where m contains the public keys $\mathsf{EK}_I^{\mathsf{pk}}, \mathsf{IK}_I^{\mathsf{pk}}, \mathsf{IK}_R^{\mathsf{pk}}, \mathsf{SPK}_R^{\mathsf{pk}}$; identifiers for $\mathsf{PQSPK}_R, \mathsf{OPK}_R$; and ct, which is the first encrypted message of the DR;
A.2) Run PQXDH to compute a shared key SK using the corresponding $\mathsf{SPK}_R^{\mathsf{sk}}$;
A.3) Initialise a DR state
$$\mathtt{st}_0 = (\mathsf{SPK}_R^{\mathsf{sk}}, \mathsf{None}, \mathsf{SK}, \mathsf{None}, \mathsf{None});$$
A.4) Process the ciphertext ct, which is both the ciphertext confirmation of PQXDH and the first ciphertext of the DR exchange;
A.5) This triggers two steps of the ratchet, generating a fresh DHs^2, and producing some DR state
$$\mathtt{st}_2 = (\mathsf{DHs}^2, \mathsf{DHr}^1, \mathsf{RK}^2, \mathsf{CKr}_0^1, \mathsf{CKs}_0^2);$$
A.6) The ct can now be decrypted with the first message key derived from CKr_0^1. If the decryption of ct fails, we should abort the initialization and not store anything in memory, otherwise, $\mathtt{st}_2$ with CKr_1^1 instead of CKr_0^1 is saved.

In this execution, notice that $\mathtt{st}_0$ is a state that contains the private key $\mathsf{SPK}_R^{\mathsf{sk}}$, and this SPK_R is shared between all PQXDH exchanges during an epoch. After the deprecation of SPK_R and thus its deletion from memory, it is expected that all the sessions that were established using this key are forward secure: even a full compromise of the current state of a device must not enable one to recompute the past SKs.

Now, if some $\mathtt{st}_0$ is saved in memory, this violates the previous statement: the private part of SPK_R might be copied in a distinct place in a memory, and thus stored and kept in memory even after any trace of this value should have been deleted from memory. And then, the compromise of the inner state of a device can lead to the compromise of a past SPK_R. With this in mind, we thus refer to the $\mathtt{st}_0$ state as a "weak-state", as it should never be saved in memory.

Saving a weak-state. Recall that based on Sesame, when receiving a ciphertext ct, instead of simply trying to decrypt ct with the current DR session, we try to decrypt ct with a list of possible DR sessions. While this makes sense when simply trying to decrypt a DR ciphertext, it leads to an attack if this is done for the ciphertext meant to act as the key confirmation for PQXDH. Indeed, when looking at the implementation, which implements PQXDH, the DR, and Sesame all together, we discovered that instead of applying step A.6, if the decryption of ct failed, $\mathtt{st}_2$ was forgotten but not $\mathtt{st}_0$, and then other potential decryption sessions were tried. Further, if the decryption succeeded for a different session than the one we are creating, everything was committed to memory, including the weak state $\mathtt{st}_0$.

Attack trace. An attacker can make the responder R save $\mathsf{SPK}_R^{\mathsf{sk}}$ in its long term memory using the following attack trace:

B.1) Run as initiator PQXDH with $\mathsf{SPK}_R^{\mathsf{pk}}$, which produces (m_0, ct_0) and a DR session S_1;
B.2) Encrypt a second message using the new DR session, producing ct_1;
B.3) Run freshly as initiator PQXDH with $\mathsf{SPK}_R^{\mathsf{pk}}$, which produces (m'_0, ct'_0) and a second DR session S_2;
B.4) Send to R first (m_0, ct_0), and then (m'_0, ct_1).

R will process the first message and obtain S_1, then process m'_0, get a weak state $\mathtt{st}'_0$ corresponding to S_2, then try to decrypt ct_1 with S_2, which fails, but then try to decrypt ct_1 with S_1, which succeeds, and thus everything is committed to memory, including $\mathtt{st}_0$. We confirmed this behaviour by writing a Rust test inside libsignal, which demonstrated that R was copying $\mathsf{SPK}_R^{\mathsf{sk}}$ inside its memory.

Here, we presented this as if R would accept PQXDH sessions initialised by the attacker, and thus the attacker must own an identity trusted by R. However, an untrusted machine-in-the-middle attacker can also intercept the messages produced by other initiators trying to reach R and mount the attack, as it only requires mixing two distinct messages together.

Severity. Looking back to the computation of SK, out of the 5 secret elements in its derivation, the first three can be recomputed by an attacker that later learns the corresponding $\mathsf{SPK}_R^{\mathsf{sk}}$. Then, the forward secrecy hinges on the DH one-time prekey OPK_R, which is optional, and the shared secrecy from Kyber, SS. In the current hybridization of PQXDH and the DR, it is expected that it provides forward secrecy, either only using security assumptions over the curve X25519, or only using Kyber based assumptions. Our attack violates this property, as under the assumption that Kyber is insecure and X25519 is secure, an attacker can first force a user to store in memory $\mathsf{SPK}_R^{\mathsf{sk}}$ even after its deprecation, then in the case of a later compromise of the device state, learn $\mathsf{SPK}_R^{\mathsf{sk}}$. It can thereby recompute the SK of all sessions that relied on SPK_R and did not use an OPK if Kyber is insecure, and thus recompute all the message keys mk_j^1 of the first receiving chain of those sessions. Note that in an untrusted server setting, the server can make it so that one-time prekeys are never used. We stress that our attack does not affect the secrecy or authentication of the messages in the absence of key compromise, and are thus in an advanced threat model.

Discovery. This attack was not fully automatically found in PROVERIF, notably because we do not model the required Sesame components. However, while trying to model the composition of PQXDH with the DR we made a model that overapproximated the attacker's power, hoping it would be simpler to verify, by removing the key confirmation that takes place during PQXDH and instead relying solely on the key confirmation provided by the DR. This modelling gave us an attack trace, illustrating that an unconfirmed DR state produced by PQXDH should never be stored and amenable to compromise, as it contains a copy of $\mathsf{SPK}_R^{\mathsf{sk}}$ (a PROVERIF model illustrating this attack trace can be found in our artifact [11]). PROVERIF thus gave us the hindsight that what we called 'weak-states' exists and should never be stored. Through several code inspections with this hindsight in mind, we were able to discover how to make Signal store a weak state in practise.

Fix. The intuitive fix is to never try to decrypt ct with other sessions in the case of a new session initialization. Along with a proof of concept in Rust demonstrating that the official libsignal implementation did suffer from this issue, this fix was proposed to the Signal developers at the end of March 2025, and pushed inside the main branch of Signal a week later[1], and was then part of a responsible disclosure process with Meta for 5 months.

3.2.2. Replay attacks. Replay attacks are an inherent concern in PQXDH. Due to its asynchronous design, sessions built without an OPK_R lack any fresh randomness from R. This is a well-known fact mentioned in the PQXDH specification, which recommends implementing mitigation measures. In particular, incorporating a DR-like mechanism helps reduce the impact of potential replay attacks. In the libsignal implementation, some replay attacks are thwarted as R will not process twice messages that use the same $\mathsf{EK}_I^{\mathsf{pk}}$.

Small subgroup attack. X25519 is not a prime order group, and so called point-mangling attacks are possible (see e.g. [21]). Without delving into the mathematics behind it, for any $\mathsf{EK}_I^{\mathsf{pk}}$, there exists a simple algorithm that yields a second distinct $\mathsf{EK}_I^{*\mathsf{pk}}$, but such that for any honest secret key $\mathsf{SPK}_R^{\mathsf{sk}}$, $(\mathsf{EK}_I^{\mathsf{pk}})^{\mathsf{SPK}_R^{\mathsf{sk}}} = (\mathsf{EK}_I^{*\mathsf{pk}})^{\mathsf{SPK}_R^{\mathsf{sk}}}$. We can then easily mount a replay attack as follows:

C.1) Intercept any PQXDH message (m, ct) meant for R, and produce m^* by swapping $\mathsf{EK}_I^{\mathsf{pk}}$ by $\mathsf{EK}_I^{*\mathsf{pk}}$. Forward in a row (m, ct) and (m^*, ct) to R.

C.2) R computes two distinct sessions, both with the same SK, and then the same CKr_0^1 storing two states:
$\mathtt{st}_2 = (\mathsf{DHs}^2, \mathsf{DHr}^1, \mathsf{RK}^2, \mathsf{CKr}_0^1, \mathsf{CKs}_0^2)$
$\mathtt{st}'_2 = (\mathsf{DHs}'^2, \mathsf{DHr}^1, \mathsf{RK}'^2, \mathsf{CKr}_0^1, \mathsf{CKs}'^2_0)$

Severity. A replay attack leads to the initialization of two sessions with the same SK and then the same CKr_0^1. The compromise of the replayed session can then be used to break the forward secrecy of all the message keys mk_j^1 with $j > 0$ (the first message key mk_0^1 is still safely ratcheted forward in the copied session when it processes the first ciphertext). Such replay attacks break FS in a common scenario, for any sessions without any OPK and using the last-resort PQSPK. Note that in a scenario with a dishonest server, this can be the case for all sessions.

Discovery. Our standalone DR ratchet naturally required uniqueness of the SK for each session in order to prove FS or PCS. When attempting to compose with PQXDH, we effectively uncovered an attack on forward secrecy, as the PQXDH model did not account for replay protection (a PROVERIF model illustrating this attack trace can be found in our artifact [11]). Thus, PROVERIF gave us the hindsight that replay attacks on PQXDH can actually lead to a loss of FS, which in turn led us to investigate the concrete status of replay attacks on PQXDH and to carefully review the code.

1. https://github.com/signalapp/libsignal/commit/cd361186fba60c83186262d91b03af608fb8c7c3

Fix. It is possible to detect that we are receiving a mangled key, by checking if the key is "torsion-free" [21]. We reported this issue, once again with a small Rust proof of concept illustrating that the replay attack was indeed possible in libsignal, and with the recommendation to add the torsion free check. This was promptly pushed to libsignal in under a week[2]. Following our warning, Signal developers also discovered another way EK_I could be mangled to create an equivalent key, also fixed in the same commit.

3.2.3. Combining both. The two previous attacks enabled us to:

- Force the long-term storage of some fresh SK using the weak state attack;
- Create a duplicate session using the same SK as a target session using the replay attack.

As a final variant, we could in fact combine the two previous independent attacks into one, and make a target user store for the long term the SK of a session of their choice.

D.1) Intercept any session initialization message (m, ct) meant for R, as well as another ct' meant for R.
D.2) Mangle m into m^* using the small subgroup attack. Then, send in a row (m, ct) and (m, ct').

Compared with the replay attack, this also compromises the FS of mk_0^1, that is, of the first message received by R and corresponding to ct. Hence, we were able to break the FS of all the messages sent in the first exchange from I to R, for sessions without an OPK_R and using the last-resort PQSPK_R. This attack variant was of course fixed by solving both previous issues.

4. ProVerif model

We begin with a brief introduction to PROVERIF and discuss the challenges involved in faithfully translating the Signal specification, along with its intended security properties, into the tool's syntax. The resulting model is quite large and covers all aspects of the protocols. In this section, our focus will be on how states and the skipped-message features of Signal are represented. Although PROVERIF is capable of automatically analyzing many models, it cannot directly draw conclusions for complex ones such as ours. For this reason, we also explain in this section how we guided PROVERIF in order to obtain our proofs.

4.1. Modelling the Signal protocol

4.1.1. Background. PROVERIF [10] is a tool for automated verification of cryptographic protocols in the symbolic model against a Dolev-Yao attacker. Informally, this means the attacker can see and manipulate any message passed along the network on public channels.

Cryptographic primitives. The symbolic approach abstracts messages as *terms*, which capture the way messages are constructed rather than their concrete bit-level values. As such, cryptographic primitives are represented by function symbols, and their behaviour is idealised through an equational theory or a set of rewrite rules. For example, symmetric encryption can be modelled using a constructor–destructor pair:

```
fun encrypt(message,key):bitstring.
fun decrypt(bitstring,key):message
reduc forall k:key,msg:message;
      decrypt(encrypt(msg, k), k) = msg.
```

Here, `encrypt` acts as a constructor and `decrypt` as a destructor, with the single rewrite rule expressing that decryption inverts encryption when the same key is used. By specifying only this equation, we assume an idealised setting: decryption is possible only with the correct key, ciphertexts reveal no information about the plaintext (e.g., length or structure), and no other algebraic properties are exploitable.

Input language. The input to PROVERIF is a protocol specified in the applied π-calculus. This calculus resembles a functional programming language, featuring variables as well as `if` and `let` statements. Communication over the network is modelled by synchronous message passing on named channels, using the constructs `in` and `out` to receive and send messages respectively. The PROVERIF language also provides operators tailored to cryptographic modelling. For instance, the `new` operator generates a fresh value of a given type, representing random secret keys or other private data. The `event` constructs allow us to add annotations, invisible to the attacker, which are useful to express security properties. As an illustration, the process `System` below specifies the execution of an unbounded number of Signal Double Ratchet (DR) sessions in our model:

2. https://github.com/signalapp/libsignal/commit/5293caa6cad81566b80f3c3731857bedaa78737b

```
let System =
  !  (* Attacker decides participants *)
     in(att, (I:id, R:id));
     (* Data obtained from PQXDH *)
     let key_pair = generate_dh() in
     new RK:root_key;
     let AD = idPair(I,R) in
     (* Internal identifier for states *)
     new s_id:session_id;
     (* Participant session identifiers *)
     let s_infoI = session(s_id,I,R) in
     let s_infoR = session(s_id,R,I) in
     (* Event for queries *)
     event SPK(s_infoR, key_pair);
     let KeyPair(_,pub_key) = key_pair in
     out(att,pub_key);
       I_DR(s_infoI,RK,pub_key,AD)
     | R_DR(s_infoR,RK,key_pair,AD).
```

The operator `!` denotes replication of the process. With `att` being declared as a public channel, the construct `in(att, (I:id, R:id))` asks the attacker to supply a pair of agent identifiers that will initiate a DR session. The declarations `new RK:root_key` and `new s_id:session_id` specify the generation of fresh values: a root key and a session identifier, the latter serving to model session state. The event `SPK(s_infoR, key_pair)` indicates that a new session between `R` and `I` has started with `key_pair` being the DH keys of `R`. Finally, the construct `I_DR(s_infoI,...) | R_DR(...)` launches the two processes `I_DR` and `R_DR` concurrently, each instantiated with its corresponding arguments.

4.1.2. States. Signal provides documentation for the Double Ratchet protocol, including pseudocode specifications for its main functions [19], [20]. In principle, these specifications can be translated into PROVERIF almost line for line. For example, the pseudocode for ratchet encryption is given as:

```
state.CKs, mk = KDF_CK(state.CKs)
header=HEADER(state.DHs,state.PN,state.Ns)
state.Ns += 1
return header,
       ENCRYPT(mk, msg, CONCAT(AD,header))
```

which corresponds to the following PROVERIF snippet

```
let (cks,mk) = kdf_ck(CKs(st)) in
let hd = header(DHs(st),PN(st),Ns(st)) in
let new_state = update_cks_ns(st, cks) in
out(att,(hd,
         encrypt(mk, msg, concat(ad,hd))))
```

However, PROVERIF natively lacks certain features present in the pseudocode, most notably mutable *state*. The standard way to model mutable state in the applied π-calculus is through synchronous communication over private channels using the standard pattern below. We showcase here a state containing a single counter incremented by 1 at every step:

```
out(prv_ch,init_value) |
! in(prv_ch,cur_value:internal_state);
  P(cur_value);
  out(prv_ch,cur_value+1)
```

Assuming `prv_ch` is private, and since communication in the applied π-calculus is synchronous, an input `in(prv_ch, cur_value:internal_state)` can only occur when a corresponding output on the same channel is available, and vice versa. Thus, even though replication induces concurrent executions of the process `P`, the reliance on private channels ensures that executions effectively correspond to a sequential order:

```
P(init_value); P(init_value+1); ...
```

In this way, the private channel `prv_ch` serves as a representation of a *mutable state*. In Signal, the `internal_state` is not just a single counter but consists of a collection

```
 state(dhs,dhr,rk,cks,ckr,ns,nr,pn)
```

```
let RatchetInitInitiator(sinfo:session_info, SK:root_key, SPK_R:point) =
  let dhs = generate_dh() in
  let (rk:root_key,cks:chain_key) = kdf_rk(SK, dh(dhs,SPK_R)) in
  out(state_chan(sinfo), cell(state(dhs,SPK_R,rk,cks,none_ck,0,0,0),1,0,0)).

let RatchetEncrypt(sinfo:session_info, msg:bitstring, ad:associated_data) =
  in(state_chan(sinfo), cell(st,r_idx,m_idx,rcv_idx));
  let (cks:chain_key,mk:message_key) = kdf_ck(CKs(st)) in
  let hd = header(DHs(st),PN(st),Ns(st)) in
  let new_state = update_cks(st, cks) in
  out(att, (hd, encrypt(mk, msg, concat(ad,hd))));
  out(state_chan(sinfo), cell(new_state, r_idx, m_idx+1, rcv_idx)).

let InitiatorSendRatchet(sinfo:session_info, SK:root_key, SPK_R:point, ad:associated_data) =
  RatchetInitInitiator(sinfo,SK,SPK_R)
  | ! new msg:bitstring; RatchetEncrypt(sinfo, msg, ad).
```

Figure 2. PROVERIF modelling of a Signal participant initialising and sending an arbitrary number of messages

of current keys: the sending and receiving Diffie-Hellman keys (`dhs` and `dhr`), the root key (`rk`) and the sending and receiving chain keys (`cks` and `ckr`). In addition, it contains counters: the number of messages sent and received in the current ratchet step (`ns` and `nr`), and the number of messages sent in the previous ratchet step (`pn`).

We encapsulate this internal state inside a *cell*, which augments the protocol state with auxiliary metadata. These annotations are not part of the protocol itself but are introduced to facilitate the specification and proof of security properties, as discussed in Section 4.2. A cell is a collection of values, of the form:

```
cell(state(dhs,...,pn),r_idx,m_idx,rcv_idx)
```

Here, `r_idx` denotes the ratchet index, which tracks the current asymmetric ratchet; `m_idx` is the modification index, recording how many times the state has been updated; and `rcv_idx` is the receive index, which counts how many messages have been received, including skipped ones.

Since each participant must maintain a distinct mutable state for every session, we assign a private channel `state_chan(sinfo)` to each participant's mutable state. Here, `sinfo` is a tuple `session(s_id, A, B)`, where `s_id` is a fresh session identifier, `A` denotes the participant owning the state, and `B` is the communication partner of `A`. Figure 2 illustrates how mutable states are used within the process `InitiatorSendRatchet` where I initialises and sends an arbitrary number of messages.

4.1.3. Skipped messages. Signal allows messages to be received out of order. As a result, when a user receives a ciphertext, its corresponding symmetric key may not match the next expected symmetric ratchet step. Instead of failing, the user is required to advance through the ratchets (both symmetric and asymmetric) until the correct key is derived. Since the protocol is explicitly designed to prevent recovery of past message keys, the user must also maintain a record of skipped keys in order to later decrypt messages that arrive out of order. The essential features of this record are:

- Whenever a message key is skipped, it is added to the record.
- Whenever a message key is retrieved, it is deleted from the record.

To model a record in PROVERIF, the standard construct is **`table`**, which behaves as a global persistent database and which is invisible to the attacker but accessible to all honest agents. We therefore define a table `mkskipped` to store every skipped message key using the construct:

```
new ts:timestamp;
insert mkskipped(sinfo, dhr, nr, mk, m_idx, r_idx, ts);
```

Here, `mk` is the skipped message key, `ts` is a timestamp marking the insertion time, and the other arguments are taken from the internal state at the time of insertion. Message keys can later be retrieved using the **`get`** operator, which selects records according to specific arguments (here, both the session info and message number must match).

However, faithfully modelling the skipped-keys mechanism of Signal requires two additional features: skipped keys must be *deleted* once they are used, and they must be computed iteratively *until* a condition is satisfied. As with mutable states, PROVERIF does not natively support either deletion from tables or while loops. We now explain how we encode these features.

Deletion of skipped keys. Rather than attempting to explicitly delete entries from the table, we allow honest processes to potentially look up the same skipped key multiple times, but enforce via *restrictions* that each skipped key can only be looked up once in any valid execution trace. To do this, we annotate table lookups with an event `MKSKIPPED_del`:

```
(* Lookup some mk given a sinfo, dhr, nr *)
get mkskipped(=sinfo,=dhr,=nr,mk,_,_,_) in
event MKSKIPPED_del(sinfo,dhr,nr,mk);
```

and add the restriction:

```
restriction ...,i,j:time;
event(MKSKIPPED_del(sinfo,dhr,nr,mk))@i &&
event(MKSKIPPED_del(sinfo,dhr,nr,mk))@j ==> i = j.
```

This restriction ensures that if the event `MKSKIPPED_del` is triggered at two time steps `i` and `j` in the execution trace, then `i = j`, i.e. each skipped message key is consumed at most once. We show that this encoding is sound by proving in PROVERIF that no message key `mk` can be inserted twice into the table `mkskipped`.

Compromise of skipped keys. Our analysis considers the possibility that an attacker may compromise message keys. Since the attacker cannot directly access the table, we allow them to compromise skipped keys via a dedicated process `compromise_mkskipped`:

```
let compromise_mkskipped(...) =
  in(att,dhr:point,n:nat);
  get mkskipped(_,=dhr,=n,mk,_,r_idx,_) in
  event CompromiseMK(sif,mk,n,r_idx);
  out(att,mk).
```

This introduces a subtle issue: combined with our skipped key deletion model, the attacker could retrieve a message key `mk` *after it has already been used* (and should therefore no longer be available). To prevent this, we enforce an additional restriction:

```
restriction ...,i,j:time;
  event(CompromiseMK(sif,mk,n,r_idx))@i &&
  event(MessageKey(sif,_,_,mk,n,r_idx))@j
  ==> i < j.
```

Here, the event `MessageKey(sif,_,_,mk,n,r_idx)` denotes that the key `mk` was used to decrypt a message. The restriction enforces that compromise can only occur before a key is used, never afterwards.

While loops. In the Signal specification, the **`while`** loop is described by pseudocode of the form:

```
  while state.Nr < until:
    state.CKr, mk = KDF_CK(state.CKr)
    state.MKSKIPPED[state.DHr,state.Nr] = mk
    state.Nr += 1
```

Although PROVERIF does not provide native support for loops, we can exploit the fact that the loop condition depends only on a counter incremented by one at each iteration. Our encoding therefore models the loop body as an independent replicated process:

```
let SkipMessageKeysBody(sinfo,...) =
 !
 in(st_ch, cell(st,r_idx,m_idx,rcv_idx));
 let (ckr,mk) = kdf_ck(get_CKr(st)) in
 insert_skipped_message(..., mk,...);
 let st=set_nr(set_ckr(st,ckr), Nr(st)+1) in
 out(st_ch, cell(st,r_idx,m_idx+1,rcv_idx+1))
```

Meanwhile, the code to be executed after the loop is handled concurrently as illustrated below.

```
(* Process before the while loop *)
let until_condition = ... in
out(st_ch,val_before_loop) |
(* Process after the while loop *)
in(st_ch,val_after_loop);

(* Check that only skipped messages were executed *)
```

```
if only_skipped_keys(val_before_loop,val_after_loop) then

(* Check that loop condition was met *)
let cell(st2,_,_,_) = val_after_loop in
if until_condition = Nr(st2) then
  ...
```

Intuitively, once the output `out(st_ch, val_before_loop)` is executed, the scheduler may run the process `SkipMessageKeysBody` concurrently, thereby executing the body of the loop until the condition is satisfied. This is achieved by reading the mutable state via `in(st_ch, val_after_loop)` and checking that the condition `until_condition = Nr(st2)` holds.

Our encoding constitutes an overapproximation of the pseudocode. For example, it permits execution traces in which the loop body runs more times than intended, thus overshooting the `until` condition. Importantly, this overapproximation does not compromise the correctness of our analysis: forward secrecy and post-compromise security are verified independently for each execution trace. To limit excessive overapproximation, we additionally enforce that only skipped-message computations occur between the states *before* and *after* the loop, which we capture through the predicate `only_skipped_keys`.

4.1.4. Integration with Triple Ratchet. The Triple Ratchet contains the DR and an additional ratcheting key generation mechanism, SPQR. These two components sit alongside each other, with completely separate and non-communicating states. When a message needs to be encrypted or decrypted, the corresponding message key is taken from the DR and SPQR and these are combined using `KDF_HYBRID`. We modify our code so that, whenever we encrypt or decrypt a message, the attacker provides the SPQR message key. For example:

```
let RatchetEncrypt =
  ...
  let (cks,dr_mk) = kdf_ck(CKs(st)) in
  (* SPQR message key chosen by attacker *)
  in(att, pq_mk);
  let tr_mk = kdf_hybrid(dr_mk, pq_mk) in
  let ciphertext = encrypt(tr_mk,plaintext, concat(ad,hd)) in
    ...
```

In this sense we model the Triple Ratchet, under the assumption that SPQR is completely insecure.

4.2. Modelling and proving security properties

We are interested in two security properties for our model: forward secrecy and post-compromise security. Both properties concern the secrecy of critical components in the presence of key compromise by an attacker. Forward secrecy ensures that any data sent before a compromise remains confidential, even when long-term or session keys are compromised. Post-compromise security, on the other hand, captures the self-healing nature of the protocol: even after some keys have been compromised, the protocol regains security provided the attacker does not interfere during a sufficient recovery window. Both properties can be naturally expressed in PROVERIF using *correspondence queries*.

4.2.1. Compromise of keys. We showed before how the compromise of skipped message keys is modelled by allowing the attacker to run a process that explicitly leaks these keys on a public channel. To track such events, we emit an event `CompromiseMK(sif,mk,n,r_idx)` immediately *before* the key is leaked. We apply the same principle to all other keys generated during the protocol and stored in memory cells. For instance, the following process models the compromise of root keys contained within a cell:

```
! in(state_chan(sinfo),val_cell);
  out(state_chan(sinfo),val_cell);
  let cell(st,r_idx,_,_) = val_cell in
  event CompromiseRK(sinfo,RK(st),r_idx);
  out(att,RK(st)).
```

Similarly, we allow compromise of Diffie-Hellman sending keys and of the sending and receiving chain keys, with corresponding events

```
CompromiseDHs(sinfo,DHs(st),r_idx)
CompromiseCKs(sinfo,CKs(st),NS(st),r_idx)
CompromiseCKr(sinfo,CKs(st),NR(st),r_idx)
```

Note that each event is annotated with the ratchet index, and in the case of chain keys, also with the ratchet step. These annotations enable us to write precise formal statements of forward secrecy and post-compromise security.

```
query
  let sinfoS = session(sd,idA,idB) in
  let sinfoR = session(sd,idB,idA) in
  (* If mk is a key used by idA to send a message and the attacker knows it then... *)
  event(MessageKey(sinfoS,Sender,mk,n,r_idx)) && attacker(mk) ==>
    (* ... The DR part of the mk was leaked while stored in the skipped message store *)
    event(CompromiseMK(sinfoR,dr_mk,n,r_idx+1)) && mk = kdf_hybrid(dr_mk,pq_mk) ||
    (* ... or a chain key of the chain corresponding to mk was leaked on the sender's side *)
    (event(CompromiseCKs(sinfoS,ck,ns,r_idx)) && ns <= n ) ||
    (* ... or on the receiver's side *)
    (event(CompromiseCKr(sinfoR,ck,nr,r_idx+1)) && nr <= n) ||
    (* ... or a past root key was leaked on either side. *)
    (event(CompromiseRK(sinfo,rk,r_idx')) && r_idx' < r_idx && (sinfo = sinfoS || sinfo =
    sinfoR)).
```

Figure 3. PROVERIF encoding of forward secrecy on the sender's side for the Signal Protocol.

4.2.2. Modelling. Similarly to the restrictions introduced earlier, correspondence queries are logical implications of the form $F_1 \wedge \ldots \wedge F_n \Rightarrow \Phi$ which hold if, for every execution trace T, whenever $F_1, \ldots, F_n$ hold in T, then Φ also holds in T. Here, $F_1, \ldots, F_n$ are facts and Φ is a formula built from facts using conjunctions and disjunctions. As we have seen, facts may be of the form `event(ev)@i`, where `i` is a time variable marking the occurrence of the event. PROVERIF also provides built-in facts such as: `attacker(t)@i`, stating that the attacker knows term `t` at time step `i`; and `mess(c,t)@i`, stating that term `t` was sent over channel `c` at time step `i`.

This allows us to state, in Figure 3, the PROVERIF query modelling forward secrecy of message keys when used to encrypt a message on the sender's side. Note that we refer here to the agents as A and B, as inside the ratchet, they are alternating between sender and receiver roles, and who initiated the original key-exchange is not relevant anymore. Intuitively, if a message key used by a sender `A` in its `r_idx`-th ratchet with receiver `B` becomes known to the attacker, then one of the following conditions must hold:

1) The message key was compromised as a skipped message by the receiver `B` in its current `r_idx+1`-th receiving ratchet.
2) The current `r_idx`-th sending ratchet was compromised prior to the encryption (`ns <= n`).
3) The current `r_idx+1`-th receiving ratchet was compromised prior to the decryption (`nr <= n`).
4) A root key was compromised in some prior ratchet (`r_idx' < r_idx`) on either side.

Recall that the sender and receiver ratchets always differ by one, so that a participant receiving messages is one ratchet ahead of the sender. This is why the `r_idx+1`-th ratchet is the correct one to consider for the receiver in this property.

The formalisation of post-compromise security, shown in Figure 4, is more involved than that of forward secrecy, as it requires reasoning about a healing process. The query begins in a similar manner: we assume that a message key `mk`, used by `A` in its `r_idx`-th sending ratchet with `B`, is known to the attacker. In addition, we assume the existence of a *healing ratchet*, identified as `A`'s `r_idx_heal`-th ratchet where the receiving Diffie-Hellman key is honest, and which occurs before the problematic ratchet (`r_idx_heal <= r_idx`). Under these assumptions, PCS requires that the same compromise conditions as in forward secrecy apply with respect to skipped message keys and chain keys. The main difference concerns root keys, PCS requires that either:

1) A root key was compromised after the healing ratchet: `r_idx_heal <= r_idx_comp < r_idx`
2) Or, a root key was compromised before the healing ratchet, but one of the Diffie-Hellman keys of the healing ratchet (sending or receiving) must also have been compromised.

For completeness, Section A provides similar queries for the forward secrecy and post-compromise security of message keys on the receiver's side, when used for decryption.

4.2.3. Guiding PROVERIF to complete proofs. The problem of verifying security properties, even simple secrecy, is undecidable in general. Tools such as PROVERIF are always sound, but they may fail to terminate and may sometimes report false attacks (false negatives). For many reasonable protocol models, PROVERIF requires little to no guidance and can reach satisfactory conclusions. However, our model pushes the limits of PROVERIF's automation, particularly because it relies on mutable state, table deletions, and while loops. As a consequence, without additional guidance, PROVERIF is unable to prove forward secrecy or post-compromise security.

```
query
  let sinfoS = session(sd,idA,idB) in
  let sinfoR = session(sd,idB,idA) in
  (* If mk is a key used by idA to send a message and the attacker knows it ... *)
  (* And there was some 'honest' DHs key received in the past by idA, forming a 'heal' ... *)
  event(MessageKey(sinfoS,Sender,dhs, mk,n,r_idx)) && attacker(mk) &&
  mess(state_chan(sinfoS,cell(st,r_idx_heal,_,_))) &&
  event(HonestDH(sinfoR,dhr_prv',dhr',r_idx')) &&
  DHr(st) = dhr' && r_idx_heal <= r_idx ==>
  ( (* Then the skipped messages or chain keys were compromised as in FS ... *)
     event(CompromiseMK(sinfoR,dr_mk,n,r_idx+1)) && mk=kdf_hybrid(dr_mk,pq_mk) ||
    (event(CompromiseCKs(sinfoS,_,ns,r_idx)) && ns <= n ) ||
    (event(CompromiseCKr(sinfoR,_,nr,r_idx+1)) && nr <= n) ||
    (* Or a past root key was compromised ... *)
     event(CompromiseRK(sinfo,_,r_idx_comp)) && (sinfo = sinfoS || sinfo = sinfoR) && (
      (* ... after the heal *)
      (r_idx_heal <= r_idx_comp < r_idx) ||
      (* ... or before the heal but one of DH keys of healing session is compromised *)
      (r_idx_comp < r_idx_heal &&
        (event(CompromiseDHs(sinfoR, dhr_prv', r_idx'+2))) ||
        (event(CompromiseDHs(sinfoS, dhs_prv', r_idx_heal))))))
```

Figure 4. PROVERIF encoding of post-compromise security on the sender's side for the Signal Protocol.

Lemmas and proof by induction. A now standard way to handle proofs in complex models with loops is to use *lemmas* together with PROVERIF's ability to prove queries by induction. Since the protocol loops around mutable internal states, we configure PROVERIF to prove FS and PCS by induction over the time step at which the facts in the premises of the queries occur. This prevents PROVERIF from attempting to explore an infinite chain of keys, and thereby ensures termination. Intuitively, PROVERIF inductively assumes that a participant has reached some internal state, typically represented by an input such as:

```
in(state_chan(sinfo), cell(st,r_idx,m_idx,rcv_idx))
```

and then attempts to prove the queries for the next value of the mutable state.

However, this approach alone is insufficient, as PROVERIF has no built-in knowledge of the structure of mutable states: it only assumes their existence. To overcome this, we must supply PROVERIF with invariants describing the properties of mutable states. This is achieved through the use of lemmas, a feature introduced in [22]. Lemmas are auxiliary queries that PROVERIF first proves and then uses in the proofs of the main queries (FS and PCS). For example, the lemma

```
lemma
  mess(state_ch(s),cell(st,r_idx,m_idx,rcv_idx)) &&
  mess(state_ch(s),cell(st',r_idx',m_idx',rcv_idx')) &&
  DHs(st) = DHs(st') ==>
    DHr(st) = Dhr(st') && r_idx = r_idx' && PN(st) = PN(st') && RK(st) = RK(st').
```

states that if two instances of a mutable state for the same user share the same sending Diffie-Hellman key (`DHs(st) = DHs(st')`), then they must also share the ratchet index (`r_idx = r_idx'`), the ratchet steps (`PN(st) = PN(st')`), the receiving Diffie-Hellman keys (`DHr(st) = DHr(st')`) and the root keys (`RK(st) = RK(st')`).

A substantial part of our effort was devoted to identifying and formalising the correct invariants over mutable states, and to understanding how to strengthen the inductive hypotheses necessary for PROVERIF to prove FS and PCS by induction. This involved both diagnosing why PROVERIF would not terminate or yielded false negatives, and expressing all intermediate security properties verified in the DR by the various protocol elements. Once these invariants and lemmas were properly defined however, the remainder of the process was fully automatic: PROVERIF could then complete all proofs without further manual intervention.

Concretely, our model includes 7 generic lemmas common to both FS and PCS, covering over 30 distinct invariants over the states, as well as 2 lemmas specific to FS (showing first FS of root and chain keys) and 7 lemmas specific to PCS (similarly showing PCS of root and chain keys, as well as the compromise properties of Diffie-Hellman and root keys). Some of these auxiliary lemmas, for instance variants of PCS for root and chain keys, were instrumental in allowing PROVERIF to establish PCS for message keys.

GSVERIF and counters. Although modelling mutable state via private channels is standard in the applied π-calculus, PROVERIF has historically struggled to verify protocols that rely on this encoding. To address this, a front-end called

GSVERIF was developed in [23] to better handle mutable state. Their methodology is particularly well-suited for states containing monotonically increasing counters, as in our case with the ratchet index (`r_idx`), modification index (`m_idx`), and receive index (`rcv_idx`). However, directly using the GSVERIF front-end resulted in non-termination in our setting, due to its highly generic transformations being applied indiscriminately. Instead, we relied on a PROVERIF library introduced in [24], which allowed us to apply the methodology of GSVERIF in a more fine-grained and controlled way.

4.3. Extension of our model

Although our primary focus is the Double Ratchet specification, we also consider several variants of the Signal model. These variants differ according to whether Signed Pre-Keys (SPKs) can be reused, whether header encryption is enabled, and whether the attacker is allowed to compromise sender Diffie-Hellman keys. Instead of developing a separate model for each variant, we construct a single parameterised model. An example configuration is:

```
letfun allow_same_SPK = true.
letfun allow_HE = false.
letfun allow_compromise_DHs = true.
```

This particular configuration (allowing SPK reuse and all key compromises, including DHs, but disabling header encryption) corresponds to the Signal implementation under our threat model. We however must analyse multiple variants, as the more general the model, the more challenging it is for PROVERIF to complete proofs. For example, we encountered out-of-memory errors when attempting to prove PCS with both header encryption and DH compromise enabled, but could successfully obtain proofs by disallowing DH key compromise.

Reuse of SPKs. In a session between I and R initiated via PQXDH, the two initial parameters are the shared secret root key SK and R's signed pre-key (SPK_R). In principle, SK should be unique for each session. However, as shown in Section 3.2.2, this is not guaranteed on R's side, which may lead to replay attacks. To simplify the expression of security properties (e.g. forward secrecy, which only holds after the third ratchet with SK reuse), we assume fresh SK values in our model. Signed pre-keys, on the other hand, are explicitly medium-term keys and may be reused across sessions. Accordingly, our model includes a concurrent process `genSPKs` which generates SPKs for each identity and stores them in a global table `SPKs`:

```
(* Generate SPKs for ids *)
let genSPKs =
  ! in(att,idR:id);
  let spk = generate_dh() in
  insert SPKs(idR, spk).
```

When the Signal protocol assigns to R an SPK, if `allow_same_SPK` is enabled, R retrieves its key from the `SPKs` table (which may or may not be fresh); otherwise, a new SPK is generated:

```
  ! in(att, (idI:id, idR:id));
    let responder_dh_key_pair =
      if allow_same_SPK then
        (get SPKs(=idR, spk) in spk)
      else generate_dh()
    in ...
```

Header Encryption. The Signal documentation provides a specification for encryption of headers. Headers contain the message number (n), the number of messages in the previous ratchet (pn) and the current ratchet public key (DHr). The purpose of this encryption is to prevent the attacker from distinguishing messages or inferring their ordering. Achieving indistinguishability proofs in PROVERIF, while possible, is considerably more difficult and would likely require substantial modifications to our model. Therefore, within the scope of this paper, we restrict our analysis to proving that header encryption does not break forward secrecy or post-compromise security. To this end, we deliberately overapproximate the attacker's ability to interact with headers. Specifically, the attacker may decide, for each session, whether header encryption is used. If it is, the attacker can additionally choose the first two header keys and each nonce used for header encryption. This reflects the fact that the derivation of the initial header keys is not given in the specification, and there are multiple options of where to source the nonce.

Composition with PQXDH. Our ultimate goal is to obtain a fully automated proof of the entire Signal protocol, including the initial setup via PQXDH. We reuse a PROVERIF model [4] for standalone PQXDH, which has been analysed for various security properties and which establishes the (out of band) authentication of the identity keys, a shared root key SK and an initial DHr key SPK. The out-of-band authentication is modelled via a table mapping identities to public keys. As this table is inaccessible to the attacker, it suffices to model authenticated key distribution.

Proved	DR	DR-HE*	DR-HE	DR-PQXDH*
Sanity	✓(7s/0.3GB)	✓(5m/2.6GB)	✓(5m/2.6GB)	✓(38m/18GB)
Lemmas	✓(1m/0.3GB)	✓(1h/2.2GB)	✓(1h/2.6GB)	✓(4h/10GB)
FS	✓(8s/0.1GB)	✓(2m/0.2GB)	✓(12m/0.5GB)	✓(10m/1.0GB)
PCS	✓(40m/2.7GB)	✓(22h/38GB)	?	✓(7h/46GB)

TABLE 2. Time (in second, minutes or hours) and memory for proofs of variants of the Double Ratchet (DR) protocol in PROVERIF. DR refers to the version without header encryption (HE) but with SPK reuse and DHs compromise. PQXDH and HE refer to the protocol composed with PQXDH and with header encryption, respectively. The star $*$ indicates a restriction on the scenario considered: for DR-HE*, we disable the compromise of DHs, and for DR-PQXDH*, there is a single PQXDH session without OPKs, DHs compromise, or PQXDH key compromises.

In practice, I initiates the session asynchronously: when it decides to contact R, I queries the server for the necessary information and uses it to generate cryptographic material. In standalone PQXDH, I would then send R a message containing public key material, which it would use to derive the matching key. This message would also carry some encrypted data, often the first message of the Double Ratchet.

When composing PQXDH with DR, however, we must allow for out-of-order delivery: R's first received message might be any message from I's initial ratchet. Therefore, I must send the initial key material together with every message from its initial ratchet. This is specified in the Signal documentation, but complicates modular composition of PQXDH and DR, since R's setup process may involve advanced DR functions such as Skipped Messages keys.

To address this, we slightly overapproximate the protocol behaviour. When I initiates a session, I derives the initial cryptographic values and publicly outputs the key bundle while also initialising its ratchet. At this stage, no messages have yet been sent to R, unlike in the standalone PQXDH model. R, in turn, listens for its first combined PQXDH + DR message: R first receives the initial key bundle to initialise its ratchet, and then waits for the first DR message to arrive and decrypts it in the standard DR manner. Importantly, this 'first' DR message is not necessarily the first DR message output by I, as these may be received out of order. Rather, any DR message in the first ratchet could be received here. In a nutshell, we conceptually split the initial PQXDH + DR packet into its components.

Note that, as discussed in Section 3.2.1, allowing storage of weak states such as this does make the protocol weaker. In order to mitigate this we could add restrictions, for example disallowing compromise of 0th ratchet states.

5. ProVerif Double Ratchet analysis

For the DR, we proved forward secrecy and post-compromise security. This required proving several additional helping lemmas. In addition, to strengthen confidence in our model, we conducted a set of sanity checks confirming its executability, particularly by illustrating that two participants are able to exchange messages successfully. We considered several variants with different numbers of messages and alternating the sender and receiver roles.

Our main results are shown in Table 2. The server on which these were run had 378GB of RAM, and a 20 core Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz, though PROVERIF is single threaded. The proofs are run on a version of the protocol that does not suffer from the weaknesses outlined in Section 3, in that: the `none_ck` cannot be used in decryption, and each session has a distinct SK. We did not explicitly model the fix for the weak state attack, since it only appears when composing with PQXDH. In this case, due to memory limitations we had additional restrictions on the attacker's behaviour that meant that the attack was not possible. We considered several variants of the protocol as discussed in the earlier section on parametrisation, we discuss the results for the most interesting versions.

The Double Ratchet (DR). For the model which we consider to be accurate to the standalone Double Ratchet as implemented, all proofs were successful. Most ran quickly, with PCS being the only proof to take significantly longer than a minute, taking 40 minutes. The memory requirements are also reasonably modest, with the exception of PCS at 2.7GB all are well under 1GB. This gives an automated proof of the Double Ratchet, assuming the freshness and secrecy of SK.

The Double Ratchet with header encryption (DR-HE* / DR-HE). Header encryption adds little complexity to the protocol but it does result in a larger state space for the protocol since we now have four additional state variables. As a result we were unable to prove PCS on the full version (DR-HE) of this, since we ran out of memory. We ran the same proofs with DHs compromise disabled (denoted DR-HE*) and in this case we were able to prove all results. This still takes significantly more resources than the basic proof, with post-compromise security requiring about 38 GB and taking just under 22 hours. It's possible that this is due to our lemmas not proving enough information about the structure of header keys, and that if there were more lemmas then these proofs would use less resources.

The Double Ratchet composed with PQXDH. Composition with PQXDH adds significant extra information to the cell. For example, the initial SK now contains three or four Diffie-Hellman calculations, rather than just a fresh value. The resulting state space increase makes even the simplest lemmas currently impractical. In order to get some guarantees, we restrict the possible attacks, and the behaviours of a process. We disallowed the use of one-time pre keys, restricted to one session, and did not allow attacks on either PQXDH or DHs keys. With these restrictions in place, we were able to show sanity, the state lemmas, forward secrecy, and post-compromise security.

Attack traces and fixes. To ensure the correctness of our model and consequently the validity of our proofs we conducted a series of sanity check queries. These queries aimed to verify the executability of the model by examining small scenarios involving only a few message exchanges, allowing us to test the various components of DR. As shown in Table 2, PROVERIF may still require a significant amount of time to verify these sanity checks. This is partly because the tool is optimized for proof generation rather than attack discovery, especially in models as large as ours.

In addition, we verified that our model could capture the attacks described in Section 3, or at least some of their core aspects. For instance, the attack on the specification was mitigated in our model by checking that CKr is different from `None` before any decryption attempt. Removing this check enables PROVERIF to rediscover the attack that breaks forward secrecy. Regarding replay attacks, we did not directly model point-mangling attacks on X25519 or their interactions with PQXDH. However, focusing solely on the DR specification, we demonstrated using PROVERIF that allowing an SK to be reused across two distinct sessions compromises forward secrecy.

6. Limitations and future work

Our analysis of the DR specification is rather complete, but we do not provide strong results for the composition of the DR with either PQXDH or SPQR, nor do we include the analysis of Sesame in our models. Bridging these gaps would be of strong interest, especially as the main weaknesses we uncovered lie precisely at the intersection of the core three components. However, as we believe that we have pushed as far as we can with the capabilities of PROVERIF, such future work might either need new developments on the PROVERIF side, or novel ideas on how to verify and model the composition of the protocols.

Further, PROVERIF makes several approximations. Notably, it models Diffie-Hellman operation in a very abstract way and typically does not model that the attacker may compute the inverse of group elements, hence, an analysis of DR using TAMARIN might be of interest, or a more precise DH model in PROVERIF. Finally, PROVERIF only provides guarantees against a symbolic attacker where cryptography is idealized, so obtaining computer-aided security proofs with computational guarantees is also an important goal.

Ethics considerations

Weak State Attack and Replay Attack. For both of these attacks, a report and a small prototype based on the libsignal implementation illustrating the attacks were transmitted to Signal Messenger's security team. The weak state attack was reported at the end of March 2025, and the replay attack in September 2025. For both, a fix was pushed to the main branch of libsignal in under a week.

To follow up, as WhatsApp and Facebook Messenger both license libsignal, the reports were transmitted to Meta's security team. For the weak state attack, a fix was deployed for WhatsApp Web and Android in early September 2025. For the replay attack, at the time of submission, we did not know whether the issue had been fixed, as we had not received updates from Meta for some time. During the reporting process, we encountered some difficulties with access to the bug reporting platform. In particular, the Facebook account created for submission was subsequently deactivated, which limited our ability to manage the report through the interface. Although we were able to maintain partial communication via email, this did not fully resolve the situation, and some requested actions required access to the interface we could not use. In the absence of further updates, and in accordance with the responsible disclosure timeline agreed with Meta until December 2025, we proceeded with the publication of this article.

Null Decryption Key. The null decryption key attack on the DR specification was reported and acknowledged in February 2024. After some delay due to the major SPQR update that was in progress at the time, it was fixed in November 2025 by updating the specification of the `KDF_CK` function which must fail on a None input.

Acknowledgments

The authors would like to thank Rolfe Schmidt for the interesting discussions, and all the Signal Messenger developers for very efficiently implementing solutions to every issue. This work benefited from funding managed by the French National

Research Agency under the France 2030 programme with the reference ANR-22-PECY-0006. The third author was supported by an Oxford-DeepMind Graduate Scholarship. This work was supported by the European Research Council (ERC) under the European Union's Horizon Europe research and innovation programme through the ERC Synergy Grant VePaSS (Grant agreement No. 101224640).

References

[1] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A formal security analysis of the signal messaging protocol," *Journal of Cryptology*, vol. 33, pp. 1914–1983, 2020.

[2] J. Alwen, S. Coretti, and Y. Dodis, "The double ratchet: security notions, proofs, and modularization for the signal protocol," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2019, pp. 129–158.

[3] A. Bienstock, J. Fairoze, S. Garg, P. Mukherjee, and S. Raghuraman, "A more complete analysis of the signal double ratchet algorithm," in *Annual International Cryptology Conference*. Springer, 2022, pp. 784–813.

[4] K. Bhargavan, C. Jacomme, F. Kiefer, and R. Schmidt, "Formal verification of the PQXDH Post-Quantum key agreement protocol for end-to-end secure messaging," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 469–486.

[5] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *2017 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 435–450.

[6] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Wurtele, "Dy*: A modular symbolic verification framework for executable cryptographic protocol code," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021, pp. 523–542.

[7] C. Cremers, C. Jacomme, and A. Naska, "Formal analysis of Session-Handling in secure messaging: Lifting security from sessions to conversations," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1235–1252.

[8] B. Schmidt, S. Meier, C. Cremers, and D. A. Basin, "Automated analysis of diffie-hellman protocols and advanced security properties," in *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, S. Chong, Ed. IEEE Computer Society, 2012, pp. 78–94. [Online]. Available: https://doi.org/10.1109/CSF.2012.25

[9] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Trans. Dependable Secur. Comput.*, vol. 5, no. 4, pp. 193–207, 2008.

[10] B. Blanchet *et al.*, "Modeling and verifying security protocols with the applied pi calculus and proverif," *Foundations and Trends® in Privacy and Security*, vol. 1, no. 1-2, pp. 1–135, 2016.

[11] V. Cheval, C. Jacomme, and J. Richards. (2026) ProVerif models of the Double Ratchet. [Online]. Available: https://github.com/VincentCheval/SignalDoubleRatchetVerif

[12] D. Collins, D. Riepel, and S. A. O. Tran, "On the tight security of the double ratchet," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4747–4761.

[13] R. Canetti, P. Jain, M. Swanberg, and M. Varia, "Universally composable end-to-end secure messaging," in *Annual International Cryptology Conference*. Springer, 2022, pp. 3–33.

[14] O. Blazy, I. Boureanu, P. Lafourcade, C. Onete, and L. Robert, "How fast do you heal? a taxonomy for post-compromise security in secure-channel establishment," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5917–5934.

[15] F. E. Linker, C. Sprenger, C. Cremers, and D. Basin, "Looping for good: Cyclic proofs for security protocols," in *32nd ACM Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, 2025.

[16] F. Linker, R. Sasse, and D. Basin, "A formal analysis of apple's iMessage PQ3 protocol," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 5015–5034.

[17] C. Cremers, G. Horowitz, C. Jacomme, and E. Ronen, "Token weaver: Privacy preserving and post-compromise secure attestation," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 4173–4191.

[18] I. Boureanu, C. Onete, S. Wesemeyer, L. Robert, R. Miller, P. Lafourcade, and F. Rajaona, "Post-compromise security with application-level key-controls–with a comprehensive study of the 5g akma protocol," in *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, 2025, pp. 231–247.

[19] T. Perrin, M. Marlinspike, and R. Schmidt. (2025) The Double Ratchet Algorithm. [Online]. Available: https://signal.org/docs/specifications/doubleratchet/

[20] E. Kret and R. Schmidt. (2023) The PQXDH Key Agreement Protocol. [Online]. Available: https://signal.org/docs/specifications/pqxdh/

[21] C. Cremers and D. Jackson, "Prime, order please! revisiting small subgroup and invalid curve attacks on protocols using diffie-hellman," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 78–7815.

[22] B. Blanchet, V. Cheval, and V. Cortier, "Proverif with lemmas, induction, fast subsumption, and much more," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 69–86.

[23] V. Cheval, V. Cortier, and M. Turuani, "A little more conversation, a little less action, a lot more satisfaction: Global states in proverif," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 344–358.

[24] V. Cheval, V. Cortier, and A. Debant, "Election verifiability with proverif," in *36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 10-14, 2023*. IEEE, 2023, pp. 43–58. [Online]. Available: https://doi.org/10.1109/CSF57540.2023.00032

Appendix A. Additional PROVERIF queries

We include here the PROVERIF encodings for the remaining proof statements. Forward secrecy on the receiver's side is in Figure 5 and post-compromise security on the receiver's side is in Figure 6.

```
query
  let sinfoS = session(sd,idA,idB) in
  let sinfoR = session(sd,idB,idA) in
  (* If mk is a key used by idB to receive a message and the attacker knows it then... *)
  event(MessageKey(sinfoR,Receiver,mk,n,r_idx)) && attacker(mk) ==>
    (* ... The DR part of the mk was leaked while stored in the skipped message store *)
    event(CompromiseMK(sinfoR,dr_mk,n,r_idx+1)) && mk = kdf_hybrid(dr_mk, pq_mk) ||
    (* ... or a chain key of the chain corresponding to mk was leaked on the sender's side *)
    (event(CompromiseCKs(sinfoS,ck,ns,r_idx')) && r_idx' + 1 = r_idx && ns <= n ) ||
    (* ... or on the receiver's side *)
    (event(CompromiseCKr(sinfoR,ck,nr,r_idx)) && nr <= n) ||
    (* ... or a past root key was leaked on either side. *)
    (event(CompromiseRK(sinfo,rk,r_idx')) && r_idx' + 2 <= r_idx && (sinfo = sinfoS || sinfo
    = sinfoR)).
.
```

Figure 5. PROVERIF encoding of forward secrecy on the receiver's side for the Signal Protocol.

```
query
  let sinfoS = session(sd,idA,idB) in
  let sinfoR = session(sd,idB,idA) in
  (* If mk is a key used by idB to receive a message and the attacker knows it ... *)
  (* And there was some 'honest' DHs key received in the past by idB, forming a 'heal' ... *)
  event(MessageKey(sinfoR,Receiver,dhs, mk,n,r_idx)) && attacker(mk) &&
  mess(state_chan(sinfoR,cell(st,r_idx_heal,_,_))) &&
  event(HonestDH(sinfoR,dhr_prv',dhr',r_idx')) &&
  DHr(st) = dhr' && r_idx_heal <= r_idx ==>
  (
    (* Then the skipped messages or chain keys were compromised as in FS ... *)
    event(CompromiseMK(sinfoR, dr_mk,n,r_idx)) && mk=kdf_hybrid(dr_mk,pq_mk)||
    (event(CompromiseCKs(sinfoS,_,ns,r_idx'')) && r_idx'' + 1 = r_idx && ns <= n ) ||
    (event(CompromiseCKr(sinfoR,_,nr,r_idx)) && nr <= n) ||
    (* Or a past root key was compromised ... *)
    event(CompromiseRK(sinfo,_,r_idx_comp)) && (sinfo = sinfoS || sinfo = sinfoR) && (
      (* ... after the heal *)
      (r_idx_heal <= r_idx_comp + 2 <= r_idx) ||
      (* ... before the heal but one of DH keys of healing session is compromised *)
      (r_idx_comp < r_idx_heal &&
        (event(CompromiseDHs(sinfoR, dhr_prv', r_idx'+2))) ||
        (event(CompromiseDHs(sinfoS, dhs_prv', r_idx_heal)))
      )))
```

Figure 6. PROVERIF encoding of post-compromise security on the receiver's side for the Signal Protocol.