Multi-Device for Signal

Sébastien Campion[3], Julien Devigne[1], Céline Duguey[1,2], and Pierre-Alain Fouque[2]

[1] DGA Maîtrise de l'information, Bruz, France julien.devigne@intradef.gouv.fr
[2] Irisa, Rennes, France, celine.duguey@irisa.fr, pierre-alain.fouque@irisa.fr

Abstract. Nowadays, we spend our life juggling with many devices such as smartphones, tablets or laptops, and we expect to easily and efficiently switch between them without losing time or security. However, most applications have been designed for single device usage. This is the case for secure instant messaging (SIM) services based on the Signal protocol, that implements the Double Ratchet key exchange algorithm. While some adaptations, like the Sesame protocol released by the developers of Signal, have been proposed to fix this usability issue, they have not been designed as specific multi-device solutions and no security model has been formally defined either. In addition, even though the group key exchange problematic appears related to the multi-device case, group solutions are too generic and do not take into account some properties of the multi-device setting. Indeed, the fact that all devices belong to a single user can be exploited to build more efficient solutions.
In this paper, we propose a Multi-Device Instant Messaging protocol based on Signal, ensuring all the security properties of the original Signal.

Keywords: cryptography, secure instant messaging, ratchet, multi-device

1 Introduction

1.1 Context

Over the last years, secure instant messaging has become a key application accessible on smartphones. In parallel, more and more people started using several devices - a smartphone, a tablet or a laptop - to communicate. They need to be able to frequently and rapidly switch between them. Security protocols such as SIM have to be adapted to this ever-changing multi-device setting. However, the modifications have to be as light as possible for the users and efficient so that it will be the same if we use this or that device.
The Double Ratchtet algorithm, implemented in the Signal protocol, is currently the leading key management algorithm for SIM. It is implemented in WhatsApp (1.5 billion of users, for 60 billions of messages sent each day [3]), in Facebook Messenger as an optional secret conversation mode (1,3 billion of users [4]), in

[3] http://techcrunch.com - Facebook Q4 2017 earnings announcement
[4] www.socialmediatoday.com/news Facebook Messenger by the numbers 2019

Wire, Viber, Google Allo and, of course, in the Signal app itself. It has been released in 2016 by its designers Perrin and Marlinspike [21]. The idea of the Double Ratchet, and of other ratcheted key exchanges (RKE), is to propose a continuous update of session keys. The interesting property of this protocol is that the confidentiality of past and future messages is still guaranteed even after an exposure of long-term keys or even of state secrets by a passive adversary. This forces the adversary to expose keys regularly. Those features are often identified as forward secrecy and healing (or future secrecy, post-compromise security). Regrettably, the Double Ratchet algorithm has been designed for device to device interaction and its use in a multi device context is more difficult. Consequently, each SIM application has developed its own strategy to solve this problem.

1.2 Existing solutions

WhatsApp. The most widely used SIM is designed to be used on a single phone. However, in order to enable its users to communicate from a computer, WhatsApp developers released WhatsApp web. This interface establishes a secure channel between the "master phone" and the computer, with the former just pushing data from the server to the latter, and conversely. Thus, a user can use WhatsApp from its computer only if his phone is also connected.

Facebook Messenger. This SIM enables end-to-end encryption as an option, called secret conversation. A technical white paper issued in May 2017 [13] explains that "Secret conversations with more than two devices use the Signal Protocol's group Messaging Protocol". In this solution, called Sender's Key, each device sends (through a Signal Channel unused afterward) a same symmetric key: the Sender's key. This key is ratcheted through a key derivation function (KDF), without additional key exchange information. This protocol does not achieve future secrecy and does not offer the security we are looking for.

Signal. In April 2017, Open Whisper Systems (the company who developed Signal) released Sesame, a new protocol dedicated to multi-device secure messaging [23]. Sesame consists in establishing Signal sessions between all devices, as shown in Figure 1. If Alice has n_A devices and Bob n_B, it requires for Alice $(n_A - 1) \cdot n_B$ encryptions for each message she sends, and as many ratchet executions. Adding or removing a device from a user's pool of devices is possible through opening/closing the corresponding pairwise channel. In Sesame as in Facebook Messenger, Alice knows that Bob communicates from several devices. She can even identify which channel - hence which device - sent the message.

Messaging Layer Security. In a related area, Cohn-Gordon *et al.* proposed in [8] a solution for groups based on Diffie-Hellman trees. This solution could be adapted to the multi-device context, by considering each device as a single user. However, secure group messaging tries to tackle a broader and more complicated problem than secure multi-device messaging. We detail below some particularities to multi-device messaging that we take advantage of. More generally, we

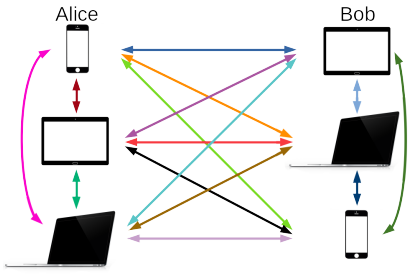


Fig. 1: Sesame multi-device protocol. Each array corresponds to pairwise Signal channel.

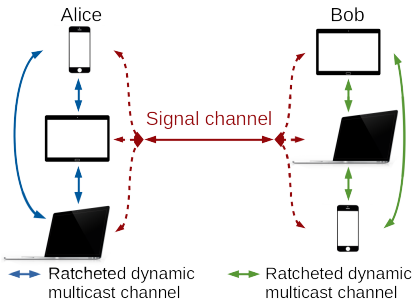


Fig. 2: Our Multi-Device Dynamic Ratcheted Key Exchange protocol. Only one Signal channel is needed between Alice and Bob.

believe that designing a solution for the multi-device case is of prime importance given the evolution of users' practices, and that such a solution, besides being secure, must also be efficient and easy to use in order to be widespread.

Multi-device messaging vs. group messaging. In multi-device messaging, a single user owns and controls the different devices, while in group messaging, multiple users discuss using a single device each. Passive authentication is therefore easier to achieve in the multi-device case: received messages are authenticated as coming from a valid device but the identity of the sending device does not need to be revealed - the owner of the devices knows this information. Moreover, to authenticate a new device to another one of the same user, one can easily assume the devices will be physically close at some point. This means that a QR code can be used to exchange data between them (as it is the case in Sesame). Finally, assuming average usage, we will not take into account concurrent actions, such as revoking one's phone from one's tablet and conversely, at the same time. This also exclude the case when one honest device and a malicious one try to revoke each other at the same time. This could be handled at an application level by requiring a password or some personal data before revoking, what we consider out of the scope of this paper.

1.3 The Signal Protocol

Here we briefly describe Signal and its Double Ratchet algorithm. Signal is a non interactive key exchange that proposes a continuous update of the message key used to encrypt the messages. To achieve this, parties have to store many intermediary keys. There are two kinds of update: the symmetric and the asymmetric ratchet. The symmetric part is a one-way evolution of the message key that ensures *forward secrecy*: past message keys can not be deduced from the current one. The asymmetric ratchet brings new entropy: a Diffie-Hellman computation (DH) is performed with new random values to update the state, and ensures the *healing* property. If some past keys are revealed, the privacy will be recovered

thanks to this new entropy. The security of Signal relies on a trusted distribution server, to avoid interactivity between users. During the registration phase, each user U sends to the server some public credentials: a long term (unchangeable) user key upk_U and some ephemeral initialization keys $ephpk_U$. When Alice wants to open a session with Bob, she asks the server for Bob's credentials. Then she can execute the non interactive key exchange protocol X3DH specified in [22]: Alice and Bob compute a shared root secret, even if they are not on-line to exchange data. From then, Alice and Bob store a common root key (rk_x), a chain key ($ck_{x,y}$), a DH ratchet secret key (their own $rchsk_A$, $rchsk_B$), a DH ratchet public key ($rchpk_B$, respectively $rchpk_A$, corresponding to the other's secret key). Those ratchet keys will be regularly renewed - this is how new entropy is injected in the protocol. From a chain key $ck_{x,y}$ are derived a new chain key $ck_{x,y+1}$ and a message key $mk_{x,y+1}$ - that will be used to encrypt messages - with a key derivation function as: $KDF\_CK(ck_{x,y}) \rightarrow ck_{x,y+1}, mk_{x,y+1}$. This is the symmetric ratchet.
As long as Alice sends messages, she updates her chain and message keys with this symmetric ratchet procedure. Bob does the same on its side to obtain the same message keys so that he can decrypt the messages he receives. Once Bob wants to answer, he updates the root key rk_x with an asymmetric ratchet. He generates a new ratchet key pair: $DHKeyGen(1^n) \rightarrow rchsk'_B, rchpk'_B$. He performs a DH between this new secret key and the ratchet public key of Alice ($rchpk_A$), obtaining a value E: $DH(rchsk'_B, rchpk_A) \rightarrow E$.
Then using a second KDF, he updates his rootkey and his chain key as follow: $KDF\_RK(rk_x, E) \rightarrow rk_{x+1}, ck_{x+1,0}$. The chain key $ck_{x+1,0}$ initiates a new chain of (chain key, message key) pairs. Bob sends his new ratchet public key $rchpk'_B$ as an associated authenticated data with its message m: $AEAD(m_{x+1,1}, m, rchpk'_B)$, so that Alice updates the root key the same way. An asymmetric ratchet is performed each time Alice or Bob sends a first response to the other. The following messages sent by the same sender only require the symmetric ratchet.

1.4 Our contributions

We propose a multi-device protocol based on the classical two users Signal. In our solution, one user does not need to know how many devices the other has. Neither can he find which device his correspondent uses. This is an improvement in terms of privacy, as, for instance, the use of a particular device can leak information about your location. The idea is to open a specific multicast channel between a user's devices to broadcast the one Signal secret essential to perform the protocol: the ratchet secret key ($rchsk_A$ in section 1.3). As illustrated in Figure 2, each time one device of Alice sends a Signal message to Bob, it also sends a specific message to Alice other devices, containing the new Signal ratchet secret key. Thanks to this non interactive synchronization, all Alice devices have the same voice in the Signal conversation: they speak through the same Signal channel to Bob. On the way back, when Bob answers Alice through the unique Signal channel, his message is duplicated by the Server to all of Alice devices. A multicast channel is created for each Signal's session. To keep the security

properties offered by the two-users ratchet, the multicast must guarantee these properties.

We propose as a first step a new primitive: a **Ratcheted dynamic multicast (RDM).** As for a traditional multicast, our RDM establishes a secure channel shared between several participants (in our case devices). It is dynamic since one can add or revoke devices during the execution of the protocol. The novelty is that the keys used to secure this channel are regularly updated, so as to obtain the forward secrecy and healing properties. This is the ratchet feature. The update can be done independently by any party. It is of utmost importance that each device remains independent in its ratcheting process, because in real life, one does not want to wait for all - or even a small part - of its devices to interact together before sending a message. For a similar reason, it is essential that our RDM is decentralized, as we want to avoid having a master device that one cannot afford to lose, have corrupted, or run out of battery. We propose a security model in Section 2.1 for this new primitive, as well as a construction in Section 2.2, that we prove secure. Our construction is based on standard well known primitives: an authenticated asymmetric encryption and a MAC scheme, that we recall in the Appendix A.

In a second step, we instantiate the integration of our multicast with Signal, to obtain **a Multi-Device version of Signal**. Figure 3 represents a high level view of our solution, with Alice sending messages to Bob. Alice sends messages from any devices and Bob receives them on all of its devices. The square box numbers highlight some of our design particularities that we motivate hereafter. We consider the sending device is $d_{A,i}$.

1. When Alice sends a message from any of her devices, this message is identically duplicated by the server and distributed to each of Bob's device. This can be done through mailboxes handled by the Server, who needs to know about Bob's devices (or at least about their numbers). This mailbox system is already offered by the Sesame solution in [23].
2. When the other devices receive a message corresponding to a symmetric Signal step performed by $d_{A,i}$, they have to perform the symmetric ratchet on their own to maintain their chain key up-to-date.
3. When the other devices receive a message corresponding to an asymmetric Signal step performed by $d_{A,i}$, they receive the corresponding ratchet secret key $rchsk_A$. From this key, they can perform the asymmetric ratchet, to derive the needed keys and maintain their state up-to-date.
4. As devices now share the ratchet secret, we need to change this secret when a device is revoked. A revocation hence induces an extra ratchet in the Signal conversation between Alice and Bob. As we do not want a newcomer to be able to read past messages, an additional ratchet also comes up with the joining process. Bob needs to know about this ratchet, otherwise the next message he sends would correspond to old keys, that the revoked device knows. An update message is sent to Bob, to let him know about the ratchet. Bob knows there has been a ratchet, but he can not know if it corresponds to an addition, a revocation, or a security update, and he has no clues about

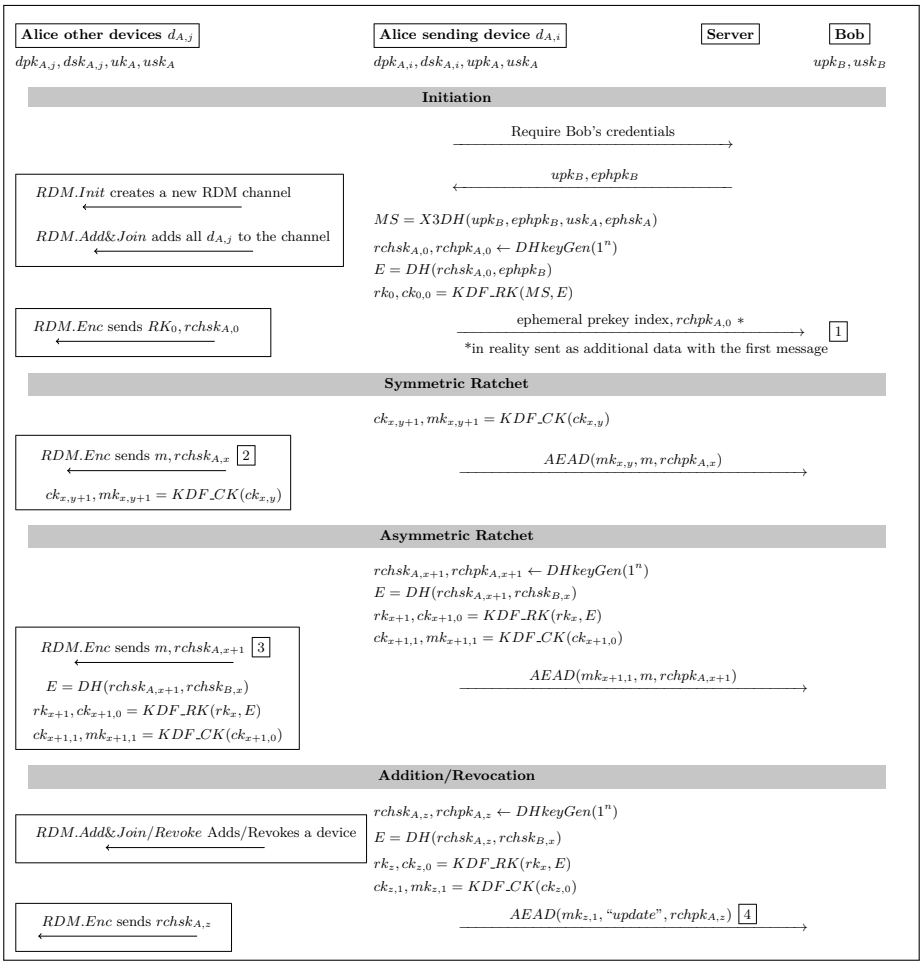


Fig. 3: Multi-Device Signal protocol. Signal procedures $KDF\_RK, KDF\_CK, X3DH, AEAD$ are defined in § 1.3. Boxed messages are sent between Alice devices. The figure without them corresponds to Signal. Boxed numbers 1 to 4 are justified in Section 1.4.

which devices are concerned. We take those extra ratchets into account in our security analysis.

We detail the above description in Section 3. We explain in Section 3.1 how we mix our RDM model with the Signal security model, to obtain a valid security model for our Multi-Device Signal protocol. We introduce some important definitions and we detail how the freshness conditions in Signal's model need to be updated to take into account the multi-device feature, in particular the dynamic aspects. Our model is based on the one issued from the first analysis of the Signal protocol in [9]. However, one could plug our RDM on another RKE security model, with the same adaptations on the freshness conditions, obtaining a flavor

of Multi-Device Ratcheted Key Exchange (MDRKE). We implement our solution over the Signal library libsignal-protocol-java accessible on Signal GitHub account. We give details and results in Appendix D.

How do we deviate from Signal. One of our goal is to upgrade the existing Signal protocol in a transparent way. However, one modification was unavoidable: the introduction of a device key, that every device generates for itself, before registering to the Signal server. This key is used to initiate the RDM channels between devices, and to add a new device. This key also plays a main part during the revocation process. In this precise case, we allow the renewal of the Signal ephemeral keys $(ephpk, ephsk)$ and user keys (upk, usk). In the original Signal, the user key cannot be modified without unregistering then registering again and thus closing all current conversations. In our solution, the server accepts a new user key for Alice if it is authenticated with one of Alice's device key. On Bob's side, this will be exactly as if Alice had registered a new account (as it is now in Signal). The main advantage is for Alice to keep her current conversations when revoking a device. If she had registered again, she also would have to add her devices again. Another deviation from the original Signal is that we make it possible to achieve several ratchets in a row on Alice's side (instead of the ping-pong pattern adopted by the original Double Ratchet). We show that this has no consequence on the security, nor on the possibility to deal with out-of-order messages. However, it implied for us a small patch in the Signal library as explained in Appendix D.

Our choices vs. Signal's Sesame solution. Our solution differs from Sesame or Facebook solutions in that, in our construction, a user is ignorant about his correspondent's devices. A message sent by Bob is only encrypted once for Alice, instead of being encrypted for each device of Alice. This message is also encrypted only once for all of Bob's other devices, instead of once per device. The server will be in charge of broadcasting the message to the appropriate devices. The authentication of a new device is also different. The Sesame protocol offers two options: the first requires all the devices of Alice to share a common IDkey. When a new device is added, it obtains this IDkey. Bob recognizes the new device as a device of Alice since it has the same IDkey. This makes the IDkey a very sensitive data. In [10], authors clearly stipulate that this feature prevented the TextSecure messaging app (the ancestor of Signal) from achieving post-compromise security. In the second option, the devices do not share a common key. When Alice adds a device, Bob should physically authenticate this device to be sure it is honest and belongs to Alice. In our solution, we only require a new device of Alice to be authenticated by another device of hers.

1.5 PFS, revocation and out-of-order messages

Forward secrecy ensures that a leakage of secret keys at some time does not compromise the confidentiality of past exchanges. When confronted to reality however, this ideal is hard to achieve perfectly. The first difficulty is to deal

with out-of-order messages. The Signal protocol handles with those messages by keeping unused keys in memory. However, this option seriously weakens forward secrecy and is not taken into account in [9]. Hence we do not consider out-of-order messages neither. The other fundamental reason why we made this choice is the revocation feature. If Bob still accepts unused old keys to face up with the arrival of delayed messages, a revoked device of Alice can also use these keys to infiltrate maliciously the session. Revocation would not be efficient. The second obstacle to forward secrecy in a multi-device context is that we consider each device shall receive all the messages in the conversation. If a device stays offline for a long-time, it will process all the updates from the moment he went offline until the moment he is back online. All the corresponding keys are still sensible data. Forward secrecy is to be considered only for the messages sent before the "oldest offline device" went offline. This highlights that a multi-device application should consider a process to prevent the devices from being offline for a time too long (automatic revocation for instance). We consider this out of the scope of this work.

2 Ratcheted Dynamic Multicast (RDM)

We introduce a new protocol for multicast communication. The idea behind the ratchet feature is that the protocol is stateful and the state evolves during the execution of the protocol. The goal is to strengthen the security of the channel. In the security model, it means that the adversary can be given more abilities than in a non-ratcheted version. From then, we will consider participants in the RDM as devices.
We start by giving a formal description of a RDM. Each device i maintains two states. The device state, π_i, is valid for all the sessions of the protocol. It registers long-term private key and public key: $\pi_i.sk$, $\pi_i.pk$. The session state π_i^s is valid only for the session s of the protocol. It contains the following information:

- $rand$, the ephemeral information of the state.
- $devices$, the public keys of all devices involved in the session.
- PK, the current session public key for the group $\pi_i^s.devices$.

Protocol description. A RDM is defined by nine algorithms:

- $\mathsf{SetUp}(1^n, i) \to \pi_i$. Generates secret and public keys (sk_i, pk_i) and creates a device state π_i.
- $\mathsf{Init}(\pi_i, s) \to \pi_i^s$. Initiates a new session s of the protocol. Generates a session state π_i^s for this session.
- $\mathsf{Enc}(m, \pi_i^s) \to C_{enc}, \pi_i^s$. On input a message m and a session state π_i^s, returns a ciphertext C_{enc} and the updated state π_i^s.
- $\mathsf{Dec}(C_{enc}, \pi_j^r) \to m, \pi_j^r$. On input a ciphertext C_{enc} and a session state π_j^r, returns a message m and the updated state π_j^r.
- $\mathsf{Add\&Join}(\{pk_{j_\ell}\}_{\ell\in[1,z]}, \pi_i^s) \to C_{add}, C_{join}, \pi_i^s$. On input a set of public keys $\{pk_{j_\ell}\}_{\ell\in[1,z]}$ and a session state π_i^s (of the device that adds), returns infor-

mation C_{join} for the new devices, C_{add} for the already enrolled devices and the updated state π_i^s.

- $\mathsf{DecJoin}(C_{join}, \pi_j, r) \rightarrow \pi_j^r$. On input a ciphertext C_{join}, a device state π_j, and a session identifier r, returns a new session state π_j^r.
- $\mathsf{DecAdd}(C_{add}, \pi_k^o) \rightarrow \pi_k^o$. On input a ciphertext C_{add} and a session state π_k^o, returns the updated session state π_k^o.
- $\mathsf{Revoke}(pk, \pi_i^s) \rightarrow C_{rev}, \pi_i^s$. On input a public key pk and a session state π_i^s, returns a ciphertext C_{rev} and the updated state π_i^s.
- $\mathsf{DecRevoke}(C_{rev}, \pi_k^o) \rightarrow \pi_k^o$. On input a ciphertext C_{rev} and a session state π_k^o, returns the updated state π_k^o.

2.1 RDM security model

In this section we give an intuition of the security expected from a RDM primitive. A more formal and detailed description is given in Appendix B.1. We expect a RDM to provide indistinguishability under chosen-ciphertext attacks, as defined in Appendix A, as well as forward secrecy and healing. We define our security model by starting from an ideal case where the adversary has full powers, and then excluding the attacks that we consider as unavoidable. The adversary controls the execution of s sessions of the protocol and he can obtain all the secret information he wishes. At some point, he can query an indistinguishability challenge on one session. He then has to distinguish between a real ciphertext honestly produced by this session or some randomness. We exclude the cases where he could trivially win, or the attacks that we consider as unavoidable by defining some freshness conditions. We introduce three necessary definitions. Firstly, we formalize the notion of step of a protocol. A session can live for a long life time (weeks, months) and some secret data may evolve during this period. Steps are meant to follow this evolution. Secondly, we define matching sessions, based on [5]. This is necessary because we consider a multi-session context. Our definition helps us to define the correctness of our protocol: if two participants are involved in a same execution and have reached corresponding steps - *i.e.* if they are matching, they should be able to communicate together. Moreover, as matching sessions may share common secret data, the adversary's powers are also defined "matching-session" wise. Finally, because of the dynamic feature, several sessions that correspond to a same execution of the RDM may not be present at the same time, and so do not match. They are however related. We introduce the notion of chained sessions, to take this relationship into account.

Let $\{\mathtt{d}_1, \ldots, \mathtt{d}_{n_d}\}$ be the devices participating in the protocol. Each device $\mathtt{d}_i$ is modeled by an oracle π_i and each session s executed by a device $\mathtt{d}_i$ (session (i, s)) is modeled by an oracle π_i^s. Oracles maintain states as defined in Section 2. In the following, the oracles and their state will be considered as equal.

Protocol steps. Data registered in a device state for a session will change during the execution of the protocol. To model this phenomenon, we consider

steps of the protocol. Each Enc, Add&Join, Revoke or corresponding decryption algorithm advances the protocol to a new step. Steps are formalized through a counter t, set to 0 at initiation and incremented by oracle queries. This counter is included in the oracle session state with $\pi_i^s.\texttt{step}$. It is not necessary in an implementation but needed by the model. (In a general way, we use the `typewriter` typo for model specific elements). Going from one step to another indicates that the algorithm has processed without error. Intuitively, steps will embody the healing and forward secrecy properties: some restrictions can be needed at some step t and released at step $t+1$ (or reversely), meaning that the confidence is back (is still there for past steps). We refer with (i, s, t) to the session (i, s) at step t. We note $\pi_i^s[t]$ when we refer to oracle's state π_i^s as it was at step t. We note $\pi_i^s[t].\texttt{X}$ the access to item `X` at step t.

Matching sessions. We now define the notion of matching sessions. We denote $\pi_i^s.\texttt{sid}$ the transcript of the protocol executed in session (i, s), that is, the concatenation of all messages C_i sent or received by π_i^s. We write $\pi_i^s[t_s].\texttt{sid} = C_i[0]\|C_i[1]\|\ldots\|C_i[t_s]$. As no message is sent or received for the initiation, the first component of a `sid` for a session running the Init procedure is set to INIT. We refer to a session created by an Init algorithm as an initial session. As all devices are playing similar roles, we do not consider roles in our definition of the matching sessions. Since devices can join and leave during the protocol, we define a matching that is step-wise.

Definition 1. *(Matching sessions at some step.) One says (i, s, t_s) and (j, r, t_r), $t_s \geq t_r$ ((i, s) joined first), are matching if $\exists$ $\texttt{sid}'$ substring of $\pi_i^s[t_s].\texttt{sid}$ such that $\pi_i^s[t_s].\texttt{sid} \doteq \texttt{sid}'\|\pi_j^r[t_r].\texttt{sid}$ ($\texttt{sid}'$ eventually empty). The symbol $\doteq$ stands for the following definition.*
$\pi_i^s[t_s].\texttt{sid} \doteq \texttt{sid}'\|\pi_j^r[t_r].\texttt{sid}$ if, $\forall t \in [0; t_r]$:

- *either $C_i[t_s - t_r + t] = C_j[t]$,*
- *either $C_k[t_s - t_r + t] = (C_{add}, C_{join})$ or C_{add} and $C_\ell[t] = C_{add}$, $k, \ell \in \{i, j\}, k \neq \ell$,*
- *or $t = 0$ and $C_i[t_s - t_r] = (C_{add}, C_{join})$ or C_{add} and $C_j[0] = C_{join}$ with (C_{add}, C_{join}) having been produced by the same Add&Join call.*

As devices can join the protocol at any moment, we define a way to link sessions that corresponds to a same execution but were not present at the same time. This composes chains of sessions, as illustrated in Figure 4.

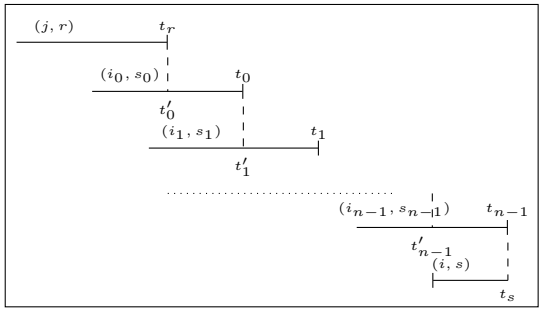


Fig. 4: A chain of sessions between (i, s, t_s) and (j, r, t_r).

Definition 2. *(Chained sessions.) A session (j, r, t_r) is chained with (i, s, t_s) if t_r is maximal and there exists n sessions (i_α, s_α), and n couples (t'_α, t_α), $t'_\alpha \leq t_\alpha$, $\alpha \in [0, n-1]$ such that:*

- *(j, r, t_r) and (i_0, s_0, t'_0) are matching,*
- *$\forall \alpha \in [0, n-2]$, $(i_\alpha, s_\alpha, t_\alpha)$ and $(i_{\alpha+1}, s_{\alpha+1}, t'_{\alpha+1})$ are matching,*
- *$(i_{n-1}, s_{n-1}, t_{n-1})$ and (i, s, t_s) are matching.*

$\{(i_\alpha, s_\alpha, t_\alpha)\}_{\alpha \in [0,n-1]}$ is called a chain of session between (i, s, t_s) and (j, r, t_r).

Definition 3. *(Correctness.) Suppose a passive adversary that sees communications and may only disturb their delivery. A RDM is said to be correct if, for all matching sessions (i, s, t_s) and (j, r, t_r), for all messages m,*

$$\mathsf{Dec}(\mathsf{Enc}(m, \pi_i^s[t_s]), \pi_j^r[t_r])) = m.$$

RDM indistinguishability. As in the original IND-CCA experiment, the adversary $\mathcal{A}$ can query for one Challenge of indistinguishability. He is given access to oracles that enables him to perfom the whole protocol: OInit, OEnc, ODec, OAdd&Join, ODecAdd, ODecJoin, ORevoke, ODecRevoke. The oracle OInit defines the experiment in a multi-session context. Finally, the adversary can Corrupt a device to obtain its long term secret key, and he can choose to Reveal the state secrets of any device.

Freshness. The natural restrictions defined here are meant to exclude unavoidable attacks or cases where the adversary could win trivially. These restrictions are often valid for a session and all the corresponding chained sessions (not only matching sessions). This expresses the fact that a device has to participate regularly to the protocol to update its state. This is inherent to the ratchet process: the participants have to be actively involved for the ratchet to be operational. One of the direct consequence of this remark, is that we consider that the session specific data are equal to the long term data until a participant is active. This gives the adversary two ways of accessing long-term data, Corrupt and Reveal. We carefully take into account these two paths for the adversary to trivially win the Game.

1. $\mathcal{A}$ shall not Reveal state secrets just before the challenge.
2. $\mathcal{A}$ shall not Reveal a device concerned with the challenge. The sequels of a Reveal are canceled by a OEnc. This ensures the security is back for d_i's secret after d_i performs an encryption. A Reveal on a device only threatens the steps from the last encryption and until the next. This corresponds to the healing property. It also means that forward secrecy depends on devices regularly sending messages, as discussed in paragraph 1.5.
3. $\mathcal{A}$ shall not Corrupt a non active device before or after the challenge. A device comes active as soon as it performs an encryption (it enters the ratcheting process).
4. $\mathcal{A}$ shall not Reveal random secrets and use them to maliciously send an encrypted message with its own new random, revoke someone, or join a non authorized corrupted device. There is nothing we can do against this kind of impersonation attack, and the two user ratchet is also vulnerable to it.

5. We prevent $\mathcal{A}$ from joining a device in a non existing session or after an exposure. This models a physical authentication procedure between the device that adds and the new device.

Definition 4. *(Secure Ratcheted Dynamic Multicast) A RDM running with n_d devices is a secure Ratcheted Dynamic Multicast if it is correct and for all adversaries $\mathcal{A}$, running in polynomial time, making at most q queries to the oracles, there exists a negligible function $\mathsf{negl}(n)$ such that:*

$$\mathsf{Adv}^{\mathsf{RDM-IND}}_{\mathcal{A},RDM,n_d,q}(n) = \left|\Pr\left[\mathsf{Exp}^{\mathsf{RDM-IND}}_{\mathcal{A},RDM,n_d,q}\right] - \frac{1}{2}\right| \leq \mathsf{negl}(n)\,.$$

We denote $\epsilon_{\mathsf{RDM-IND}}$ this advantage.

2.2 RDM construction

We give a high-level view of our construction in Figure 5. The detailed pseudocode description is given in Appendix B.2. The main idea is that the keys used to encrypt the multicast messages are updated regularly. We base our solution on parallel asymmetric encryption, as studied in [2] or [3]. The authenticated asymmetric encryption scheme is used as a multicast in an obvious manner: PK is the set of public keys of all devices d_i concerned with the encryption. $\mathsf{EncAsym}(m, PK)$ stands for $\{\mathsf{EncAsym}(m, pk)\}_{\forall pk \in PK}$. We consider that the number of devices remains reasonable: around ten for each user does not seem so restrictive. This design allows us to choose among well-known and proven secure primitives, as detailed in Appendix A. Decentralized broadcast solutions, as studied in [24], either do not offer dynamism properties or do not enable regular key resetting and offer fewer implementation evaluations.

An asymmetric ratchet. The Diffie-Hellman ratchet implemented in Signal is only possible for two users. With more than two parties, multiparty computation could be thought of as an option, but we do not want to wait for all, or even a minimum number of devices to be present before sending a message: each device has to be autonomous in its ratcheting process. Our ratchet consists in generating new ephemeral asymmetric keys epk, esk for the device which sends a message. The multicast public key is updated with the new ephemeral public key epk. Here, we take advantage of the multi-device context. As all the devices belong to a single person, we consider that no honest device will try to exclude another device maliciously (by mis-updating the multicast public key). When any device updates its ephemeral key pair, the others only receive the updated common public key. They do not need to know about who updated it. However, when a device wants to revoke another device, it has to know which ephemeral public key to erase from PK. We deal with this by considering there exists a correspondence between the list *devices* of long term keys recorded in each device state and the list PK of ephemeral public keys. Requirements in the Add&Join algorithm prevent a device from being present in the group several times. In such a case, revoking this device once would not be enough to be sure it is definitively out of the protocol.

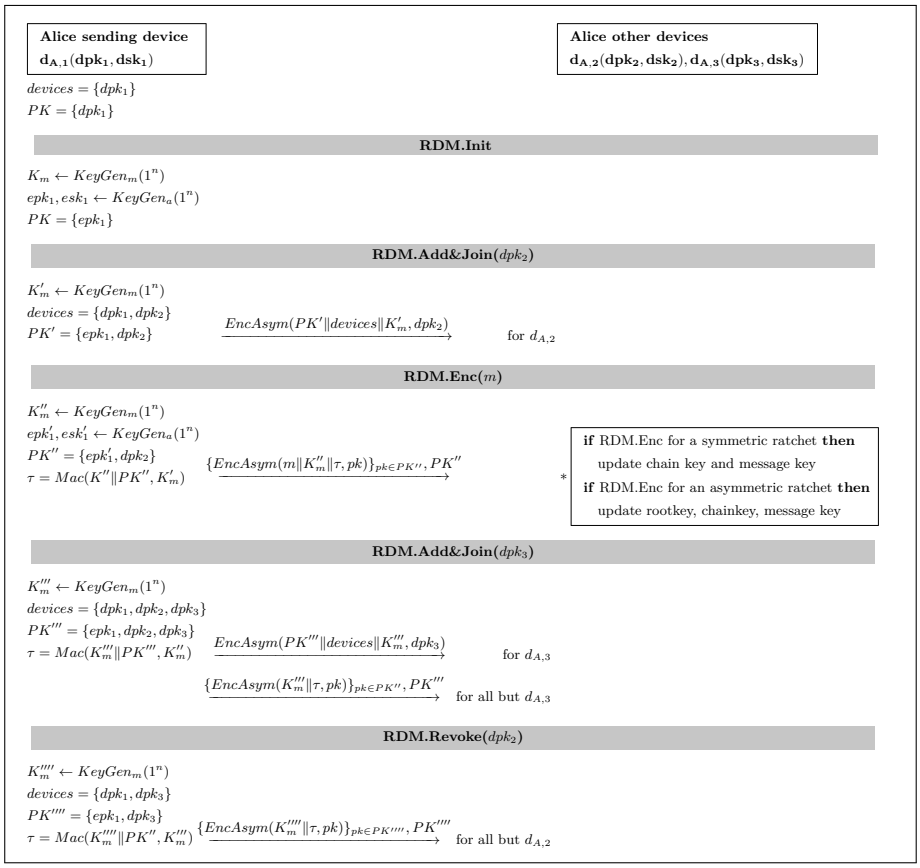


Fig. 5: Our RDM protocol. The sending device can change for each procedure. $KeyGen_{m/a}$ are Key Generation algorithm for the MAC/the asymmetric encryption $EncAsym$ schemes. The instructions * detail the integration of our RDM with Signal.

Passive authentication. Another important point is that the messages have to be identified as coming from an honest device, but again, its identity does not matter. Our solution provides passive authentication thanks to a MAC key K_m shared between the devices. A new MAC key is generated with every action: sending a message, joining, or revoking a device. Otherwise, an adversary who could access the MAC key at some step could impersonate any device at any step further. This new MAC key is authenticated under the previous one, creating an authentication chain. This solution is less expensive than generating new signature key pairs regularly.

Efficiency-wise, we generate two new keys for each encryption and only one for additions and revocations. Maintaining several Signal channels requires a number of key generations that grows linearly with the number of devices.

Ephemeral data esk, epk and K_m constitute the randomness $\pi_i^s.rand$ of the model. For readability reasons, we keep them separate in the construction and refer to them with $\pi_i^s.esk$, $\pi_i^s.epk$, and $\pi_i^s.PK$. To be able to initialize a session, a device must have processed a SetUp to generate its global state π_i.
The following theorem enunciates the security of our construction relatively to the RDM security model described in Section 2.1. A proof is given in Appendix B.3.

Theorem 1. *If* ENC *is an* IND-CCA *secure asymmetric encryption scheme, and* MAC *is secure under multi-instance strong unforgeability, the above construction is a secure ratcheted dynamic multicast for* n_d *devices, such that, for any PPT adversary making at most* q *queries to the oracles:*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{RDM-IND}}_{RDM,n_d,q,\mathcal{A}}(n) &= \left|\Pr\left[Exp^{\mathsf{RDM-IND}}_{RDM,n_d,q,\mathcal{A}}(n)\right] - \frac{1}{2}\right| \\ &\leq q \cdot \epsilon_{\mathsf{SUF}} + (q+1) \cdot n_d \cdot \epsilon_{\text{IND-CCA}}.\end{aligned}$$

In practice, one would use hybrid encryption instead of a single asymmetric encryption scheme in this construction. It means that the asymmetric encryption is used to transmit a common symmetric key to all devices and that data are then encrypted with this key. This would modify the security argument only by a negligible term due to the symmetric encryption. We decide to present our construction with the asymmetric part so as not to add extra lines in an already complex construction.

3 Multi-Device Signal

We detail our Multi-Device Signal solution, depicted in Figure 3. It is built from the Signal protocol and our RDM protocol described in Section 2.2. The pseudocode description is given in Appendix C.2. The RDM enables us to share the DH secrets, for all devices to perform the operation. It is also used to share the message's "body". Every Signal sending is doubled with a RDM sending. This way, any device can follow the conversation, can speak for itself, and can directly receive messages sent by Bob. For each Signal session, a specific RDM channel is opened between Alice's devices. Addition and revocation induce extra ratchets, for the joining/revoked device not to access previous/future conversations. We introduce a new procedure, `ExtraRatchet`. When a device receives a ratchet secret through the RDM channel, he has to update its Signal state accordingly. This is done in an Update procedure. Those new procedures are given in Appendix C.2. We recall that our user key is equivalent to the traditional Signal longterm key. It is shared among devices, as well as Signal's ephemeral keys, through the joining process. This is necessary for each device to be able to send or receive an initiation message. The device key never changes.
To execute the protocol, a device has to record some information. To do so, a device state $\pi_{u,i}$ aggregates all non session-specific elements, and a session state $\pi^s_{u,i}$ records all the session-specific ones. A device state $\pi_{u,i}$ is composed of:

- dID, the device identifier.
- uID, the user identifier.
- dsk, the device's secret key.
- $sprekeys$, the user's secret keys registered in Signal. This comprises the user key and the ephemeral keying material needed for initialization (detailed in MedTerm and One-Time prekeys in Signal).
- $Devices$, the public keys of the owner other devices.
- $Sessions$, a list of all sessions the device i is engaged in.

A session state $\pi^s_{u,i}$ gathers a Signal part:
- $role$, the role of the user u: initiator or receiver,
- $peer$, the intended peer user of this session,
- $rand$, the current ratchet secret,
- $rand_{peer}$, the current public ratchet value of the intended peer,
- $sessionkey$, the current messaging session key,
- $state$, all other secret information needed,

and a RDM part (the *device* item is already in the global state):
- $devrand$, the RDM randomness,
- PK the common public key.

A session state $\pi^s_{u,i}$ has access to general information of the device state $\pi_{u,i}$. Conversely, a device state $\pi_{u,i}$ gives implicit access to every session state $\pi^s_{u,i}$. We describe Signal as a multi-stage key exchange as in [9] (detailed in Appendix A), except that we split the algorithm Run defined in into Sig.Send and Sig.Receive. We detail Sig.Register to take into account the device key in addition to the user key and the ephemeral keys usually used by Signal.

A device shall perform a `RDM.SetUp` to obtain its devices keys before he registers. We gather `Sig.KeyGen` and Sig.MedTermKeyGen in a UserKeyGen procedure that returns a set of prekeys $prekeys$. Those keys are registered to the server with the device key. We obtain a Multi-Device Instant Messaging protocol, as formally defined in Appendix C.1.

3.1 Security model

We build a security model for a MDIM by mixing our RDM model with Signal security model described in [9]. A formal description can be found in the full version. The joint between the two models is highly related to the way the two primitives overlap in practice. As said before, the present model is there to ensure that we keep the Signal security when adding our RDM. Let $\{\mathcal{P}_1, \ldots, \mathcal{P}_{n_U}\}$ be the set of users in the protocol and $\{\mathtt{d}_{u,1}, \ldots, \mathtt{d}_{u,n_d}\}$ the set of devices of the user $\mathcal{P}_u$. Each device $\mathtt{d}_{u,i}$ is modeled by an oracle $\pi_{u,i}$ and each session s executed by a device $\mathtt{d}_{u,i}$ is modeled by an oracle $\pi^s_{u,i}$. Device oracles maintain device states and session oracles maintain session states as defined in Section 3. In the following, device or session oracles and their state will be considered as equal. We identify sessions that were present during the initial step of the protocol as initial sessions.

Protocol steps As in the RDM model, we consider the steps of the protocol. Each Send, Add, Revoke, or corresponding $\mathsf{Receive}$ or Dec algorithm brings the session to a new step and corresponding oracles will increment the step counter. If no change occurs, values are transmitted from one step to another (e.g. if state does not change at step t_s, then $\pi^s_{u,i}[t_s+1].state = \pi^s_{u,i}[t_s].state$). We refer to session s run by $\pi_{u,i}$ at step t_s as (u, i, s, t_s).

Matching sessions. In order to define the matching between sessions run by different users, we first need to consider the relationship between devices belonging to a single user. We introduce the notion of partnered sessions. For this definition, we separate the conversation between devices (written in a session identifier $\mathtt{sid_1}$) from the Signal messages (gathered in $\mathtt{sid_2}$). Partnered sessions correspond to devices of a single user that are online at the same moment.

Definition 5. *(Partnered sessions at some step.) Two sessions* (u, i, s, t_s) *and* (u, j, r, t_r) *are partnered if:*

- $\pi^s_{u,i}.role = \pi^r_{u,j}.role$,
- $\pi^s_{u,i}.peer = \pi^r_{u,j}.peer$,
- $\pi^s_{u,i}.uID = \pi^r_{u,j}.uID$,
- (i, s, t_s) *and* (j, r, t_r) *are matching in the sense of RDM (relatively to* $\mathtt{sid_1}$*).*

We define chains of partnered sessions, as for the RDM, to connect sessions that are active on different devices that were present at different steps. Those structures are necessary to link any session to the initiation step when the authentication is performed.

Definition 6. *(Chained sessions.) A session* (u, j, r, t_r) *is chained with* (u, i, s, t_s) *if* t_r *is maximal and there exists* n *sessions* (u, i_α, s_α)*, and* n *couples* (t'_α, t_α)*,* $t'_\alpha \leq t_\alpha$*,* $\alpha \in [0, n-1]$ *such that:*

- (u, j, r, t_r) *and* (u, i_0, s_0, t'_0) *are partnered,*
- $\forall \alpha \in [0, n-2]$*,* $(u, i_\alpha, s_\alpha, t_\alpha)$ *and* $(u, i_{\alpha+1}, s_{\alpha+1}, t'_{\alpha+1})$ *are partnered,*
- $(u, i_{n-1}, s_{n-1}, t_{n-1})$ *and* (i, s, t_s) *are partnered.*

$\{(u, i_\alpha, s_\alpha, t_\alpha)\}_{\alpha\in[0,n-1]}$ *is called a chain of sessions between* (u, j, r, t_r) *and* (u, i, s, t_s).

We consider matching sessions relatively to Signal conversation. We define $\pi^s_{u,i}.\mathtt{sid_2}$ as the concatenation of ciphertexts c_{out} received by $\mathsf{ReceiveOut}(\pi^{\mathsf{s}}_{\mathsf{u,i}}, \cdot)$ or produced either by $\pi^s_{u,i}$ or by any partnered session. The matching is defined between two sessions (u, i, s) and (v, ℓ, o). One session (u, i, s) can match several other sessions (v, ℓ_z, o_z). Our definition is recursive: a matching is well-defined if one can trace the conversations from the very beginning on u and v's side. This is done by calling chains of sessions, and each chain element should match an element in the other chain.

Definition 7. *(Matching sessions at some step.) Two sessions* (u, i, s, t_s) *and* (v, ℓ, p, t_p)*,* $t_s \geq t_p$ *are matching if:*

- $\pi^s_{u,i}.role \neq \pi^p_{v,\ell}.role$,

- $\pi^s_{u,i}.peer = \pi^p_{v,\ell}.user$ *and* $\pi^s_{u,i}.user = \pi^p_{v,\ell}.peer$,
- (u, i, s, t_s) *and* (v, ℓ, p, t_p) *are chained with respective initial sessions* (u, i_0, s_0, t_{s_0}), (v, l_0, p_0, t_{p_0}), *through respective chains* $\{(u, i_\alpha, s_\alpha, t_\alpha)\}_{\alpha\in[1,n]}$, $\{(v, \ell_\beta, p_\beta, t_\beta)\}_{\beta\in[1,m]}$,
- $\exists$ $\mathtt{sid}$ *subset of* $\pi^s_{u,i}[t_s].\mathtt{sid}_2$ *such that* $\pi^s_{u,i}[t_s].\mathtt{sid}_2 = \mathtt{sid}\|\pi^p_{v,\ell}[t_p].\mathtt{sid}_2$,
- *if* (u, i, s, t_s) *is an initial session, then* $\forall\ \beta \in [0, m]$, $\exists\ \tilde{t}_\beta$ *and* $\mathtt{sid}_\beta$ *such that* $\pi^s_{u,i}[\tilde{t}_\beta].\mathtt{sid}_2 = \mathtt{sid}_\beta\|\pi^{p_\beta}_{v,\ell_\beta}[t_\beta].\mathtt{sid}_2$,
- *else* (u, i, s, t_s) *and* (v, ℓ_m, p_m, t_m) *are matching.*

We can estimate $\tilde{t}_\beta = \tilde{t}_{\beta-1} + t_\beta - t'_\beta$, t'_β *as in Definition 6.*

Definition 8. *(Correctness of MDIM.) A Multi-Device Instant Messaging is said to be correct if:*

- *for all users* u, *all devices* i, j, *all session identifiers* s, r, *and all* t_s, t_r *such that* (u, i, s, t_s) *and* (u, j, r, t_r) *are partnered,* $\pi^s_{u,i}[t_s].rand = \pi^r_{u,j}[t_r].rand$ *and* $\pi^s_{u,i}[t_s].sessionkey = \pi^r_{u,j}[t_r].sessionkey$.
- *For all users* u, v, *all devices* i, ℓ, *all session identifiers* s, p, *and all* t_s, t_p *such that* (u, i, s, t_s) *and* (v, ℓ, p, t_p) *are matching,* $\pi^s_{u,i}[t_s].sessionkey = \pi^p_{v,\ell}[t_p].sessionkey$.

MDIM indistinguishability. We will consider an adversary $\mathcal{A}$ that has access to a pool of registered devices. He controls communications through oracles corresponding to the protocol algorithms. $\mathcal{A}$ has access to the oracles corresponding to RDM corruptions (`CorruptDevice`, `RevealDevRand`) and to those corresponding to the Signal security model we consider (`RevealSessionKey`, `CorruptUser`, `CorruptOpt`, `RevealState`, `RevealRandom`) .

Freshness. Freshness conditions are obtained by considering the RDM freshness conditions and upgrading the original Signal freshness in the following way: each time an element of a session was concerned in the original freshness, the same element has now to be considered for this session and all the partnered and chained sessions. More precisely, we try to stipulate clearly all the ways that an element can leak to the adversary: directly from the targeted device or a partnering or matching one, or through the communication between devices. For the latter, data of a session (u, i, s) at step t can leak if a device (u, i)'s randomness (or device keys if it has not been active in the ratcheting process yet) is compromised, or if the device randomness of a partnered session is compromised or if there exists a session chained with (u, i, s) whose device randomness is compromised (if it is chained, it means it will not perform any action until it matches (u, i, s, t) - if not revoked - it just has not received all of its messages).

Initial freshness. As in the original model for Signal, we treat separately the initiation phase, and it has to be treated very carefully. As written in the definition, Signal prekeys (including user key) are renewed with every revocation. On the one hand, this restricts the consequences of a user corruption to the

period between two revocations. On the other hand, it means that the prekeys have to be shared with the other devices. Hence the security of a session initiation is now related to the security of the communication between the devices since the last renewal of prekeys, that is, since the last revocation.

Definition 9. *(Secure Multi-Device Instant Messaging.) A MDIM executed with n_p users, each having n_d devices is said to be secure in the above model if it is correct and for all adversary $\mathcal{A}$ running in polynomial time, the following advantage is negligible:*

$$\mathsf{Adv}^{\mathsf{MDIM-IND}}_{\mathcal{A},MDIM,n_p,n_d}(n) = \left|\Pr\left[\mathsf{Exp}^{\mathsf{MDIM-IND}}_{\mathcal{A},MDIM,n_p,n_d}(n)\right] - \frac{1}{2}\right|.$$

Theorem 2. *Let Signal be a secure multi-stage key-exchange protocol with advantage ϵ_{sig} and RDM a $\mathsf{RDM-IND}$ secure ratcheted dynamic multicast with advantage $\epsilon_{\mathsf{RDM-IND}}$, the above construction is a secure MDIM such that, for any PPT adversary running n_s sessions from n_d devices of n_p users, making at most q queries to the oracles:*

$$\mathsf{Adv}^{\mathsf{MDIM-IND}}_{\mathcal{A}}(n) \leq n_p^2 \cdot (2 \cdot \epsilon_{\mathsf{RDM-IND}} + \epsilon_{sig}).$$

We give a proof of Theorem 2 in Appendix C.3.

4 Related Work

The first ideas for ratcheting appeared in protocol such as Off-the-Record or TextSecure (the predecessor of Signal) and were studied respectively in [7] and [14]. In [17], Green and Miers studied the interest of puncturable encryption instead of using ratchet to achieve forward security in messaging. The first formal analysis of the Signal protocol has been given by Cohn-Gordon *et al.* in [9]. The same year, Kobeissi *et al.* proposed in [20] a formal verification of a variant of the Signal protocol with ProVerif and CryptoVerif. Later, Bellare *et al.* formalized in [6] the idea of ratcheting, and proposed a security model, as well as constructions, for a single unilateral ratchet key exchange, and the corresponding communication channel. The authors target the already identified security offered by the ratchet: forward secrecy and healing. They clearly stipulate that an active attack after exposure can lead to a continued violation of integrity. At Crypto'18, [26] and [18] extended Bellare *et al.*'s work to define and evaluate the optimal security of bilateral ratcheted key exchanges (messaging channels respectively). Their models define the best security one can expect from such protocols. The drawback is that the associated constructions require less efficient primitives like hierarchical identity-based encryption (HIBE, [16]). One remarkable point of the first paper is that the model allows for concurrent actions. However, in both cases messages still need to be received in the right order.

Recently, Durak and Vaudenay in [12] proposed a solution using standard public key cryptography. They assume the order of transmitted messages is preserved and their model does not allow for exposure of randomness. They also introduced the Recover security to prevent from being safe again after a trivial impersonation. They claim that this kind of recovery not only is not reached by RKE, but also undesirable. Another recent work concerning the two-user ratcheted key exchange setting is the paper by Jost *et al.*, [19]. They introduce a security model that considers suboptimal security. This includes forward secrecy and healing, but they also consider post-impersonation security, under some condition. They claim their model is a bit less permissive than the ones of [26] and [18] but the constructions they propose only require standard public-key cryptography. In [1] Alwen *et al.* proposed their own definition for secure messaging, introducing a continuous key agreement primitive to cover the ratcheting process. They consider a new security notion: the immediate decryption. This property requires that any incoming message can be decrypted, even if some previous messages were lost or arrive out-of-order. According to the authors, Signal achieves this property, but all recent RKE propositions do not. In fact, we discussed this out-of-order problematic in our introduction. Finally, the recent papers [6], [26], [18], [19], [12] and [1] concern the ratcheting process and do not analyze the initial non-interactive key exchange.

5 Conclusion

In this paper, we provide a solution to address the multi-device issue. For the first time, this work introduces some questions that we think can provide great motivation for future work. Studying how recently proposed RKE or messaging schemes as [26], [19], [18] or [12] could be adapted to the multi-device context with our multicast solution, seems a legitimate follow-up. In addition, the out-of-order message, revocation, and PFS problematics, formally introduced in [1], is definitely worth considering.

References

1. Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the signal protocol. In: Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I. pp. 129–158 (2019)
2. Baudron, O., Pointcheval, D., Stern, J.: Extended notions of security for multicast public key cryptosystems. In: Montanari, U., Rolim, J.D.P., Welzl, E. (eds.) ICALP 2000: 27th International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 1853, pp. 499–511. Springer, Heidelberg (Jul 2000)
3. Bellare, M., Boldyreva, A., Micali, S.: Public-key encryption in a multi-user setting: Security proofs and improvements. In: Preneel, B. (ed.) Advances in Cryptology – EUROCRYPT 2000. Lecture Notes in Computer Science, vol. 1807, pp. 259–274. Springer, Heidelberg (May 2000)

4. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. pp. 1–15. Springer-Verlag (1996)
5. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: Stinson, D.R. (ed.) Advances in Cryptology – CRYPTO'93. Lecture Notes in Computer Science, vol. 773, pp. 232–249. Springer, Heidelberg (Aug 1994)
6. Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: The security of messaging. In: Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III. pp. 619–650 (2017)
7. Borisov, N., Goldberg, I., Brewer, E.A.: Off-the-record communication, or, why not to use PGP. In: Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004. pp. 77–84 (2004)
8. Cohn-Gordon, K., Cremers, C., Garratt, L., Millican, J., Milner, K.: On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. CCS '18 (2018)
9. Cohn-Gordon, K., Cremers, C.J.F., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017. pp. 451–466 (2017)
10. Cohn-Gordon, K., Cremers, C.J.F., Garratt, L.: On post-compromise security. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016. pp. 164–178 (2016)
11. Cramer, R., Shoup, V.: A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In: Krawczyk, H. (ed.) Advances in Cryptology – CRYPTO'98. Lecture Notes in Computer Science, vol. 1462, pp. 13–25. Springer, Heidelberg (Aug 1998)
12. Durak, F.B., Vaudenay, S.: Bidirectional asynchronous ratcheted key agreement without key-update primitives. Cryptology ePrint Archive, Report 2018/889 (2018)
13. Facebook: Messenger secret conversation, technical whitepaper, version 2.0 (May 2017)
14. Frosch, T., Mainka, C., Bader, C., Bergsma, F., Schwenk, J., Holz, T.: How secure is TextSecure? Cryptology ePrint Archive, Report 2014/904 (2014)
15. Fujisaki, E., Okamoto, T., Pointcheval, D., Stern, J.: RSA-OAEP is secure under the RSA assumption. Journal of Cryptology **17**(2), 81–104 (Mar 2004)
16. Gentry, C., Silverberg, A.: Hierarchical ID-based cryptography. In: Zheng, Y. (ed.) Advances in Cryptology – ASIACRYPT 2002. Lecture Notes in Computer Science, vol. 2501, pp. 548–566. Springer, Heidelberg (Dec 2002)
17. Green, M.D., Miers, I.: Forward secure asynchronous messaging from puncturable encryption. In: 2015 IEEE Symposium on Security and Privacy. pp. 305–320. IEEE Computer Society Press (May 2015)
18. Jaeger, J., Stepanovs, I.: Optimal channel security against fine-grained state compromise: The safety of messaging. In: Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I. pp. 33–62 (2018)
19. Jost, D., Maurer, U., Mularczyk, M.: Efficient ratcheting: Almost-optimal guarantees for secure messaging. IACR Cryptology ePrint Archive **2018**, 954 (2018)
20. Kobeissi, N., Bhargavan, K., Blanchet, B.: Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. pp. 435–450 (April 2017)

21. Marlinspike, M., Perrin, T.: The double ratchet algorithm. Signal's web site (2016)
22. Marlinspike, M., Perrin, T.: The x3dh key agreement protocol. Signal's web site (2016)
23. Marlinspike, M., Perrin, T.: The sesame algorithm: Session management for asynchronous message encryption. Signal's web site (2017)
24. Phan, D.H., Pointcheval, D., Strefler, M.: Decentralized dynamic broadcast encryption. In: Visconti, I., Prisco, R.D. (eds.) SCN 12: 8th International Conference on Security in Communication Networks. Lecture Notes in Computer Science, vol. 7485, pp. 166–183. Springer, Heidelberg (Sep 2012)
25. Poettering, B., Rösler, P.: Asynchronous ratcheted key exchange. Cryptology ePrint Archive, Report 2018/296 (2018)
26. Poettering, B., Rösler, P.: Towards bidirectional ratcheted key exchange. In: Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I. pp. 3–32 (2018)
27. Shoup, V.: A proposal for an ISO standard for public key encryption. Cryptology ePrint Archive, Report 2001/112 (2001)

A Preliminaries and definitions

Notations We note $\mathcal{M}$ the message space, $\mathcal{SK}_a$ and $\mathcal{PK}_a$, asymmetric secret key and public key spaces, $\mathcal{C}_a$ an asymmetric ciphertext space, $\mathcal{K}_m$ a MAC key space and $\mathcal{T}$ a tag space.

Asymmetric encryption An authenticated asymmetric encryption scheme ENC is composed of the three algorithms: $\mathsf{KeyGen_a} : 1^n \to \mathcal{SK}_a \times \mathcal{PK}_a$, $\mathsf{EncAsym} : \mathcal{M} \times \mathcal{PK}_a \to \mathcal{C}_a$, and $\mathsf{DecAsym} : \mathcal{C}_a \times \mathcal{SK}_a \to \mathcal{M} \times \perp$.
We consider the traditional IND-CCA security experiment as defined in Figure 6.

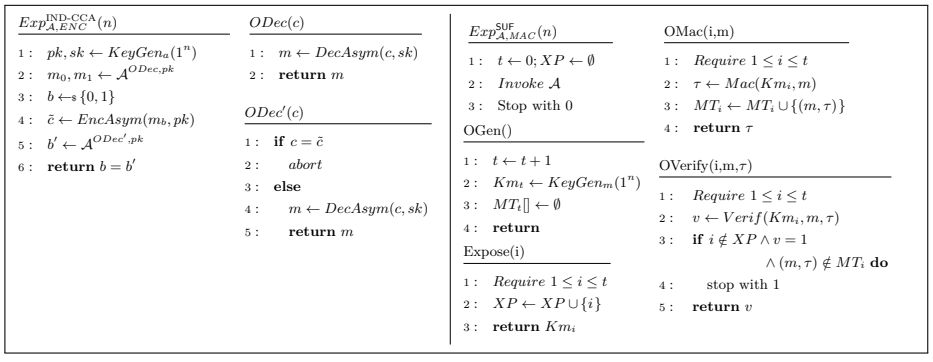


Fig. 6: IND-CCA and multi-instance Strong Unforgeability security experiments.

Definition 10. *We say that an asymmetric encryption scheme* $\mathsf{ENC} = (\mathsf{KeyGen_a}, \mathsf{EncAsym}, \mathsf{DecAsym})$ *has indistinguishability under chosen ciphertext attack (is* IND-CCA*-secure), if for all probabilistic polynomial-time adversaries* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(n)$ *such that:*

$$\mathsf{Adv}^{\text{IND-CCA}}_{\mathcal{A},ENC}(n) = \left|\Pr\left[\mathsf{Exp}^{\text{IND-CCA}}_{\mathcal{A},ENC}(n) = 1\right] - \frac{1}{2}\right| \leq \mathsf{negl}(n)\,.$$

We denote $\epsilon_{\text{IND-CCA}}$ *this advantage.*

To give some examples, RSA-OAEP is proven IND-CCA secure in [15]. In [11], Cramer and Shoup also propose an IND-CCA secure scheme. Another IND-CCA secure solution based on Elliptic Curve (known as $ECIES$) is defined in [27].

Message authentication code A message authentication code scheme MAC is composed of three algorithms: $\mathsf{KeyGen_m} : 1^n \to \mathcal{K}_m$, $\mathsf{Mac} : \mathcal{M} \times \mathcal{K}_m \to \mathcal{T}$, $\mathsf{VerifMac} : \mathcal{M} \times \mathcal{T} \times \mathcal{K}_m \to \{\mathsf{true}, \mathsf{false}\}$.

Multi-instance Strong Unforgeability Strong UnForgeability (SUF) of a MAC scheme in a multi-instance setting has been defined in [25] and we recall the associated experiment in Figure 6. This definition is similar to the classical SUF except that the adversary can play with several instantiations of the MAC by generating several MAC keys, and can expose MAC keys. He will try to forge an instance whose key has not been exposed and on a pair (message, tag) that is not registered as being created by the MAC oracle. The reduction from multi-instance to the classical unforgeability induces a loss factor in the number of instances.

Definition 11. *A scheme* $\mathsf{MAC} = (\mathsf{KeyGen_m}, \mathsf{Mac}, \mathsf{Verif})$ *is secure under strong unforgeability attacks, if for all probabilistic polynomial-time adversaries* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(n)$ *such that:*

$$\mathsf{Adv}^{\mathsf{SUF}}_{\mathcal{A},MAC}(n) = \Pr\left[\mathsf{Exp}^{\mathsf{SUF}}_{\mathcal{A},MAC}(n) = 1\right] \leq \mathsf{negl}(n).$$

We denote ϵ_{SUF} *this advantage.*

For instance, HMAC defined in [4] is proven to have strong existential unforgeability. Its security can then be extended to the multi instance setting as explained in [25].

Multi-stage key-exchange. In their security analysis, Cohn-Gordon *et al.* modeled Signal as a multi-stage key exchange composed of the following algorithms: KeyGen, MedTermKeyGen, Activate, and Run. They define the multistage key-indistinguishability security. For sake of brevity, we do not recall the complete security game here but we just give the following definition.

Definition 12. *A key exchange scheme* $\mathsf{KE} = (\mathsf{KeyGen}, \mathsf{MedTerm\text{-}KeyGen}, \mathsf{Activate}, \mathsf{Run})$ *is a multi-stage key-indistinguishable scheme, if, for all probabilistic polynomial-time adversaries* $\mathcal{A}$*, running* n_P *parties, recording each* n_M *MedTermKeys,* n_S *sessions with maximum* n_s *stages on a session, there exists a negligible function* $\mathsf{negl}(n)$ *such that:*

$$\mathsf{Adv}^{\mathrm{MS-IND}}_{n_P,n_M,n_S,n_s,\mathcal{A},\mathsf{KE}}(n) \leq \mathsf{negl}(n).$$

We denote ϵ_{sig} *this advantage for the Signal protocol.*

The parameter n_M is introduced by the authors to authorize the adversary to choose among a pool of MedTermKeys, which is actually stronger than the actual Signal description. For readability reasons, we consider n_M equal to 1, but our solution works the same with any n_M.

B RDM security analysis

B.1 RDM security model

RDM Indistinguishability. Our experiment is detailed in Figure 7. We complete the state description with flags, not required for the implementation, linked with different oracles:

- `corrupt`. Flag on global state. Set to $false$ at Register, set to $true$ whenever Corrupt is called.

All the followings are session state flags:

- `reveal`. Set to $false$ at Init, set to $true$ whenever Reveal is called on this session.
- `challenge`. Set to $false$ at Init, set to $true$ whenever Challenge is called on this session or on a matching session.
- `active`. Set to $true$ if the device i has been called by oracle OEnc in session s. A device gets active in the protocol as soon as it sends a message.

We note $\pi_i^s.\texttt{flag}$ for $\pi_i^s.\texttt{flag} = true$ and $\neg\pi_i^s.\texttt{flag}$ for $\pi_i^s.\texttt{flag} = false$. We identify a session s by the number s_i of sessions already run by i.

Freshness. We detail below the natural restrictions of our model. The first two are traditional trivial attacks. The third means forward secrecy can be achieved solely for active participants. The fourth models the healing property of the ratchet. The fifth excludes impersonation attacks and finally, the last point models an authentication procedure when adding a new participant, which we consider out of the scope of this protocol. We define:

$\texttt{NoReveal} - \texttt{NoInactiveCorrupt}(\texttt{i}, \texttt{s}, \texttt{t}) =$

- $\neg[\pi_i^s[t].\texttt{reveal} \vee (\pi_i.\texttt{corrupt} \wedge \neg\pi_i^s[t].\texttt{active})]$ and
- $\forall(j, r, t_r)$ chained with (i, s, t),
  $\neg[\pi_j^r[t_r].\texttt{reveal} \vee (\pi_j.\texttt{corrupt} \wedge \neg\pi_j^r[t_r].\texttt{active})]$.

1. $\mathcal{A}$ shall not Reveal state secrets just before the challenge. This is prevented by line 2 of Challenge.

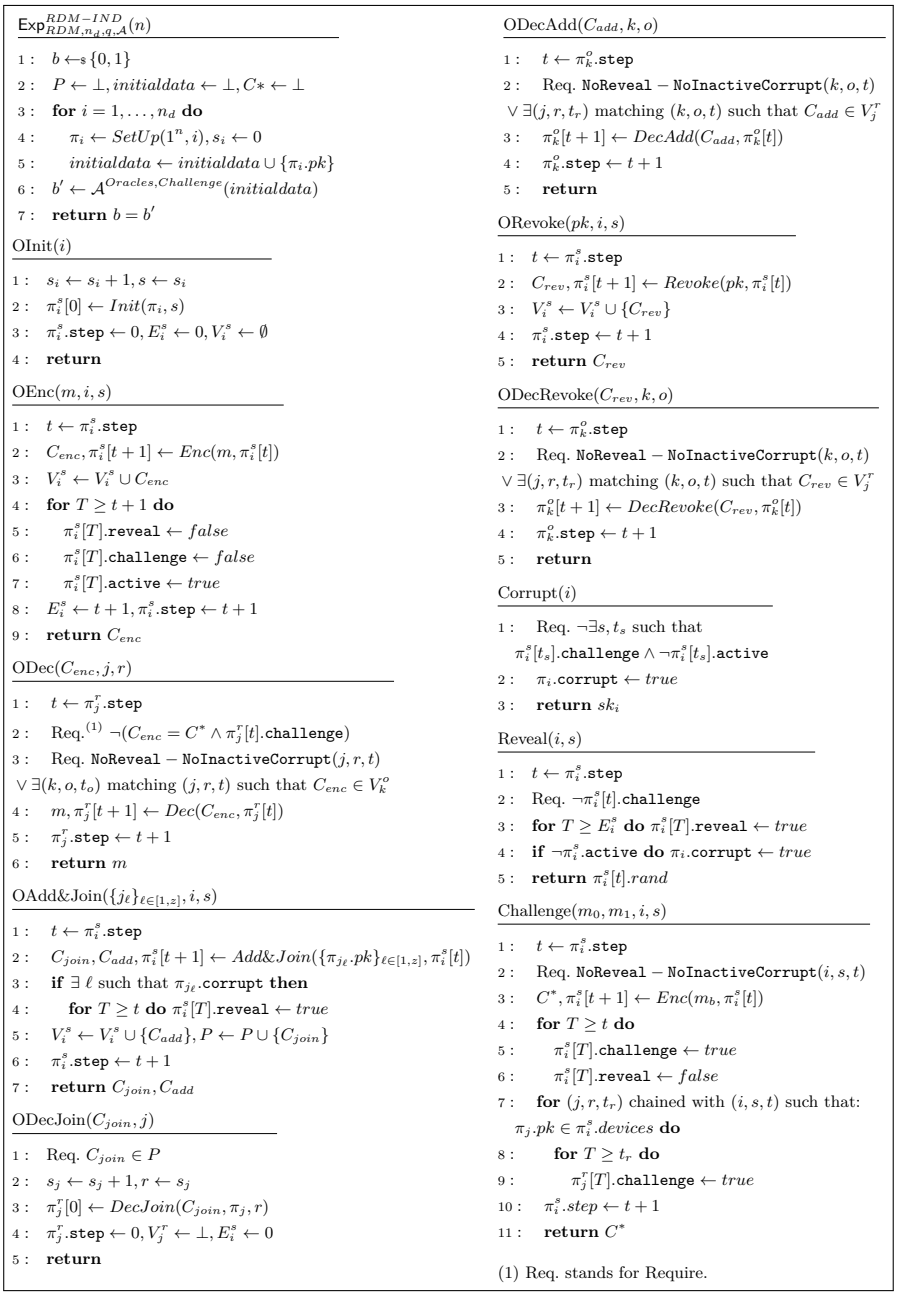


Fig. 7: Ratcheted Dynamic Multicast Indistinguishability experiment.

2. $\mathcal{A}$ shall not Reveal a device concerned with the challenge. This is prevented by line 2 in Reveal and lines 7-9 in Challenge. Line 6 in OEnc ensures the security is back for i's secret after i performs an encryption.

3. $\mathcal{A}$ shall not corrupt a non active device before or after the challenge. This is prevented by line 2 of Challenge and lines 1-2 of Corrupt. A device comes active as soon as it performs an encryption (it enters the ratcheting process).
4. $\mathcal{A}$ shall not reveal random secrets and use them to maliciously send an encrypted message with its own new random, revoke someone, or join a non authorized corrupted device. We prevent this by introducing the register V_i^s that keeps track of ciphertexts produced by the OEnc, OAdd&Join and ORevoke oracles. Line 3 of Oracle ODec, and line 2 of ODecAdd and ODecRevoke check whether the incoming ciphertext has been produced legitimately. If not, those lines verify whether there was a Reveal at the same step.
5. Register P initialized in line 2 of the Exp prevents $\mathcal{A}$ from joining a device in a non existing session or after an exposure. This models an authentication procedure between the device that adds and the new device.

About healing. The `reveal` flag can only be turned back to $false$ with an encryption (line 5 in OEnc). The counter E on line 2 of OEnc registers the step of the last encryption. In Reveal, flags are turned to *true* only from this last encryption. This means a Reveal on a device only threatens the steps from the last encryption and until the next. This corresponds to the healing property. Note that $E_i^s = 0$ is equivalent to $\pi_i^s.\texttt{active} = false$.

Definition. *(Secure Ratcheted Dynamic Multicast) A RDM running with n_d devices is a secure Ratcheted Dynamic Multicast if it is correct and for all adversaries $\mathcal{A}$, running in polynomial time, there exists a negligible function* $\mathsf{negl}(n)$ *such that:*

$$\mathsf{Adv}^{\mathsf{RDM-IND}}_{\mathcal{A},RDM,n_d}(n) = \left|\Pr\left[\mathsf{Exp}^{\mathsf{RDM-IND}}_{\mathcal{A},RDM,n_d}\right] - \frac{1}{2}\right| \leq \mathsf{negl}(n)\,.$$

We denote $\epsilon_{\mathsf{RDM-IND}}$ *this advantage.*

B.2 RDM construction

We give a pseudocode detailed description of our RDM protocol in Figure 8. The restrictions in the Add&Join procedure are there to prevent a same device to be added several times. The requirement in the Revoke procedure prevents a device to revoke itself.

B.3 Proof of Theorem 1

We recall Theorem 1 and provide a detailed proof of this result.

Theorem. *If* ENC *is an* IND-CCA *secure asymmetric encryption scheme, and* MAC *is secure under multi-instance strong unforgeability, the above construction*

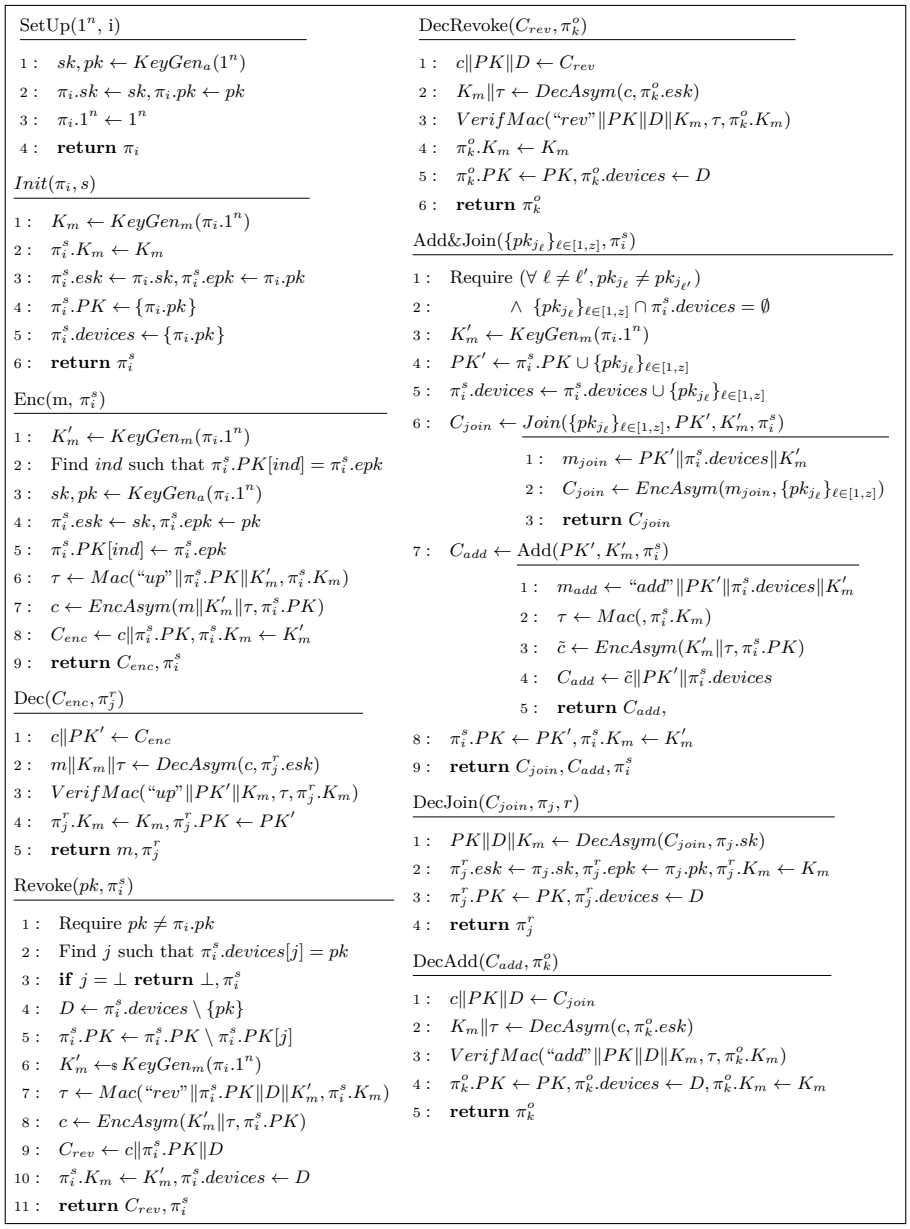


Fig. 8: Ratcheted Dynamic Multicast construction.

is a secure ratcheted dynamic multicast for n_d devices, such that, for any PPT

adversary making at most q queries to the oracles:

$$\mathsf{Adv}^{\mathsf{RDM-IND}}_{RDM,n_d,q,\mathcal{A}}(n) = \left| \Pr\left[Exp^{\mathsf{RDM-IND}}_{RDM,n_d,q,\mathcal{A}}(n) \right] - \frac{1}{2} \right| \leq q \cdot \epsilon_{\mathsf{SUF}} + (q+1) \cdot n_d \cdot \epsilon_{\text{IND-CCA}}.$$

Proof. We consider a lookup table $\mathbb{B}$ in which we will write each Corrupt or Reveal query made by the adversary. Corrupt and Reveal are written and erased in $\mathbb{B}$ the same way corresponding flags are turned to $false$ or $true$, but we need this table to identify the last reveal/corruption healing. We choose to observe the experiment from one session (i, s) but what we actually look at is the behaviour of the group at each step. Hence we consider in the lookup table lines that concerns (i, s) or any matching session. If $\mathbb{B}$ contains only lines for non matching session, we say it is empty. Of course adversary does not only play with (i, s) but also with others. We therefore consider queries on (i, s) or on other sessions (j, r) if it is a matching sessions because it has an effect on (i, s) state. We consider a proof by iteration.

Case of initial sessions

Initialization. We consider an initial session (i, s). We initialize our iteration by considering step 0 of the session. As (i, s) is an initial session, its step 0 concerns only device i. First, a MAC key K_m is generated and is not used at this moment, so it can be considered as non revealed.

Case A: fresh departure. Step 0 is not corrupted nor revealed. $\mathbb{B}$ is empty. It means i is not corrupted and K_m is safe. As K_m is safe, we show $\mathcal{A}$ can not interfere.

Game 0. Let Game 0 be the original game with this configuration.

Game A.1. Let Game A.1 be as Game 0 except the simulator respectively reject all ciphertexts C_{enc} and C_{add} not produced by OEnc and OAdd&Join on input to ODec and ODecAdd (Revocation is not considered here as i cannot revoke himself). Otherwise, we can use it to build an adversary against strong unforgeability of the MAC scheme. We obtain:

$$|\Pr[S_{A.1}] - \Pr[S_0]| \leq \epsilon_{\mathsf{SUF}}.$$

At this point, we consider there are no forgeries, this means that all ciphertexts accepted[5] by different oracles are produced by other oracles and $\mathcal{A}$ cannot join or send encrypted message maliciously.

[5] For ciphertexts accepted by the ODecJoin, the security model already restricts the ciphertext to belong to P (line 1. of this oracle). This is due to the fact that we do not have made a choice to authenticate a device to another. By considering a specific solution to authenticate them, we could delete this restriction in the security model and describe here an additional game A.1' in which we would apply the security of this chosen authentication to obtain C_{join} on input to ODecJoin has been produced by OAdd&Join.

From there, $\mathcal{A}$ can query for an OEnc, $\mathsf{OAdd\&Join}$, $\mathsf{ORevoke}$, Reveal or $\mathsf{Corrupt}$. We do not consider decryptions here as there is still no ciphertext produced by oracles OEnc, $\mathsf{OAdd\&Join}$, and $\mathsf{ORevoke}$. We study each option:

- $\mathsf{Corrupt}(i)$. This query will lead to a non fresh step (step has not changed, it just got corrupted) and (i, s) is written as corrupted in $\mathbb{B}$ (actually, we shall write (i, s') for all sessions runned by i, but, since we are only interested in (i, s) now, we only consider this one).
- $\mathsf{Reveal}(i, s, 0)$. This query will lead to a non fresh step and (i, s) is written as corrupted in $\mathbb{B}$.
- $\mathsf{Enc}(m, i, s)$. Let **Game A.2.Enc** be as Game A.1 but ensures $\mathcal{A}$ learns nothing about the newly generated MAC key K'_m. That is we replace the ciphertext $c = \mathsf{EncAsym}(m\|K'_m\|\tau, \pi_i^s.PK)$ produced in OEnc by a random $c' = \mathsf{EncAsym}(rand, \pi_i^s.PK)$ and keep the couple $(c', m\|K'_m\|\tau)$ in a list L for later decryption. i is not corrupted and is the only receiver of this encryption ($\pi_i^s.PK = \pi_i.pk$). IND-CCA security of the encryption scheme ensures we can do this substitution properly with loss:

  $$|\Pr[S_{A.2.Enc}] - \Pr[S_{A.1}]| \leq \epsilon_{\text{IND-CCA}}.$$

  We are in a fresh state with secure MAC key K_m.
- $\mathsf{OAdd\&Join}\left(\{j_\ell\}_{\ell\in[1,z]}, i, s\right)$ with none of the j_ℓ corrupted. Consider a **Game A.2.Join** that runs as Game A.1 except that we ensure that $\mathcal{A}$ learns nothing about the newly generated MAC key K'_m. We replace the ciphertext $C_{join} = \mathsf{EncAsym}(PK'\|K'_m, \{pk_{j_\ell}\}_{\ell\in[1,z]})$ produced in Join of $\mathsf{OAdd\&Join}$ by $C_{join} = \mathsf{EncAsym}(rand, \{pk_{j_\ell}\}_{\ell\in[1,z]})$ and ciphertext $\tilde{c} = \mathsf{EncAsym}(K'_m\|\tau, \pi_i^s.PK)$ produced in Add of $\mathsf{OAdd\&Join}$ by $\tilde{c}' = \mathsf{EncAsym}(rand, \pi_i^s.PK)$. We keep both couples $(C'_{join}, PK'\|K'_m, \{pk_{j_\ell}\}_{\ell\in[1,z]})$ and $(\tilde{c}', K'_m\|\tau, \pi_i^s.PK)$ in the list L for later decryption. None of the $\{j_\ell\}_{\ell\in[1,z]}$ is corrupted and they are the only receivers of the encryption C_{join} and i is not corrupted and is the only receiver of the encryption $\tilde{c}$. IND-CCA security of the encryption scheme - extended to parallel encryption - ensures we can do both substitutions properly with loss:

  $$\begin{aligned}|\Pr[S_{A.2.Join}] - \Pr[S_{A.1}]| &\leq \epsilon_{\text{IND-CCA}} + z \cdot \epsilon_{\text{IND-CCA}} \\ &\leq n_d \cdot \epsilon_{\text{IND-CCA}}.\end{aligned}$$

  We are in a fresh state with secure MAC key K_m.
- $\mathsf{OAdd\&Join}(\{\mathsf{j}_\ell\}_{\ell\in[1,\mathsf{z}]}, \mathsf{i}, \mathsf{s})$ with one or more j_ℓ corrupted. For each j_ℓ corrupted, $(j_\ell, allsessions)$ is written as corrupted in $\mathbb{B}$. Adversary obtains the new MAC key K'_m through the joining process and we are in a non fresh step with unsafe MAC key.

Case B: unfresh departure. If i is corrupted, (i, s) is written as corrupted in $\mathbb{B}$ and step 0 is not suitable for a $\mathsf{Challenge}$ query.
As before, $\mathcal{A}$ can query for an OEnc, $\mathsf{OAdd\&Join}$, and Reveal (a query $\mathsf{OCorrupt}$ is useless here as i is already corrupted). We do not consider decryption oracles here as there is still no ciphertext produced by oracles OEnc, $\mathsf{OAdd\&Join}$, and $\mathsf{ORevoke}$ and $\mathcal{A}$ cannot join, revoke, or send encrypted message maliciously. This

last point is due to the natural restrictions in the security model: the model does not allow $\mathcal{A}$ to forge a message after a corruption or an exposure.

Game 0. Let Game 0 be the original game with this configuration. We study each option:

- $\mathsf{Reveal}(i, s, 0)$. This step remains a non fresh step and i is still written as corrupted in $\mathbb{B}$.
- $\mathsf{OEnc}(m, i, s)$. Let **Game B.1.Enc** be as Game 0 but ensures $\mathcal{A}$ learns nothing about the newly generated MAC key K'_m. We replace the ciphertext $c = \mathsf{EncAsym}(m \| K'_m \| \tau, \pi_i^s.PK)$ produced in OEnc by $c' = \mathsf{EncAsym}(rand, \pi_i^s.PK)$ and keep the couple $(c', m \| K'_m \| \tau)$ in the list L for later decryption. i is corrupted, but during the execution of OEnc, this oracle generates for i a new secret/public key (sk, pk) (line 3. of Enc) unknown to $\mathcal{A}$ and we can delete the entry (i, s) from the table $\mathbb{B}$. $\mathbb{B}$ is now empty. The ciphertext c is intended for the sole new key of i, non corrupted at this step. IND-CCA security of the encryption scheme ensures we can do this substitution properly with loss:

$$|\Pr[S_{B.1.Enc}] - \Pr[S_0]| \leq \epsilon_{\text{IND-CCA}}.$$

  We are in a fresh state with secure MAC key K_m.
- $\mathsf{OAdd\&Join}(\{j_\ell\}_{\ell \in [1,z]}, i, s)$ with none of the j_ℓ corrupted. As $\mathcal{A}$ get the new MAC key K'_m through the Add part of $\mathsf{Add\&Join}$, we demand no security on the private side apart from the model restrictions. We are in a non fresh state with unsafe MAC key.
- $\mathsf{OAdd\&Join}(\{j_\ell\}_{\ell \in [1,z]}, i, s)$ with one or more j_ℓ corrupted. For each j_ℓ corrupted, $(j_\ell, allsessions)$ is written as corrupted in $\mathbb{B}$. Adversary obtains the new MAC key K'_m through the joining process and we are in a non fresh step with unsafe MAC key.

Iteration. We now consider a session (i, s) at step t in the protocol after q queries. Each reveal or corrupt has been registered in $\mathbb{B}$. As before, we consider case A when we start from a fresh step and case B when we start from a non fresh step.

Case A: fresh departure ($\mathbb{B}$ is empty).

Game 0. Let Game 0 be the original game with this configuration.

Game A.1. Let Game A.1 be as Game 0 except the simulator respectively reject all ciphertexts C_{enc}, C_{add}, and C_{rev} not produced by OEnc, $\mathsf{OAdd\&Join}$, and $\mathsf{ORevoke}$ on input to ODec, $\mathsf{ODecAdd}$, and $\mathsf{ORevoke}$. Otherwise, we can use it to build an adversary against strong unforgeability of the MAC scheme. We obtain:

$$|\Pr[S_{A.1}] - \Pr[S_0]| \leq \epsilon_{\mathsf{SUF}}.$$

At this point, all ciphertexts accepted[6] by different oracles are produced by other oracles and $\mathcal{A}$ cannot join or send encrypted message maliciously.

[6] For ciphertexts accepted by the $\mathsf{ODecJoin}$, as already explained, $\mathcal{A}$ cannot join a maliciously *DecJoin* due to the security model.

From there $\mathcal{A}$ can query for an OEnc, $\mathsf{OAdd\&Join}$, $\mathsf{ORevoke}$, Reveal, or $\mathsf{Corrupt}$. Decryption oracles do not induce changes on the honesty of the devices, on the secret of the MAC key, or on the freshness of the step due to Game A.1, so we do not detail them. We note $t = \pi_i^s.\texttt{step}$ We study each option:

- $\mathsf{Corrupt}(i)$. If i is active in the session - meaning that it already does not use his long term secret key $\pi_i.sk$ to communicate with the group of devices -, then nothing happens. Else (i, s) is written as corrupted in $\mathbb{B}$.
- $\mathsf{Corrupt}(j)$. If there exists r, $t_s \leq t$ such that $(j, r, \pi_j^r.\texttt{step})$ is non active and matches (i, s, t_s), (j, r) is written as corrupted in $\mathbb{B}$. Otherwise, as j is active on all sessions matching with (i, s), nothing happens.
- $\mathsf{Reveal}(i, s)$. (i, s) is written as corrupted in $\mathbb{B}$.
- $\mathsf{Reveal}(j, r)$. If there exists $t_s \leq t$ such that $(j, r, \pi_j^r.\texttt{step})$ matches (i, s, t_s), (j, r) is written as corrupted in $\mathbb{B}$. If $t_s < t$, then the MAC key has changed but as the secret key of j is known to the adversary, he has access to the new MAC key also.
- $\mathsf{OEnc}(m, i, s)$. Let **Game A.2.Enc** be the same as Game A.1 but ensures $\mathcal{A}$ learns nothing about the newly generated MAC key K'_m. That is we replace the ciphertext produced in OEnc, $c = \mathsf{EncAsym}(m \| K'_m \| \tau, \pi_i^s.PK)$, by $c' = \mathsf{EncAsym}(rand, \pi_i^s.PK)$ and keep the couple $(c', m \| K'_m \| \tau)$ in the list L for later decryption. $\mathbb{B}$ is empty, IND-CCA security of the encryption scheme ensures we can do this substitution properly with loss $n_{|PK|}$ where $n_{|PK|}$ is the number of public keys in $\pi_i^s.PK$, equals to the number of devices in the group at this step:

$$\begin{aligned} |\Pr[S_{A.2.Enc}] - \Pr[S_{A.1}]| &\leq n_{|PK|} \cdot \epsilon_{\text{IND-CCA}} \\ &\leq n_d \cdot \epsilon_{\text{IND-CCA}}. \end{aligned}$$

  We are in a fresh state with secure MAC key K_m.
- $\mathsf{OEnc}(m, j, r)$. If $(j,r,\pi_j^r.step)$ matches $(i, s, t_s < t)$ then the message produced is not valid. In fact, (j, r) matches on an older MAC key than the one (i, s) uses now. If $(j,r,\pi_j^r.step)$ matches (i, s, t), just process the same as for $\mathsf{OEnc}(m, i, s)$. Otherwise do nothing.
- $\mathsf{OAdd\&Join}(\{k_\ell\}_{\ell \in [1,z']}, i, s)$ with none of the k_ℓ corrupted. Let the next **Game A.2.Join** be as Game A.1 but ensures $\mathcal{A}$ learns nothing about the newly generated MAC key K'_m. That is we replace the ciphertext $C_{join} = \mathsf{EncAsym}(PK' \| K'_m, pk_k)$ produced in the Join subprocedure of $\mathsf{OAdd\&Join}$ by $C_{join} = \mathsf{EncAsym}(rand, pk_k)$ and the cipher $\tilde{c} = \mathsf{EncAsym}(K'_m \| \tau, \pi_i^s.PK)$ produced in Add of $\mathsf{OAdd\&Join}$ by $\tilde{c}' = \mathsf{EncAsym}(rand, \pi_i^s.PK)$. We keep both couples $(C'_{join}, PK' \| K'_m, pk_k)$ and $(\tilde{c}', K'_m \| \tau, \pi_i^s.PK)$ in the list L for later decryption. None of the k_ℓ is corrupted and they are the only receivers of the encryption C_{join}. $\mathbb{B}$ is empty which means that the receivers of $\tilde{c}$ are not corrupted. IND-CCA security of the encryption scheme ensures we can do both substitutions properly with loss where $n_{|PK|}$ is the number of public keys in $\pi_i^s.PK$, equals to the number of devices in the group:

$$\begin{aligned} |\Pr[S_{A.2.Join}] - \Pr[S_{A.1}]| &\leq (n_{|PK|} + z') \cdot \epsilon_{\text{IND-CCA}} \\ &\leq n_d \cdot \epsilon_{\text{IND-CCA}}. \end{aligned}$$

We are in a fresh state with secure MAC key K_m.

- $\mathsf{OAdd\&Join}(\{\mathsf{k}_\ell\}_{\ell\in[1,z']}, \mathsf{i}, \mathsf{s})$ with one or more k_ℓ corrupted. For each k_ℓ corrupted, $(k_\ell, allsessions)$ is written as corrupted in $\mathbb{B}$. Adversary obtains the new MAC key K'_m through the joining process and we are in a non fresh step with unsafe MAC key.
- $\mathsf{OAdd\&Join}(\{k_\ell\}_{\ell\in[1,z']}, j, r)$. If $(j,r,\pi_j^r.step)$ matches $(i, s, t_s < t)$ then the message produced is not valid. In fact, (j, r) matches on an older MAC key than the one (i, s) uses now. If $(j,r,\pi_j^r.step)$ matches (i, s, t), just processes the same as for the previous case $\mathsf{OAdd\&Join}(\{k_\ell\}_{\ell\in[1,z']}, i, s)$. Otherwise do nothing.
- $\mathsf{ORevoke}(pk, i, s)$. The construction verifies that $\exists k$ such that $pk = pk_k$[7]. $\mathbb{B}$ is empty, we revoke k even if it was not written as corrupted - in reality this allows to prevent a further corruption. Let **Game A.2.Rev** be as Game A.1 but ensures $\mathcal{A}$ learns nothing about the newly generated MAC key K'_m. Again, we replace the ciphertext $c = \mathsf{EncAsym}(K'_m \| \tau, \pi_i^s.PK)$ produced in $\mathsf{ORevoke}$ by $c' = \mathsf{EncAsym}(rand, \pi_i^s.PK)$ and keep the couple $(c', K'_m \| \tau)$ in the list L for later decryption. $\mathbb{B}$ is empty, IND-CCA security of the encryption scheme ensures we can do this substitution properly with loss $n_{|PK|}$ where $n_{|PK|}$ is the number of public keys in $\pi_i^s.PK$, equals to the number of devices in the group:

$$\begin{aligned} |\Pr[S_{A.2.Rev}] - \Pr[S_{A.1}]| &\leq n_{|PK|} \cdot \epsilon_{\text{IND-CCA}} \\ &\leq n_d \cdot \epsilon_{\text{IND-CCA}}. \end{aligned}$$

  We are in a fresh state with secure MAC key K_m.
- $\mathsf{ORevoke}(pk_k, j, r)$. If $(j,r,\pi_j^r.step)$ matches $(i, s, t_s < t)$ then the message produced is not valid. If $(j,r,\pi_j^r.step)$ matches (i, s, t),just processes the same as $\mathsf{ORevoke}(pk_k, i, s)$. Otherwise do nothing.

Case B: unfresh departure. If $\mathbb{B}$ is not empty, there is at least an identifier (device, session) written as corrupted in $\mathbb{B}$ and step t is not suitable for the $\mathsf{Challenge}$ query.

As before, $\mathcal{A}$ can query for an OEnc, $\mathsf{OAdd\&Join}$, $\mathsf{ORevoke}$, Reveal, and $\mathsf{Corrupt}$. As for the case B of the initialization, $\mathcal{A}$ cannot join, revoke, or send encrypted message maliciously. Again, this is due to natural restrictions in the security model.

Game 0. Let Game 0 be the original game with this configuration. We study each option:

- $\mathsf{Corrupt}(i)$. If i is active in the session - meaning that it already does not use his long term secret key $\pi.sk$ to communicate with the group of devices -, then nothing happens. Else (i, s) is written as corrupted in $\mathbb{B}$.
- $\mathsf{Corrupt}(j)$. If there exists r, $t_s \leq t$ such that $(j, r, \pi_j^r.\mathtt{step})$ is non active and matches (i, s, t_s), (j, r) is written as corrupted in $\mathbb{B}$. Otherwise, as j is active on all sessions matching with (i, s), nothing happens.
- $\mathsf{Reveal}(i, s)$. (i, s) is written as corrupted in $\mathbb{B}$.

[7] In the following, we assume that revoke queries are always on a valid long term key pk_k.

- $\mathsf{Reveal}(j, r)$. If there exists $t_s \leq t$ such that $(j, r, \pi_j^r.step)$ matches (i, s, t_s), (j, r) is written as corrupted in $\mathbb{B}$. If $t_s < t$, then the MAC key has changed but as the secret key of j is know to adversary, he will access to the new MAC key also.
- $\mathsf{OEnc}(m, i, s)$. If (i, s) is the sole entry written as corrupted in $\mathbb{B}$, then as for the case B in step 0, we regain security. (i, s) is now not written as corrupted in $\mathbb{B}$. Let in this case, **Game 1.Enc.fresh** be as Game 0 but ensures $\mathcal{A}$ learns nothing about the newly generated MAC key K'_m. We replace the ciphertext $c = \mathsf{EncAsym}(m \| K'_m \| \tau, \pi_i^s.PK)$ produced in OEnc by $c' = \mathsf{EncAsym}(rand, \pi_i^s.PK)$ and keep the couple $(c', m \| K'_m \| \tau)$ in the list L for later decryption. $\mathbb{B}$ is empty, CCA-security of the encryption scheme ensures we can do this substitution properly with loss $n_{|PK|}$ where $n_{|PK|}$ is the number of public key in $\pi_i^s.PK$, equals to the number of devices in the group:

$$\begin{aligned} |\Pr[S_{A.2.Enc}] - \Pr[S_{A.1}]| &\leq n_{|PK|} \cdot \epsilon_{\text{IND-CCA}} \\ &\leq n_d \cdot \epsilon_{\text{IND-CCA}}. \end{aligned}$$

  We are in a fresh state with secure MAC key K_m.
  If $\mathbb{B}$ contains lines other than (i, s), security is still not back. (i, s) is erased from $\mathbb{B}$ if it was previously written. Nothing more is expected since the new MAC key K'_m is accessible through remained corrupted/revealed devices and we are in a non fresh state.
- $\mathsf{OEnc}(m, j, r)$. If $(j,r,\pi_j^r.step)$ matches $(i, s, t_s < t)$ then the message produced is not valid, do nothing. If $(j,r,\pi_j^r.step)$ matches (i, s, t), just process the same as for $\mathsf{OEnc}(m, i, s)$. Otherwise do nothing.
- $\mathsf{OAdd\&Join}(\{k_\ell\}_{\ell \in [1,z']}, i, s)$ with none of the $\{k_\ell\}_{\ell \in [1,z']}$ corrupted. We do nothing as the adversary will obtain the new MAC key K'_m through the ciphertext C_{add}. We remain in a non fresh step with unsafe MAC key.
- $\mathsf{OAdd\&Join}(\{k_\ell\}_{\ell \in [1,z']}, i, s)$ with one or more of the $\{k_\ell\}_{\ell \in [1,z']}$ corrupted. For all corrupted k_ℓ we write the newly generated (k_ℓ, r_ℓ) as corrupted in $\mathbb{B}$ as soon as $\mathsf{DecJoin}(k_\ell, c_{join})$ is queried on the returned C_{join} message and do nothing else as the adversary is able to obtain the new MAC key K'_m. We remain in a non fresh step with unsafe MAC key.
- $\mathsf{OAdd\&Join}(\{k_\ell\}_{\ell \in [1,z']}, j, r)$. If $(j,r,\pi_j^r.step)$ matches $(i, s, t_s < t)$ then the message produced is not valid, do nothing. If $(j,r,\pi_j^r.step)$ matches (i, s, t), just process the same as for the previous case $\mathsf{OAdd\&Join}(\{k_\ell\}_{\ell \in [1,z']}, i, s)$. Otherwise do nothing.
- $\mathsf{ORevoke}(pk_k, i, s)$. For all $o, t_o, t_s \leq t$ such that (k, o) is written as corrupted in $\mathbb{B}$ and (k, o, t_o) is matching (i, s, t_s), we erase (k, o) from $\mathbb{B}$. If (k, o) was the sole entry written as corrupted in $\mathbb{B}$, then as for the case B in step 0, we regain security. Let in this case, **Game 1.Rev.fresh** be as Game 0 but ensures $\mathcal{A}$ learns nothing about the newly generated MAC key K'_m. We replace the ciphertext $c = \mathsf{EncAsym}(K'_m \| \tau, \pi_i^s.PK)$ produced in $\mathsf{ORevoke}$ by $c' = \mathsf{EncAsym}(rand, \pi_i^s.PK)$ and keep the couple $(c', K'_m \| \tau)$ in the list L for later decryption. $\mathbb{B}$ is now empty, IND-CCA security of the encryption scheme ensures we can do this substitution properly with loss $n_{|PK|}$ where $n_{|PK|}$ is the number of public keys in $\pi_i^s.PK$, equals to the number of devices

in the group:

$$|\Pr[S_{A.2.Rev}] - \Pr[S_{A.1}]| \leq n_d \cdot \epsilon_{\text{IND-CCA}}.$$

We are in a fresh state with secure MAC key K_m. If $\mathbb{B}$ contains other lines not concerning (k, o), security is still not back. Nothing more is attended since the new MAC key K'_m is accessible through remained corrupted/revealed devices and we are in a non fresh state.

- $\mathsf{ORevoke}(pk_k, j, r)$. If $(j,r,\pi_j^r.step)$ matches $(i, s, t_s < t)$ then do nothing. If $(j,r,\pi_j^r.step)$ matches (i, s, t), just process the same as for $\mathsf{ORevoke}(pk_k, i, s)$. Otherwise do nothing.

Case of non initial sessions

Through restrictions, a session can only be created by the OInit. This means that every session is chained with an initial session. The only modification in our game hops is that we now consider chained session instead of solely matching sessions. (For instance in a case $\mathsf{OEnc}(m, j, r)$, replace "If $(j,r,\pi_j^r.step)$ matches $(i, s, t_s < t)$ then the message produced is not valid, do nothing" by "If $(j,r,\pi_j^r.step)$ is chained with (i, s, t) then the message produced is not valid, do nothing.") The above game hops are still valid because one can go from the initial session to the one of our interest following a chain of sessions. As we consider actions on chained sessions and as those chains are taken into account in our restrictions, the same logic applies.
Consider the event E: after q queries any session state is either corrupted or a fresh state with no adversary having interfered. Suppose, n_d is a fixed value and correspond to the maximum number of devices the adversary can play with, we obtain:

$$\Pr[E] \leq q \cdot \epsilon_{\mathsf{SUF}} + q \cdot n_d \cdot \epsilon_{\text{IND-CCA}}.$$

Finally when adv queries a $\mathsf{Challenge}$, if the step is not fresh, challenge is not valid. If the step is fresh with secret keys and no adversary having infiltrated the group, IND-CCA security of the encryption scheme used n_d parallel times gives:

$$\begin{aligned}\mathsf{Adv}^{\mathsf{RDM-IND}}_{RDM,n_d,q,\mathcal{A}}(n) &= \left|\Pr\left[Exp^{\mathsf{RDM-IND}}_{RDM,n_d,q,\mathcal{A}}(n)\right] - \frac{1}{2}\right| \\ &\leq \Pr[E] + n_d \cdot \epsilon_{\text{IND-CCA}} \\ &\leq q \cdot \epsilon_{\mathsf{SUF}} + (q+1) \cdot n_d \cdot \epsilon_{\text{IND-CCA}}.\end{aligned}$$

C Multi-Device Signal security analysis

C.1 Multi-Device Instant Messaging ($MDIM$) formal definition

Let $\{\mathcal{P}_1, \ldots, \mathcal{P}_{n_U}\}$ be the set of users in the protocol and for each user $\mathcal{P}_u$, let $\{\mathtt{d}_{u,1},\ldots,\mathtt{d}_{u,n_d}\}$ be the set of his devices. Each device $\mathtt{d}_{u,i}$ is modeled by an oracle

$\pi_{u,i}$ and each session s executed by a device $\mathtt{d}_{u,i}$ is modeled by an oracle $\pi^s_{u,i}$. Device oracles maintain device states and session oracles maintain session states as defined in Section 3. In the following, device or session oracles and their state will be considered as equals.

Protocol description We formalize here the description of our Multi Device Signal protocol.

- $\mathsf{UserKeyGen}(1^n) \rightarrow \mathit{pprekeys}, \mathit{sprekeys}$. Generates two lists of prekeys. These lists gathers respectively public and private user keys, and optional extra keying material. We call these keys the "prekey bundle".
- $\mathsf{DeviceSetUp}(1^n, (u, i)) \rightarrow \pi_{u,i}$. Generates the device key and records them in a new state $\pi_{u,i}$.
- $\mathsf{Register}(\mathsf{uID}, \pi_{u,i}) \rightarrow \pi_{u,i}$. Creates the prekey bundle and uID, and registers them together with the device key on the server. Completes the device state $\pi_{u,i}$ with the prekey bundle and uID.
- $\mathsf{InitSession}(\mathit{role}, \mathit{pprekeys}_v, \pi_{u,i}, s) \rightarrow C_{init}, c_{out}, \pi^s_{u,i}$. On input a role, the $\mathit{pprekeys}_v$ of the intended peer v, a device state $\pi_{u,i}$, and a session identifier s, returns a (eventually empty) ciphertext C_{init} for u's other devices, a (possibly empty) ciphertext c_{out} for the intended peer, and a session state $\pi^s_{u,i}$.
- $\mathsf{ReceiveInitSession}(C_{init}, \pi_{u,j}, r) \rightarrow \pi^r_{u,j}$. On input a ciphertext C_{init}, a device state $\pi_{u,j}$, and a session identifier r, outputs a session state $\pi^r_{u,j}$.
- $\mathsf{Send}(m, \pi^s_{u,i}) \rightarrow C_{in}, c_{out}, \pi^s_{u,i}$. On input a plaintext m and a session state $\pi^s_{u,i}$, returns one ciphertext C_{in} for u's other devices, a second ciphertext c_{out} for the intended peer $\pi^s_{u,i}.\mathit{peer}$, and an updated state $\pi^s_{u,i}$.
- $\mathsf{ReceiveIn}(C_{in}, \pi^r_{u,j}) \rightarrow m, \pi^r_{u,j}$. On input a ciphertext C_{in} and a session state $\pi^r_{u,j}$, outputs a (eventually empty) message m and an updated session state $\pi^r_{u,j}$.
- $\mathsf{ReceiveOut}(c_{out}, \pi^p_{v,\ell}) \rightarrow m, \pi^p_{v,\ell}$. On input a ciphertext c_{out} and a session state $\pi^p_{v,\ell}$, outputs a (potentially empty) message m and an updated session state $\pi^p_{v,\ell}$.
- $\mathsf{Add\&Join}(\mathit{dpk}_j, \pi_{u,i}) \rightarrow \{C_{join,s}, C_{add,s}, c_{out,s}\}_{s \in S}, \pi_{u,i}$
  with $S = \pi_{u,i}.\mathit{Sessions}$. On input a device public key dpk_j and a device state $\pi_{u,i}$, returns, for each session s the device i is engaged in, one ciphertext $C_{join,s}$ for the new device j, one ciphertext $C_{add,s}$ for all the other devices, a ciphertext $c_{out,s}$ for the intended peer, and an updated state $\pi_{u,i}$ (comprising updated session states $\pi^s_{u,i}$ for each s).
- $\mathsf{DecJoin}\left(\{C_{join,r}\}_{r \in [1,R]}, \pi_{u,j}\right) \rightarrow \pi_{u,j}$. On input R ciphertexts $C_{join,r}$ and a device state $\pi_{u,j}$, returns an updated device state $\pi_{u,j}$ (comprising R new session states $\pi^r_{u,j}$).
- $\mathsf{DecAdd}\left(\{C_{add,o}\}_{o \in [1,O]}, \pi_{u,k}\right) \rightarrow \pi_{u,k}$. On input O ciphertexts $C_{add,o}$ and a device state $\pi_{u,k}$, returns an updated device state $\pi_{u,k}$ (comprising updated session states $\pi^o_{u,k}$ for each o).

- $\mathsf{Revoke}(dpk, \pi_{u,i}) \rightarrow \{C_{rev,s}, c_{out,s}\}_{s \in \pi_{u,i}.Sessions}$, $pprekeys$, $\pi_{u,i}$. On input a device public key dpk and a user state $\pi_{u,i}$, returns, for each session s the device i is engaged in, a ciphertext $c_{rev,s}$ for all the other devices of user u and a ciphertext $c_{out,s}$ for the intended peer, and a new user state $\pi_{u,i}$ (comprising updated session states $\pi^s_{u,i}$ for each s).
- $\mathsf{DecRevoke}\left(\{C_{rev,o}\}_{o \in [1,O]}, \pi_{u,k}\right) \rightarrow \pi_{u,k}$. On input O ciphertexts $C_{rev,o}$ and a device state $\pi_{u,k}$, returns an updated device state $\pi_{u,k}$ (comprising updated sessions states $\pi^o_{u,k}$ for each o).

Protocol steps. As in the RDM model, we consider the steps of the protocol. Each Send, Add, Revoke, or corresponding Receive or Dec algorithm brings the session to a new step and corresponding oracles will increment the step counter. If no change occurs, values are transmitted from one step to another (e.g. if state does not change at step t_s, then $\pi^s_{u,i}[t_s + 1].state = \pi^s_{u,i}[t_s].state$). We refer to session s run by $\pi_{u,i}$ at step t_s as (u, i, s, t_s)

Matching sessions. Considering partnering sessions, we define the session identifier $\mathtt{sid_1}$ as the concatenation of ciphertexts C_{in} sent by $\mathsf{Send}(\cdot, \pi^{\mathsf{s}}_{\mathsf{u,i}})$ or received through $\mathsf{ReceiveIn}(\cdot, \pi^{\mathsf{s}}_{\mathsf{u,i}})$. Hence we can write the definition of partnering as follow.

Definition. *(Partnered sessions at some step.) Two sessions (u, i, s, t_s) and (u, j, r, t_r) are partnered if:*

- $\pi^s_{u,i}.role = \pi^r_{u,j}.role$,
- $\pi^s_{u,i}.peer = \pi^r_{u,j}.peer$,
- $\pi^s_{u,i}.uID = \pi^r_{u,j}.uID$,
- *there exists* $\mathtt{sid'}$ *subset of* $\pi^s_{u,i}[t_s].\mathtt{sid_1}$ *such that* $\pi^s_{u,i}[t_s].\mathtt{sid_1} \tilde{=} \mathtt{sid'} \| \pi^r_{u,j}[t_r].\mathtt{sid_1}$ *(or reversely for $\pi^s_{u,i}$ and $\pi^r_{u,j}$ if j was there for longer than i), where $\tilde{=}$ is defined as in Definition 1.*

The definition for a chain of sessions and matching sessions are the one given in paragraph 3.1.

MDIM indistinguishability The MDIM-IND complete experiment is described in Figure 9. We complete a session state $\pi^s_{u,i}$ with flags linked to different oracles. All flags are initialized with false. Flag `active`, is set to *true* as soon as $\mathsf{OSend}(\pi^{\mathsf{s}}_{\mathsf{u,i}})$ is called. In a general way, `revX` is set to *true* whenever revealX is invoked on this session. A flag value is transmitted from one step to another.
We detail below the different registers and counters we need to define our freshness.

- The register $V^s_{u,i}$, as in RDM model in Section 2.1, is there to prevent $\mathcal{A}$ from interfering if a device randomness is corrupted.
- The counter $E^s_{u,i}$ records the step of the last Send.

- The counter $F^s_{u,i}$ records the step when the last change of randomness (what we call ratchet) occurred.
- The counter $G^s_{u,i}$ records the step when the last change of state occurred. $E^s_{u,i}$, $F^s_{u,i}$ and $G^s_{u,i}$ are useful to turn all necessary flags to $true$ when Reveal queries occur. If the Reveal query happens, then we turn all flags from $E^s_{u,i}$ (resp. $F^s_{u,i}$, $G^s_{u,i}$) to $true$. Some actions as Send or Add&Join or Revoke may turn them back to $false$. This corresponds to the healing property. Remark that in OSend and OReceiveIn, we identify ratchets by comparing random values. In OReceiveOut, we identify on state changes by comparing state values.
- The list $A^s_{u,i}$ records all steps when an addition of a new device occurs. We need to keep track of these because information is sent to the newcomer. This list is emptied with every revocation.

For readability reasons we define:
ResetRand − State − Session(u, i, s, t)=
$\pi^s_{u,i}[T].\texttt{revRand} \leftarrow false$ and $\pi^s_{u,i}[T].\texttt{revState} \leftarrow false$ and
$\pi^s_{u,i}[T].\texttt{revSessionKey} \leftarrow false$.
ResetState − Session(u, i, s, t)=
$\pi^s_{u,i}[T].\texttt{revState} \leftarrow false$ and $\pi^s_{u,i}[T].\texttt{revSessionKey} \leftarrow false$.
NoReveal − NoInactiveCorrupt(u, i, s, t)=
$\neg[\pi^s_{u,i}[t].\texttt{revDevRand} \vee(\pi_{u,i}.\texttt{corruptDevice} \wedge \neg\pi^s_{u,i}[t].\texttt{active})]$ and
$\forall(u, j, r, t_r)$ chained with (u, i, s, t)
$\neg[\pi^r_{u,j}[t_r].\texttt{revDevRand} \vee (\pi_{u,j}.\texttt{corruptDevice} \wedge \neg\pi^r_{u,j}[t_r].\texttt{active})]$.

Freshness. We formalize here the intuition of the freshness given in Section 3.1.
We define:
DeviceLeak(u, i, s, t_s) =
· $\pi^s_{u,i}[t_s].\texttt{revDevRand}$
· or $(\pi_{u,i}.\texttt{corruptDevice} \wedge \neg\pi^s_{u,i}[t_s].\texttt{active})$
· or $\exists$ (u, j, r, t_r) partnering (u, i, s, t_s) such that:
  · $\pi^r_{u,j}[t_r].\texttt{revDevRand}$
  · or $(\pi_{u,j}.\texttt{corruptDevice} \wedge \neg\pi^r_{u,j}[t_r].\texttt{active})$
· or $\exists$ (u, j, r, t_r) chained with (u, i, s, t_s) such that:
  · $\pi_{u,j}.pk \in \pi^s_{u,i}[t_s].devices$ and
    * $\pi^r_{u,j}[t_r].\texttt{revDevRand}$
    * or $(\pi_{u,j}.\texttt{corruptDevice} \wedge \neg\pi^r_{u,j}[t_r].\texttt{active})$.

Initial freshness Freshness of the initial non interactive key exchange (NIKE) is treated separately, because it is proper to X3DH, the NIKE used by Signal, specified in [22]. One can imagine adopting another NIKE and designing a new ad hoc initiation freshness. Let I be an interval of protocol steps. For readability reasons we define:
DevLeakPrekeys(u, i, s, I) = $\exists$ a step $t \in Add^s_{u,i} \cap I$ such that DeviceLeak(u, i, s, t).
CorruptUser(u, I) = $u.\texttt{corruptUser}$ within the period I or $\exists$ (u, i, s) such that

$Exp^{\mathsf{MDIM-IND}}_{\mathcal{A},MDIM,n_p,n_d}(n)$

$b \leftarrow\!\!\$\ \{0,1\}$
$tested \leftarrow \bot$
$initialdata \leftarrow \bot$
for $u = 1, \ldots, n_p$ **do**
  $P_u \leftarrow \bot$
  for $i = 1, \ldots, n_d$ **do**
    $s_{u,i} \leftarrow 0$
    $dpk_{u,i}, \pi_{u,i} \leftarrow DeviceSetup(1^n, (u,i))$
    $\pi_{u,i}, pubprekeys_u \leftarrow Register(id_u, id_i)$
    $initialdata += u, pubprekeys_u, dpk_{u,i}$
$b' \leftarrow \mathcal{A}^{Oracles, Challenge}(initialdata)$
return $tested \neq \bot \wedge Fresh(tested) \wedge b = b'$

OInitSession($role$, u, i, v)

$s_{u,i} \leftarrow s_{u,i} + 1, s \leftarrow s_{u,i}$
$C_{init}, c_{out}, \pi^s_{u,i}[0] \leftarrow InitSession(role, \pi_v.pubprekeys, \pi_{u,i}, s)$
$P_u \leftarrow P_u \cup C_{init}$
$\pi_{u,i}.Sessions \leftarrow s, \pi^s_{u,i}.\texttt{step} \leftarrow 0$
$V^s_{u,i} \leftarrow \emptyset$
$E^s_{u,i} \leftarrow 0,\ F^s_{u,i} \leftarrow 0,\ G^s_{u,i} \leftarrow 0,\ A^s_{u,i} \leftarrow \{0\}$
return c_{out}, C_{init}

OReceiveInitSession(C_{init}, u, j)

$s_{u,j} \leftarrow s_{u,j} + 1, r \leftarrow s_{u,j}$
$\pi^r_{u,j} \leftarrow ReceiveInitSession(C_{init}, \pi_{u,j}, r)$
$\pi_{u,j}.sessions \leftarrow r,\ \pi^r_{u,j}.\texttt{step} \leftarrow 0$
$V^r_{u,j} \leftarrow \emptyset,\ A^r_{u,j} \leftarrow \{0\}$
$E^r_{u,j} \leftarrow 0,\ F^r_{u,j} \leftarrow 0,\ G^r_{u,j} \leftarrow 0$
return

OSend(m, u, i, s)

$t \leftarrow \pi^s_{u,i}.\texttt{step}$
$C_{in}, c_{out}, \pi^s_{u,i}[t+1] \leftarrow Send(m, \pi^s_{u,i}[t])$
$V^s_{u,i} \leftarrow V^s_{u,i} \cup C_{in},\ E^s_{u,i} \leftarrow t+1$
for $T \geq t+1$ **do**
  $\pi^s_{u,i}[T].\texttt{revDevRand} \leftarrow false$
if $\pi^s_{u,i}[t].\texttt{active} = false$ **then**
  for $T \geq t+1$ **do**
    $\pi^s_{u,i}[T].\texttt{active} \leftarrow true$
if $\pi^s_{u,i}[t+1].rand \neq \pi^s_{u,i}[t].rand$ **then**
  $F^s_{u,i} \leftarrow t+1, G^s_{u,i} \leftarrow t+1$
  for $T \geq t+1$ **do**
    $ResetRand - State - Session(u, i, s, T)$
$\pi^s_{u,i}.\texttt{step} \leftarrow t+1$
return C_{in}, c_{out}

OReceiveIn(C, u, j, r)

$t \leftarrow \pi^r_{u,j}.\texttt{step}$
Req. $NoReveal - NoInactiveCorrupt(u, j, r, t)$
  $\vee\ C \in V^r_{u,j}$
  $\vee\ \exists (u, k, o, t_o)$ partnered with (u, j, r, t)
  such that $C \in V^o_{u,k}$
$m, \pi^r_{u,j}[t+1] \leftarrow ReceiveIn(c, \pi^r_{u,j}[t])$
if $\pi^r_{u,j}[t+1].rand \neq \pi^r_{u,j}[t].rand$ **then**
  $F^r_{u,j} \leftarrow t+1,\ G^r_{u,j} \leftarrow t+1$
  for $T \geq t+1$ **do**
    $ResetRand - State - Session(u, j, r, T)$
$\pi^r_{u,j}.\texttt{step} \leftarrow t+1$
return m

OReceiveOut(c, v, ℓ, p)

$t \leftarrow \pi^p_{v,\ell}.\texttt{step}$
$m, \pi^p_{v,\ell}[t+1] \leftarrow ReceiveOut(c, \pi^p_{v,\ell}[t])$
if $\pi^p_{v,\ell}[t+1].state \neq \pi^p_{v,\ell}.state$ **then**
  $G^p_{v,\ell} \leftarrow t+1$
  for $T \geq t+1$ **do** $ResetState - Session(v, j, r, T)$
$\pi^p_{v,\ell}.\texttt{step} \leftarrow t+1$
return m

OAdd&Join(j, u, i)

$\{C_{join,s}, C_{add,s}, c_{out,s}\}_{s \in \pi_{u,i}.sessions}, \pi_{u,i} \leftarrow Add\&Join(pk_j, \pi_{u,i})$
for $s \in \pi_{u,i}.Sessions$ **do**
  $t_s \leftarrow \pi^s_{u,i}.\texttt{step}$
  $P_u \leftarrow P_u \cup \{C_{join,s}\}, V^s_{u,i} \leftarrow V^s_{u,i} \cup \{C_{add,s}\}$
  $F^s_{u,i} \leftarrow t_s + 1,\ G^s_{u,i} \leftarrow t_s + 1$
  $A^s_{u,i} \leftarrow A^s_{u,i} \cup \{t_s + 1\}$
  $\pi^s_{u,i}.\texttt{step} \leftarrow t_s + 1$
  for $T \geq t_s + 1$ **do**
    $ResetRand - State - Session(u, i, s, T)$
return $\{C_{join,s}, C_{add,s}, c_{out,s}\}_{s \in \pi_{u,i}.Sessions}$

ODecAdd($\{C_o\}_{o \in [1,O]}$, u, k)

Req. $O = \#\pi_{u,k}.Sessions$
for $1 \leq o \leq O$ **do**
  $t_o \leftarrow \pi^o_{u,k}.step$
  Req. $NoReveal - NoInactiveCorrupt(u, k, o, t_o)$
    $\vee\ C_o \in V^o_{u,k}$
    $\vee\ \exists (u, i, s, t_s)$ partnered with (u, k, o, t_o)
    such that $C_o \in V^s_{u,i}$
$\pi_{u,k} \leftarrow DecAdd(\{C_o\}_{o \in [1,O]}, \pi_{u,k})$
for $1 \leq o \leq O$ **do**
  for $T \geq t_o + 1$ **do**
    $ResetRand - State - Session(u, k, o, T)$
  $F^o_{u,k} \leftarrow t_o + 1,\ G^o_{u,k} \leftarrow t_o + 1$
  $A^o_{u,k} \leftarrow A^o_{u,k} \cup \{t_o\}$
  $\pi^o_{u,k}.step \leftarrow t_o + 1$
return

ODecJoin($\{C_r\}_{r \in [1,R]}$, u, j)

for $1 \leq r \leq R$ **do** Req. $C_r \in P_u$
$\pi_{u,j} \leftarrow DecJoin(\{C_r\}_{r \in [1,R]}, \pi_{u,j}, r)$
$s_{u,j} \leftarrow R$
for $1 \leq r \leq R$ **do**
  $E^r_{u,j} \leftarrow 0,\ F^r_{u,j} \leftarrow 0,\ G^r_{u,j} \leftarrow 0, V^r_{u,j} \leftarrow \bot$
  $A^r_{u,j} \leftarrow \{0\},\ \pi^r_{u,j}.step \leftarrow 0$
return

ORevoke(dpk, u, i)

$\{C_{rev,s}, c_{out,s}\}_{s \in \pi_{u,i}.Sessions}, \pi_{u,i}, pubprekeys \leftarrow Revoke(dpk, \pi_{u,i})$
for $s \in \pi_{u,i}.Sessions$ **do**
  $t_s \leftarrow \pi^s_{u,i}.\texttt{step}$
  $V^s_{u,i} \leftarrow V^s_{u,i} \cup \{C_{rev,s}\}$
  $F^s_{u,i} \leftarrow t_s + 1,\ G^s_{u,i} \leftarrow t_s + 1,\ A^s_{u,i} \leftarrow \emptyset$
  $\pi^s_{u,i}.\texttt{step} \leftarrow t_s + 1$
  for $T \geq t_s + 1$ **do**
    $ResetRand - State - Session(u, i, s, T)$
$u.\texttt{corruptUser} \leftarrow false$
$u.\texttt{corruptOpt} \leftarrow false$
return $\{C_{rev,s}, c_{out,s}\}_{s \in \pi_{u,i}.Sessions}, pubprekeys$

ODecRevoke($\{C_o\}_{o \in [1,O]}$, u, k)

Req. $O = \#\pi_{u,k}.Sessions$
for $1 \leq o \leq O$ **do**
  $t_o \leftarrow \pi^o_{u,k}.\texttt{step}$
  Req. $NoReveal - NoInactiveCorrupt(u, k, o, t_o)$
    $\vee\ C_o \in V^o_{u,k}$
    $\vee\ \exists (u, i, s, t_s)$ partnered with (u, k, o, t_o)
    such that $C_o \in V^s_{u,i}$
$\pi_{u,k} \leftarrow DecRevoke(\{C_o\}_{o \in [1,O]}, \pi_{u,k})$
for $1 \leq o \leq\ O$ **do**
  for $T \geq t_o + 1$ **do**
    $ResetRand - State - Session(u, k, o, T)$
  $F^o_{u,k} \leftarrow t_o + 1,\ A^o_{u,k} \leftarrow \emptyset$
  $\pi^o_{u,k}.\texttt{step} \leftarrow t_o + 1$
return

RevealDevRand(u, i, s)

$t \leftarrow \pi^s_{u,i}.step$
for $T \geq E^s_{u,i}$ **do**
  $\pi^s_{u,i}[T].\texttt{revDevRand} \leftarrow true$
  if $\neg\pi^s_{u,i}[t].\texttt{active}$ **then**
    $\pi_{u,i}.\texttt{corruptDevice} \leftarrow true$
return $\pi^s_{u,i}[t].devrand$

RevealSessionKey(u, i, s)

$t \leftarrow \pi^s_{u,i}.step$
for $T \geq t$ **do**
  $\pi^s_{u,i}[T].\texttt{revSessionKey} \leftarrow true$
return $\pi^s_{u,i}[t].sessionkey$

RevealState(u, i, s)

$t \leftarrow \pi^s_{u,i}.step$
for $T \geq G^s_{u,i}$ **do**
  $\pi^s_{u,i}[T].\texttt{revState} \leftarrow true$
return $\pi^s_{u,i}[t].state$

RevealRandom(u, i, s)

for $T \geq F^s_{u,i}$ **do**
  $\pi^s_{u,i}[T].\texttt{revRand} \leftarrow true$
return $\pi^s_{u,i}[t].Random$

CorruptDevice(u, i)

$\pi_{u,i}.\texttt{corruptDevice} \leftarrow \texttt{true}$
return $\pi_{u,i}.dsk_{u,i}$

CorruptUser(u)

$u.\texttt{corruptUser} \leftarrow \texttt{true}$
return $\pi_u.usk_u$

CorruptOpt(u)

$u.\texttt{corruptOpt} \leftarrow true$
return $\pi_u.mtsk_u$

Challenge(u, i, s, t)

$tested \leftarrow (u, i, s, t)$
if $b = 0$ $k \leftarrow \pi^s_{u,i}[t].sessionkey$
else $k \leftarrow\!\!\$\ \mathcal{K}$
return k

Fig. 9: Multi-Device Ratcheted Key Exchange Indistinguishability experiment.

DevLeakPrekeys(u, i, s, I).
CorruptOpt(u, I) = u.`corruptOpt` within the period I or $\exists$ (u, i, s) such that DevLeakPrekeys(u, i, s, I).

For a session (u, i, s), let I_u identify the interval that starts with the last revocation on u's side before (u, i, s) is initialized and ends with the first revocation on u's side after (u, i, s) is initialized. If no revocation occurs after initialization, I_u ends with the experiment. If no revocation had occurred before the initialization, I_u starts with the experiment. The same way, define $I_{\bar{u}}$ as the interval that starts with the last revocation on $\pi^s_{u,i}.peer$'s side before (u, i, s) is initialized and ends with the first revocation on $\pi^s_{u,i}.peer$'s side after (u, i, s) is initialized.

Definition 13. *(Initiation freshness.) Consider a session (u, i, s). A session $\pi^s_{u,i}$ such that $\pi^s_{u,i}.role$ = initiator has a fresh initiation if:*
- $\neg$CorruptUser$(u, I_u) \vee \neg$CorruptOpt$(\pi^s_{u,i}.peer, I_{\bar{u}})$ *or*
- $\neg\pi^s_{u,i}[0]$.`revRandom` $\vee \neg$CorruptOpt$(\pi^s_{u,i}.peer, I_{\bar{u}})$ *or*
- $\neg\pi^s_{u,i}[0]$.`revRandom` $\vee \neg$CorruptUser$(\pi^s_{u,i}.peer, I_{\bar{u}})$.

A session $\pi^s_{u,i}$ such that $\pi^s_{u,i}.role$ = responder has a fresh initiation if:
- $\neg$CorruptUser$(\pi^s_{u,i}.peer, I_{\bar{u}}) \vee \neg$CorruptOpt$(u, I_u)$ *or*
- $\neg\pi^o_{v,k}[0]$.`revRandom` *for all* $(v, k, o, 0)$ *matching* $(u, i, s, 0)$ *if it exists* $\vee \neg$CorruptOpt$(\mathrm{u}, \mathrm{I_u})$ *or*
- $\neg\pi^o_{v,k}[0]$.`revRandom` *for all* $(v, k, o, 0)$ *matching* $(u, i, s, 0)$ *if it exists* $\vee \neg$CorruptUser$(\mathrm{u}, \mathrm{I_u})$.

Freshness of the following steps Here we consider the same restrictions as in [9] but we extend them relatively to partnered sessions, multiple matching sessions and device leakage for data that are transmitted between the devices (randomness and state data when a device is added).

Definition 14. *(Freshness at some step.) A session (u, i, s, t), $t > 0$, is fresh if:*
- $\neg(\pi_{u,i}$.`corruptDevice` $\wedge \neg\pi^s_{u,i}[t]$.`active`$)$ *and*
- $\neg$RevealSessionKey*(u,i,s,t) and*
- $\neg$RevealState*$(u,i,s,t-1)$* *or* $\neg$RevealRandom(u, i, s, t).

RevealSessionKey(u, i, s, t) stands for:
- $\pi^s_{u,i}[t]$.`revSessionKey`
- or $\exists$ (u,j,r,t_r) partnered with (u,i,s,t) such that $\pi^r_{u,j}[t_r]$.`revSessionKey`
- or $\exists$ (v,ℓ,o,t_o) matching (u,i,s,t) such that $\pi^o_{v,\ell}[t_o]$.`revSessionKey`.

RevealState(u, i, s, t) stands for:
- $\pi^s_{u,i}[t]$.`revState`
- or $\exists$ (u,j,r,t_r) partnered with (u,i,s,t) such that $\pi^r_{u,j}[t_r]$.`revState`
- or $\exists$ (v,ℓ,o,t_o) matching (u,i,s,t) such that $\pi^o_{v,\ell}[t_o]$.`revState`
- or $t \in A^s_{u,i}$ and DeviceLeak(u, i, s, t)
- or $\exists$ (v,k,o,t_o) matching (u,i,s,t) such that $t_o \in A^o_{v,k}$ and DeviceLeak(v, k, o, t_o)

RevealRandom(u, i, s, t) stands for:

· $\pi^s_{u,i}[t]$.revRandom
· or $\exists$ (u,j,r,t_r) partnering (u,i,s,t) such that $\pi^r_{u,j}[t_r]$.revRandom
· or DeviceLeak(u,i,s,t).

Definition. *(Secure Multi-Device Instant Messaging.) A MDIMexecuted with n_p users, each having n_d devices is said to be secure in the above model if it is correct and for all adversary $\mathcal{A}$ running in polynomial time, the following advantage is negligible:*

$$\mathsf{Adv}^{\mathsf{MDIM-IND}}_{\mathcal{A},MDIM,n_p,n_d}(n) = \left|\Pr\left[\mathsf{Exp}^{\mathsf{MDIM-IND}}_{\mathcal{A},MDIM,n_p,n_d}(n)\right] - \frac{1}{2}\right|.$$

C.2 *MDIM* construction

Our construction is detailed in Figure 11.
Based on [9], we consider the following description for the Signal protocol:
- $\mathsf{Sig.KeyGen}(1^n) \to pk, sk$.
- $\mathsf{Sig.MedTermKeyGen}(1^n) \to ephpk, ephsk$.
- $\mathsf{Sig.Activate}(role, ephpk, \pi^s_{u,i}) \to c_{out}, \pi^s_{u,i}$. Computes the initial shared secret, the first rootkey rk and the first chainkey ck. Returns a message c_{out} and an updated state $\pi^s_{u,i}$.
- $\mathsf{Sig.Send}(m, \pi^s_{u,i}) \to c_{out}, \pi^s_{u,i}$ and $\mathsf{Sig.Receive}(c, \pi^s_{u,i}) \to m$, $\pi^s_{u,i}$. Two probabilistic algorithms that take as input a session state $\pi^s_{u,i}$ and either a message m or a ciphertext c and return an updated state $\pi^s_{u,i}$ and either a ciphertext c or a message m. The original run algorithm also takes as input ephemeral and user keys, they are accessible from our session state.

We formalize the registering procedure as follow:
- $\mathsf{Sig.Register}(uID, \pi_{u,i}, pprekeys) \to \pi_{u,i}$. On input a user ID, a device state and a set of prekeys $prekeys$, registers on the server and updates the device state $\pi_{u,i}$ with Signal data.

We detail in Figure 10 below the ExtraRatchet and Update procedures. The first is needed to ensure confidentiality of conversation before adding a device or after revoking one. The second enables devices that receive a ratchet secret through the RDM channel (in a ReceiveIn), to maintain their Signal state up-to-date.

About addition and revocation Our Add&Join sends *sprekeys* and *Devices* to everybody (the newcomer as the already enrolled devices). This last point is done on purpose to be sure a newcomer cannot receive session specific information without receiving global ones. Sending these data only with the RDM.Join would lead to a more complicated model for the RDM and we choose to keep the joining action unrelated to the shipping of encrypted messages.
As all Signal and RDM procedures take as an entry the device state and update it, we simplify (except for initiation and key generation procedures) $\pi^s_{u,i}, y \leftarrow \mathsf{Proc}(\pi^s_{u,i}, x)$ as $y \leftarrow \mathsf{Proc}(x)$.

Update($\pi^s_{u,i}, rchsk$)

1 : **if** $rchsk \neq \pi^s_{u,i}.rand$
2 : $\quad E \leftarrow DH(rchsk, \pi^s_{u,i}.rand_{peer})$
3 : $\quad RK, CK \leftarrow KDF\_RK(\pi^s_{u,i}.RK, E)$
4 : $\quad \pi_{u,i}s.RK \leftarrow RK, \pi^s_{u,i}.CK \leftarrow CK$
5 : $CK, MK \leftarrow\leftarrow KDF\_CK(CK)$
6 : $\pi^s_{u,i}.CK \leftarrow CK, \pi^s_{u,i}.MK \leftarrow MK$
7 : **return** $\pi^s_{u,i}$

ExtraRatchet($\pi^s_{u,i}$)

1 : $rchsk, rchpk \leftarrow DHKeyGen(1^n)$
2 : $E \leftarrow DH(rchsk, \pi^s_{u,i}.rand_{peer})$
3 : $RK, CK \leftarrow KDF\_RK(\pi^s_{u,i}.RK, E)$
4 : $\pi^s_{u,i}.RK \leftarrow RK, \pi^s_{u,i}.CK \leftarrow CK$
5 : $\pi^s_{u,i}.rand \leftarrow rchsk$
6 : **return** $rchpk, \pi^s_{u,i}$

Fig. 10: The `ExtraRatchet` and Update procedures.

C.3 Proof of Theorem 2

We recall Theorem 2 and provide a detailed proof of this result.

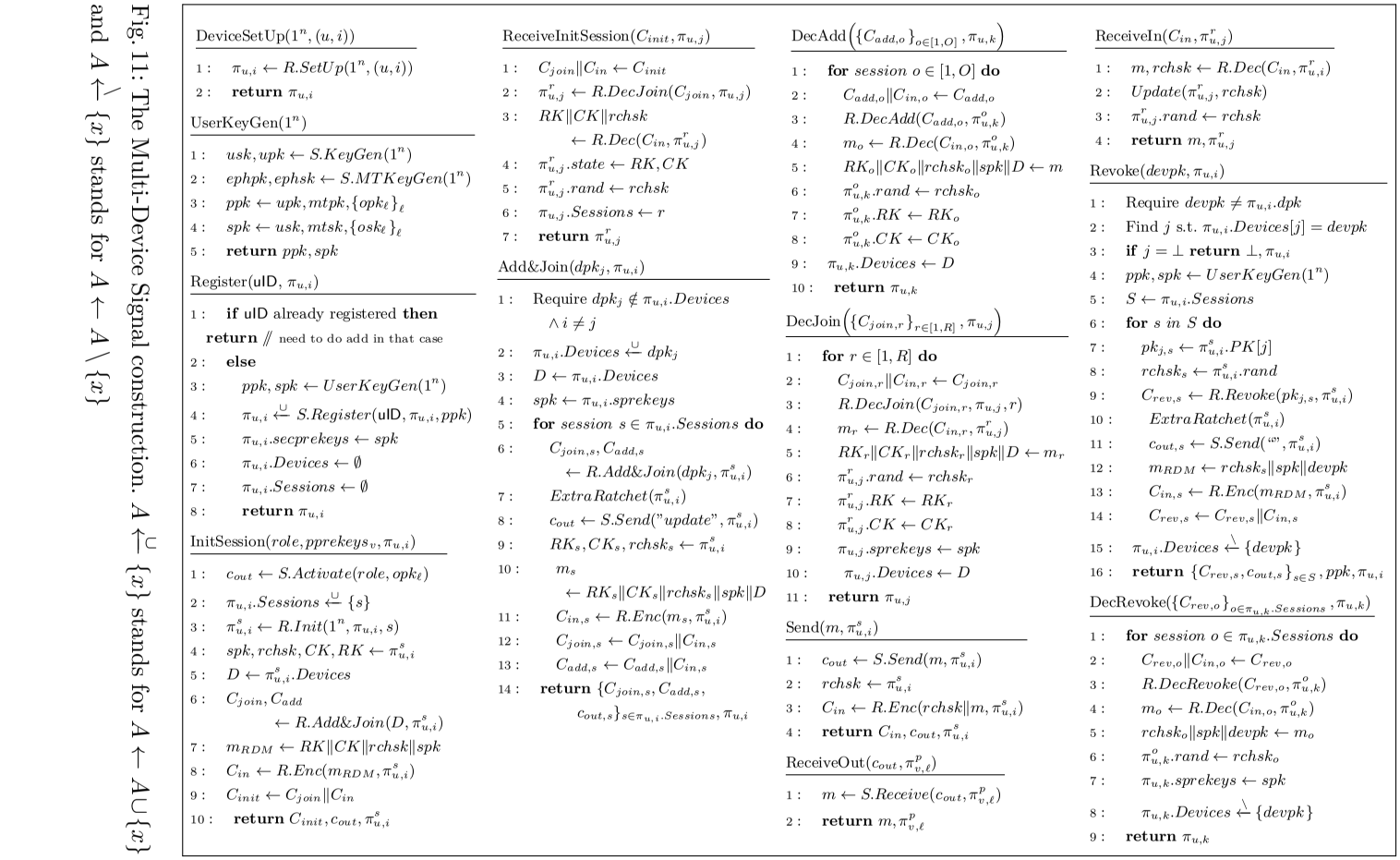
Theorem. *Let Signal be a secure multi-stage key-exchange protocol with advantage ϵ_{sig} and RDM a $\mathsf{RDM-IND}$ secure ratcheted dynamic multicast with advantage $\epsilon_{\mathsf{RDM-IND}}$, the above construction is a secure MDIM such that, for any PPT adversary running n_s sessions from n_d devices of n_p users, making at most q queries to the oracles:*

$$\mathsf{Adv}^{\mathsf{MDIM-IND}}_{\mathcal{A}}(n) \leq n_p^2 \cdot (2 \cdot \epsilon_{\mathsf{RDM-IND}} + \epsilon_{sig}).$$

MDIM correctness

Proof. Correctness of the partnering. Suppose (u,i,s,t_s) and (u,j,r,t_r) are partnered sessions, with (u,j,r) being alive for longer than (u,i,s) $(t_r > t_s)$. That means $\pi^r_{u,j}[t_r].\mathtt{sid_1} \tilde{=} sid' \| \pi^s_{u,i}[t_s].\mathtt{sid1}$. By definition of $\mathtt{sid_1}$, the two sessions are matching in the sense of ratcheted multicast. By the construction we have that $\pi^s_{u,i}[t_s].\mathtt{sid_1}[0]{=}C_{join}$ where C_{join} is obtained from $\mathsf{Add\&Join}(pk_i, \pi^o_{u,k})$ for some session (u,k,o). By the partnering definition, two cases are possible as C_{join} is intended to (u,i,s): either $\pi^r_{u,j}[t_r].\mathtt{sid_1} = sid' \| (C_{join}, C_{add}) \| \cdots$, (meaning that $(u,k,o){=}(u,j,r)$) or $\pi^r_{u,j}[t_r].\mathtt{sid_1} = sid' \| C_{add} \| \cdots$ (meaning that $(u,k,o) \neq (u,j,r)$) and both sessions share the same RK and CK, as they are encrypted in the ciphertext C_{in} included in both C_{join} and C_{add} ciphertexts of the *Add&Join* algorithm. From this step, both sessions are included in the group of the multicast and by correctness of the multicast, all messages exchanged via the multicast canal are the same for (u,i,s) and (u,j,r): they have access to the same successive *randsk*. We now consider the chain of session from (u,i,s,t_s) to an initial session $(u,i_{init},s_{init},t_{init})$. We can see that the same chain links (u,j,r,t_r) to the same initial session. By construction, (u,j,r,t_r) and (u,i,s,t_s) will receive the same Signal message in ReceiveOut hence the same public randomness from the conversation peer. Since they have common root key and then common secret and public ratchet randomness they compute the same *sessionkey*.

Proof. Correctness of the matching. Suppose (u,i,s,t_s) and (v,ℓ,p,t_p) are matching sessions. Consider the chains of sessions from (u,i,s) to an initial session



DeviceSetUp$(1^n,(u,i))$

$\pi_{u,i} \leftarrow R.SetUp(1^n,(u,i))$

return $\pi_{u,i}$

UserKeyGen(1^n)

$usk, upk \leftarrow S.KeyGen(1^n)$

$ephpk, ephsk \leftarrow S.MTKeyGen(1^n)$

$ppk \leftarrow upk, mtpk, \{opk_\ell\}_\ell$

$spk \leftarrow usk, mtsk, \{osk_\ell\}_\ell$

return ppk, spk

Register(uID, $\pi_{u,i}$)

if uID already registered **then**

return // need to do add in that case

else

$ppk, spk \leftarrow UserKeyGen(1^n)$

$\pi_{u,i} \xleftarrow{\cup} S.Register(\mathsf{uID}, \pi_{u,i}, ppk)$

$\pi_{u,i}.secprekeys \leftarrow spk$

$\pi_{u,i}.Devices \leftarrow \emptyset$

$\pi_{u,i}.Sessions \leftarrow \emptyset$

return $\pi_{u,i}$

InitSession$(role, pprekeys_v, \pi_{u,i})$

$c_{out} \leftarrow S.Activate(role, opk_\ell)$

$\pi_{u,i}.Sessions \xleftarrow{\cup} \{s\}$

$\pi^s_{u,i} \leftarrow R.Init(1^n, \pi_{u,i}, s)$

$spk, rchsk, CK, RK \leftarrow \pi^s_{u,i}$

$D \leftarrow \pi^s_{u,i}.Devices$

$C_{join}, C_{add} \leftarrow R.Add\&Join(D, \pi^s_{u,i})$

$m_{RDM} \leftarrow RK\|CK\|rchsk\|spk$

$C_{in} \leftarrow R.Enc(m_{RDM}, \pi^s_{u,i})$

$C_{init} \leftarrow C_{join}\|C_{in}$

return $C_{init}, c_{out}, \pi^s_{u,i}$

ReceiveInitSession$(C_{init}, \pi_{u,j})$

$C_{join}\|C_{in} \leftarrow C_{init}$

$\pi^r_{u,j} \leftarrow R.DecJoin(C_{join}, \pi_{u,j})$

$RK\|CK\|rchsk \leftarrow R.Dec(C_{in}, \pi^r_{u,j})$

$\pi^r_{u,j}.state \leftarrow RK, CK$

$\pi^r_{u,j}.rand \leftarrow rchsk$

$\pi_{u,j}.Sessions \leftarrow r$

return $\pi^r_{u,j}$

Add&Join$(dpk_j, \pi_{u,i})$

Require $dpk_j \notin \pi_{u,i}.Devices \wedge i \neq j$

$\pi_{u,i}.Devices \xleftarrow{\cup} dpk_j$

$D \leftarrow \pi_{u,i}.Devices$

$spk \leftarrow \pi_{u,i}.sprekeys$

for *session* $s \in \pi_{u,i}.Sessions$ **do**

$C_{join,s}, C_{add,s} \leftarrow R.Add\&Join(dpk_j, \pi^s_{u,i})$

$ExtraRatchet(\pi^s_{u,i})$

$c_{out} \leftarrow S.Send(\text{"update"}, \pi^s_{u,i})$

$RK_s, CK_s, rchsk_s \leftarrow \pi^s_{u,i}$

$m_s \leftarrow RK_s\|CK_s\|rchsk_s\|spk\|D$

$C_{in,s} \leftarrow R.Enc(m_s, \pi^s_{u,i})$

$C_{join,s} \leftarrow C_{join,s}\|C_{in,s}$

$C_{add,s} \leftarrow C_{add,s}\|C_{in,s}$

return $\{C_{join,s}, C_{add,s}, c_{out,s}\}_{s\in\pi_{u,i}.Sessions}, \pi_{u,i}$

DecAdd$\left(\{C_{add,o}\}_{o\in[1,O]}, \pi_{u,k}\right)$

for *session* $o \in [1,O]$ **do**

$C_{add,o}\|C_{in,o} \leftarrow C_{add,o}$

$R.DecAdd(C_{add,o}, \pi^o_{u,k})$

$m_o \leftarrow R.Dec(C_{in,o}, \pi^o_{u,k})$

$RK_o\|CK_o\|rchsk_o\|spk\|D \leftarrow m$

$\pi^o_{u,k}.rand \leftarrow rchsk_o$

$\pi^o_{u,k}.RK \leftarrow RK_o$

$\pi^o_{u,k}.CK \leftarrow CK_o$

$\pi_{u,k}.Devices \leftarrow D$

return $\pi_{u,k}$

DecJoin$\left(\{C_{join,r}\}_{r\in[1,R]}, \pi_{u,j}\right)$

for $r \in [1,R]$ **do**

$C_{join,r}\|C_{in,r} \leftarrow C_{join,r}$

$R.DecJoin(C_{join,r}, \pi_{u,j}, r)$

$m_r \leftarrow R.Dec(C_{in,r}, \pi^r_{u,j})$

$RK_r\|CK_r\|rchsk_r\|spk\|D \leftarrow m_r$

$\pi^r_{u,j}.rand \leftarrow rchsk_r$

$\pi^r_{u,j}.RK \leftarrow RK_r$

$\pi^r_{u,j}.CK \leftarrow CK_r$

$\pi_{u,j}.sprekeys \leftarrow spk$

$\pi_{u,j}.Devices \leftarrow D$

return $\pi_{u,j}$

Send$(m, \pi^s_{u,i})$

$c_{out} \leftarrow S.Send(m, \pi^s_{u,i})$

$rchsk \leftarrow \pi^s_{u,i}$

$C_{in} \leftarrow R.Enc(rchsk\|m, \pi^s_{u,i})$

return $C_{in}, c_{out}, \pi^s_{u,i}$

ReceiveOut$(c_{out}, \pi^p_{v,\ell})$

$m \leftarrow S.Receive(c_{out}, \pi^p_{v,\ell})$

return $m, \pi^p_{v,\ell}$

ReceiveIn$(C_{in}, \pi^r_{u,j})$

$m, rchsk \leftarrow R.Dec(C_{in}, \pi^r_{u,i})$

$Update(\pi^r_{u,j}, rchsk)$

$\pi^r_{u,j}.rand \leftarrow rchsk$

return $m, \pi^r_{u,j}$

Revoke$(devpk, \pi_{u,i})$

Require $devpk \neq \pi_{u,i}.dpk$

Find j s.t. $\pi_{u,i}.Devices[j] = devpk$

if $j = \perp$ **return** $\perp, \pi_{u,i}$

$ppk, spk \leftarrow UserKeyGen(1^n)$

$S \leftarrow \pi_{u,i}.Sessions$

for s *in* S **do**

$pk_{j,s} \leftarrow \pi^s_{u,i}.PK[j]$

$rchsk_s \leftarrow \pi^s_{u,i}.rand$

$C_{rev,s} \leftarrow R.Revoke(pk_{j,s}, \pi^s_{u,i})$

$ExtraRatchet(\pi^s_{u,i})$

$c_{out,s} \leftarrow S.Send(\text{“}\text{”}, \pi^s_{u,i})$

$m_{RDM} \leftarrow rchsk_s\|spk\|devpk$

$C_{in,s} \leftarrow R.Enc(m_{RDM}, \pi^s_{u,i})$

$C_{rev,s} \leftarrow C_{rev,s}\|C_{in,s}$

$\pi_{u,i}.Devices \xleftarrow{\setminus} \{devpk\}$

return $\{C_{rev,s}, c_{out,s}\}_{s\in S}, ppk, \pi_{u,i}$

DecRevoke$(\{C_{rev,o}\}_{o\in\pi_{u,k}.Sessions}, \pi_{u,k})$

for *session* $o \in \pi_{u,k}.Sessions$ **do**

$C_{rev,o}\|C_{in,o} \leftarrow C_{rev,o}$

$R.DecRevoke(C_{rev,o}, \pi^o_{u,k})$

$m_o \leftarrow R.Dec(C_{in,o}, \pi^o_{u,k})$

$rchsk_o\|spk\|devpk \leftarrow m_o$

$\pi^o_{u,k}.rand \leftarrow rchsk_o$

$\pi_{u,k}.sprekeys \leftarrow spk$

$\pi_{u,k}.Devices \xleftarrow{\setminus} \{devpk\}$

return $\pi_{u,k}$

Fig. 11: The Multi-Device Signal construction. $A \xleftarrow{\cup} \{x\}$ stands for $A \leftarrow A \cup \{x\}$ and $A \xleftarrow{\setminus} \{x\}$ stands for $A \leftarrow A \setminus \{x\}$

$(u,i_0,\ s_0)$ (chain U) and from (v,ℓ,p) to the initial session (v,ℓ_0,p_0) (chain V). Those two chains exist according to the definition 7. Now we consider an abstract super device S_u that is present from the initiation step of session (u, i_0, s_0), and until the step t_s of session (u, i, s). Its state is limited to RK, $randsk$ and $\mathtt{sid}_2$. The state of this super device takes successively the values of the state of the different sessions composing the chain U. This is possible without conflict, because, when two partners are present at the same time, they share the same RK, $randsk$ and $\mathtt{sid}_2$. (For $\mathtt{sid}_2$ this is by definition, for the others, by correctness of partnering). $S_u.\mathtt{sid}_2$ contains all Signal messages received and sent from the initialization to the present step t_r of (u, j, r). We consider a similar super device S_v that aggregates state information along the V chain. The recursive definition of the matching ensures those two users are matching in terms of Signal transcripts at each moment, including the initialization step. The correctness of the Signal protocol provides that S_u and S_v share the same session key at each moment. As $S_u.sessionkey$ (respectively $S_v.sessionkey$) is defined as $\pi^s_{u,i}.sessionkey$ (resp. $\pi^s_{u,i}.sessionkey$) when session (u,i,s) (resp. (v,j,r)) is alive, we obtain that (u,j,i,t_s) and (v,j,r,t_r) share the same $sessionkey$.

MDIM indistinguishability

Proof. For all games G_x, we denote S_x the event $\mathcal{A}$ wins in G_x.

Game 0 is the original $\mathsf{MDIM-IND}$ Game as described in 9.

In **Game 1**, we guess which pair of users (initiator/receiver) (u, v) will be targeted by the adversary in order to apply only on them the $RDM-IND$ security. We have n_p users.

$$\mathsf{Adv}^{\mathrm{G}_0}_{\mathcal{A}}(n) \leq n_p^2 \cdot \mathsf{Adv}^{\mathrm{G}_1}_{\mathcal{A}}(n).$$

In **Game 2**, we will collapse all devices of each of these two users into a super single user. First, as the $\mathsf{MDIM-IND}$ security game respects the same restrictions as in RDM ind game for the RDM part, RDM security of the multicast ensures none of the information sent through the RDM canal leaks with security loss bounded by the $\mathsf{RDM-IND}$ factor. Second, correctness of the RDM ensures us at each step all "alive" devices share same secrets and transcripts for signal part and that the device that sent the signal message is present in the partnering pool. With this consideration, one can think of the pool of devices of one user as a single signal superdevice.

$$|\Pr[S_2] - \Pr[S_1]| \leq 2 \cdot \epsilon_{\mathsf{RDM-IND}}.$$

What we have to show is that these superdevices execute the Signal protocol as a classical device. What is different from the original Signal design? There are additions and revocations of devices. Once the multicast communication are "erased" (swallowed by superdevice), there remains additional asymmetric signal ratchets (not at initial stage) and prekey bundle updates.

About the asymmetric ratchets. Several ratchets in a row on Alice's side for instance means that Bob's ratchet random will be used several times. In Signal proof [9] we are in [asym-ir or ri] case. There are two options, either freshness relays on the root key, then randoms are not concerned and we are still in the model. Either freshness relays on randoms. In our model, freshness flag on a random is maintained from step to step: if it is corrupted at a step t, it will remain corrupted for the followings steps. So assumptions on the freshness on the root key or on the random still holds. If the tested session is an initiator type session and that there has been several [asym-ir] type step before challenge, there is no problem. We will receive values (X, Y) from the GDH Challenger and replace last initiator public random by X, and last responder public random by Y as much time as needed (the number of ratchet step there has been on the initiator's side). We knows the successive initiator's ratchet secret and can evaluate the root key as needed. If test occurs on responder's side after several ratchet on responder side it is the same. Responder's public key eventually has been used many steps before but we can simulate perfectly the corresponding steps (not concerned with the DDH challenge) as we know initiator's secret in this case. Additional ratchets do not alter Signal's model with GDH assumption.

About the additional keys. The devices keys are only concerned with the multicast. The main difference is that the entire Signal prekey bundle can be refreshed with the revoke algorithm. This refresh should correspond to an unregistering then registering again with the difference here that already ongoing sessions are still alive. Indeed, these ongoing sessions continue to be used as an asymmetric ratchet has already been performed during the revoke algorithm but the Signal prekey bundle related to these sessions is not used anymore (due to the ratchets). For future new session, not initiated yet, the new Signal prekey bundle is used and it can be viewed as a Signal prekey bundle for a new user. In others terms, it is like if each superdevice were composed of many independant Signal users. In Game 2, we are in the traditional Signal protocol execution. In **Game 3**, in the Challenge, if b=0, we replace the session key by a random. Signal security ensures we can do this substitution and we have:
$|\Pr[S_3] - \Pr[S_2]| = \epsilon_{MS-IND}$. Finally, in Game 3, $\mathcal{A}$ has no advantage since the returned key is always random and we obtain:

$$\mathsf{Adv}_{\mathcal{A}}^{\mathrm{G}_0}(n) \leq n_p^2 \cdot (2 \cdot \epsilon_{\mathsf{RDM-IND}} + \epsilon_{sig}).$$

D Implementation

We implement our solution over the Signal library libsignal-protocol-java accessible on https://github.com/signalapp/libsignal-protocol-java. This is the one library considered in [9]. We build our test in the experimental InMemory version of the Library. This version does not use a physical server but simulates the transport layout. We use the JCE and the BouncyCastle libraries for cryptography services. We implemented our RDM with $ECIES$ on curve $secp256r1$

and AES with 128 bits keys for hybrid encryption, and $HMAC\_SHA256$ for the MAC scheme. We play a scenario where Alice uses three devices and Bob one. We run n iterations of the following: Alice sends a message from a random device, Bob and Alice other devices decrypt, Bob answers, all devices of Alice decrypt. Finally, to compare our implementation, we run a similar scenario with Alice devices represented by 3 different users in the Sesame solution. Figure 12 presents the results in term of time and number of connection (each time a message is sent or received) for a run of 1 000 exchanges. We run our test on a 2,9 GHz IntelCore i7 with 16 Go of LPDDR3 RAM memory at 2 133 MHz. Our code is accessible on `https://github.com/multidevicerke`. In Appendix D, we detail how we locally patched the original library.

	time (ms)	number of connections	data weight (bits)
our solution	20 690	8 000	2 186 000
Sesame	3 007	12 000	990 000

Fig. 12: Results for a run of $n = 1\ 000$ exchanges.

The Sesame solution is quicker which can be explained by the use of asymmetric encryption in the RDM scheme. However, the difference can be minimized since asymmetric like computation in the Sesame version (Diffie Hellman asymmetric ratchet for all channels) are done using the native elliptic curve library available in the original library, whereas we employ an external BouncyCastle Library for the encryption computations. Finally, our solution requires one-third less connections. This corresponds to Alice sending only one message for Bob and one for all her other devices instead of one message for each device. As a connection is an irreducible time-consuming operation, this gain is not negligible. Considering the amount of exchanged data, we have a ratio of 273 bits/connection which is largely acceptable.

Patch on the Signal Library. We add a $\mathsf{half-ratchet}$ method that corresponds to our $\mathsf{ExtraRatchet}$ procedure described in Section 3. In the original code, as soon as Bob receives a reply, he performs two asymmetric step. He computes the ratchet with Alice new randomness and obtains the root key, chain key pair: (RK_r, CK_r) also computed by Alice. Then Bob immediately prepares his keys for his reply. He generates his new randomness and computes a new pair (RK_s, CK_s). RK_s becomes Bob's current root key. CK_s defines Bob's future sending chain. The positive of this solution is that the sending procedure does not have to consider whether to perform an asymmetric ratchet or not. The negative is that Bob stores his future secret keys before it is necessary, which downgrades the future secrecy property. In our solution, Alice can send another ratchet. She uses her current root key, which is equal to RK_r. When Bob tries to computes the ratchet from his current root key RK_s, he fails. We separate

the two ratchet step: the receiver chain is updated when receiving a message, and the sending chain is updated, if necessary, before sending a message. We add a `RatchetCounter` in the Signal state to deal with whether or not perform a ratchet in the Encrypt procedure.