SAS-based authentication for secure messaging

Sébastien Hauri

School of Computer and Communication Sciences

Master Semester Project

June 2022

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

Supervisor
Daniel Collins
EPFL / LASEC

Contents

1 Introduction

Primitives such as ratcheted key exchange have gained a lot of popularity in recent years. This is mainly due to the fact that they provide strong security guarantees. They can achieve *forward secrecy* and *post-compromise* security to ensure *confidentiality*. A main example is the Signal protocol which provides end-to-end encryption in secure messaging. It is one of the the most popular protocols used in the wild. We find it implemented in the eponymous application, WhatsApp, Facebook Messenger Secret Conversations and a lot more. Inspired by this protocol, researchers proposed formalism and new protocols to achieve these security guarantees in different forms [BSJ+17].

On another side, there has not been a lot of development in the domain of entity authentication. The Signal app uses so called "safety numbers" [Mar16] to solve this problem. Basically, the two parties involved produce some fingerprint based on a mix of their long-term secrets that they encode in a QR code[1]. They then compare those code they each produced out-of-band. This has been proven to not be very secure since it relies only on long-term secrets, which can be exposed or stolen. Thus any key derived after the initial key exchange cannot be authenticated properly. [DH20] proposed a modified version to make it more robust. It authenticates the asymmetric keying material and allows for out-of-band detection of an active adversary each time the direction of communication changes.

It makes sense to consider short authenticated strings (SAS) [Vau05] here since an amount of data would have to be compared out-of-band, and we would intuitively like this amount be as short as possible. Short messages can be used in situations where QR codes can't, can reduce user mistakes and lead to time savings, e.g. over the phone. A short authenticated string is, as in its name a short string, used to authenticate a message with the help of an authenticated channel. Two SAS-based message authentication protocols were first introduced by [Vau05], based on commitment schemes. They have then been refined in [PV06]. The main primitives are *mutual message authentication* and *message cross-authentication*. The big difference between the two is that one is used to authenticate a commonly shared message and the other to authenticate two[2] messages.

In this work, we first begin by explaining Signal's key exchange, double ratchet algorithm and its out-of-band authentication safety numbers protocol, followed by the modified version proposed by [DH20]. Later, we in-

[1]There are in fact two encodings: a QR code and a numeric code. The former is a QR encoding of the serialization of the fingerprint. The latter is based on a truncation of the numeric representation of the fingerprint.

[2]One per party.

troduce SAS-based cryptography and mutual message authentication from [PV06] and we explain how it could fit in our authentication scheme. Finally, we define our a modification of DH's proposed messaging scheme based on the previous primitives. We will present the building blocks of the scheme and also give a construction. We will finish by giving some intuition on why it is secure in the security model we propose and a way to prove it.

2 Signal

We give in this section a brief explanation on Signal's key exchange, along with its double ratchet algorithm and its authentication mechanism. We list below all types of keys we can find in the protocol. All public keys have some corresponding private keys: we simplify the description by focusing only on public keys here as in [MP16].

- *Identity keys (IK):* those are long-term DH key pairs. They are used to sign and derive other DH keys.
- *Signed prekeys (SPK)*: medium-term DH key pairs. They are signed with the identity key.
- *One-time prekeys (OPK)*: ephemeral DH key pairs. They are used in a single X3DH protocol run.
- *Ephemeral keys (EK)*: ephemeral DH key pair used when deriving a shared key between two participants.
- *Ratchet keys (RCK)*: ephemeral DH key pair used each time the conversation changes sense in the asymmetric ratchet.
- *Root keys (RK)*: symmetric secret value, used as input to a KDF to derive new root and chain keys.
- *Chain key (CK)*: symmetric secret value, used as input to a KDF to get new encryption keys.
- *Message (Encryption) keys (MK)*: symmetric secret keys used to encrypt messages.

Signal uses some defined functions $KDF(\cdot)$ which denotes some key derivation function, $KDF(K, \cdot)$ which relates to some keyed key derivation function and $DH(X, Y)$ which is a Diffie-Hellman computation between the public key Y and the private key related to X.

2.1 The X3DH key agreement protocol

The Extended Triple Diffie-Hellman (X3DH) protocol [MP16] aims at establishing a shared secret key between two parties (*Alice* and *Bob*). One of the main advantage of such a protocol is that it allows the key exchange (and thus the conversation) to begin with only one party being online. When a user (we take here Bob) registers in Signal, the following key material is generated (client-side):

- IK_B, Bob's identity key
- SPK_B, Bob's signed prekey
- $\{OPK_B^i\}_{i=1,\dots,n}$, a set of n one-time prekeys
- $Sig(IK_B, SPK_B)$, the signature of Bob's signed prekey

All of this is packed in what is called a "prekey bundle" and is published on a Signal's server. The medium and short-term key will eventually be deleted and/or replaced by new key material for *forward security*. When Alice wants to start a conversation with Bob (we assume here Alice has already generated her key material), she fetches Bob's prekey bundle from the server, verifies the signature of Bob's signed prekey, chooses one of Bob's one-time prekey, generates a new ephemeral and ratchet keys EK_A, RCK_A and derives a shared secret in the following way [MP16]:

$$\begin{aligned}
DH_1 &= DH(IK_A, SPK_B)\\
DH_2 &= DH(EK_A, IK_B)\\
DH_3 &= DH(EK_A, SPK_B)\\
DH_4 &= DH(EK_A, OPK_B)\\
SK &= KDF(DH_1||DH_2||DH_3||DH_4)
\end{aligned}$$

She then uses this shared secret to create the first root, chain and message keys:

$$\begin{aligned}
RK^0, CK_0^0 &= KDF(SK, DH(RCK_A, SPK_B))\\
CK_1^0, MK_1^0 &= KDF(CK_0^0)
\end{aligned}$$

This marks the end of the X3DH key agreement protocol. A simplified scheme is depicted in figure 1. Alice can then send an initial message to Bob, encrypted with MK_1^0 and associated data RCK_A plus other identification material in order to Bob getting the same secret.

2.2 The double ratchet algorithm

The asymmetric ratchet is used to enforce *future secrecy* or post-compromise security. Each time a new conversation begins or each time the direction

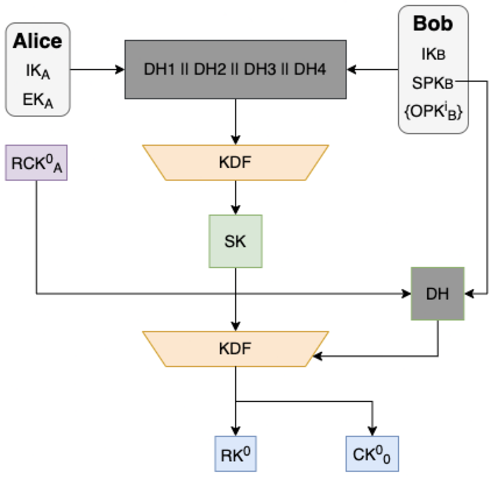

Figure 1: *The (simplified) X3DH protocol*

of messaging changes, a new iteration of this protocol is applied[3]. Each party generates a fresh ratchet key pair. *"Every message from either party begins with a header which contains the sender's current ratchet public key"* [PM16]. Those new ratchet keys are used along with previous ratchet and root keys to derive new root and chain keys in the following way:

$$RK^i, CK_0^i \leftarrow KDF(RK^{i-1}, DH(RCK^i, RCK^{i-1}))$$

This new chain key is used to derive new message keys:

$$CK_1^i, MK_1^i \leftarrow KDF(CK_0^i)$$

This is what is called the symmetric ratchet. We depict this key schedule in figure 2. We don't elaborate on the symmetric ratchet since it is out of the scope of this work.

2.3 Entity authentication mechanism

The Signal app comes with a user mediated entity authentication mechanism called "safety numbers". The software produces fingerprints based on long-term keying material who can take form of either a QR code or a numeric code. We only present here the QR code representation since it is the mostly used in practice and also that the numeric one can be computed in a quite similar manner. Also note that since Signal does not provide any documentation on this entity authentication mechanism, this description is

[3]We denote by epoch a time when the direction of the communication changes. The first epoch is epoch 0.

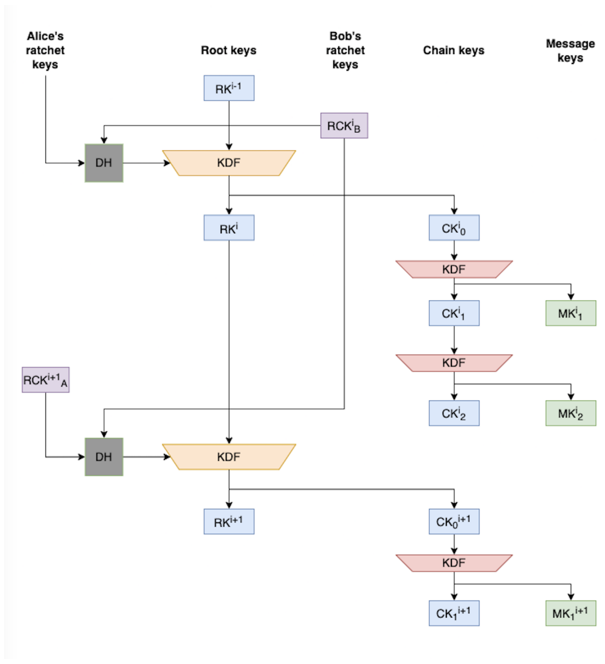

Figure 2: *Key schedule for the Signal double ratchet algorithm. It is adapted from [DH20] with the notation of [PM16] used in this work.*

taken from [DH20]. The advantage of using scannable fingerprints is that it effectively removes the risk of human error.
The QR code is built in the following way (following the perspective of A):

$$\begin{aligned} \texttt{local\_fprint} &= H_i(0||\texttt{fvers}||{IK_A}^{4}||ID_A, IK_A) \\ \texttt{remote\_fprint} &= H_i(0||\texttt{fvers}||IK_B||ID_B, IK_B) \end{aligned}$$

where $H_i(x,y)$ is an iterative hash function such that $H_0 = H(x)$, $H_i = H(H_{i-1}, y)$, and $\texttt{fvers} = 0$. In practice, H is the SHA2 function with 512 output bits with an iteration number of 5200. The resulting hashes are then truncated to 240 bits. This impacts directly the security regarding collisions between hashes and it is thus lowered. The QR code is then the representation of $\{\texttt{svers}^{5}, 0, \texttt{local\_fprint}, \texttt{remote\_fprint}\}$ serialised.

[4] ID is the output of a function $IDGen$ which takes as inout the private key associated to IK

[5] svers corresponds to the scannable fingerprint version number. There are two versions (namely 0 and 1) and version 1 is the latest.

3 Modified Signal key exchange and authentication

In this section, we explain the work of [DH20]. We state how the current Signal's so-called *Safety Numbers* could be manipulated to trick entity authentication. We will also explain their modified key-schedule and see why it could fit our model.

As Signal has made great advancements towards post-compromise security and forward secrecy for *confidentiality*, we can find some flaws in the engineering of their entity authentication mechanism. The main problem pointed out is that the "safety numbers" only considers long-term secret keys IK. Thus, it does not contain any information about the state of the session. Assume that an adversary manages to expose one of the party's states after initialization. Then, he will be able to inject messages using ephemeral keys alone in the conversation and impersonate the participant without being detected by the protocol.

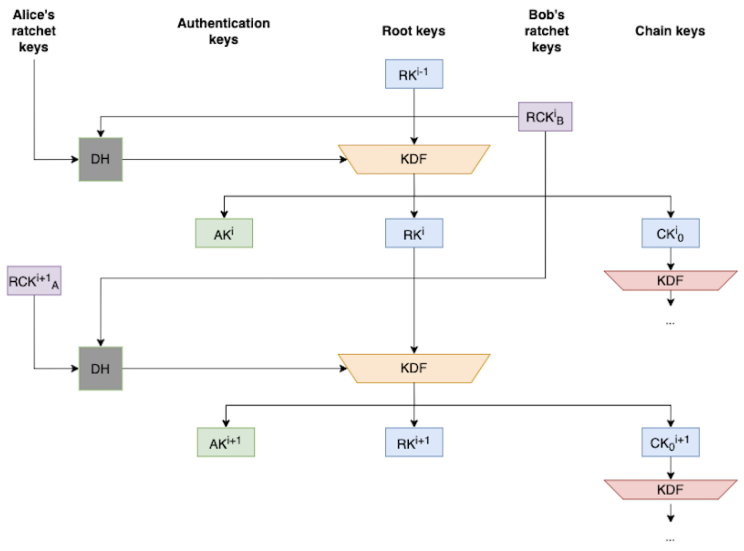

Figure 3: *The modified key schedule described in [DH20] with modified notations.*

To cope with this problem, [DH20] came with a new key schedule that contains an authentication key (AK) used as input key to an HMAC function of the fingerprints. This way, the QR code is always bound to the session state and authenticates it. One intuition about why authenticating the entire conversation is better than just the initial keying material, even if we do it only once, is that more messages will be authenticated.

4 SAS-based authentication

Authenticated key-agreement has been a widely researched area since the beginning of public key cryptography. We have seen the rise of the use of certificates in the public key infrastructure model in protocols such as TLS. This model is based on a trusted third party. On another side, we have seen protocols such as PGP who relies on a web-of-trust model. The SAS-based authentication came with the idea of authenticating the output of a key agreement protocol with the use of a commitment scheme which can then be verified using an authenticate channel (e.g. in person, by recognising voice over the phone). In [PV06], the authors propose two protocols: (i) a mutual-authentication protocol where two parties authenticate a common value, and (ii) a cross-authentication protocol where the value is different for each of the two user. We will only focus on the former as it is the one used in this work.

$$\begin{array}{ccc}
\textbf{Alice's input}: m_A & & \textbf{Bob's input}: m_B \\
\\
\text{Pick } R_A \in \{0,1\}^k & & \\
(c, d) \leftarrow Commit(m_A, R_A) & \xrightarrow{\quad c \quad} & \text{Pick } R_B \in \{0,1\}^k \\
 & \xleftarrow{\quad R_B \quad} & \\
 & \xrightarrow{\quad d \quad} & \hat{R}_A \leftarrow Open(m_B, \hat{c}, \hat{d}) \\
SAS \leftarrow R_A \oplus \hat{R}_B & \xrightarrow{\text{authenticate}_A(SAS)} & SAS \overset{?}{=} \hat{R}_A \oplus R_B \\
\text{Check } SAS \text{ is the same} & \xleftarrow{\text{authenticate}_B(SAS)} &
\end{array}$$

Figure 4: *The SAS-based message mutual authentication protocol [PV06]*

The SAS-based message mutual authentication protocol is, as previously stated, based on a commitment scheme. Assume the two parties of the protocol are Alice and Bob and that the values to be authenticate are respectively m_A and m_B, the output of some key agreement protocol (without an attack on this protocol we would have $m_A = m_B$). Alice picks a random number and makes a commitment on this value using m_A. Bob picks a random value and sends it back to Alice. Finally, Alice sends the decommit value in order to Bob opening the random value chosen by Alice. All of those communications are made on the insecure channel. They then both compute some value based on the random values they exchanged and compare it out-of-band (i.e. on an authenticated channel). This short out-of-band channel is the main key of the protocol. It allows the authentication and integrity of the short computed strings and thus authenticates the messages. We depict the protocol in figure 4. The security of this protocol relies on the fact that an active adversary has only one shot to equivocate the commitment of Alice in each protocol run. This is mainly due to the fact that an adversary cannot precompute anything since he is not aware of Alice's (and thus Bob's) input of the protocol. In other words, the adversary has to choose R_B without knowing R_A in the first place. As he cannot guess those values, offline attacks fail (assuming in this case that the commitment scheme is secure).

5 The ARC scheme

We describe in this section how the modified Signal key schedule mixed with a SAS-based authentication protocol could be a solution to achieve entity authentication. Suppose the key schedule of figure 3 is used. The authentication key is supposed to be shared between the two parties. We can take advantage of this as input to a SAS-based message mutual authentication run. Mainly the idea is to start new instances of the SAS authentication protocol each epoch. . To do so, at each epoch i, the sending party would send a new commitment using value AK^i, a random challenge value R^{i-1} used for the authentication of epoch $i-1$ and the decommit value d^{i-2} to authenticate epoch $i-2$ [6]. In this system, if current epoch is i, one can authenticate all epochs j such that $1 < j < i-1$. A series of messages should look like described in figure 5.

This way, the authentication does not rely on only long-term secrets and we can reduce the amount of computations done by each device. On what could be an implementation, as the SAS should be short enough, we could output it as a small numeric value, e.g. 6 digits, and use that representation in a similar manner as the safety numbers implemented by Signal. Note that a QR representation loses its sense due to the shortness of the SAS.

We hereafter define formally that new model. We first define a syntax and a security model for it. The scheme will then combine different primitives that are presented before describing the scheme itself.

5.1 Syntax and security

Definition 1 (Authenticated ratcheted communication) *An* authenticated ratcheted communication *scheme (ARC) is composed of the following PPT algorithms:*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$ *takes a unary string* 1^λ *and outputs public parameters* pp.
- $\mathsf{Gen}(\mathsf{pp}) \to (\mathsf{pk}, \mathsf{sk})$ *takes public parameters* pp *and outputs a public/secret key pair* $(\mathsf{pk}, \mathsf{sk})$.

[6] Note that this behaviour cannot work for the first and second epoch of the communication. Thus the values with a negative superscript should (in fact cannot since they don't exist) be sent.

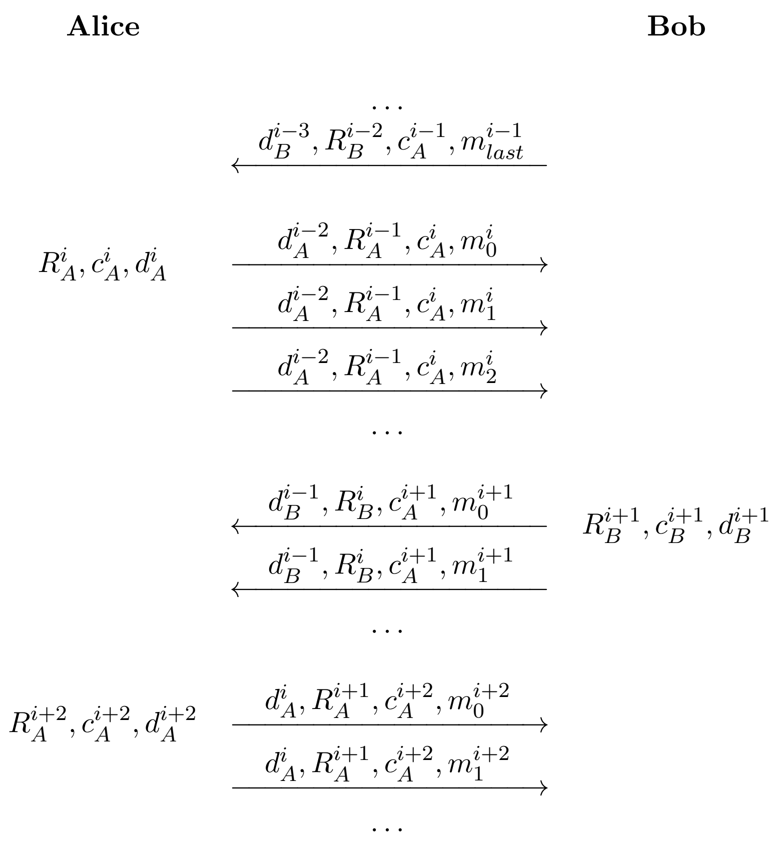
Alice
Bob
$d_B^{i-3}, R_B^{i-2}, c_A^{i-1}, m_{last}^{i-1}$
R_A^i, c_A^i, d_A^i
$d_A^{i-2}, R_A^{i-1}, c_A^i, m_0^i$
$d_A^{i-2}, R_A^{i-1}, c_A^i, m_1^i$
$d_A^{i-2}, R_A^{i-1}, c_A^i, m_2^i$
$d_B^{i-1}, R_B^i, c_A^{i+1}, m_0^{i+1}$
$R_B^{i+1}, c_B^{i+1}, d_B^{i+1}$
$d_B^{i-1}, R_B^i, c_A^{i+1}, m_1^{i+1}$
$R_A^{i+2}, c_A^{i+2}, d_A^{i+2}$
$d_A^i, R_A^{i+1}, c_A^{i+2}, m_0^{i+2}$
$d_A^i, R_A^{i+1}, c_A^{i+2}, m_1^{i+2}$

Figure 5: *How the direction changes through the discussion. In this example, when Bob receives one of the m^{i+2} messages, he can compute the SAS of epoch i with d_A^i and the random number he sent in the previous round. Note that with the receiving of R_A^{i+1} he can already compute the SAS of epoch $i+1$.*

- $\mathsf{Init}(\mathsf{pp}, \mathsf{sk}_\mathcal{P}, \mathsf{pk}_{\overline{\mathcal{P}}}, \mathcal{P}) \rightarrow \mathsf{st}_\mathcal{P}$ *for* $\mathcal{P} \in \{\mathsf{A}, \mathsf{B}\}$ *takes public parameters* pp*,* $\mathcal{P}$*'s secret key* $\mathsf{sk}_\mathcal{P}$*, partner* $\overline{\mathcal{P}}$*'s public key* $\mathsf{pk}_{\overline{\mathcal{P}}}$ *and party identifier* $\mathcal{P}$ *and outputs a state* $\mathsf{st}_\mathcal{P}$ *for* $\mathcal{P}$*.*

- $\mathsf{Send}(\mathsf{st}_\mathcal{P}, \mathsf{ad}, \mathsf{pt}) \rightarrow (\mathsf{st}'_\mathcal{P}, \mathsf{ct})$ *takes a state* $\mathsf{st}_\mathcal{P}$*, associated data* ad *and a plaintext* pt *and outputs a new state* $\mathsf{st}'_\mathcal{P}$ *and ciphertext* ct.

- $\mathsf{Receive}(\mathsf{st}_\mathcal{P}, \mathsf{ad}, \mathsf{ct}) \rightarrow (\mathsf{acc}, \mathsf{st}'_\mathcal{P}, \mathsf{pt}, \mathsf{t}, \mathsf{i})$ *takes a state* $\mathsf{st}_\mathcal{P}$*, associated data* ad *and ciphertext* ct *and outputs an acceptance bit* $\mathsf{acc} \in \{\mathsf{true}, \mathsf{false}\}$*, state* $\mathsf{st}'_\mathcal{P}$*, plaintext* pt*, epoch number* t *and message number* i.

- $\mathsf{AuthSend}(\mathsf{st}_\mathcal{P}) \rightarrow (\mathsf{at}, \mathsf{st}'_\mathcal{P})$ *takes a state* $\mathsf{st}_\mathcal{P}$ *and outputs an authetication tag* at *(or* $\perp$ *if it failed) and a new state* $\mathsf{st}'_\mathcal{P}$*.*

- $\mathsf{AuthReceive}(\mathsf{st}_\mathcal{P}, \mathsf{at}) \rightarrow (\mathsf{auth}, \mathsf{st}'_\mathcal{P})$ *takes a state* $\mathsf{st}_\mathcal{P}$ *and an authen-*

tication tag at *and outputs a bit* auth *(0 if authentication failed, 1 otherwise) and a new state* $\mathsf{st}'_{\mathcal{P}}$*.*

As done by CDV in [CDV21], an additional $\mathsf{InitAll}(\mathsf{pp}) \to (\mathsf{st}_\mathsf{A}, \mathsf{st}_\mathsf{B}, z)$ *algorithm returning the initial states of parties* A *and* B *together with public information* z *is defined as follows:*

```
InitAll(pp)
1 :  $(\mathsf{pk_A}, \mathsf{sk_A}) \leftarrow \mathsf{Gen}(\mathsf{pp})$
2 :  $(\mathsf{pk_B}, \mathsf{sk_B}) \leftarrow \mathsf{Gen}(\mathsf{pp})$
3 :  $\mathsf{st_A} \leftarrow \mathsf{Init}(\mathsf{pp}, \mathsf{sk_A}, \mathsf{pk_B}, \mathsf{A})$
4 :  $\mathsf{st_B} \leftarrow \mathsf{Init}(\mathsf{pp}, \mathsf{sk_B}, \mathsf{pk_A}, \mathsf{B})$
5 :  $z \leftarrow (\mathsf{pp}, \mathsf{pk_A}, \mathsf{pk_B})$
6 :  return $(\mathsf{st_A}, \mathsf{st_B}, z)$
```

To complete that definition, we need to formalize some security notions for the scheme.

The Signal security. As said before, we defined the Signal security from [ACD19]. As our scheme is based on theirs, with some little changes in the notations and syntax, it maintains the security guarantees that they define.

Auth security. The Auth game formalises the security of the entity authentication mechanism. The adversary is given access to the following oracles :

- $\mathsf{OSend}(\mathcal{P}, \mathsf{ad}, \mathsf{pt})$ which sends a message pt with associated data ad from a party $\mathcal{P} \in \{\mathsf{A}, \mathsf{B}\}$,
- $\mathsf{ORecv}(\mathcal{P}, \mathsf{ad}, \mathsf{ct})$ which receives a ciphertext ct with associated data ad at $\mathcal{P}$,
- $\mathsf{OAuthSend}$ and $\mathsf{OAuthRecv}$ which make parties use the authentication channel,
- $\mathsf{OExp}(\mathcal{P})$ which exposes the state of party $\mathcal{P}$ and
- $\mathsf{OExpPT}(\mathcal{P}, \mathsf{t}, \mathsf{i})$ which exposes the i-th plaintext-ciphertext pair of epoch t.

The goal of the adversary is to make $\mathsf{OAuthRecv}$ output true when it is not supposed to. A ciphertext can be abstracted with 4 different parts: the asymmetric, the SAS, the symmetric and the message part. The adversary must forge a message with a new asymmetric part, and the authentication of

```
$\mathsf{Auth}^{\mathcal{A}}_{\mathsf{ARC}}()$ :
1 : $\mathsf{st_A}, \mathsf{st_B}, \mathsf{z} \leftarrow \mathsf{InitAll}(\mathsf{pp})$
2 : $\mathsf{sent_A}, \mathsf{sent_B} \leftarrow \{\}$
3 : $\mathsf{authsent_A}, \mathsf{authsent_B} \leftarrow \{\}$
4 : $\mathsf{exp}, \mathsf{frgd} \leftarrow \{\}$
5 : $\mathsf{win} \leftarrow 0$
6 : $\mathsf{t_A}, \mathsf{i_A}, \mathsf{t_B}, \mathsf{i_B} \leftarrow 0$
7 : $\mathcal{A}^{\mathsf{OSend},\ldots,\mathsf{OExp}} \rightarrow \perp$
8 : return $\mathsf{win} = 1$
```

```
$\mathsf{OSend}(\mathcal{P}, \mathsf{ad}, \mathsf{pt})$ :
1 : if ($\mathcal{P} = \mathsf{A}$ and $\mathsf{t_A}$ is even)
2 :   or ($\mathcal{P} = \mathsf{B}$ and $\mathsf{t_B}$ is odd) then
3 :     $\mathsf{t}_\mathcal{P} \leftarrow \mathsf{t}_\mathcal{P} + 1$
4 :     $\mathsf{i}_\mathcal{P} \leftarrow 0$
5 : $\mathsf{st}_\mathcal{P}, \mathsf{ct} \leftarrow \mathsf{Send}(\mathsf{st}_\mathcal{P}, \mathsf{ad}, \mathsf{pt})$
6 : $\mathsf{sent}_\mathcal{P}[(\mathsf{t}_\mathcal{P}, \mathsf{i}_\mathcal{P})] \leftarrow (\mathsf{pt}, \mathsf{ct})$
7 : $\mathsf{i}_\mathcal{P} \leftarrow \mathsf{i}_\mathcal{P} + 1$
8 : return $\mathsf{ct}$
```

```
$\mathsf{ORecv}(\mathcal{P}, \mathsf{ad}, \mathsf{ct})$ :
1 : $\mathsf{acc}, \mathsf{st}_\mathcal{P}, \mathsf{pt}, \mathsf{t}, \mathsf{i} \leftarrow \mathsf{Receive}(\mathsf{st}_\mathcal{P}, \mathsf{ad}, \mathsf{ct})$
2 : if $\mathsf{acc} = \mathsf{true}$ then
3 :     $\mathsf{t}_\mathcal{P} \leftarrow max(\mathsf{t}_\mathcal{P}, \mathsf{t})$
4 :     if $\mathsf{sent}[(\mathsf{t}, \mathsf{j})] = \perp, \forall \mathsf{j}$ then
5 :         $\mathsf{frgd}[\mathsf{t}] \leftarrow \mathsf{true}$
6 :         return $(\mathsf{acc}, \mathsf{pt})$
7 :     $(asym, sas, sym, mess) \leftarrow \mathsf{ct}$
8 :     for $\mathsf{j}$ s.t. $\mathsf{sent}[(\mathsf{t}, \mathsf{j})] \neq \perp$ do
9 :         $(\mathsf{pt}', \mathsf{ct}') \leftarrow \mathsf{sent}[(\mathsf{t}, \mathsf{j})]$
10 :        $(asym', \ldots) \leftarrow \mathsf{ct}'$
11 :        if $asym \neq asym'$ then
12 :            $\mathsf{frgd}[\mathsf{t}] \leftarrow \mathsf{true}$
13 :            break
14 : return $(\mathsf{acc}, \mathsf{pt})$
```

```
$\mathsf{OExp}(\mathcal{P})$ :
1 : $\mathsf{exp}[\mathsf{t}_\mathcal{P}] \leftarrow \mathcal{P}$
2 : return $\mathsf{st}_\mathcal{P}$
```

```
$\mathsf{OExpPT}(\mathcal{P}, \mathsf{t}, \mathsf{i})$
1 : return $\mathsf{sent}_\mathcal{P}[(\mathsf{t}, \mathsf{i})]$
```

```
$\mathsf{OAuthSend}(\mathcal{P})$ :
1 : $\mathsf{at}, \mathsf{st}_\mathcal{P} \leftarrow \mathsf{AuthSend}(\mathsf{st}_\mathcal{P})$
2 : $\mathsf{authsent}_\mathcal{P}[\mathsf{t}_\mathcal{P}] \leftarrow \mathsf{at}$
3 : return $\mathsf{t}_\mathcal{P}$
```

```
$\mathsf{OAuthRecv}(\mathcal{P}, \mathsf{i})$ :
1 : $\mathsf{at} \leftarrow \mathsf{authsent}_{\overline{\mathcal{P}}}[\mathsf{i}]$
2 : $\mathsf{auth}, \mathsf{st}_\mathcal{P} \leftarrow \mathsf{AuthReceive}(\mathsf{st}_\mathcal{P}, \mathsf{at})$
3 : if $\mathsf{exp}[\mathsf{i} - 2] \neq \mathcal{P}$ and $\mathsf{exp}[\mathsf{i} - 2] \neq \overline{\mathcal{P}}$ then
4 :     if $\mathsf{auth}$ and $\exists \mathsf{j} \leq \mathsf{i} - 2$ s.t. $\mathsf{frgd}[j] = \mathsf{true}$ then
5 :         $\mathsf{win} \leftarrow 1$
6 : return $\mathsf{auth}$
```

Figure 6: *The* Auth *security game and its oracles*

this "forged" epoch should pass. To check that this asymmetric part of the message has changed, we check all the previously sent messages, split them into different parts and compare those. The way forgeries are recorded is

simple: it is a map from an epoch number t to $\mathsf{b} \in \{\mathsf{true}, \mathsf{false}\}$. If there has been a forgery at epoch i, then there will a true at $\mathsf{frgd}[i]$. Another point is that, intuitively, the adversary cannot expose both states of A and B. Exposures are handled in the same way as forgeries but the output domain is $\{\mathsf{A}, \mathsf{B}\}$ to denote which party has been exposed. Also when the state of a party $\mathcal{P}$ is exposed, we do not provide authentication for the epochs that he can trivially win in. This is captured by recording the epochs A and B are in along with the last epoch where there has been an exposure. Variables $sent_{\mathcal{P}}$ keep track of messages sent by both parties in a map.
We then define the advantage of the adversary as:

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{auth}}(\lambda) = \Pr[\mathsf{Auth} \to \mathsf{true}]$$

5.2 Primitive definitions

We now describe all the primitives we need in order to build our *authenticated ratcheted communication scheme*: initial key agreement (IKA), continuous key-agreement (CKA), forward-secure authenticated encryption with associated data (FS-AEAD), PRF-PRNGs, two-output pseudo-random function (PRF) and SAS-based entity authentication (SAS-EA).

Definition 2 (CKA, FS-AEAD) *We define CKA and FS-AEAD in a same sense as what is described in section 4 of [ACD19].*

A continuous key agreement *is an asynchronous protocol between two parties, composed of the following PPT algorithms:*

- $\mathsf{CKA\text{-}Init\text{-}A}(\mathsf{k}) \to \gamma^{\mathsf{A}}$ *which takes as input a shared key* k *and outputs an initial state* γ^{A}*.*
- $\mathsf{CKA\text{-}S}(\gamma) \to (\gamma', \mathsf{T}, \mathsf{I})$ *which takes as input a state* γ *and outputs a new state* γ'*, a message* T *and a key* I*.*
- $\mathsf{CKA\text{-}R}(\gamma) \to (\gamma', \mathsf{I})$ *which takes as input a state* γ *and outputs a new state* γ' *and a key* I*.*

A forward-secure authenticated encryption with associated data *scheme is a state-full primitive between two users* A *and* B*. It is composed of the following PPT algorithms and memory management functions:*

- $\mathsf{FS\text{-}Init\text{-}S}(\mathsf{k}) \to \mathsf{v_A}$ *(similarly* $\mathsf{FS\text{-}Init\text{-}R}$*) which takes a symmetric key* k *an outputs a state* $\mathsf{v_A}$*.*
- $\mathsf{FS\text{-}Send}(\mathsf{v}, \mathsf{a}, \mathsf{m}) \to (\mathsf{v}', \mathsf{e})$ *which takes as input as state* v*, associated data* a *and message* m *and outputs a new state* v' *and a ciphertext* e*.*

- $\mathsf{FS\text{-}Recv}(\mathsf{v}, \mathsf{a}, \mathsf{e}) \rightarrow (\mathsf{v}', \mathsf{i}, \mathsf{m})$ *which takes a state* v*, associated data* a *and ciphertext* e *and outputs new state* v'*, an index* i *and a message* m*.*

- $\mathsf{FS\text{-}Stop}(\mathsf{v}) \rightarrow \mathsf{n}$ *which given a state* v *outputs the number* n *of messages that have been received and then "erases" the* $\mathsf{FS\text{-}AEAD}$ *session corresponding to* v *from memory.*

- $\mathsf{FS\text{-}Max}(\mathsf{v}, \mathsf{n})$ *which given a state* v *and integer* n *stores* n *in memory and erases the corresponding state* v *as soon as* n *messages have been received.*

A PRF-PRNG *is a primitive which relates to both a* pseudo-random function *and a* pseudo-random number generator*. It is composed of the two following PPT algorithms:*

- $\mathsf{P\text{-}Init}(\mathsf{k}) \rightarrow \sigma$ *which takes a key* k *and outputs a state* σ*.*

- $\mathsf{P\text{-}Up}(\sigma, \mathsf{I}) \rightarrow (\sigma', \mathsf{R})$ *which takes as input a state* σ *and an input* I *and returns a new state* σ' *and an output* R*.*

In what follows for the scheme, the output R *will be a pair of keys which are statistically independent.*

For convenience of the reader, we recall here only the definitions of those primitives. The correctness and security notions of all of them can be found in section 4 of [ACD19].

To be more precise, our scheme takes into account the initial key agreement made by both the sender and the receiver. This primitive is defined hereafter.

Definition 3 (IKA) *An* initial key agreement *is a primitive used to derive a shared secret between two parties (A and B). As an example, the X3DH protocol described in figure 1 is an instance of an initial key agreement protocol. More formally, it is composed of the following PPT algorithm:*

- $\mathsf{IKA}(\mathsf{pp}, \mathsf{sk}_{\mathcal{P}}, \mathsf{pk}_{\overline{\mathcal{P}}}) \rightarrow \mathsf{k}$ *for* $\mathcal{P} \in \{\mathsf{A}, \mathsf{B}\}$ *which takes as input public parameters* pp*,* $\mathcal{P}$*'s secret key* $\mathsf{sk}_{\mathcal{P}}$ *and partner* $\overline{\mathcal{P}}$*'s public key* $\mathsf{sk}_{\overline{\mathcal{P}}}$ *and outputs a commonly shared key* k*.*

Note that this algorithm could use randomness.

We also need to define a way to "split" or expand a key in two statistically independent new keys. This can be implemented using a pseudo-random function that we define in what follows.

Definition 4 (PRF) *We define a two-output pseudo-random function as one of the form:*

$$f : \mathcal{K} \times \mathcal{I} \rightarrow \{0,1\}^\lambda \times \{0,1\}^\lambda$$
$$(\mathsf{k}, \mathsf{i}) \mapsto (\mathsf{k}_1, \mathsf{k}_2)$$

i.e. which takes as input a key $\mathsf{k} \in \mathcal{K}$ *and an input* $\mathsf{i} \in \mathcal{I}$ *and outputs two independent keys* $\mathsf{k}_1, \mathsf{k}_2 \in \{0,1\}^\lambda$.

The security of such a primitive can be formalized in the following game:

```
PRF_b :
1 : $k \leftarrow\$ \{0,1\}^\lambda$
2 : $b' \leftarrow \mathcal{A}^{\mathsf{OChal}}()$
3 : $map \leftarrow \{\}$
4 : return $b'$
```

```
OChal(i) :
1 : if $b = 0$
2 :     return $f(\mathsf{k}, \mathsf{i})$
3 : else
4 :     if $map[i] = \perp$ then
5 :         $(\mathsf{k}_1, \mathsf{k}_2) \leftarrow\$ \{0,1\}^\lambda \times \{0,1\}^\lambda$
6 :         $map[i] \leftarrow (\mathsf{k}_1, \mathsf{k}_2)$
7 :         return $(\mathsf{k}_1, \mathsf{k}_2)$
8 :     else
9 :         $(\mathsf{k}_1, \mathsf{k}_2) \leftarrow map[i]$
10 :        return $(\mathsf{k}_1, \mathsf{k}_2)$
```

We say that f is a secure pseudo-random function if for all λ, $\mathsf{Adv}^{\mathsf{f}}_{\mathcal{A},\mathsf{OChal}}(\lambda) = \Pr[\mathcal{A} \rightarrow 1 | b = 1] - \Pr[\mathcal{A} \rightarrow 1 | b = 0]$ is negligible where the probability is taken over the choice of random coins.

Finally we need to define our new entity authentication mechanism based on SAS cryptography. We first define the primitive and then explain how it works internally as this will be part of our new model of messaging scheme.

Definition 5 (SAS-EA) SAS-based entity authentication *is composed of the following PPT algorithms:*

- $\mathsf{SAS\text{-}EA\text{-}Init}(\mathsf{m}) \rightarrow \mathsf{w}$ *which takes a message* m *as input and outputs a state* w*,*
- $\mathsf{SAS\text{-}EA\text{-}Send}(\mathsf{m}) \rightarrow (\mathsf{w}, \mathsf{R})$ *which takes a message* m *and returns a state* w *and a new random number* R*,*
- $\mathsf{SAS\text{-}EA\text{-}Recv}(\mathsf{m}, \mathsf{drc}, \mathsf{w}^{-2}, \mathsf{w}^{-1}) \rightarrow \mathsf{w}'$ *which takes a message* m*, an input* drc *and two states* $\mathsf{w}^{-2}, \mathsf{w}^{-1}$ *and returns a new state* w'.

Note that both algorithms are not deterministic. The scheme is depicted in figure 7. A state is composed of the following variables :

- *a computed* SAS*, which will be used to authenticate a message,*
- *the shared message* m *that we want to authenticate,*
- *a random number* R *used to commit on the message,*
- *the committed an decommit values.*

```
SAS-EA-Init(m) :
1 : $\mathsf{SAS}, \mathsf{R}, \mathsf{c}, \mathsf{d} \leftarrow \perp$
2 : $\mathsf{w} \leftarrow (\mathsf{SAS}, \mathsf{m}, \mathsf{R}, \mathsf{c}, \mathsf{d})$
3 : return $\mathsf{w}$
```

```
SAS-EA-Send(m) :
1 : $\mathsf{R}_{new} \leftarrow\$ \{0,1\}^{\lambda}$
2 : $(\mathsf{c}_{new}, \mathsf{d}_{new}) \leftarrow \mathit{Commit}(\mathsf{m}, \mathsf{R}_{new})$
3 : $\mathsf{w} \leftarrow (\perp, \mathsf{m}, \mathsf{R}_{new}, \mathsf{c}_{new}, \mathsf{d}_{new})$
4 : $\mathsf{R}_{resp} \leftarrow\$ \{0,1\}^{\lambda}$
5 : return $(\mathsf{w}, \mathsf{R}_{resp})$
```

```
SAS-EA-Recv(m, drc, w^{-2}, w^{-1}) :
1 : $(\hat{d}^{-2}, \hat{R}^{-1}, \hat{c}) \leftarrow \mathsf{drc}$
2 : $\mathsf{w}^{-2}.d \leftarrow \hat{d}^{-2}$
3 : $\hat{R}^{-2} \leftarrow \mathit{Open}(\mathsf{w}^{-2}.m,\ \mathsf{w}^{-2}.c,\ \hat{d}^{-2})$
4 : $\mathsf{w}^{-2}.SAS \leftarrow \mathsf{w}^{-2}.R \oplus \hat{R}^{-2}$
5 : $\mathsf{w}^{-1}.SAS \leftarrow \mathsf{w}^{-1}.R \oplus \hat{R}^{-1}$
6 : $\mathsf{w} \leftarrow (\perp, \mathsf{m}, \perp, \hat{c}, \perp)$
7 : return $\mathsf{w}$
```

Figure 7: *The SAS-based authentication mechanism scheme*

The idea behind the algorithms is that at each round the sender computes new commit and decommit values on the message by sampling a random value in $\{0,1\}^{\lambda}$ *and using a* Commit *function, and storing all those values in a new state. It then samples the new random value that it will send in the current round, in response to the previous instance of the SAS protocol. The receiver can then compute, with the help of the current* drc *variable, the two previous SASs. Note that the states are updated in place in the reception procedure. As this primitive is used continuously, and as at each round the sender and receiver exchange role, at the end of each epoch, they should both share the same* $n-2$ *computed SAS* [7]*.*

We only give some intuition on why this mechanism is secure. By formalising and adapting some notations of the oracles defined in section 2.1 of [PV06], we can prove the same security guarantees.

[7]Assuming the current epoch is epoch n.

5.3 The scheme

In this section we formalise the *ARC* scheme of definition 1. The scheme is highly inspired by the one described in [ACD19] and de facto the Signal protocol. As explained before, the main difference with Signal (other than the ones introduced by [ACD19]) is that the scheme keeps tracks of an SAS-based entity authentication state which will be used to authenticate the communication. The scheme if depicted in figure 8. In order to ease readability, the states are not made explicit but they contain all the variables defined in the Init procedure. Also note that the Send and Receive algorithms describe the behaviour for $\mathcal{P} = \mathsf{A}$. We assume it is the first party sending some messages. To get the algorithms of B, one simply need to replace "even" by "odd" in the procedures. We explain hereafter the different procedures and the slight changes we have made.

Setting up parameters. The Setup procedure is used, as its name and definition say, to set up general public parameters used in the different cryptographic primitives. It is not made explicit since this depends highly on their implementations. For example in the X3DH protocol of Signal, one has to decide several public parameters such as an elliptic curve (typically the $X25519$ curve) or an hash function.

Generating the keying material. The Gen procedure is used to generate the keying material for both parties. It uses a previously generated public parameters to do so. This algorithm is also not made explicit since, as for setting up public parameters, it is really dependent of the cryptographic primitives. Typically one could think of generating the elliptic curve public/private key pairs used as input to the X3DH protocol.

Initialising a party. The Init procedure is used to handle the initialisation of the state of both parties. It makes explicit the initial key agreement part since our model of initialisation takes into account the secret and public keys of both sender and receiver. It also takes into account the initialisation of the SAS-based entity authentication mechanism. A new variable drc is added to the state of each party. This variable represents a triplet which contains the decommit value of the commitment made two epochs before, the random value needed to compute the previous SAS and the current commitment.

Sending messages. The Send procedure is used to send a message to the other party. It now contains directly the associated data as input unlike Signal in [ACD19]. This could contain for example the output of the initial key material, some constants or whatever. The SAS mechanism is triggered whenever a new epoch occurs. The current epoch drc variable is sent each time, since the protocol can handle out-of-order messages.

```
Init(pp, sk_P, pk_P̄, P) :
    $\mathsf{id} \leftarrow \mathcal{P}$
    $\mathsf{k} \leftarrow\$ \mathsf{IKA}(\mathsf{pp}, \mathsf{sk}_{\mathcal{P}}, \mathsf{pk}_{\overline{\mathcal{P}}})$
    $(\mathsf{k}_{root}, \mathsf{k}_{CKA}) \leftarrow \mathsf{KSM}(\mathsf{k}, \perp)$
    $\sigma_{root} \leftarrow \mathsf{P\text{-}Init}(\mathsf{k}_{root})$
    $(\sigma_{root}, (\mathsf{k}_{SAS}, \mathsf{k}_{FS})) \leftarrow \mathsf{P\text{-}Up}(\sigma_{root}, \perp)$
    $\mathsf{v}[\cdot], \mathsf{w}[\cdot] \leftarrow \perp$
    if $\mathsf{id} = \mathsf{A}$ then
        $\mathsf{v}[0] \leftarrow \mathsf{FS\text{-}Init\text{-}R}(\mathsf{k}_{FS})$
    else
        $\mathsf{v}[0] \leftarrow \mathsf{FS\text{-}Init\text{-}S}(\mathsf{k}_{FS})$
    $\mathsf{w}[0] \leftarrow \mathsf{SAS\text{-}EA\text{-}Init}(\mathsf{k}_{SAS})$
    $\gamma \leftarrow \mathsf{CKA\text{-}Init\text{-}A}(\mathsf{k}_{CKA})$
    $\mathsf{T}_{cur} \leftarrow \perp$
    $l_{prv}, \mathsf{t}_{cur} \leftarrow 0$
```

```
Send(ad, pt) :
    if $\mathsf{t}_{cur}$ is even then
        $l_{prv} \leftarrow \mathsf{FS\text{-}Stop}(\mathsf{v}[\mathsf{t}_{cur} - 1])$
        $\mathsf{t}_{cur} \leftarrow \mathsf{t}_{cur} + 1$
        $(\gamma, \mathsf{T}_{cur}, \mathsf{I}) \leftarrow \mathsf{CKA\text{-}S}(\gamma)$
        $(\sigma, (\mathsf{k}_{FS}, \mathsf{k}_{SAS})) \leftarrow \mathsf{P\text{-}Up}(\sigma, \mathsf{I})$
        $\mathsf{v}[\mathsf{t}_{cur}] \leftarrow \mathsf{FS\text{-}Init\text{-}S}(\mathsf{k}_{FS})$
        $\mathsf{w}[\mathsf{t}_{cur}], \mathsf{w}[\mathsf{t}_{cur} - 1].R$
            $\leftarrow \mathsf{SAS\text{-}EA\text{-}Send}(\mathsf{k}_{SAS})$
    $\mathsf{drc} \leftarrow (\mathsf{w}[\mathsf{t}_{cur} - 2].d, \mathsf{w}[\mathsf{t}_{cur} - 1].R,$
            $\mathsf{w}[\mathsf{t}_{cur}].c)$
    $\mathsf{h} \leftarrow (\mathsf{t}_{cur}, \mathsf{T}_{cur}, l_{prv}, \mathsf{drc})$
    $\mathsf{a} \leftarrow (\mathsf{ad}, \mathsf{h})$
    $(\mathsf{v}[\mathsf{t}_{cur}], \mathsf{e}) \leftarrow \mathsf{FS\text{-}Send}(\mathsf{v}[\mathsf{t}_{cur}], \mathsf{a}, \mathsf{pt})$
    $\mathsf{ct} \leftarrow (\mathsf{h}, \mathsf{e})$
    return ct
```

```
Receive(ad, ct) :
    $(\mathsf{h}, \mathsf{e}) \leftarrow \mathsf{ct}$
    $(\mathsf{t}, \mathsf{T}, l, \mathsf{drc}) \leftarrow \mathsf{h}$
    req t even and $\mathsf{t} \leq \mathsf{t}_{cur} + 1$
    if $\mathsf{t} = \mathsf{t}_{cur} + 1$ then
        $\mathsf{t}_{cur} \leftarrow \mathsf{t}_{cur} + 1$
        $\mathsf{FS\text{-}Max}(\mathsf{v}[\mathsf{t} - 2], l)$
        $(\gamma, \mathsf{I}) \leftarrow \mathsf{CKA\text{-}R}(\gamma, \mathsf{T})$
        $(\sigma, (\mathsf{k}_{FS}, \mathsf{k}_{SAS})) \leftarrow \mathsf{P\text{-}Up}(\sigma, \mathsf{I})$
        $\mathsf{v}[t] \leftarrow \mathsf{FS\text{-}Init\text{-}R}(\mathsf{k}_{FS})$
        $\mathsf{w}[t] \leftarrow \mathsf{SAS\text{-}EA\text{-}Recv}(\mathsf{k}_{SAS}, \mathsf{drc},$
            $\mathsf{w}[\mathsf{t} - 2], \mathsf{w}[\mathsf{t} - 1])$
    $\mathsf{a} \leftarrow (\mathsf{ad}, \mathsf{h})$
    $(\mathsf{v}[\mathsf{t}], \mathsf{i}, \mathsf{pt}) \leftarrow \mathsf{FS\text{-}Recv}(\mathsf{v}[\mathsf{t}], \mathsf{a}, \mathsf{e})$
    if $\mathsf{pt} = \perp$ then
        error
    return (acc, pt, t, i)
```

```
AuthSend(st_P)
    $\mathsf{t} \leftarrow \mathsf{st}_{\mathcal{P}}.t - 2$
    $\mathsf{SAS} \leftarrow \mathsf{st}_{\mathcal{P}}.\mathsf{w}[\mathsf{t}].\mathsf{SAS}$
    return (t, SAS)
```

```
AuthReceive(st_P, at)
    $(\mathsf{t}, \mathsf{SAS}) \leftarrow \mathsf{at}$
    $\mathsf{SAS}' \leftarrow \mathsf{st}_{\mathcal{P}}.\mathsf{w}[\mathsf{t}].\mathsf{SAS}$
    return $\mathsf{SAS} = \mathsf{SAS}'$
```

Figure 8: *ARC scheme*

Receiving messages. The Receive procedure handles the reception of messages. It also gets the associated data directly as input, as explained before. The drc variable is used only at the reception of a new epochs message. This is done in order to avoid changing each time receiving a message. Note that if the decryption of the ciphertext fails, an error is returned. In that case, the current state of the party receiving is rolled back to what it was before the function call. We do not make it explicit but this is what is captured by the acc variable, that tells if the decryption has been accepted or not.

Authenticating epochs. The algorithms AuthSend and AuthReceive are used to send and receive the computed SAS from the internal state of a party, for an epoch, through the out-of-band authenticated channel. As soon as an SAS is verified, one can delete all the previous ones as it authenticates the discussion until that specific epoch. This procedures can be easily implemented by just getting the SAS from memory and outputting it concatenated with its corresponding epoch number. Then receiving would compare the received SAS with the one in memory and output the result of this comparison. As one can imagine, in the case of receiving an SAS that does not match the stored one, this would then trigger some alert mechanism. Note that the third message of the SAS containing the decommit value may not have been received by the party at the time of receiving the SAS but in this case we just reject the SAS for simplicity.

5.4 Security of the scheme

We present some intuition about why the Auth security should hold. First of all, recall that the so-called "authentication keys" AK^i are derived one after the other using the asymmetric ratchet. By constriction, this is done by the mean of a secure PRF-PRNG. As they are the product of the initial key agreement, authenticating those keys would reflect authenticating the long-term key material used in that initial key exchange. This authentication is done through the SAS-EA mechanism. Intuitively, if there is no collision on the PRF-PRNG output, then the AKs will all be different and any disagreement between the parties will be detected assuming the SAS-EA is secure.[8] Thus, if the SAS-EA scheme can be proven secure, then the Auth security game defined can be reduced to the SAS-EA security game. Thus our ARC scheme would be proven to be secure. Properly defining and formalising the SAS-EA game and the related reductions could be the work for a continuation of this project.

[8]Note that we do not use the same hashing mechanism as [DH20] since it appears unnecessary.

6 Conclusion

We have defined and built an authenticated ratcheted communication (ARC) scheme. It is based on a modified version of the Signal protocol developed by [DH20] and the SAS-based mutual message authentication defined in [PV06]. We introduced the foundations of the Signal protocol in section 2. In particular, we focused on the key agreement, the asymmetric ratchet of the double ratchet protocol and the entity authentication mechanism. We have then explained the modification of the key exchange in section 3. As previously said this modification in the asymmetric ratchet is a foundation for the ARC scheme. In section 4, we presented the SAS-based mutual message authentication protocol. We then defined formally the ARC scheme itself by the mean of a syntax, a security definition, the primitives used in the scheme and an implementation of the scheme itself. It allows to keep all the security guarantees of the Signal protocol, added of a simple and working entity-authentication mechanism providing authentication of the initial key material.

Future work. As stated in the previous section, some security proofs still need to be formalized, particularly the SAS-EA game and its oracles. This way, we could ensure the Auth security for our ARC scheme. One could also think about implementation details for the SAS representation on the authenticated channel or generalising the use of SAS-based cryptography in other protocols.

References

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. In Vincent Rijmen and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 129–158. Springer Verlag, 2019.

[BSJ$^+$17] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 619–650, Cham, 2017. Springer International Publishing.

[CDV21] Andrea Caforio, F. Betül Durak, and Serge Vaudenay. Beyond security and efficiency: On-demand ratcheting with security awareness. In Juan A. Garay, editor, *Public-Key Cryptography – PKC 2021*, pages 649–677, Cham, 2021. Springer International Publishing.

[DH20] Benjamin Dowling and Britta Hale. There can be no compromise: The necessity of ratcheted authentication in secure messaging. Cryptology ePrint Archive, Report 2020/541, 2020. https://ia.cr/2020/541.

[Mar16] Moxie Marlinspike. Safety number updates. *Signal's blog*, Nov 2016. https://www.signal.org/blog/safety-number-updates/.

[MP16] Moxie Marlinspike and Trevor Perrin. The x3dh key agreement protocol. *Open Whisper Systems*, 283, 2016. https://www.signal.org/docs/specifications/x3dh/.

[PM16] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. *GitHub wiki*, 2016. https://www.signal.org/docs/specifications/doubleratchet/.

[PV06] Sylvain Pasini and Serge Vaudenay. Sas-based authenticated key agreement. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 395–409, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Vau05] Serge Vaudenay. Secure communications over insecure channels based on short authenticated strings. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, pages 309–326, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.