MARSHAL: Messaging with Asynchronous Ratchets and Signatures for faster HeALing

Abstract. Secure messaging applications are deployed on devices that can be compromised, lost, stolen, or corrupted in many ways. Thus, recovering from attacks to get back to a clean state is essential and known as *healing*. Signal is a widely-known, privacy-friendly messaging application, that uses key-ratcheting mechanism updates keys at each stage to provide end-to-end channel security, forward secrecy, and post-compromise security. We strengthen this last property, by providing a faster healing. Signal needs up to two full chains of messages before recovering, our protocol enables recovery after the equivalent of a chain of only one message. We also provide an extra protection against session-hijacking attacks. We do so, while building on the pre-existing Signal backbone, without weakening its other security assumptions, and still being compatible with Signal's out-of-order message handling feature.

1 Introduction

Asynchronous messaging protocols like Signal [11] or OTR [5] allow two parties that are not always simultaneously online to communicate securely. The protocol must guarantee the confidentiality and authenticity of the exchanged messages with respect to a PitM[1]. *Forward security* additionally requires that if a party is compromised (loses its long-term keys), past communications are still secure. Signal pioneered a new property, formalized by Cohn-Gordon *et al.* as *post-compromise security*[2] (PCS) [7]. It entails a renewal of the protocol's original security guarantees even after a complete compromise of a party's secrets. Cohn-Gordon *et al.* later showed that Signal attains PCS [6] under certain assumptions. Signal's healing ability follows from the gradual insertion of fresh Diffie-Hellman (DH) material into the current session secrets.

In Fig. 1, we give a toy-example of a Signal conversation between an *initiator* Alice and the *responder* Bob. Each message comes at a protocol *stage*[3], denoted by (x, y). The y value increases when the speaker changes (Alice starts at $y = 1$, then y turns to 2 when Bob speaks, etc.). The x value increases with each new message from the same speaker, and is reset to 1 for each new value of y. Each stage (x, y) is associated with a message key $\mathsf{mk}^{x,y}$, used to encrypt and authenticate that stage's message. To evolve from a key $\mathsf{mk}^{x,y}$ to $\mathsf{mk}^{x+1,y}$ (next message, same speaker), the two peers use a key-derivation function (KDF) with no further freshness. This is called a *symmetric ratchet*, denoted by [S] in Fig. 1. To update a key $\mathsf{mk}^{x,y}$ to $\mathsf{mk}^{1,y+1}$ (new speaker), a DH share is used as freshness into the KDF. This is called the *asymmetric ratchet*, denoted by [A].

Signal's PCS guarantee is limited by two main factors: the lack of persistent authentication (noticed by Blazy *et al.* [4]) and the frequency of asymmetric ratchets, which is our key motivation.

[1] Person-in-the-Middle is a politically-correct version of Man-in-the-Middle.
[2] This property is also called *healing*.
[3] We use a different notation of stages from [6].

Sender	**Key(s)**	**AD**	**Message**	MARSHAL	**Signal**
Alice	$\mathsf{mk}^{1,1}$:	$(1, \mathsf{Rchpk}_A^1)$	Hi Bob	✓	✓
	[S] $\mathsf{mk}^{2,1}$	$(2, \mathsf{Rchpk}_A^1)$	How are you ?	×	×
	[S] $\mathsf{mk}^{3,1}$-$\mathsf{mk}^{17,1}$	$(3, \mathsf{Rchpk}_A^1)$-$(17, \mathsf{Rchpk}_A^1)$	(... *15 messages*)	✓	×
	[S] $\mathsf{mk}^{18,1}$	$(18, \mathsf{Rchpk}_A^1)$	Cinema tonight ?	✓	×
Bob :	[A] $\mathsf{mk}^{1,2}$	$(1, \mathsf{Rchpk}_B^2)$	Hi Alice	✓	×
	[S] $\mathsf{mk}^{2,2}$	$(2, \mathsf{Rchpk}_B^2)$	I'm good, thanks	✓	×
	[S] $\mathsf{mk}^{3,2}$-$\mathsf{mk}^{12,2}$	$(3, \mathsf{Rchpk}_B^2)$-$(12, \mathsf{Rchpk}_B^2)$	(... *10 messages*)	✓	×
Alice :	[A] $\mathsf{mk}^{1,3}$	$(1, \mathsf{Rchpk}_A^3)$	Great	✓	✓

Fig. 1: Toy example for Signal and MARSHAL. Messages are encrypted with the keys in the second column (indexed by the stage) and have the associated data (AD) in the third column. The labels [A] and [S] indicate asymmetric and symmetric ratcheting respectively. The security (✓) and insecurity (×) of messages is given, assuming Alice is compromised at message 2. Italics show that several messages are sent in the same chain.

Persistent authentication [4]. In Signal, the two parties initially use long-term identity-keys to authenticate. However, subsequent authentication only relies on knowledge of a previous stage-specific key. This allows an adversary to impersonate entities by only compromising a party's ephemeral state; then the adversary hijacks the session by forcing a ratchet. Two events follow: (i) the keys between the honest endpoints diverge irrevocably from those derived by the adversary and the non-compromised endpoint; and (ii) the non-compromised endpoint is unaware of this. In Fig. 1, an adversary can compromise Alice after her second message, learning the private key rchk_A^1 corresponding to the public key[4] Rchpk_A^1. The attacker then blocks all messages from Alice to Bob, and waits for Bob to ratchet (with "Hi Alice"). The adversary, impersonating Alice, forces a ratchet by sending a new message. At this point, Bob and the adversary ratchet to keys depending on the DH product of $\mathsf{Rchpk}_{\mathcal{A}}$ and Rchpk_B^2. Alice, however, can no longer ratchet to those keys, even given Bob's and the adversary's transcript.

Frequency of asymmetric ratchets. In Signal, parties asymmetrically ratchet whenever the speaker changes. The private key for that ratchet can, however, be leaked to an adversary. A compromise of, say, Alice compromises the security of Alice's entire chain of messages, then Bob's entire chain of responses. The protocol only heals once Alice chooses new ratcheting material and safely sends it to Bob. In Fig. 1, if the adversary compromises Alice after she's sent the first message, it obtains 30 message keys.

Contributions. Our protocol MARSHAL (Messaging with Asynchronous Ratchets and Signatures for faster HeALing) achieves persistent authentication and faster healing than Signal, requiring just one message to recover MARSHAL's original security once the attacker has compromised all of Alice's ephemeral information and even some combination of long-term keys. In MARSHAL, the parties ratchet at every stage, even when the speaker has not changed. This causes two technical challenges. First, there is the

[4] The adversary also has access to chain and message keys computed at this point, and to the root key necessary for the ratchet. However, none of Alice's long-term information is necessary for the attack.

question of how Alice (the initiator) will ratchet at the start of the protocol, before Bob comes online. We handle this by requiring Bob to register an extra ephemeral DH element on the semitrusted server. A second challenge concerns out-of-order messages. To handle those, the sender will send at each stage a *concatenation* of all the public ratchet keys used so far in that chain, in order, in the associated data.

In MARSHAL, session-hijacking attacks require long-term credentials such as the party's identity- or signature key. By contrast, an adversary can hijack a Signal session with only ephemeral information, like the current message and/or chain key.

MARSHAL provides almost instant healing: unless it holds the party's long-term keys *and* ephemeral information, the attacker cannot compromise more than one message at a time. As soon as either party ratchets honestly, the adversary loses the ability to decrypt any fresh messages.

These strong properties come at a cost. Apart from registering an additional DH element and having to perform DH computations at each stage, MARSHAL adds to the complexity of Signal in two ways: (1) requiring the transmission of a number of DH elements that is linear in the maximal depth of the chain; (2) using signatures to transmit the encrypted messages and stage metadata. The former allows us to provide message-loss resilience: if this is not needed, metadata size can be reduced. The second source of complexity, the signatures, serve a double purpose: they help preserve AKE security, and they restrict an adversary's ability to impersonate parties upon corruption.

Related Work. Ratcheted key-exchange (RKE) was introduced as a unidirection, single-move primitive by Bellare *et al.* [3], who used it to define and instantiate ratcheted encryption. This security model was later extended by work such as [15,9] to treat double ratchets, but also more generic RKE. A crucial difference between generic RKE and our work is that we focus on the full message transmission process, as in the case of [6,4].

The work of Alwen *et al.* [1] provides a complete security model for protocols like Signal, which also handles out-of-order messages (which they call *immediate decryption*). Alwen *et al.* view asynchronous messaging protocols as a composition of three parts: a hash function that generates pseudorandom output (PRF-PRNG), a primitive called forward-secure AEAD (FS-AEAD) which captures symmetric ratchets, and continuous key-agreement (CKA) which captures asymmetric ratchets. While this work does capture Signal and allows for modular security proofs, it is not so well suited to the analysis of our protocol, for three main reasons. First, MARSHAL does not employ any symmetric ratcheting; second, we want to capture the properties of the actual message transmission; third, we do not use AEAD solely, but rather combine it with a public-key authentication mechanism. This would minimally indicate a need to modify the FS-AEAD primitive. We therefore prefer a security model that is less modular, but comes closer to the protocol, similar to the one used in [4]. We have adapted one of the properties they consider to our security model, namely that of message-loss resilience.

The works of Jost *et al.* [10,8] provide efficient instantiations of bidirectional ratcheted key-exchange by using relatively inexpensive primitives (unlike previous work such as *e.g.*, that of Poettering and Rössler [15]). However, these protocols are different and do not follow the structure of Signal. In addition, features such as out-of-order messages are not included, because some of these constructions *require* the parties to receive each message. Starting from Signal's structure, we preserve properties such as out-of-order messages, and have stronger healing by persistent authentication and more frequent asymmetric ratchets.

Our work comes closest to the SAID protocol of Blazy *et al.* [4], whose notion of persistent authentication prevents hijacking attacks. SAID therefore authenticates each ratchet by using identity keys. As long as the identity keys are safely stored, session-hijacking cannot happen because the adversary cannot convince Bob he is ratcheting with the correct person. While we also ensure persistent authentication, our work uses the backbone of Signal and its security assumptions: public-key cryptography and a semi-trusted middle server. By contrast, Blazy *et al.* constructed their protocol in the paradigm of identity-based cryptography.

2 Background

Notations. Let g be a generator of a cyclic group $\mathbb{G}$ of prime order q. A user's Diffie-Hellman public key is an exponentiation of g to the private exponent k: $pk = g^k \mod p$ for a large prime p. Like [6] we end names of public keys in pk and private keys ending in k. For instance Rchpk is a ratchet public key with corresponding private key rchk. Let $DH(x, y) = x^y$ denote the exponentiation of $x \in \mathbb{G}$ to a power $y \in \mathbb{Z}_q$. A key generated by party P is denoted by ik_P while the public key is denoted ipk_P. Stage-specific keys have stages as superscript *e.g.*, , $\mathsf{mk}^{1,1}$. In this paper we assume that all signature schemes involve hashing, and omit the hashing in the notation, *i.e.*, $\mathtt{SIGN}_{\mathsf{sk}}(m) := \mathtt{Sign}_{\mathsf{sk}}(H(m))$ for a hash function H. We use the notations $\mathtt{AEAD.Enc}$ and $\mathtt{AEAD.Dec}$ for encryption and decryption respectively of an AEAD-scheme.

The Signal Protocol. While Signal is *post-compromise secure* and has *forward secrecy*, a session that is severely compromised by an adversary can take a long time to heal. In Signal, message keys are derived from chain keys by means of a key-derivation function (KDF), which outputs a new chain key $\mathsf{ck}^{(x+1,y)}$ and a message key $\mathsf{mk}^{(x,y)}$ from a chain key $\mathsf{ck}^{(x,y)}$. If an adversary compromises $\mathsf{ck}^{(1,1)}$, it will have access to every message on the chain $y = 1$ (stages from $(1, 1)$ up to, but not including $(1, 2)$). However, if the adversary does not know Alice's private ratchet key rchk^1 for the stages with $y = 1$, then as soon as Bob's asymmetric ratchet comes (at stage $(1, 2)$), the chain heals. If rchk^1 *is* known, then *two* full chains of messages are compromised.

Active attacks are even more dangerous. So far, attacks consisted of compromising a party's state, then inspecting and decrypting traffic. If, upon corrupting $\mathsf{ck}_A^{(1,1)}$, the adversary replaces Alice's original ratchet key by a key of its own, then the attack renders the chain impossible to heal without starting a new conversation. The fact that the adversary can do this while in possession of a single chain key (which is a piece of temporary, stage-specific data) is troubling. Adding persistent authentication would prevent such attacks, ensuring that all key derivations require long-term keys. MARSHAL enhances the original Signal protocol in order to achieve near-optimal security against both passive and active attackers. More details are given in Appendix A.

3 The MARSHAL protocol

The protocol we propose, MARSHAL, runs –like Signal– in several stages: registration, session setup, and communication. We describe in Fig. 2 the session-setup and communication phases of MARSHAL. As a novelty, MARSHAL requires two types of ratchet keys: *same-user* ratchet keys, and *cross-user* ratchet keys. Same-user ratchet keys are indexed by stage, and generated whenever a new message is sent: for instance $\mathsf{Rchpk}^{2,1}$ denotes the ratchet public key at stage $(2, 1)$ (the second message sent in the first message chain). Cross-user ratchet keys are only generated at the beginning of a chain of messages and indexed only by the y-component of the stage (called a *chain index*). We denote by T_i the public key generated during the i-th message chain and by T_0 an initial public key registered by the session's responder (and recovered by its initiator).

While stages are indexed as (x, y) with $x, y \geq 1$, special indexes $x = 0$ and $y = 0$ denote special ratchet keys used for initialization. The first same-user ratchet key $\mathsf{Rchpk}^{0,1}$ is only used to compute the master secret of a session. Additionally, a ratchet key T_0 is registered by each user. The initiator of a session uses its correspondent's initial ratchet key during the first chain of communication ($y = 1$). Note that this method of ratcheting uniquely associates stages and chain indexes to the party generating them.

3.1 Registration

To use MARSHAL, each party P must first *register*, by generating private keys and uploading the corresponding public keys to the server: a long-term identity key ik_P; a medium-term prekey prek_P, and a signature on that key (generated with the identity key ik_P); multiple ephemeral one-time-use prekeys ephpk_P; multiple medium-term stage keys T_0. The last of these keys is a cross-user ratchet key (see above): a novelty with respect to Signal, which will help Alice asymmetric-ratchet in the first chain of messages, when she has not yet had a message (and therefore a ratcheting key) from Bob. In addition to these keys users will also generate and subsequently use a pair of long-term signature keys (sk_P, pk_P). These keys will not be registered on the server, but rather included in the associated data in each partner's first respective chain of messages.

3.2 Session Setup

Whenever Alice A wants to contact Bob B, she runs a protocol similar to that of Signal and [12], with some small tweaks.

The master secret. To initiate a session with B, Alice queries the server for Bob's following values: the identity key ipk_B, a signed prekey prepk_B, a one-time prekey ephpk_B (if available), and a medium-term stage key T_B^{0}, denoted in short T_0. Having received those keys, A generates its own ephemeral key ek_A. The master secret $ms = \mathsf{prepk}_B^{\mathsf{ik}_A} || \mathsf{ipk}_B^{\mathsf{ek}_A} || \mathsf{prepk}_B^{\mathsf{ek}_A} || \mathsf{ephpk}_B{}^{\mathsf{ek}_A}$ is a concatenation of DH values, as computed in Signal.

First keys. Alice randomly generates a same-user ratchet keypair $(\mathsf{rchk}^{0,1}, \mathsf{Rchpk}^{0,1})$. She computes a DH of her ratchet key and prepk_B, and the result is fed to a key derivation function along with ms to produce a chain key $\mathsf{ck}^{1,1}$.

Alice $(\mathsf{ik}_A, \mathsf{ipk}_B, \mathsf{prepk}_B, \mathsf{ephpk}_B, \boxed{\mathsf{T}_0})$		**Bob** $(\mathsf{ik}_B, \mathsf{ipk}_A, \mathsf{prek}_B, \mathsf{ephk}_B, \boxed{\mathsf{T}_0})$
Session initialization: initiator Alice, responder Bob.		
$\mathsf{ek}_A, \mathsf{rchk}^{0,1}, \boxed{\mathsf{t}_1, \mathsf{rchk}^{1,1}} \xleftarrow{\$} \mathbb{Z}_q;$ $\mathsf{T}_1 = g^{t_1}; \mathsf{Epk}_A = g^{\mathsf{ek}_A};$ $\mathsf{Rchpk}^{0,1} = g^{\mathsf{rchk}^{0,1}}; \mathsf{Rchpk}^{1,1} = g^{\mathsf{rchk}^{1,1}}$ $ms = \mathsf{prepk}_B^{\mathsf{ik}_A} \|\| \mathsf{ipk}_B^{\mathsf{ek}_A} \|\| \mathsf{prepk}_B^{\mathsf{ek}_A} \|\| \mathsf{ephpk}_B{}^{\mathsf{ek}_A}$ $\mathsf{ck}^{1,1} = \mathsf{KDF}_r\left(\mathsf{prepk}_B^{\mathsf{rchk}^{0,1}} \|\| ms\right)$ $(\mathsf{ck}^{2,1}, \mathsf{mk}^{1,1}) = \mathsf{HKDF}(\mathsf{ck}^{1,1} \boxed{\|\| \sigma_{1,1} \|\| (\mathsf{ipk}_B)^{\mathsf{rchk}^{1,1}}})$		
First message: stage $(1,1)$, Alice is the sender, Bob, the receiver.		
$\boxed{AD_{y=1}} = \mathsf{Epk}_A \|\| \mathsf{ipk}_A \|\| \mathsf{ipk}_B \|\| \mathsf{prepk}_B \|\|$ $\mathsf{ephpk}_B \|\| \mathsf{T}_0 \|\| \mathsf{Rchpk}^{0,1} \|\| \mathsf{T}_1$ $\boxed{AD_{1,1}} = (1,1) \|\| \mathsf{Rchpk}^{1,1} \|\| \sigma_{1,1}$ $c_{1,1} = \mathtt{AEAD.Enc}_{\mathsf{mk}^{1,1}}(M_{1,1}; AD_1 \|\| AD_{1,1})$	$\xrightarrow[pk_A, \mathtt{SIGN}_{\mathsf{ik}_A}(pk_A)]{c_{1,1}, \mathtt{SIGN}_{sk_A}(c_{1,1}),}$	Verify signature on pk_A and $\sigma_{1,1}$ $ms = \mathsf{ipk}_A^{\mathsf{prek}_B} \|\| \mathsf{Epk}_A^{\mathsf{ik}_B} \|\| \mathsf{Epk}_A^{\mathsf{prek}_B} \|\| \mathsf{Epk}_A^{\mathsf{ephk}_B}$ $\mathsf{ck}^{1,1} = \mathsf{KDF}_r((\mathsf{Rchpk}^{0,1})^{\mathsf{prek}_B} \|\| ms)$ $(\mathsf{ck}^{2,1}, \mathsf{mk}^{1,1}) = \mathsf{HKDF}(\mathsf{ck}^{1,1} \|\| \sigma_{1,1} \|\| (\mathsf{Rchpk}^{1,1})^{\mathsf{ik}_B})$ $M_{1,1} = \mathtt{AEAD.Dec}_{\mathsf{mk}^{1,1}}(c_{1,1}).$
ℓ-th **message**: stage $(\ell, 1)$, Alice is the sender, Bob, the receiver.		
$\mathsf{rchk}^{\ell,1} \xleftarrow{\$} \mathbb{Z}_q$, set $\mathsf{Rchpk}^{\ell,1} = g^{\mathsf{rchk}^{\ell,1}}$ $(\mathsf{ck}^{\ell+1,1}, \mathsf{mk}^{\ell,1}) = \mathsf{HKDF}(\mathsf{ck}^{\ell,1} \|\| \sigma_{\ell,1} \|\| \mathsf{ipk}_B^{\mathsf{rchk}^{\ell,1}})$ $\boxed{AD_{\ell,1}} = (\ell, 1) \|\| \{\mathsf{Rchpk}^{x,1}\}_{1 \le x \le \ell} \|\| \sigma_{\ell,1}$ $c_{\ell,1} = \mathtt{AEAD.Enc}_{\mathsf{mk}^{\ell,1}}(M_{\ell,1}; AD_1 \|\| AD_{\ell,1})$	$\xrightarrow[pk_A, \mathtt{SIGN}_{\mathsf{ik}_A}(pk_A)]{c_{\ell,1}, \mathtt{SIGN}_{pk_A}(c_{\ell,1}),}$	Verify leftover signatures $(\mathsf{ck}^{\ell+1,1}, \mathsf{mk}^{\ell,1}) = \mathsf{HKDF}(\mathsf{ck}^{\ell,1}, \|\| \sigma_{\ell,1} \|\| (\mathsf{rchk}^{\ell,1})^{\mathsf{ik}_B})$ $M_{\ell,1} = \mathtt{AEAD.Dec}_{\mathsf{mk}^{\ell,1}}(c_{\ell,1}).$
Switching speakers: Bob comes online and begins a new ratcheting chain.		
		$\boxed{\mathsf{t}_2, \mathsf{rchk}^{1,2}} \xleftarrow{\$} \mathbb{Z}_q; \mathsf{T}_2 = g^{\mathsf{t}_2}, \boxed{\mathsf{Rchpk}^{1,2} = g^{\mathsf{rchk}^{1,2}}}$ $\mathsf{ck}^{1,2} = \mathsf{HKDF}(\boxed{\mathsf{T}_1{}^{\mathsf{ik}_B} \|\| \mathsf{ipk}_A{}^{\mathsf{t}_0}})$ $(\mathsf{ck}^{2,2}, \mathsf{mk}^{1,2}) = \mathsf{HKDF}(\mathsf{ck}^{1,2}, \boxed{\sigma_{1,2} \|\| (\mathsf{ipk}_A)^{\mathsf{rchk}^{1,2}}})$
Bob's message, stage $(1,2)$: Bob is the sender, Alice is the receiver.		
Verify signature on pk_B and $\sigma_{1,2}$ $\mathsf{ck}^{1,2} = \mathsf{KDF}_r((\mathsf{ipk}_B)^{\mathsf{t}_1} \|\| (\mathsf{T}_0)^{\mathsf{ik}_A})$ $(\mathsf{ck}^{2,2}, \mathsf{mk}^{1,2}) = \mathsf{HKDF}(\mathsf{ck}^{1,2} \|\| \sigma_{1,2} \|\| (\mathsf{Rchpk}^{1,2})^{\mathsf{ik}_A})$ $M_{1,2} = \mathtt{AEAD.Dec}_{\mathsf{mk}^{1,2}}(c_{1,2})$	$\xleftarrow[pk_B, \mathtt{SIGN}_{\mathsf{ik}_B}(pk_B)]{c_{1,2}, \mathtt{SIGN}_{pk_B}(c_{1,2}),}$	$\boxed{AD_{y=2}} = \mathsf{T}_2$ $\boxed{AD_{1,2}} = (1,2) \|\| \mathsf{Rchpk}^{1,2} \|\| \sigma_{1,2}$ $c_{1,2} = \mathtt{AEAD.Enc}_{\mathsf{mk}^{1,2}}(M_{1,2}; AD_2 \|\| AD_{1,2})$

Fig. 2: MARSHAL protocol execution between Alice and Bob for the first few stages. The yellow boxes indicate modifications with respect to Signal protocol [6]. The transmitted data is also different and not in yellow for more clarity.

Signature keys. A also needs a signature key, as described in Section 3.1.

3.3 Communication phase

We first make a few remarks about the protocol.

MARSHAL **ratcheting.** Our protocol heals faster than Signal because both parties ratchet asymmetrically at every stage. Thus, even at stage $(1,1)$, Alice needs ratcheting randomness from Bob, which in MARSHAL comes in the form of the registered public key T_0 (see Section 3.1). For stages $(1, m)$ with integer $m \geq 1$, the party whose

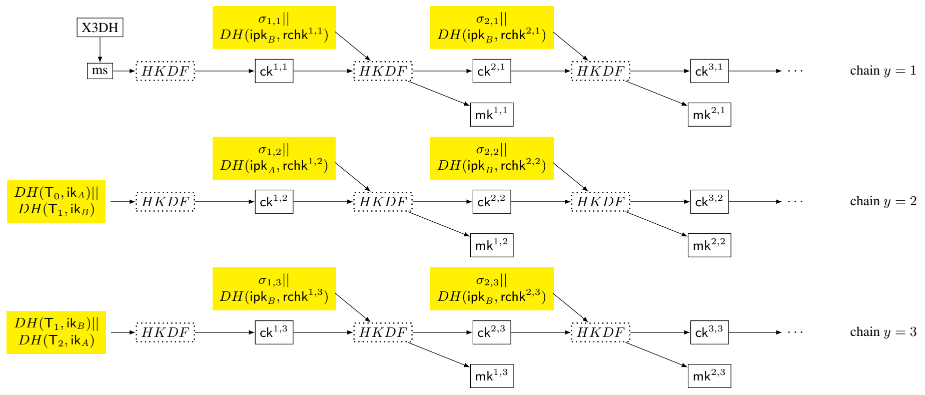

X3DH
ms
HKDF
$\sigma_{1,1}||$ $DH(\mathsf{ipk}_B, \mathsf{rchk}^{1,1})$
$\sigma_{2,1}||$ $DH(\mathsf{ipk}_B, \mathsf{rchk}^{2,1})$
$\mathsf{ck}^{1,1}$ $\mathsf{ck}^{2,1}$ $\mathsf{ck}^{3,1}$
$\mathsf{mk}^{1,1}$ $\mathsf{mk}^{2,1}$
chain $y = 1$
$DH(\mathsf{T}_0, \mathsf{ik}_A)||$ $DH(\mathsf{T}_1, \mathsf{ik}_B)$
$\sigma_{1,2}||$ $DH(\mathsf{ipk}_A, \mathsf{rchk}^{1,2})$
$\sigma_{2,2}||$ $DH(\mathsf{ipk}_B, \mathsf{rchk}^{2,2})$
$\mathsf{ck}^{1,2}$ $\mathsf{ck}^{2,2}$ $\mathsf{ck}^{3,2}$
$\mathsf{mk}^{1,2}$ $\mathsf{mk}^{2,2}$
chain $y = 2$
$DH(\mathsf{T}_1, \mathsf{ik}_B)||$ $DH(\mathsf{T}_2, \mathsf{ik}_A)$
$\sigma_{1,3}||$ $DH(\mathsf{ipk}_B, \mathsf{rchk}^{1,3})$
$\sigma_{2,3}||$ $DH(\mathsf{ipk}_B, \mathsf{rchk}^{2,3})$
$\mathsf{ck}^{1,3}$ $\mathsf{ck}^{2,3}$ $\mathsf{ck}^{3,3}$
$\mathsf{mk}^{1,3}$ $\mathsf{mk}^{2,3}$
chain $y = 3$

Fig. 3: MARSHAL key schedule diagram, where $\sigma_{x,y} = \mathtt{SIGN}_{sk_A}\left(\mathsf{T}_{y-1}||\mathsf{Rchpk}^{x,y}\right)$ for y odd and $\sigma_{x,y} = \mathtt{SIGN}_{sk_B}\left(\mathsf{T}_{y-1}||\mathsf{Rchpk}^{x,y}\right)$ for y even. The yellow boxes indicate modifications with respect to Signal protocol [6].

turn it is to speak will generate a cross-user ratcheting value t_m and compute the corresponding public value T_m. The T_m value is sent as part of the metadata of all messages with chain index m, and will be used for the ratchet at stage $(1, m+1)$.

Moreover at each stage (ℓ, m) for $\ell, m \geq 1$, the current speaker also generates a same-user key-pair $(\mathsf{rchk}^{\ell,m}, \mathsf{Rchpk}^{\ell,m})$ which will be used to generate chain and message keys for stage $\mathsf{mk}^{\ell+1,m}$. To account for out-of-order messages we then concatenation of all the public ratchet keys as metadata to each stage message.

MARSHAL **auxiliary data.** Each message will be sent end-to-end encrypted, together with some additional metadata, which is meant to tell Bob how to run the key-schedule. At each stage (ℓ, m) with $\ell, m \geq 1$, the auxiliary value will consist of two elements: $\mathsf{AD}_{y=m}$ and $\mathsf{AD}_{\ell,m}$. The former will include elements of the metadata that are universal across the chain (*i.e.*, all stages $(\cdot, m)$), whereas the second includes metadata that is stage-specific.

We detail each of the classes of stages, *cf.* Fig. 2 and 3 for further reference.

Alice's first message. At session setup, Alice has generated its cross-user ratchet keys $(\mathsf{t}_1, \mathsf{T}_1)$, and computed the chain key $\mathsf{ck}^{1,1}$. Now she generates the same-user ratchet key $\mathsf{rchk}^{1,1}$ and computes $\mathsf{Rchpk}^{1,1} = g^{\mathsf{rchk}^{1,1}}$. The message and chain-keys are computed as follows $(\mathsf{ck}^{2,1}, \mathsf{mk}^{1,1}) \leftarrow \mathsf{HKDF}(\mathsf{ck}^{1,1}||\sigma_{1,1}||(\mathsf{ipk}_B)^{\mathsf{rchk}^{1,1}})$ where $\sigma_{1,1} := \mathtt{SIGN}_{sk_A}\left(T_0||\mathsf{Rchpk}^{1,1}\right)$. We use:

$$\sigma_{x,y} := \begin{cases} \mathtt{SIGN}_{sk_A}\left(\mathsf{T}_{y-1}||\mathsf{Rchpk}^{x,y}\right), y \text{ odd} \\ \mathtt{SIGN}_{sk_B}\left(\mathsf{T}_{y-1}||\mathsf{Rchpk}^{x,y}\right), y \text{ even} \end{cases}$$

At chains $y = 1$ and $y = 2$, apart from cross-user ratchet keys, each user will need to include metadata that is universal for the session, and which helps at session setup. The metadata for $\mathsf{AD}_{y=1}$ thus includes the public identity keys of Alice and Bob, the medium-term and ephemeral keys of Bob as recovered by Alice from server, the value T_0 from the server, Alice's ephemeral public key used in the computation of the master secret, and two of Alice's ratchet public keys: its first same-user ratchet key $\mathsf{Rchpk}^{0,1}$, and its first cross-user ratchet key t_1. Finally, the stage-specific data contains: the stage index $(1,1)$ and the same-user ratcheting public key $\mathsf{Rchpk}^{1,1}$. Alice computes the authenticated encryption of the message herself: $c_{1,1} = \mathtt{AEAD.Enc}_{\mathsf{mk}^{1,1}}(M_{1,1}; \mathsf{AD}_{y=1} || \mathsf{AD}_{1,1})$ and sends: $c_{1,1}$, a signature on it, Alice's public signature key pk_A, and a signature on it.

Alice's $(\ell, 1)$ **message,** $\ell > 1$**.** Having already computed $\mathsf{t}_1, \mathsf{T}_1$, $\mathsf{AD}_{y=1}$, ratcheting material $\mathsf{Rchpk}^{1,1}, \mathsf{Rchpk}^{2,1}, \ldots, \mathsf{Rchpk}^{\ell-1,1}$, and the key $\mathsf{ck}^{l,1}$, Alice generates new same-user ratcheting key $\mathsf{rchk}^{\ell,1}$ and computes $\mathsf{Rchpk}^{\ell,1} = g^{\mathsf{rchk}^{\ell,1}}$. The key update relies on both long-term keys, for persistent authentication, and this same-user ratcheting key, for healing: $(\mathsf{ck}^{\ell+1,1}, \mathsf{mk}^{\ell,1}) \leftarrow \mathsf{HKDF}(\mathsf{ck}^{\ell,1}, \sigma_{\ell,1} || \mathsf{ipk}_B^{\mathsf{rchk}^{\ell,1}})$. The stage-specific metadata consists of the stage $(\ell, 1)$ and *all the ratcheting keys* $\{\mathsf{Rchpk}^{x,1}\}_{1 \geq x \geq \ell}$. Then Alice computes $c_{\ell,1}$ and sends: the ciphertext, a signature on it, its signature public key, and a signature on that.

Note that this procedure applies to *all* messages (ℓ, m) for $\ell > 1$ and $m \geq 1$, in replacing the y stage-index above, from 1 to m.

Decryption (Bob side). When B comes online, he first needs to compute the same session-setup values as Alice, including the master secret ms and the first chain key $\mathsf{ck}^{1,1}$. To do so, B queries the server for A's registered identity key and verifies that it is identical to the one included in $\mathsf{AD}_{y=1}$. Then, B verifies the signature on pk_A, and, if the verification returns 1, it stores that key as A's signature key. From now on, B will use that key to verify A's signatures. In particular, the verification of pk_A is only done for the first message that Bob actually checks in the $y = 1$ chain. Once pk_A is validated, B retraces Alice's steps to compute ms, the chain keys, and eventually, the first message key. Then he uses authenticated decryption to decrypt the first message.

Bob's first message. B generates a new cross-user ratcheting value t_2 with corresponding public value T_2 and a same-user ratcheting key $\mathsf{rchk}^{1,2}$ and computes $\mathsf{Rchpk}^{1,2} := g^{\mathsf{rchk}^{1,2}}$. Bob computes: $\mathsf{ck}^{1,2} \leftarrow \mathsf{KDF}_r((\mathsf{Rchpk}^{0,1})^{\mathsf{rchk}^{0,2}} || \mathsf{rk}_1 || (T_1)^{\mathsf{ik}_B} || \mathsf{ipk}_A^{\mathsf{rchk}^{1,2}})$, then its first sending keys $(\mathsf{ck}^{2,2}, \mathsf{mk}^{1,2}) \leftarrow \mathsf{HKDF}(\mathsf{ck}^{1,2} || \sigma_{1,2} || (\mathsf{ipk}_A)^{\mathsf{rchk}^{1,2}})$.

Then analogously to Alice's first message, Bob splits the metadata into the two auxiliary values $\mathsf{AD}_{y=2}$ and $\mathsf{AD}_{1,2}$. The signed public key pk_B is also appended to each of the messages in stages with chain-index $y = 2$, *cf.* Fig. 2.

Switching speakers. Similar computations will take place: generating cross-chain ratcheting public keys and new same-user ratcheting keys at every new message. The only differences with respect to stages $(1,1)$ and $(1,2)$ respectively will be that now the parties will no longer need to compute long-term keys or the master secret. In addition, starting from chain-index $y \geq 3$, the public key for signatures is no longer included in the message transmission.

Out-of-order messages/multiple messages. MARSHAL handles both out-of-order and lost messages to the same extent as Signal. Indeed, at each stage, the receiving party gets a list of ratcheting elements used along that chain, which will allow it to update correctly, even if some messages were lost in between. The parties will update their state in the order they receive the messages. In other words, say that Bob receives a message from Alice at stage $(1, 1)$, but then the next received message comes at stage $(4, 1)$ (thus, Bob is missing messages $(2, 1)$ and $(3, 1)$). Nevertheless, Bob will use the metadata at stage $(4, 1)$ to ratchet, thus computing the keys for stages $(2, 1)$ and $(3, 1)$ as well. If subsequently Bob receives message $(2, 1)$ with conflicting metadata, Bob disregards that.

In the same way, if multiple messages are received for some stage, the receiver will rely on the metadata (and message) received first, chronologically speaking.

4 Security Analysis

Our security models adapts and extends that of Blazy et al. [4]. We provide the general intuition for our framework and security games below, but since the details are lengthy, we leave the full syntax to the full version [2].

Syntax. We work in an environment with parties $P \in \mathcal{P}$, which have long-term key pairs $(\mathsf{sk}, \mathsf{pk})$, indexed by type. For instance in $\mathsf{MARSHAL}$ users have both identity and signature keys. Protocol *sessions* take part between two party *instances*. The i-th instance of P is denoted π_P^i. Beside long-term credentials, party instances also store:

pid**:** the identifier of the instance's purported partner, denoted $\pi_P^i.\mathsf{pid}$.

sid**:** the session identifier $\pi_P^i.\mathsf{sid}$: an evolving set of instance-specific values[5].

sidsk**:** instance-specific private keys (like ephemeral session keys), denoted sidsk_P^i.

sidpk**:** the public keys sidpk_P^i corresponding to sidsk_P^i.

stages**:** a list with elements $(s, \cdot)$, associating stages $s = (x, y)$ to values $v \in \{0, 1\}$ depending on whether a message was received (1) or not (0). We write $s \in \pi_P^i$ if, and only if, $(s, v) \in \pi_P^i.\mathsf{stages}$.

Tr**:** the instance's transcript $\pi_P^i.Tr$, associating to each stage s all data sent or received at that stage (in plaintext) – denoted $\pi_P^i.Tr[s]$.

rec**:** a list of subsets $\pi_P^i.\mathsf{rec}$, indexed by stage s and indicating messages and metadata received, in order. A special symbol $\perp$ is used for sending stages.

var**:** a set $\pi_P^i.\mathsf{var}$ of ephemeral values used to compute stage keys, indexed by stage. For $\mathsf{MARSHAL}$ this includes rchk_P^s, mk^s and $\mathsf{ck}_{x-1,y}$.

While Signal only allows one conversation per pair of parties, $\mathsf{MARSHAL}$ permits multiple conversations between the same two peers. Compared to Blazy *et al.* [4], we add the private/public key-pair $(\mathsf{sidsk}, \mathsf{sidpk})$, which allows us to separate one-time session-specific randomness required for the master secret from stage-specific, recurring randomness. In addition, the same-user ratcheting at every stage compels us to extend $\pi.\mathsf{rec}$ to include *all the messages and metadata received*, not just the first one.

As in Cohn-Gordon *et al.* [6], we abstract the semitrusted server from the setup, assuming it behaves honestly. It implies an authentication of each user upon registration.

We define asynchronous messaging protocols `ID-AsynchM` to be tuples of five algorithms $(\mathtt{aKGen}, \mathtt{aStart}, \mathtt{aRKGen}, \mathtt{aSend}, \mathtt{aReceive})$, such that:

$\underline{\mathtt{aKGen}(1^\lambda) \rightarrow (\mathsf{sk}, \mathsf{pk})}$: this algorithm outputs long-term credentials.

[5] In the case of our protocol, sid will be instantiated as a concatenation of all the auxiliary information sent throughout the protocol, ordered by stage.

$\underline{\texttt{aStart}(P, \mathsf{role}, \mathsf{pid}) \to \pi_P^i}$: creates a new instance of (existing) party P with partner pid, such that P has a role role $\in \{I, R\}$ (initiator/responder). The new instance is instantiated with that party's long-term keys.

$\underline{\texttt{aRKGen}(1^\lambda) \to (\mathsf{rchk}, \mathsf{Rchpk})}$: outputs a public/private ratcheting keypair.

$\underline{\texttt{aSend}(\pi_P^i, s, M, AD, aux) \to (\pi_P^i, C, AD^*, aux^*) \cup \bot}$: runs the sending part of the protocol for π_P^i on stage s, message M associated data AD, and auxiliary data aux; it outputs a ciphertext C with new associated and auxiliary data.

$\underline{\texttt{aReceive}(\pi_P^i, s, C, AD^*, aux^*) \to (\pi_P^i, M, AD, aux) \cup \bot}$: runs the receiving part of the protocol for instance π_P^i on stage s, ciphertext C, associated data AD^*, and auxiliary data aux^*; it outputs a message M and some (possibly transformed) associated and auxiliary data.

Definition 1 (Matching conversation). *Two instances π_A^i and π_B^i of an asynchronous-messaging protocol have* matching conversation *if and only if $\pi_A^i.\mathsf{sid} = \pi_B^j.\mathsf{sid}$ and $\pi_P^i.\mathsf{pid} = B$ and $\pi_B^j.\mathsf{pid} = A$.*

Correctness. Assume π_A^i and π_B^j have matching conversation (A is the initiator). The protocol guarantees *correctness* if, in the absence of an adversary, for every stage $s = (x, y)$ with $s \in \pi_A^i$ and $s \in \pi_B^j$ it holds simultaneously:

– Both instances have identical keys $\mathsf{mk}^{x,y}$ and $\mathsf{ck}^{x-1,y}$ at stage s.
– Assuming $(\pi_P^u, s, C, AD^*, aux^*)$ was output by aSend on input (π_P^i, s, M, AD, aux) for $(P, u) \in \{(A, i), (B, j)\}$ if $(\pi_Q^v, s, C, AD^*, aux^*)$ is input to $\texttt{aReceive}$, then M is the message output by π_Q^v for $(Q, v) = \{(A, i), (B, j)\} \setminus (P, u)$.

Threat Model. The guarantees we want to prove for MARSHAL are: AKE security (including authentication), post-compromise security, and out-of-order resilience within a fully adversarially-controlled network. The first two properties make up a single security definition, written as a game between the *adversary* and the *challenger*. The adversary can register malicious users, corrupt users to obtain long-term secrets, reveal stage- and session-specific ephemeral values, access (a function of) the party's secret key as a black box, prompt new instances of existing parties, and sending/receiving messages. The adversary ultimately has to distinguish from random a real message key generated by an honest instance speaking with another honest instance. As a result, instances will also need to keep track of the following attribute:

$\pi_P^i.\mathsf{b}[s]$**:** a challenge bit randomly chosen for each instance at each stage it reaches. If set to 1 at stage s, the adversary gets a real message key; else $\mathcal{A}$ sees a random key.

We briefly review our oracles and winning conditions in the following. Our games begin with a set of $\mathsf{n}_\mathcal{P}$ honest parties $\mathcal{P}$ for which long-term keys are generated. We instantiate an empty set $\mathcal{P}^*$ of malicious parties[6]. The adversary is given all the public keys, and will have access to the following oracles:

$\mathsf{oUReg}(P, \mathsf{pk}) \to \bot \cup \mathsf{OK}$: allows $\mathcal{A}$ to register public keys pk to malicious $P \notin \mathcal{P}$.

$\mathsf{oCorrupt}(P, \mathsf{ktype}) \to \mathsf{sk} \cup \bot$: corrupts $P \in \mathcal{P}$, giving $\mathcal{A}$ P's long-term key of type ktype[7].

[6] We assume that each party in $\mathcal{P}$ has a unique identifier.

[7] In MARSHAL this could be either the identity or the signature key.

$\mathsf{oStart}(P, \mathsf{role}, \mathsf{pid}) \to \pi_P^i \cup \perp$: creates a new instance of an existing honest party with the specified role, by running `aStart`.

$\mathsf{oReveal}(\pi_P^i, \mathsf{ktypes}, s) \to \mathsf{key.set} \cup \perp$: for stage s, it leaks the set $\mathsf{key.set}$ of ephemeral values requested by key type.

$\mathsf{oAccessSK}(P, \mathsf{fct}, \mathsf{q.input}) \to \perp \cup \mathsf{q.rsp}$: gives $\mathcal{A}$ oracle access to a function of one of the long-term keys with the given input. We restrict the space of functions to those naturally occurring in the protocol, *i.e.*, exponentiations of the form h^{ik_P} for ik_P and signatures for the sk_P.

$\mathsf{oTest}_b(\pi_P^i, s) \to \perp \cup K$: for honest parties, valid instances and stages, the oracle yields either the true message key or a random key of the same length (depending on $\pi_P^i.\mathsf{b}[s] = 0$).

$\mathsf{oSend}(\pi_P^i, s, AD, aux) \to \perp \cup (AD^*, aux^*)$: this oracle works in two modes: honest and adversary-driven. For AD, aux equal to $\perp$ and valid other values, then π_P^i ratchets normally for stage s, outputting ratcheting information and auxiliary values (like signatures). For adversarially-chosen AD, aux the oracle uses the given input (as much as possible) to update.

$\mathsf{oReceive}(\pi_P^i, s, AD, aux) \to \perp$: again, this oracle has an honest or adversarial mode, simulating the reception of a message. In honest mode, AD, aux are the correct values output by oSend at stage s by π_P^i's partner. For the adversarial mode, the receiver checks the data and ratchets with the new values if it deems them correct.

Like in Blazy *et al.* [4], $\mathcal{A}$ is not given the true ciphertext, which can help $\mathcal{A}$ distinguish the message keys. However, $\mathcal{A}$ can use oSend and its own, chosen input private keys to force parties to ratchet. Then $\mathcal{A}$ receives AD, a signature on AD (in aux), and other elements like sk_P and a signature on it.

4.1 Post-compromise security game

Our PCS AKE game is parametrized by the security parameter λ and a number $\mathsf{n}_{\mathcal{P}}$ of honest parties. The challenger starts by generating the $\mathsf{n}_{\mathcal{P}}$ honest parties and long-term secrets, giving the public keys and all system parameters to $\mathcal{A}$. The adversary can access all the oracles except oTest. At each new stage of each honest-party instance, the challenger generates a fresh test bit $\pi_P^i.\mathsf{b}[s^\star]$ for that instance and stage.

At some point, $\mathcal{A}$ outputs a party instance $\pi_P^\star$ and a stage $s^\star$. The challenger runs oTest on these inputs, outputting the returned key K (either the real message key of stage $s^\star$or a randomly-chosen key from the same key-space, depending on $\pi_P^i.\mathsf{b}[s^\star]$. Finally, $\mathcal{A}$ outputs a bit d, which is the adversary's guess for $b^\star$. We say $\mathcal{A}$ *wins* the experiment if $d = b^\star$ and if the adversary has played the game such that the active danger event $\mathsf{Act.Dan}(\mathsf{sid}, \mathsf{mk}^{\mathsf{x}^*,\mathsf{y}^*})$ as defined at the end of this section is not triggered. If the adversary terminates without outputting d, the challenger picks d uniformly at random, treating it as $\mathcal{A}$'s final output.

Definition 2 (PCS-AKE security). *Let Π be an asynchronous messaging protocol. Π is said to be* PCS-AKE secure *if for any polynomial time adversary $\mathcal{A}$, the adversary's advantage is negligibly close to 0 as a function of the security parameter. We define:*

$$\mathsf{Adv}_{\Pi}^{\mathsf{PCS-AKE}}(\mathcal{A}) := \left|\mathbb{P}[\mathcal{A} \textit{ wins } \mathsf{Exp}_{\Pi}^{\mathsf{PCS\text{-}AKE}}(\lambda, \mathcal{A})] - \frac{1}{2}\right|$$

```
$\underline{\mathsf{Exp}_{\Pi}^{\mathsf{PCS\text{-}AKE}}(\lambda, \mathcal{A})}$

$(\mathcal{P} = \{P_1, \cdots P_{\mathsf{n}_\mathcal{P}}\}) \leftarrow \mathcal{C}(\lambda, \mathsf{n}_\mathcal{P})$
$(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \mathcal{C}^{\mathtt{aKGen}(1^\lambda)} \quad \forall i \in \{1, \cdots, \mathsf{n}_\mathcal{P}\}$
$\mathcal{P}^* \leftarrow \emptyset$
$\mathcal{O}_{\mathsf{PCS}} \leftarrow \left\{ \begin{array}{l} \mathsf{oUReg}(\cdot,\cdot), \mathsf{oCorrupt}(\cdot,\cdot), \mathsf{oStart}(\cdot,\cdot,\cdot), \mathsf{oReveal}(\cdot,\cdot,\cdot), \mathsf{oSend}(\cdot,\cdot,\cdot,\cdot), \\ \mathsf{oReceive}(\cdot,\cdot,\cdot,\cdot), \mathsf{oAccessSK}(\cdot,\cdot,\cdot), \mathcal{RO}_1(\cdot), \mathcal{RO}_2(\cdot) \end{array} \right\};$
$(\pi_P^\star, s^\star) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{PCS}}}(1^\lambda)$
$K \leftarrow \mathsf{oTest}_{b^\star}(\pi_P^\star, s^\star)$
$d \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{PCS}}}(\lambda, \mathsf{n}_\mathcal{P}, K)$

$\boxed{\mathcal{A} \text{ wins iff. } d = b^\star \text{ and } \neg\mathsf{Act.Dan}(\pi_P^\star.\mathsf{sid}, \pi_P^\star.\mathsf{mk}^{\mathsf{s}^*})}$
```

Fig. 4: Description of the PCS AKE game, denoted $\mathsf{Exp}_{\Pi}^{\mathsf{PCS\text{-}AKE}}(\lambda, \mathcal{A})$ between adversary $\mathcal{A}$ and chalenger $\mathcal{C}$. The game is parametrized by the security parameter λ and the number of honest parties $\mathsf{n}_\mathcal{P}$. The set of accessible oracles by $\mathcal{A}$ is denoted $\mathcal{O}$.

Trivial attacks. We retroactively restrict $\mathcal{A}$'s otherwise full control of the oracles to rule out attacks that trivially break security. For instance, the oTest oracle is queried on a malicious party, it will win right away, since the protocol is designed to let both endpoints compute message keys. In general, such restrictions to the adversary indicate potential weaknesses in the protocol (which the adversary exploits): the more restrictions, the weaker the protocol's security.

In Signal querying $\mathsf{oReveal}$ for the chain and ratchet keys used at stage $s = (x, y)$ allows $\mathcal{A}$ to know: *all* chain and message keys for stages (x^*, y) with $x^* \geq x$; *and* all chain and message keys at stages $(x^*, y + 1)$ with $x \geq 1$. Moreover, $\mathcal{A}$ can hijack the conversation, using $\mathsf{oReceive}$ to learn future keys. These attacks are weaknesses specific to Signal, but they do not translate to MARSHAL. In fact, MARSHAL provides strictly stronger security than Signal.

Winning conditions. Recall that the adversary's goal is to guess the test bit correctly for the stage and instance queried to oTest. However, we need to rule out trivial attacks, which we present as predicates (conjunctions and disjunctions of Boolean events). Each Boolean event is called a *danger* to a specific value, and we divide these into passive (compromising a party's state, but without hijacking) and active ($\mathcal{A}$ will also attempt hijacking) attacks.

Say $\mathcal{A}$ has queried oTest for instance π_A^i of initiator Alice, and stage $s^* = (x^*, y^*)$ (the winning conditions are mirrored for Bob). Assume π_A^i has session identifier sid, and let π_B^j be an honest instance with which π_A^i has matching conversation (party B is the partner of π_A^i).

We first list trivial attacks components (dangers) in passive attacks:

- $\mathsf{Danger}(\pi_A^i)$ is the event that either $A \in \mathcal{P}^*$ *or* $\pi_A^i.\mathsf{pid} \in \mathcal{P}^*$;

- $\mathsf{Danger}(\mathsf{sid}, \mathsf{ms})$ is the event that either ms is queried directly to $\mathsf{oReveal}$ for π_A^i *or* π_B^j, *or* that all these four dangers are triggered:
  - $\mathsf{Danger}(\mathsf{prepk}_B^{\mathsf{ik}_A})$: the event that $\mathcal{A}$ corrupts A *or* that it reveals the key prek_B, *or* that it learns the exponentiation result from $\mathsf{oAccessSK}$.
  - $\mathsf{Danger}(\mathsf{sid}, \mathsf{ipk}_B^{\mathsf{ek}_A})$: the event that $\mathcal{A}$ corrupts B to learn its identity key *or* that it learns the key ek_A *or* that it learns the exponentiation result through $\mathsf{oAccessSK}$.
  - $\mathsf{Danger}(\mathsf{sid}, \mathsf{prepk}_B^{\mathsf{ek}_A})$: the event that $\mathcal{A}$ reveals either prek_B *or* ek_A.
  - $\mathsf{Danger}(\mathsf{sid}, \mathsf{ephpk}_B{}^{\mathsf{ek}_A})$: the event that $\mathcal{A}$ reveals either ephk_B *or* ek_A.
- $\mathsf{Danger}(\mathsf{sid}, \mathsf{ck}^{1,1})$: the event that $\mathcal{A}$ queries $\mathsf{oReveal}$ for $\mathsf{ck}^{1,1}$ from either π_A^i *or* π_B^j *or* that *both* the following events are triggered:
  - $\mathsf{Danger}(\mathsf{sid}, \mathsf{ms})$ as defined above.
  - $\mathsf{Danger}(\mathsf{sid}, (\mathsf{prepk}_B)^{\mathsf{rchk}^{0,1}})$: the event that $\mathcal{A}$ queries $\mathsf{oReveal}$ on prek_B *or* $\mathsf{rchk}^{0,1}$.
- $\mathsf{Danger}(\mathsf{sid}, \mathsf{ck}^{1,\mathsf{y}})$ for $y > 1$: the event that $\mathcal{A}$ queries $\mathsf{oReveal}$ for $\mathsf{ck}^{1,y}$ and π_A^i *or* π_B^j *or* that *both* the following events occur:
  - $\mathsf{Danger}(\mathsf{sid}, (\mathsf{T}_{\mathsf{y}-1})^{\mathsf{ik}_B})$: the event that $\mathcal{A}$ queries $\mathsf{oReveal}$ on t_{y-1} *or* $\mathsf{oCorrupt}$ on ik_B, *or* $\mathsf{oAccessSK}$ on the exponentiation.
  - $\mathsf{Danger}(\mathsf{sid}, (\mathsf{ipk}_A)^{\mathsf{t}_{\mathsf{y}-1}})$: the event that $\mathcal{A}$ queries $\mathsf{oReveal}$ on t_{y-2} *or* $\mathsf{oCorrupt}$ on ik_A *or* $\mathsf{oAccessSK}$ on the exponentiation.
- $\mathsf{Danger}(\mathsf{sid}, \mathsf{ck}^{\mathsf{x},2\mathsf{m}+1})$ ($x > 1$, chain-index is odd): the event that $\mathcal{A}$ queries $\mathsf{oReveal}$ for $\mathsf{ck}^{x,2m+1}$ to π_A^i *or* π_B^j, *or* that *both* the following events occur:
  - $\mathsf{Danger}(\mathsf{sid}, \mathsf{ck}^{\mathsf{x}-1,2\mathsf{m}+1})$ recursive part ot the defintion.
  - $\mathsf{Danger}(\mathsf{sid}, (\mathsf{Rchpk}^{\mathsf{x}-1,2\mathsf{m}+1})^{\mathsf{ik}_B})$: the event that $\mathcal{A}$ queries $\mathsf{oCorrupt}$ on ik_B *or* $\mathsf{oReveal}$ on $\mathsf{rchk}^{x-1,2m+1}$, *or* $\mathsf{oAccessSK}$ on the exponentiation.
- $\mathsf{Danger}(\mathsf{sid}, \mathsf{ck}^{\mathsf{x},2\mathsf{m}})$ ($x > 1$, chain-index is even): the event that $\mathcal{A}$ queries $\mathsf{oReveal}$ for $\mathsf{ck}^{x,2m+1}$ to π_A^i *or* π_B^j, *or* that *both* the following events occur:
  - $\mathsf{Danger}(\mathsf{sid}, \mathsf{ck}^{\mathsf{x}-1,2\mathsf{m}})$, recursive part ot the defintion.
  - $\mathsf{Danger}(\mathsf{sid}, (\mathsf{Rchpk}^{\mathsf{x}^*-1,\mathsf{y}^*})^{\mathsf{ik}_A})$: the event that $\mathcal{A}$ queries $\mathsf{oCorrupt}$ on ik_A *or* $\mathsf{oReveal}$ on $\mathsf{rchk}^{x-1,2m}$, *or* $\mathsf{oAccessSK}$ on the exponentiation.

Regarding the (passive) danger to the message key we conclude as follows:

- $\mathsf{Danger}(\mathsf{sid}, \mathsf{mk}^{\mathsf{x},\mathsf{y}})$: the event that $\mathcal{A}$ queries $\mathsf{oReveal}$ for $\mathsf{mk}^{x,y}$ to π_A^i *or* π_B^j, *or* that $\mathsf{Danger}(\pi_A^i)$ occurs, *or* that *both* the following events occur:
  - $\mathsf{Danger}(\mathsf{sid}, \mathsf{ck}^{\mathsf{x},\mathsf{y}})$
  - $\mathsf{Danger}(\mathsf{sid}, (\mathsf{Rchpk}^{\mathsf{x}^*-1,\mathsf{y}^*})^{\mathsf{ik}_X})$: the event that $\mathcal{A}$ queries $\mathsf{oCorrupt}$ on ik_X (if y is even, then $X = A$, if y is odd, then $X = B$) *or* $\mathsf{oReveal}$ on $\mathsf{rchk}^{x-1,y}$, *or* $\mathsf{oAccessSK}$ on the exponentiation.

Beside these passive-attack strategies, $\mathcal{A}$ can also *hijack* a session, *i.e.*, obtain credentials from an endpoint and use them to impersonate that endpoint to its partner, inserting its own ratcheting information into the key schedule.

We define the *hijacking* of a session run between π_A^i and its partner π_B^j at some stage $s_h = (x_h, y_h)$ (for which we assume w.l.o.g. that A is the sender) the event that the following conditions hold simultaneously:

– $\mathcal{A}$ has queried $\mathsf{oReceive}(\pi_B^j, s_h, AD_h, aux_h)$;

– the values (AD_h, aux_h) were never output by an $\mathsf{oSend}(\pi_A^i, s_h, \cdot, \cdot)$ query;
– there exists a value $v \in AD_h \cup aux_h$, but such that $v \notin \mathsf{sidsk}_A^i \cup \pi_B^i.\mathsf{var}[s_h]$.

We call stage s_h *successfully hijacked* if, in addition to the conditions above, it also holds that the oReceive query in the first bullet point has yielded an output that is different from $\perp$. Let $\mathsf{Hijack}(\mathsf{sid}, \mathsf{s_h})$ be the event that the adversary *successfully* hijacked a session with identifier sid at stage s_h.

Let $s^* = (x^*, y^*)$ be the stage for which $\mathcal{A}$ has queried oTest. We define the active danger to mk^{x^*,y^*}, denoted $\mathsf{Act.Dan}(\mathsf{sid}, \mathsf{mk}^{\mathsf{x}^*,\mathsf{y}^*})$, as follows:

- $\mathsf{Act.Dan}(\mathsf{sid}, \mathsf{mk}^{\mathsf{x}^*,\mathsf{y}^*})$: is the event that either $\mathsf{Danger}(\mathsf{sid}, \mathsf{mk}^{\mathsf{x}^*,\mathsf{y}^*})$ is triggered, *or* the following events *all* occur:
  - $\mathsf{Hijack}(\mathsf{sid}, \mathsf{s_h})$ was triggered for sid at stage $s_h = (x_h, y_h)$, such that s_h is *prior* to s^*;
  - $\mathcal{A}$ has queried oTest for π_B^j;
  - Depending on whether $y^* - y_h$ is odd or even, precisely one of the following events is triggered:
    - If $y^* - y_h$ is even: either $\mathcal{A}$ has queried oCorrupt on sk_A *or* oAccessSK was queried on the AD input to oReceive cast on π_B^j for stage s^*.
    - If $y^* - y_h$ is odd: $\mathcal{A}$ has queried oCorrupt for ik_A, *or* oReveal on π_B^j for rchk^{x^*,y^*}, *or* oAccessSK on $(\mathsf{Rchpk}^*)^{\mathsf{ik}_A}$ such that $\mathcal{A}$ has received Rchpk^* as part of the AD of B's message at stage s^* (via oSend).

4.2 Message-loss resilience

For the message-loss resilience game we need an additional notion. Let π_P^i be such that $s \in \pi_P^i$ for some stage s. We denote by $\mathsf{next}(\pi_P^i, s)$ the stages reachable for that instance from stage s. This depends on P's role (sender or receiver) at stage $s = (x, y)$. If P was a sender, then $\mathsf{next}(\pi_P^i, s) = \{(x', y) | x' \geq x + 1\}$. If P was a receiver, then $\mathsf{next}(\pi_P^i, s) = \{(x'', y + 1) | x'' \geq 1\}$.

The message-loss resilience game begins like PCS-AKE, by creating $\mathsf{n}_{\mathcal{P}}$ honest parties and their long-term credentials. We provide an illustration of it in Fig. 5. The adversary receives the public keys, then gets access to the oracles oStart, oCorrupt, oReveal, oAccessSK, and a crippled version of oSend and oReceive ($\mathcal{A}$ must always use these oracles in honest mode). The adversary finally stops, outputting a protocol instance/stage pair π_P^i, s^*, for which the challenger must produce the key mk^{s^*} (or $\perp$ if it does not know it). We say $\mathcal{A}$ *wins* if, and only if, the challenger has output $\perp$ and there exists a stage $s \in \pi_P^i$ such that $s^* \in \mathsf{next}(\pi_P^i, s)$.

4.3 The security of MARSHAL

The following theorem describes the security of MARSHAL in terms of PCS-AKE security and MLR-security, as defined in Fig. 4 and Fig 5. This security holds in the random oracle model (we replace the two KDFs by random oracles).

Theorem 1. *If the GDH [14] assumption holds, if our signature scheme is EUF-CMA-secure, then the* MARSHAL *protocol is PCS-AKE secure in the random oracle model (we model the two KDFs as $\mathcal{RO}_1, \mathcal{RO}_2$). In addition,* MARSHAL *is MLR-secure.*

$$\underline{\mathsf{Exp}_{\Pi}^{\mathsf{MLR}}(\lambda, \mathcal{A})}$$

$$(\mathcal{P} = \{P_1, \cdots P_{\mathsf{n}_{\mathcal{P}}}\}) \leftarrow \mathcal{C}(\lambda, \mathsf{n}_{\mathcal{P}})$$
$$(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \mathcal{C}^{\mathtt{aKGen}(1^\lambda)} \quad \forall i \in \{1, \cdots, \mathsf{n}_{\mathcal{P}}\}$$
$$\mathcal{O}_{\mathsf{MLR}} \leftarrow \left\{ \begin{array}{l} \mathsf{oCorrupt}(\cdot,\cdot), \mathsf{oStart}(\cdot,\cdot,\cdot), \mathsf{oReveal}(\cdot,\cdot,\cdot), \mathsf{oSend}^*(\cdot,\cdot,\cdot,\cdot), \\ \mathsf{oReceive}^*(\cdot,\cdot,\cdot,\cdot), \mathsf{oAccessSK}(\cdot,\cdot,\cdot) \end{array} \right\};$$
$$(\pi_{P}^{\star}, s^{\star}) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{MLR}}}(1^\lambda)$$
$$\mathsf{mk} \leftarrow \mathcal{C}$$

$$\boxed{\mathcal{A} \text{ wins iff. } \mathsf{mk} \neq \bot}$$

Fig. 5: Description of the MLR game, denoted $\mathsf{Exp}_{\Pi}^{\mathsf{MLR}}(\lambda, \mathcal{A})$ between adversary $\mathcal{A}$ and chalenger $\mathcal{C}$. The game is parametrized by the security parameter λ and the number of honest parties $\mathsf{n}_{\mathcal{P}}$. Oracles oSend^* and $\mathsf{oReceive}^*$ are the crippled versions of the original oSend and $\mathsf{oReceive}$, restricted to honest mode only.

The proof of this theorem is not technically complex, but includes a lot of special cases (just like the proof of Cohn-Gordon *et al.* [6]). This is a direct consequence of our having excluded only trivial attacks from the winning conditions. However, we note that this was done in order to provide a more direct and honest comparison to the Signal protocol; indeed, with our winning conditions, Signal fails to attain security, whereas MARSHAL can be proved secure.

Proof (Sketch). The first game-hops ensure that there are no collisions between DH key values that are generated honestly, then the challenger must guess the target instance and stage that will be input to the oTest oracle. Note that we do not rely on the security of AEAD (since in fact the authentication of the AD is done also as part of the signature).

At this point, the proof moves "backwards", from the point where the test took place. What makes the proof tedious is that we have allowed the adversary a lot of power in the winning conditions; thus, it is harder to rule out any specific queries *a priori*. We replace the true message key at stage s^* with a random (consistent) one, and must show that this is not detectable by the adversary.

One key observation is that the only hijacking attempts (on the partnering instance) that we worry about for active adversaries *must* occur before the test stage. However, due to the restrictions imposed in $\mathsf{Act.Dan}(\mathsf{sid}, \mathsf{mk}^{\mathsf{s}^*})$, the adversary will not be able to impose its own key (nor compute the message key on a receiving stage) *unless* it is able to either forge a signature on behalf of the hijacked party (if s^* is a sending stage for the hijacked party) or learn a value $(\mathsf{Rchpk}^*)^{\mathsf{ik}}$ where the identity key belongs to the hijacked party. Since our predicate $\mathsf{Act.Dan}(\mathsf{sid}, \mathsf{mk}^{\mathsf{s}^*})$ forbids the adversary from learning the long-term key involved (depending on the nature of that stage), if the adversary does forge the signature or submits an input including the value $(\mathsf{Rchpk}^*)^{\mathsf{ik}}$ to $\mathcal{RO}_2$, we construct reductions to EUF-CMA security and respectively to GDH.

Note that without these two vital ingredients, even if $\mathcal{A}$ has successfully hijacked and controlled the target instance's partner so far, it cannot compute the message key by itself. We proceed to rule out other means for the adversary to distinguish the key from random. Since we have modelled both KDFs as a random oracle, our next step is to rule out an adversary learning the input that the honest party (or parties) use to compute mk^{s^*}. The event $\mathsf{Danger}(\mathsf{sid}, \mathsf{mk}^{\mathsf{s}^*})$ rules out combinations of queries that $\mathcal{A}$ could make that would give it the values directly. We bound the probability that the adversary has managed to input the correct value $(\mathsf{Rchpk}^*)^{\mathsf{ik}}$ to $\mathcal{RO}_2$ without endangering it, by a reduction to GDH. Then we move on to the input chain key and continue working through the particular cases. □

Complexity. The security of MARSHAL comes at a cost compared to Signal. First, each party must generate *2 long-term keys* (instead of 1), and it must also register (and store) *a number of medium-term ratchet keys* (these do not exist in Signal). As precomputation (for multiple sessions), each party also *signs* the signature key with the identity key. We compare computational overhead in terms of three types of computations: session setup (beginning of session to first message keys), sending messages, and receiving messages. Note that in all these stages we *remove the need for root keys*. At session setup, we require *an additional signature*, but otherwise the runtime remains comparable (although our KDFs take larger input). Sending messages adds *one random-value generation*, *two group exponentiations*, and *two signature computations* compared to Signal. Receiving messages adds *one group exponentiation*, and *two signature verifications*. Finally, sent and received messages involve much *larger associated data* than in Signal (the size is linear in the maximal chain depth in order to achieve MLR). Finally, in terms of communication overhead, all messages involve *larger associated data* (linear in the chain depth) and *one (constant-size) signature*. Finally, the first two chains (y=1,2) also serve to transmit *an additional signature*.

5 Conclusion

Our main contribution is providing an alternative design to Signal, which achieves much stronger security properties at comparatively little cost. Unlike alternative approaches to designing ratcheted key-exchange, which follow a modular design (typically based on KEMs), we try to stick close to Signal's original structure, thus showing how that protocol could be modified to achieve better post-compromise security (PCS).

Our protocol departs from the key observation that Signal's comparative lack of PCS is due to the frequency of asymmetric ratchets and lack of persistent authentication. The latter is fixed by adding long-term keys at every new stage. The former is dealt with by adding asymmetric ratchets *at every stage*. To do so, we require a long-term key stored on the semi-trusted Signal server, and we ensure that message-loss resilient is achieved by providing the a list of correct ratcheting keys at every stage of a given chain.

Our protocol's security heals after only one message, even in the presence of a strong, active adversary, assuming that at least one long-term credential remains secure (depending on the stage, this could be the signature or identity keys). This data should therefore be stored separately from ephemeral data, in a secure component.

References

1. Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the signal protocol. In: EUROCRYPT'19 (2019)
2. Anonymous: Full version (2021), `https://drive.google.com/file/d/1fw5bJf_M492dBjgygbFyNL1AdgWcE011/view?usp=sharing`
3. Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: The security of messaging. In: CRYPTO'17 (2017)
4. Blazy, O., Bossuat, A., Bultel, X., Fouque, P., Onete, C., Pagnin, E.: SAID: reshaping signal into an identity-based asynchronous messaging protocol with authenticated ratcheting. In: EuroS&P'19 (2019)
5. Borisov, N., Goldberg, I., Brewer, E.: Off-the-record communication, or, why not to use pgp. In: WPES '04. ACM (2004)
6. Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A Formal Security Analysis of the Signal Messaging Protocol. EuroS&P'17 (2017)
7. Cohn-Gordon, K., Cremers, C.J.F., Garratt, L.: On post-compromise security. In: CSF'16 (2016)
8. Durak, F.B., Vaudenay, S.: Bidirectional asynchronous ratcheted key agreement with linear complexity. In: IWSEC'19 (2019)
9. Jaeger, J., Stepanovs, I.: Optimal channel security against fine-grained state compromise: The safety of messaging. In: CRYPTO'18 (2018)
10. Jost, D., Maurer, U., Mularczyk, M.: Efficient ratcheting: Almost-optimal guarantees for secure messaging. In: EUROCRYPT'19. Springer (2019)
11. Marlinspike, M., Perrin, T.: The double ratchet algorithm (2016), https:// whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf
12. Marlinspike, M., Perrin, T.: Double Ratchet Algorithm. Signal (2016)
13. Marlinspike, M., Perrin, T.: The X3DH Key Agreement Protocol. Signal (2016)
14. Okamoto, T., Pointcheval, D.: The gap-problems: A new class of problems for the security of cryptographic schemes. In: PKC (2001)
15. Poettering, B., Rösler, P.: Towards bidirectional ratcheted key exchange. In: CRYPTO'18 (2018)

A Signal

We briefly describe the Signal protocol, see [6] for more details. Signal can be described in terms of four main steps:

Registration: Each party P registers by uploading on a semi-trusted server a number of (public) keys: a long-term key denoted ipk_P, a medium-term key prepk_P signed with ik_P, and optional ephemeral *public* keys ephpk_P.

Session Setup: Alice wants to initiate communication with Bob. She retrieves Bob's credentials from the server, generates an initial ratchet key-pair $(\mathsf{rchk}_A^1, \mathsf{Rchpk}_A^1)$ and an ephemeral key-pair $(\mathsf{Epk}_A, \mathsf{ek}_A)$, and uses the X3DH protocol [13] to generate an initial shared secret ms (master secret): $\mathsf{ms} := (\mathsf{prepk}_B)^{\mathsf{ik}_A} || (\mathsf{ipk}_B)^{\mathsf{ek}_A} || (\mathsf{prepk}_B)^{\mathsf{ek}_A} || (\mathsf{ephpk}_B)^{\mathsf{ek}_A}$. This value is used in input to a key derivation function (KDF_r), outputting the root key rk_1 and the chain key $\mathsf{ck}^{1,1}$. The latter is used to derive the first message key $\mathsf{mk}^{(1,1)}$ that Alice uses to communicate with Bob. The following associated data (AD) is appended to that message: the value 1 (for the index x), Alice's ephemeral public key Epk_A, the ratchet key Rchpk^1, as well as Alice's and Bob's identities.

Symmetric Ratchet: Whenever a sender P chooses a new message to send, the stage changes from (x, y) to $(x+1, y)$ and a new symmetric ratchet takes place. At stage (x, y), the message key is $\mathsf{mk}^{x,y}$, derived from a stage secret $\mathsf{ck}^{x,y}$. In fact, given $\mathsf{ck}^{x,y}$, the sender computed (at stage $(x-1, y)$) the values $\mathsf{ck}^{x+1,y}$ and $\mathsf{mk}^{x,y}$. At stage $(x+1, y)$, the sender inputs $\mathsf{ck}^{x+1,y}$ to the key-derivation function KDF_m and receives the output $\mathsf{ck}^{x+2,y}$ and $\mathsf{mk}^{x+1,y}$. The key $\mathsf{mk}^{x+1,y}$ is then used for the authenticated encryption of the sender's message at stage $(x+1, y)$. The AD sent at this stage will be the ratchet key Rchpk^{y} and the stage index[8] $x+1$. The same process takes place on the receiving side, in order to authenticate and decrypt messages.

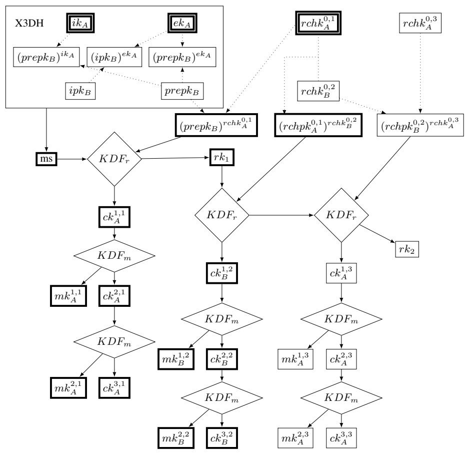


Fig. 6: Effect of a compromise at initial state for Signal protocol (we use the X3DH protocol without the optional fourth value). This figure contains the key-schedule of Signal, with A denoting the initiator Alice and B, the responder Bob. The adversary compromises A during setup. The keys that are now compromised are in bold, black boxes.

Asymmetric Ratchet: If the speaker changes (that is Alice stops sending messages and Bob starts instead), the new speaker inserts fresh Diffie-Hellman elements into the key-derivation. Assume that we are at stage (x, y) and the speaker changes (thus yielding stage $(0, y+1)$). Different computations are made depending on whether the new speaker is the initiator or the responder.

1. First assume that initiator Alice was the speaker at stages $(\cdot, y)$; therefore y is even at each stage $(\cdot, y)$ and the encrypted message included associated data Rchpk^{y}. When Bob comes online, he chooses a new ratchet key rchk^{y+1}, and the public key Rchpk^{y+1} is then computed. A temporary value t and the chain

[8] In the original protocol, the sender also sends the identity public keys of Alice and Bob; since these values are public and constant for all stages, we omit them.

key $\mathsf{ck}^{(0,y+1)}$ are calculated from the root key[9] rk_y and the Diffie-Hellman product $(\mathsf{Rchpk}^y)^{\mathsf{rchk}^{y+1}}$ via KDF_r. Then, the chain and message keys are computed as described in the previous item. From that point onwards, keys evolve by symmetric ratcheting until the speaker changes again.

2. Now assume that the responder was the speaker at stages $(\cdot, y)$; therefore y is odd and at each stage $(\cdot, y)$ the encrypted message includes associated data Rchpk^y. When Alice comes online, she chooses new ratcheting information rchk^{y+1}, Rchpk^{y+1} and computes a new root key rk_{y+1} and the base chain key $\mathsf{ck}^{(0,y+1)}$ from the value t computed at stage $(0, y)$ (see the bullet point before) and the Diffie-Hellman product $(\mathsf{Rchpk}^y)^{\mathsf{rchk}^{y+1}}$. From here the key derivation proceeds as described in the bullet point on symmetric ratcheting.

We depict in Fig. 6 the extent of a full compromise in the case of the Signal protocol. We note that a compromise of Alice's ephemeral values (including the stage-specific ratchet key) leads to two entire chains of messages being leaked.

B Winning conditions

In this appendix we describe the winning conditions in a pictographic way. Due to lack of space, we choose sometimes to only handle one of two alternative situations (for instance just y is odd or y is even, rather than both). The adversary can be a passive or an active adversary, as described in Section 4. The active adversary has all the full capacities (and restrictions in winning conditions) as the passive one, as shown in Fig. 7.

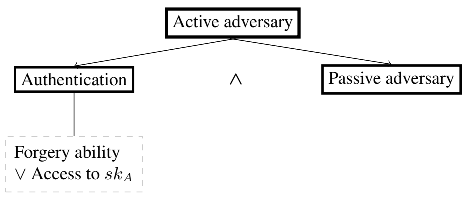


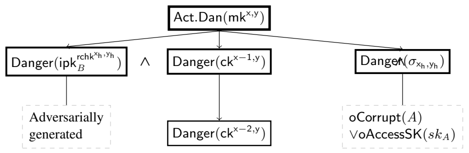
Fig. 7: General description of an active adversary.

Subsequently, we show the conditions under which the adversary successfully hijacks a session at a stage with odd y, in Fig. 8. In Fig. 9 through to Fig. 13 we represent the most important danger events for passive adversaries.

C Proof of Theorem 1

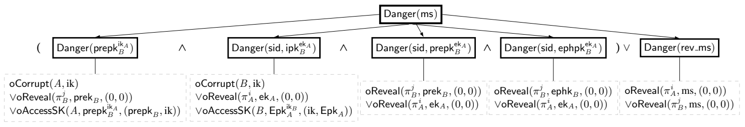
Proof. The security statement is parametrized by the maximal number of stages $\mathsf{n_S}$ run by any given instance, the number of parties generated by the adversary $\mathsf{n_P}$, the number

[9] Root keys are only computed when one reverts back to the initiator, so in our notation, on stages $(0, y)$ for even values of y.



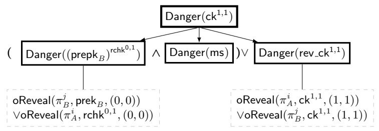
$\mathsf{Act.Dan}(\mathsf{mk}^{x,y})$
$\mathsf{Danger}(\mathsf{ipk}_B^{\mathsf{rchk}^{x_h,y_h}})$
$\wedge$
$\mathsf{Danger}(\mathsf{ck}^{x-1,y})$
$\mathsf{Danger}(\sigma_{x_h,y_h})$
Adversarially generated
$\mathsf{Danger}(\mathsf{ck}^{x-2,y})$
$\mathsf{oCorrupt}(A)$
$\vee\mathsf{oAccessSK}(sk_A)$

Fig. 8: Conditions for $\mathcal{A}$ to hijack the communication at stage $s = (x, y)$ with y_h odd.



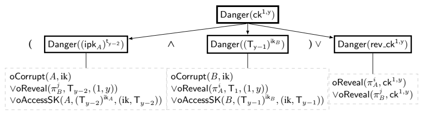
$\mathsf{Danger}(\mathsf{ms})$
$($ $\mathsf{Danger}(\mathsf{prepk}_B^{\mathsf{ik}_A})$ $\wedge$ $\mathsf{Danger}(\mathsf{sid}, \mathsf{ipk}_B^{\mathsf{ek}_A})$ $\wedge$ $\mathsf{Danger}(\mathsf{sid}, \mathsf{prepk}_B^{\mathsf{ek}_A})$ $\wedge$ $\mathsf{Danger}(\mathsf{sid}, \mathsf{ephpk}_B^{\mathsf{ek}_A})$ $)\vee$ $\mathsf{Danger}(\mathsf{rev\_ms})$
$\mathsf{oCorrupt}(A, \mathsf{ik})$ $\vee\mathsf{oReveal}(\pi_B^j, \mathsf{prek}_B, (0,0))$ $\vee\mathsf{oAccessSK}(A, \mathsf{prepk}_B^{\mathsf{ik}_A}, (\mathsf{prepk}_B, \mathsf{ik}))$
$\mathsf{oCorrupt}(B, \mathsf{ik})$ $\vee\mathsf{oReveal}(\pi_A^i, \mathsf{ek}_A, (0,0))$ $\vee\mathsf{oAccessSK}(B, \mathsf{Epk}_A^{\mathsf{ik}_B}, (\mathsf{ik}, \mathsf{Epk}_A))$
$\mathsf{oReveal}(\pi_B^j, \mathsf{prek}_B, (0,0))$ $\vee\mathsf{oReveal}(\pi_A^i, \mathsf{ek}_A, (0,0))$
$\mathsf{oReveal}(\pi_B^j, \mathsf{ephk}_B, (0,0))$ $\vee\mathsf{oReveal}(\pi_A^i, \mathsf{ek}_A, (0,0))$
$\mathsf{oReveal}(\pi_A^i, \mathrm{ms}, (0,0))$ $\vee\mathsf{oReveal}(\pi_B^j, \mathrm{ms}, (0,0))$

Fig. 9: Conditions for $\mathcal{A}$ to learn ms.



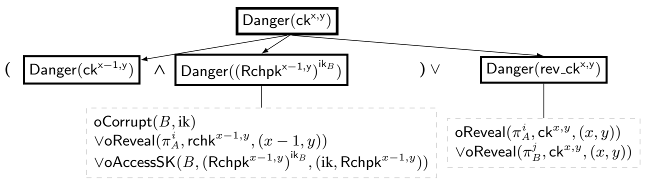
$\mathsf{Danger}(\mathsf{ck}^{1,1})$
$($ $\mathsf{Danger}((\mathsf{prepk}_B)^{\mathsf{rchk}^{0,1}})$ $\wedge$ $\mathsf{Danger}(\mathsf{ms})$ $)\vee$ $\mathsf{Danger}(\mathsf{rev\_ck}^{1,1})$
$\mathsf{oReveal}(\pi_B^j, \mathsf{prek}_B, (0,0))$ $\vee\mathsf{oReveal}(\pi_A^i, \mathsf{rchk}^{0,1}, (0,0))$
$\mathsf{oReveal}(\pi_A^i, \mathsf{ck}^{1,1}, (1,1))$ $\vee\mathsf{oReveal}(\pi_B^j, \mathsf{ck}^{1,1}, (1,1))$

Fig. 10: Conditions for $\mathcal{A}$ to learn $\mathsf{ck}^{1,1}$.



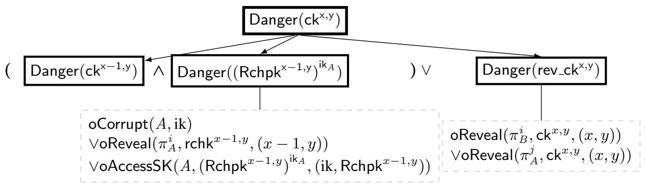
$\mathsf{Danger}(\mathsf{ck}^{1,y})$
$($ $\mathsf{Danger}((\mathsf{ipk}_A)^{\mathsf{t}_{y-2}})$ $\wedge$ $\mathsf{Danger}((\mathsf{T}_{y-1})^{\mathsf{ik}_B})$ $)\vee$ $\mathsf{Danger}(\mathsf{rev\_ck}^{1,y})$
$\mathsf{oCorrupt}(A, \mathsf{ik})$ $\vee\mathsf{oReveal}(\pi_B^j, \mathsf{T}_{y-2}, (1,y))$ $\vee\mathsf{oAccessSK}(A, (\mathsf{T}_{y-2})^{\mathsf{ik}_A}, (\mathsf{ik}, \mathsf{T}_{y-2}))$
$\mathsf{oCorrupt}(B, \mathsf{ik})$ $\vee\mathsf{oReveal}(\pi_A^i, \mathsf{T}_1, (1,y))$ $\vee\mathsf{oAccessSK}(B, (\mathsf{T}_{y-1})^{\mathsf{ik}_B}, (\mathsf{ik}, \mathsf{T}_{y-1}))$
$\mathsf{oReveal}(\pi_A^i, \mathsf{ck}^{1,y})$ $\vee\mathsf{oReveal}(\pi_B^j, \mathsf{ck}^{1,y})$

Fig. 11: Conditions for $\mathcal{A}$ to learn $\mathsf{ck}^{1,y}$.



$\mathsf{Danger}(\mathsf{ck}^{x,y})$
$($ $\mathsf{Danger}(\mathsf{ck}^{x-1,y})$ $\wedge$ $\mathsf{Danger}((\mathsf{Rchpk}^{x-1,y})^{\mathsf{ik}_B})$ $)\vee$ $\mathsf{Danger}(\mathsf{rev\_ck}^{x,y})$
$\mathsf{oCorrupt}(B, \mathsf{ik})$ $\vee\mathsf{oReveal}(\pi_A^i, \mathsf{rchk}^{x-1,y}, (x-1,y))$ $\vee\mathsf{oAccessSK}(B, (\mathsf{Rchpk}^{x-1,y})^{\mathsf{ik}_B}, (\mathsf{ik}, \mathsf{Rchpk}^{x-1,y}))$
$\mathsf{oReveal}(\pi_A^i, \mathsf{ck}^{x,y}, (x,y))$ $\vee\mathsf{oReveal}(\pi_B^j, \mathsf{ck}^{x,y}, (x,y))$

Fig. 12: Conditions for $\mathcal{A}$ to learn $\mathsf{ck}^{x,y}$ where y is odd (*i.e.*, Alice's chain).



$\mathsf{Danger}(\mathsf{ck}^{x,y})$
$($ $\mathsf{Danger}(\mathsf{ck}^{x-1,y})$ $\wedge$ $\mathsf{Danger}((\mathsf{Rchpk}^{x-1,y})^{\mathsf{ik}_A})$ $)\vee$ $\mathsf{Danger}(\mathsf{rev\_ck}^{x,y})$
$\mathsf{oCorrupt}(A, \mathsf{ik})$ $\vee\mathsf{oReveal}(\pi_A^i, \mathsf{rchk}^{x-1,y}, (x-1,y))$ $\vee\mathsf{oAccessSK}(A, (\mathsf{Rchpk}^{x-1,y})^{\mathsf{ik}_A}, (\mathsf{ik}, \mathsf{Rchpk}^{x-1,y}))$
$\mathsf{oReveal}(\pi_B^i, \mathsf{ck}^{x,y}, (x,y))$ $\vee\mathsf{oReveal}(\pi_A^j, \mathsf{ck}^{x,y}, (x,y))$

Fig. 13: Conditions for $\mathcal{A}$ to learn $\mathsf{ck}^{x,y}$ where y is even (*i.e.*, Bob's chain).

of medium-term keys $\mathsf{n_{prek}}$ deployed, and the number of instances n_π created by any given party.

We prove the statement in a sequence of game hops, as follows.

$\mathbb{G}_0$: This is the $\mathsf{Exp}_{\Pi}^{\mathsf{PCS\text{-}AKE}}(\lambda, \mathcal{A})$ described in Section 4. We denote by Adv_0 the maximal advantage an adversary $\mathcal{A}$ to win this game.

$\mathbb{G}_1$: This game is identical to $\mathbb{G}_0$, except, whenever prompted to generate randomness within a protocol step, any honestly-created instance of the protocol will produce unique, random values. In other words, we remove the probability of having a collision for any of the private material, including:
- Long-term keys: ik, sk;
- Medium-term keys: prek, t_0;
- Session-setup ephemeral keys: ek, $\mathsf{rchk}^{0,1}$;
- Stage-specific ephemeral keys: same-user ratchet keys $\mathsf{rchk}^{x,y}$ avec $x \geq 0$, and, at every fresh $y > 0$, a new cross-user ratcheting key t_y.

For ease of notation, we will upper-bound the number of fresh cross-user ratcheting keys by the total number of stages $\mathsf{n_S}$(the bound is tight if we have only one message per chain). Let Adv_1 be the advantage of the adversary $\mathcal{A}$ in this game.
It holds that:

$$\mathsf{Adv}_0 \leq \frac{\binom{\mathsf{n_P}+\mathsf{n_P}\cdot\mathsf{n_{prek}}+\mathsf{n}_\pi\cdot(2+2\mathsf{n_S})}{2}}{q} + \mathsf{Adv}_1 \ .$$

$\mathbb{G}_2$: This game is identical to $\mathbb{G}_1$, except that the challenger guesses and outputs (privately w.r.t. the adversary) a tuple consisting of an instance π_P^i and a stage index s^*. The game is lost if these do not coincide with the tuple output by the adversary to oTest. Let the advantage of the adversary in this hop be Adv_2. Then:

$$\mathsf{Adv}_1 \leq \frac{1}{\mathsf{n_P}\mathsf{n}_\pi\mathsf{n_S}}\mathsf{Adv}_2.$$

Note that the challenger's guess π_P^i, s^* gives it more information than just the target instance and stage. In particular, the challenger now knows: the target party P, the role of P in the target session (initiator/responder), the role of party P in π_P^i at stage s^* (sender/receiver), as well as the role of P at every prior or future stage in that conversation.

$\mathbb{G}_3$: This game hop is identical to $\mathbb{G}_2$ except that the challenger will now refuse to answer the adversary's hijacking queries ($\mathsf{oReceive}$ queries with adversarially-chosen input substituted for output of honest oSend queries) for the following queries:
- If $\mathcal{A}$ queries $\mathsf{oReceive}$ maliciously for an instance of some party other than P;
- If $\mathcal{A}$ queries $\mathsf{oReceive}$ maliciously for an instance of P other than π_P^i;
- If $\mathcal{A}$ queries $\mathsf{oReceive}$ maliciously for the target instance π_P^i for a stage s that comes after s^*;
- If $\mathcal{A}$ queries $\mathsf{oReceive}$ maliciously for π_P^i, such that: s precedes s^* and π_P^i already has a message key mk set for the target stage or any other stage in that chain at the time of the $\mathsf{oReceive}$ query.

We argue that the adversary's advantage in this game, denoted Adv_3, is undiminished with respect to $\mathbb{G}_2$, because: (1) as of $\mathbb{G}_1$ we have unique session identifiers;

(2) we are using random oracles for the key derivation; (3) once key material is accepted for the target stage (or an ulterior one in the same stage), that automatically sets the ratcheting information, which cannot be reset by the malicious $\mathsf{oReceive}$ query. Thus:

$$\mathsf{Adv}_2 = \mathsf{Adv}_3;$$

Note that, starting from this game, the only hijacking attempts that will work are those for the target instance, for stages prior to the target stage.

$\mathbb{G}_4$: This game behaves identically to the previous game, except that the adversary instantly loses (the game returns a random bit) if the following conditions hold:

- The adversary successfully hijacks the session run by π_P^i by a malicious $\mathsf{oReceive}$ query to π_P^i at some stage s_h prior to s^*;
- The adversary has not triggered $\mathsf{Danger}(\pi_P^i.\mathsf{sid}, \mathsf{mk}^{\mathsf{s}^*})$.

We claim that the advantage of the adversary in this game, denoted Adv_4 is such that:

$$\mathsf{Adv}_3 \leq \mathrm{MAX}\big[\mathsf{Adv}_{\mathsf{Sign}}^{\mathsf{EUF-CMA}}(\mathcal{B}), \mathsf{Adv}^{\mathsf{GDH}}(\mathcal{C})\big] + \mathsf{Adv}_4,$$

for reductions $\mathcal{B}$ against the unforgeability of the signature scheme and $\mathcal{C}$ against the Gap-DH problem.

To understand this claim, we first note that the adversary can find itself in one of two situations: at target stage s^*, the target instance π_P^i is either the sender, or the receiver. The two situations are mutually exclusive, and as soon as the challenger guesses the target instance and stage, it will know which of those situations it is in. Moreover, note that once successful hijacking is achieved, the instance π_P^i will no longer be partnered with its honest partner (which $\mathcal{A}$ has successfully impersonated), but rather, with the adversary itself.

We now consider the two options described above.

Suppose first that π_P^i is the receiver at stage s^*. In this case, we can construct a reduction $\mathcal{B}$ to the unforgeability of the signature scheme. This reduction generates all the private keys with the exception of sk_Q, where $Q = \pi_P^i.\mathsf{pid}$. Then the reduction simulates the game faithfully, querying its signature oracle whenever it needs a signature on behalf of Q.

In order to win its game $\mathcal{A}$ will need to query oTest for the honest instance π_P^i at stage s^*. Suppose that $\mathcal{A}$ has not queried $\mathsf{oReceive}$ for π_P^i at stage s^*. In this case, our reduction will fail, but so will $\mathcal{A}$, since π_P^i will have no key mk^{s^*} set. Since π_P^i no longer has an honest partner, $\mathcal{A}$ cannot receive honest input from that partner. As a result, the only way for the adversary to make P ratchet to stage s^* is to produce a valid message at stage s^* (since P is the receiver at that stage), including a valid signature using sk_Q. Note also that $\mathcal{A}$ may not simply query corrupt Q for that signature key (a query that $\mathcal{B}$ would not be able to respond to), nor can it use $\mathsf{oAccessSK}$ (since either of those actions would trigger the active danger event). Hence, the adversary's only choice is to forge the signature.

We have two situations. Either $\mathcal{A}$ does not produce that message/signature pair – in which case, both $\mathcal{A}$ and $\mathcal{B}$ lose, or $\mathcal{A}$ does produce the message/signature pair, in which case $\mathcal{B}$ can forward it, and wins.

Now we turn to the other situation, in which π_P^i is the sender at the target stage s^*. In this case, we construct an adversary $\mathcal{C}$ against the GDH problem that wins with at least as much probability as $\mathcal{A}$ wins its game.

The reduction $\mathcal{C}$ will simulate the game correctly, except that it will embed, as it elements g^a, g^b, the public identity key of Q (pk_Q) and the public ratcheting key Rchpk^{s^*} (forwarded by P at stage s^*). We note that the only way the adversary can distinguish mk^{s^*} from random is if it queried input that includes $(\mathsf{Rchpk}^{s^*})^{sk_Q}$ to the random oracle $\mathcal{RO}_2$. Note that the reduction generates all the other keys, and uses its DDH oracle to ensure consistency with respect to the challenge elements. When the adversary queries $\mathcal{RO}_2$ with an input that allows the DDH oracle to return 1, the reduction forwards that input as its guess for g^{ab}.

We have two cases. Either $\mathcal{A}$ never queries a correct input to $\mathcal{RO}_2$, in which case both $\mathcal{A}$ and $\mathcal{C}$ fail, or $\mathcal{A}$ does query the correct input, in which case $\mathcal{C}$ wins.

This gives the required bound.

Note that we have now effectively ruled out active attacks made by the adversary. We can now focus on only passive attacks.

From this point on, we will have to move through the options given in the winning conditions, starting from the target stage.

$\mathbb{G}_5$: This game is identical to the previous one, except the adversary loses (the game returns a random bit) if it does not trigger the event $\mathsf{Danger}(\pi_P^i.\mathsf{sid}, (\mathsf{pk}_R)^{\mathsf{rchk}^{s^*}})$, where R is the receiving party in the target session at stage s^* (thus, R could be either P or its partner). In the following we show that except for breaking the GDH problem, the adversary's view of the two games is identical.

Let Adv_5 be the adversary's advantage in $\mathbb{G}_5$. It holds that:

$$\mathsf{Adv}_4 \leq \mathsf{Adv}^{\mathsf{GDH}}(\mathcal{D}_0) + \mathsf{Adv}_5,$$

where $\mathcal{D}_0$ is an adversary that breaks the GDH problem.

The reduction is very similar to the second case of game $\mathbb{G}_4$ (reduction $\mathcal{C}$). The adversary cannot simply leak the value of mk^{s^*} (as per the winning conditions) and it cannot perform an active attack (as per the previous game). Its only option is to query $\mathcal{RO}_2$ for the correct input that yields mk^{s^*}, including the value $(pk_R)^{\mathsf{rchk}^{s^*}}$. The reduction only has to embed its challenge tuple into that value and return it when $\mathcal{A}$ queries it to the random oracle.

As of this game, we can therefore assume that the adversary $\mathcal{A}$ triggers the event $\mathsf{Danger}(\pi_P^i.\mathsf{sid}, (\mathsf{pk}_R)^{\mathsf{rchk}^{s^*}})$. Note that, as per the winning conditions, the adversary cannot now *also* trigger the event $\mathsf{Danger}(\pi_P^i.\mathsf{sid}, \mathsf{ck}^{s^*})$. In addition: honest instances cannot produce ratcheting keys duplicating $\mathsf{rchk}^{(x^*-(j-5),y^*)}$(as per $\mathbb{G}_1$), and while pk_R will be used in other instances, the reduction knows the other share in the DH product. Finally, note that in order to win, the adversary *must* input a value for ck^{s^*} to $\mathcal{RO}_2$.

$\mathbb{G}_6$-$\mathbb{G}_{5+x^*}$: At each subsequent game hop $\mathbb{G}_{5+j}$, with $j \in \{1, \ldots, x^* - 1\}$, we move one step backwards along the x axis of the challenge stage $s^* = (x^*, y^*)$. At each step, the adversary will lose (and the game will return a random bit) if the event

$\mathsf{Danger}(\pi_P^i.\mathsf{sid}, (\mathsf{pk}_R)^{\mathsf{rchk}^{(x^*-j,y^*)}})$ is not triggered, where j is the number of the game hop. For each game we use the same argument as above to prove:

$$\mathsf{Adv}_j \leq \mathsf{Adv}^{\mathsf{GDH}}(\mathcal{D}_j) + \mathsf{Adv}_{j+1}.$$

Here, $\mathcal{D}_j$ is at each time a new reduction ($j \in \{1, \ldots, x^* - 1\}$) against the GDH problem. The latter will be embedded at each time in the input to $\mathcal{RO}_2$, which is of the form: $(pk_R)^{\mathsf{rchk}^{(x^*-j,y^*)}}$.
After every individual game hop, the conclusion is the same: in order to produce the correct input to $\mathcal{RO}_2$, the adversary somehow has to have a correct value for the chain key – without triggering the danger event to that value.

$\mathbb{G}_{6+x^*}$-$\mathbb{G}_{6+\mathsf{n_S}}$: In these games we continue making reductions to Gap-DH as we move backwards through the various stages (chains $y^* - 1, y^* - 2, \ldots, 1$). Each time we lose a term $\mathsf{Adv}^{\mathsf{GDH}}(\mathcal{D}_j)$, for $j \in \{x^*, \ldots \mathsf{n_S}\}$.

$\mathbb{G}_{7+\mathsf{n_S}}$: This game hop consists of case-by-case reductions for the security of the master secret ms, similar to the analysis given by Cohn-Gordon *et al.* in Appendix C1 of the full version of their Signal-analysis paper. We notably replace this master secret by a consistent, but random value. For each case, we focus on which values are *not endangered* by the adversary (the winning conditions will not allow the adversary, by that point in the game, to know all the parts). In each case, we will lose a term equivalent to a reduction to GDH.

$\mathbb{G}_{8+\mathsf{n_S}}$: In this game hop we replace all the chain keys from $\mathsf{ck}^{1,1}$ to ck^{s^*} by consistent, but random values. We argue that, due to the previous gamehops, the adversary has no means of endangering this value, and as such, we can guarantee that no such input has been made to the random oracles. At this point the adversary's advantage will be $\frac{1}{2}$, thus concluding the proof.