MARSHAL: Messaging with Asynchronous Ratchets and Signatures for faster HeALing

Olivier Blazy, Pierre-Alain Fouque, Thibaut Jacques, Pascal Lafourcade, Cristina Onete, Léo Robert

► **To cite this version:**

Olivier Blazy, Pierre-Alain Fouque, Thibaut Jacques, Pascal Lafourcade, Cristina Onete, et al.. MARSHAL: Messaging with Asynchronous Ratchets and Signatures for faster HeALing. ACM Symposium on Applied Computing, Apr 2022, Virtual, Czech Republic. pp.1-8, 10.1145/3477314.3507044 . hal-03510612

HAL Id: hal-03510612

https://uca.hal.science/hal-03510612

Submitted on 4 Jan 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

MARSHAL: Messaging with Asynchronous Ratchets and Signatures for faster HeALing*

Olivier Blazy
LIX, CNRS, Inria, École Polytechnique, Institut Polytechnique de Paris
91120 Palaiseau, France
olivier.blazy@polytechnique.edu

Pierre-Alain Fouque
IRISA
Rennes, France
Pierre-Alain.Fouque@ens.fr

Thibaut Jacques
Orange Labs, IRISA, XLIM
Rennes, France
thibaut.jacques@orange.com

Pascal Lafourcade
Université Clermont-Auvergne, CNRS, Mines de Saint-Étienne, LIMOS
Clermont-Ferrand, France
pascal.lafourcade@uca.fr

Cristina Onete
XLIM
Limoges, France
cristina.onete@gmail.com

Léo Robert
Université Clermont-Auvergne, CNRS, Mines de Saint-Étienne, LIMOS
Clermont-Ferrand, France
leo.robert@uca.fr

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. Our implementation results show that, while slower than Signal (as expected), MARSHAL's spectacular gain in healing speed comes at a surprisingly low cost, with individual stages (including key-derivation, encryption, and decryption) taking less than 6 ms.

CCS CONCEPTS

• **Security and privacy** → **Public key (asymmetric) techniques**; **Security protocols**;

KEYWORDS

Secure messaging, Signal, Healing, E2E encryption.

ACM Reference format:
Olivier Blazy, Pierre-Alain Fouque, Thibaut Jacques, Pascal Lafourcade, Cristina Onete, and Léo Robert. 2022. MARSHAL: Messaging with Asynchronous Ratchets and Signatures for faster HeALing. In *Proceedings of The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, , April 25–29, 2022 (SAC '22)*, 8 pages.
https://doi.org/10.1145/3477314.3507044

*This study was partially supported by the French ANR, grants 16-CE39-0012 (SafeTLS) and 18-CE39-0019 (MobiS5). There are also the French government research program "Investissements d´Avenir" through the IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25), the IMobS3 Laboratory of Excellence (ANR-10-LABX-16-01), the French ANR project DECRYPT (ANR-18-CE39-0007) and SEVERITAS (ANR-20-CE39-0009).

1 INTRODUCTION

Asynchronous messaging protocols like Signal [13] or OTR [6] 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) [9]. 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 [8] under certain assumptions.

Currently used in messaging apps, Signal is designed for long-term conversations between two peers. In fact, implementations of this protocol tend to create a single session between the two peers, which will last for the entire lifetime of their communication. This is why post-compromise security is such an important property in Signal: if one peer is compromised and thus the confidentiality of its conversations fails, PCS guarantees that in the near future, that confidentiality will heal and become as strong as before the attack. Clearly, the faster the conversation is able to heal, the more useful this protocol will prove in real life.

Signal's healing ability follows from the gradual insertion of fresh Diffie-Hellman material into the current session secrets, which is done every time speakers change. This provides healing once speakers have changed twice, assuming the adversary has not inserted cryptographic material in the session, thus hijacking it. We aim to do better, providing healing within one single message.

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.

[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 [8].

Each stage (x, y) is associated with a message key $\mathrm{mk}^{x,y}$, used to encrypt and authenticate that stage's message. To evolve from a key $\mathrm{mk}^{x,y}$ to $\mathrm{mk}^{x+1,y}$ (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 $\mathrm{mk}^{x,y}$ to $\mathrm{mk}^{1,y+1}$ (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.* [5]) and the frequency of asymmetric ratchets, which is our key motivation.

Persistent authentication [5]. 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 $\mathrm{Rchpk}_{\mathcal{A}}$ 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 main contribution in this paper is a protocol called MARSHAL (for "Messaging with Asynchronous Ratchets and Signatures for faster HeALing"). The MARSHAL protocol reuses Signal as a backbone, but we modify the way ratcheting is done to provide both persistent authentication and faster healing.

In terms of its security properties, MARSHAL outperforms Signal. Healing occurs within one message after full compromise of a user's session-specific information. We also restrict session hijacking attacks by adding persistent authentication at each message.

In MARSHAL, the parties asymmetrically ratchet at every stage, even when the speaker has not changed. This causes two technical challenges. First, there is the 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.

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. In addition, 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 (*e.g.*, message and/or chain key).

These strong properties of MARSHAL 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.

Another aspect of our protocol that deserves some discussion is the requirement of a trusted execution environment in which the long-term identity keys are stored. A reason users might want to keep those keys safer than others is that they are meant to last a long time. But, one might wonder, if users are assumed to store some of their keys, why not all? In the interest of fairness, we state that storing long-term keys in the trusted execution environment provides us with the attractive properties we advertise: its almost-instant healing and persistent authentication. In our model, we treat the corruption of long-term keys differently than we do the leakage of ephemeral information. One reason is that storing session- and stage-specific keys of all that user's sessions may result in a large quantity of data that needs storing, updating, deleting, etc., which is something we'd like to avoid. Additionally, note that in itself, the same strategy would not enhance the security of Signal, for which the long-term keys and master secret value are only used during session initialization, for the computation of the master secret. Thus, our additional hypothesis of the existence of a trusted execution environment does not give us an unfair advantage in comparison to Signal.

In order to assess the comparative complexity of our protocol with respect to Signal, we implemented MARSHAL in Java, using libsignal. Our implementation results, described in detail in Section 4, show that our protocol is comparatively slower than Signal (as expected); however, all the individual stages of communication remain fast in terms of absolute runtime. This is particularly encouraging since the security gain of MARSHAL is massive, allowing a compromised user's keys to heal within a single stage.

Related Work. Ratcheted key-exchange (RKE) was introduced as a unidirectional, single-move primitive by Bellare *et al.* [4], who used it to define and instantiate ratcheted encryption. This security model was later extended by work such as [11, 16] 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 [5, 8].

[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.

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

Figure 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.

The work of Alwen *et al.* [3] 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 (as in [5]). 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, 12] 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 [16]). 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.* [5], 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.

An interesting work, but which is orthogonal to ours was presented at CCS '19 by Chase et al. [7]. They focus on the long-term keys of two parties, and present a way of updating those in case access is lost to a user's current account. We did not consider updates to long-term key in this paper; instead we focus on the actual session and message keys.

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 [8] 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.*, , $\mathrm{mk}^{1,1}$. In this paper we assume that all signature schemes involve hashing, and omit the hashing in the notation, *i.e.*, $\mathrm{SIGN}_{\mathrm{sk}}(m) := \mathrm{Sign}_{\mathrm{sk}}(H(m))$ for a hash function H. We use the notations $\mathrm{AEAD.Enc}$ and $\mathrm{AEAD.Dec}$ for encryption and decryption respectively of an AEAD-scheme. Finally, for simplification, we use the notation HKDF (*HMAC Key Derivation Function*) to represent several key derivation functions taking either one input or two inputs.

The Signal Protocol. We briefly describe the Signal protocol, see [8] for 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 $(\mathrm{rchk}_A^1, \mathrm{Rchpk}_A^1)$ and an ephemeral key-pair $(\mathrm{Epk}_A, \mathrm{ek}_A)$, and uses the X3DH protocol [14] to generate an initial shared secret ms (master secret): $\mathrm{ms} := (\mathrm{prepk}_B)^{\mathrm{ik}_A} || (\mathrm{ipk}_B)^{\mathrm{ek}_A} || (\mathrm{prepk}_B)^{\mathrm{ek}_A} || (\mathrm{ephpk}_B)^{\mathrm{ek}_A}$. This value is used in input to a key derivation function (KDF_r), outputting the root key rk_1 and the chain key $\mathrm{ck}^{1,1}$. The latter is used to derive the first message key $\mathrm{mk}^{1,1}$ 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.

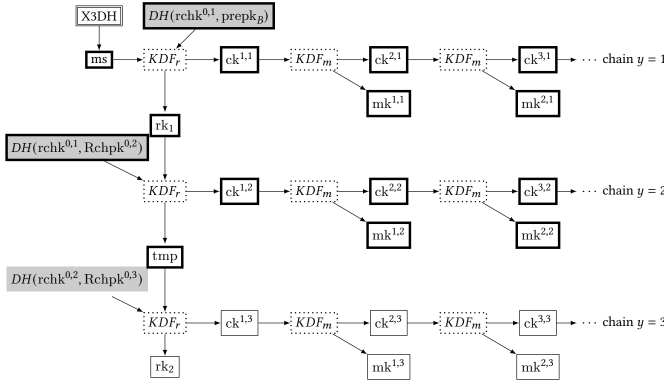

Figure 2: The key schedule of Signal where the DH values are marked with grey boxes. Each stage (x, y) has its x-coordinate corresponding to a message (horizontal moves) inside a chain (vertical moves) for y-coordinate. The compromise (in double edge) during A setup is affecting the keys (in thick edge) of the two first chains.

Symmetric Ratchet. Whenever a sender P chooses a new message to send, the stage changes from (x, y) to $(x + 1, y)$. At stage (x, y), the message key is $\text{mk}^{x,y}$, derived from $\text{ck}^{x,y}$. In fact, given $\text{ck}^{x,y}$, the sender computed (at stage $(x - 1, y)$) the values $\text{ck}^{x+1,y}$ and $\text{mk}^{x,y}$. At stage $(x+1, y)$, the sender inputs $\text{ck}^{x+1,y}$ to the key-derivation function KDF_m and receives the output $\text{ck}^{x+2,y}$ and $\text{mk}^{x+1,y}$. The key $\text{mk}^{x+1,y}$ 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[5] $x + 1$. The same process takes place on the receiving side, in order to authenticate and decrypt messages.

Asymmetric Ratchet. If the speaker changes, 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 computes a new ratchet key pair $(\text{rchk}^{y+1}, \text{Rchpk}^{y+1})$. A temporary value t and the chain key $\text{ck}^{(0,y+1)}$ are calculated from the root key[6] rk_y and the Diffie-Hellman product $(\text{Rchpk}^y)^{\text{rchk}^{y+1}}$ 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 $\text{rchk}^{y+1}, \text{Rchpk}^{y+1}$ and computes a new root key rk_{y+1} and the base chain key $\text{ck}^{(0,y+1)}$ from the value t computed at stage $(0, y)$ (see the bullet point before) and the Diffie-Hellman product $(\text{Rchpk}^y)^{\text{rchk}^{y+1}}$. From here the key derivation proceeds by symmetric ratcheting.

We depict in Fig. 2 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.

[5] 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.

[6] 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.

3 THE MARSHAL PROTOCOL

The protocol we propose, MARSHAL, runs –like Signal– in several stages: registration, session setup, and communication. We describe in Fig. 3 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 $\text{Rchpk}^{2,1}$ 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.

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 $\text{Rchpk}^{0,1}$ 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 [13], 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^0_B, denoted in short T_0. Having received those keys, A generates its own ephemeral key ek_A. The master secret *ms* is a concatenation of DH values, as computed in Signal.

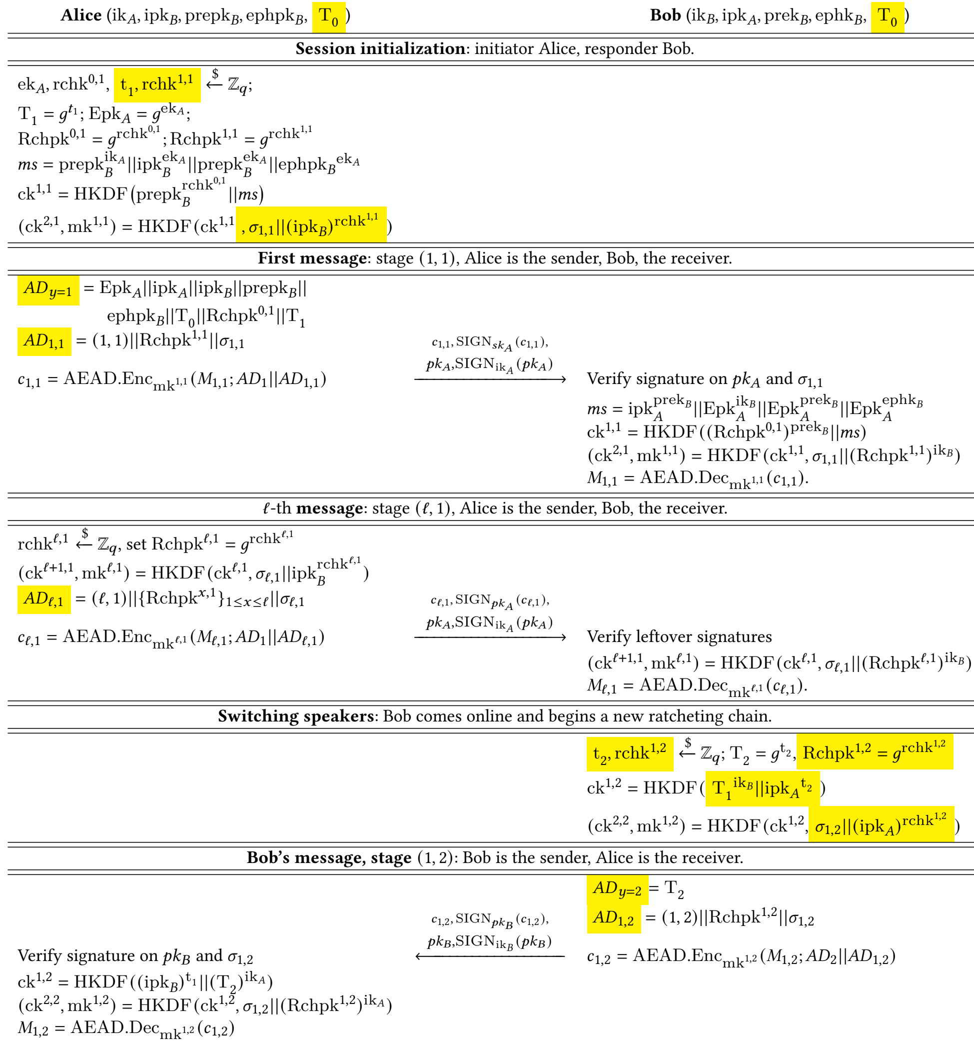

Figure 3: MARSHAL protocol execution between Alice and Bob for the first few stages. The yellow boxes indicate modifications with respect to Signal protocol [8]. The transmitted data is also different and not in yellow for more clarity.

First keys. Alice randomly generates a same-user ratchet keypair $(\text{rchk}^{0,1}, \text{Rchpk}^{0,1})$. 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 $\text{ck}^{1,1}$.

Signature keys. A also needs a signature key (Section 3.1). We choose to use a second pair of signature keys, (sk_P, pk_P). If desired, these keys *could* coincide with $(\text{ik}_P, \text{ipk}_P)$– however, this is not compulsory. By differentiating those two pairs of keys, we allow future implementations to be somewhat agnostic of the underlying mathematical structure of the signature keys (whereas this is impossible for identity-keys, whose structure must support, *e.g.*, group exponentiations/scalar multiplications). Moreover, we limit the load on the centralized PKI server by not including the signature keys amongst the credentials stored for each party; instead users can authenticate those keys at session initialisation.

3.3 Communication phase

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 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 $(\text{rchk}^{\ell,m}, \text{Rchpk}^{\ell,m})$ which will be used to generate chain and message keys for stage $\text{mk}^{\ell+1,m}$. To account for out-of-order messages the concatenation of all the public ratchet keys is included 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: $\mathrm{AD}_{y=m}$ and $\mathrm{AD}_{\ell,m}$. 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. 3 and 4).

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

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

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 $\mathrm{AD}_{y=1}$ includes public identity keys of Alice and Bob, medium-term and ephemeral keys of Bob as recovered by Alice from server, 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 $\mathrm{Rchpk}^{0,1}$, and its first cross-user ratchet key t_1. Finally, the stage-specific data contains: stage index $(1, 1)$ and same-user ratcheting public key $\mathrm{Rchpk}^{1,1}$. Alice computes $c_{1,1} = \mathrm{AEAD.Enc}_{\mathrm{mk}^{1,1}}(M_{1,1}; \mathrm{AD}_{y=1}||\mathrm{AD}_{1,1})$ 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 $\mathrm{t}_1, \mathrm{T}_1$, $\mathrm{AD}_{y=1}$, ratcheting material $\mathrm{Rchpk}^{1,1}, \mathrm{Rchpk}^{2,1}, \ldots, \mathrm{Rchpk}^{\ell-1,1}$, and the key $\mathrm{ck}^{l,1}$, Alice generates new same-user ratcheting key $\mathrm{rchk}^{\ell,1}$ and computes $\mathrm{Rchpk}^{\ell,1} = g^{\mathrm{rchk}^{\ell,1}}$. The key update relies on both long-term keys, for persistent authentication, and this same-user ratcheting key, for healing:

$(\mathrm{ck}^{\ell+1,1}, \mathrm{mk}^{\ell,1}) \leftarrow \mathrm{HKDF}(\mathrm{ck}^{\ell,1}, \sigma_{\ell,1}||\mathrm{ipk}_B^{\mathrm{rchk}^{\ell,1}})$

The stage-specific metadata consists of the stage $(\ell, 1)$ and *all the ratcheting keys* $\{\mathrm{Rchpk}^{x,1}\}_{1 \geq x \geq \ell}$. 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 $\mathrm{ck}^{1,1}$. To do so, B queries the server for A's registered identity key and verifies that it is identical to the one included in $\mathrm{AD}_{y=1}$. 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 $\mathrm{rchk}^{1,2}$ and computes $\mathrm{Rchpk}^{1,2} := g^{\mathrm{rchk}^{1,2}}$. Bob computes: $\mathrm{ck}^{1,2} \leftarrow \mathrm{HKDF}((T_1)^{\mathrm{ik}_B}||\mathrm{ipk}_A^{t_0})$, then its first sending keys $(\mathrm{ck}^{2,2}, \mathrm{mk}^{1,2}) \leftarrow \mathrm{HKDF}(\mathrm{ck}^{1,2}, \sigma_{1,2}||(\mathrm{ipk}_A)^{\mathrm{rchk}^{1,2}})$.

Then analogously to Alice's first message, Bob splits the metadata into the two auxiliary values $\mathrm{AD}_{y=2}$ and $\mathrm{AD}_{1,2}$. The signed public key pk_B is also appended to each of the messages in stages with chain-index $y = 2$, *cf.* Fig. 3.

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 $(2, 1)$ 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.

Security Analysis. 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 send/receive messages. The adversary ultimately has to distinguish from random a real message key generated by an honest instance speaking with another honest instance.

The following theorem describes the security of MARSHAL in terms of PCS-AKE security and MLR-security. This security holds in the random oracle model (KDF are replaced by random oracles).

Theorem 3.1. *If the GDH [15] assumption holds, if the signature scheme employed 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.*

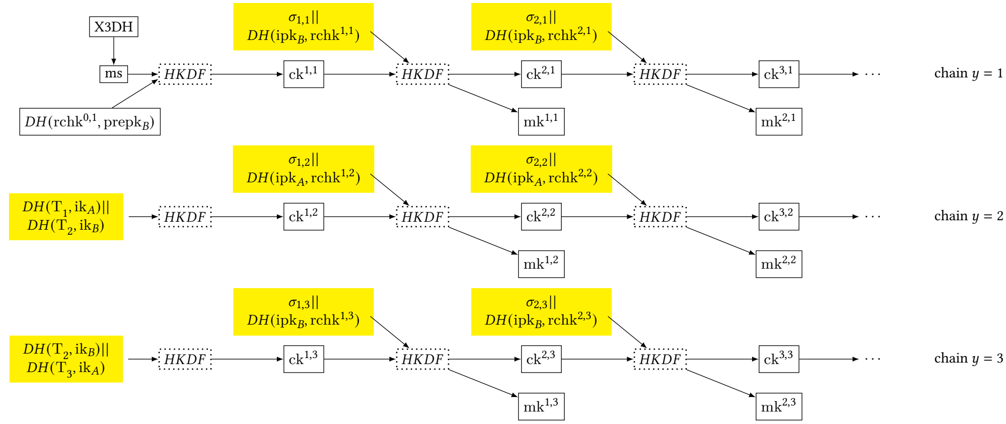

Figure 4: MARSHAL **key schedule diagram, where** $\sigma_{x,y} = \mathrm{SIGN}_{sk_A}\left(\mathrm{T}_{y-1}||\mathrm{Rchpk}^{x,y}\right)$ **for** y **odd and** $\sigma_{x,y} = \mathrm{SIGN}_{sk_B}\left(\mathrm{T}_{y-1}||\mathrm{Rchpk}^{x,y}\right)$ **for** y **even. The yellow boxes indicate modifications with respect to Signal protocol [8].**

We prove the security of our protocol with respect to a security model which is derived from the identity-based setup of [5], rather than the one used by Cohn-Gordon *et al.* for their original analysis of Signal. This is chiefly because in [8], Cohn-Gordon *et al.* bypass a feature of the protocol which we consider essential: sending metadata as AD attached to AEAD. Instead [8] assumes that the metadata is sent unauthenticated. We prefer not to modify the protocol, and use the less composable security notion proposed (for the same reasons as we described here) by Blazy *et al.*. The proof of this theorem is not technically complex, but includes a lot of special cases (as in [8]). This is a direct consequence of having excluded only trivial attacks from the winning conditions (given in [1]). However, 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. The full proof is given in [1].

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. 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, 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 $(\mathrm{Rchpk}^*)^{\mathrm{ik}}$ where the identity key belongs to the hijacked party. Our winning conditions forbid the adversary from learning the long-term key involved, if the adversary forges the signature or submits an input including $(\mathrm{Rchpk}^*)^{\mathrm{ik}}$ to $\mathcal{RO}_2$, we construct reductions to EUF-CMA security and 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 use to compute mk^{s^*}. We ruled 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 $(\mathrm{Rchpk}^*)^{\mathrm{ik}}$ 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. □

4 IMPLEMENTATION

We give a proof-of-concept implementation of MARSHAL in Java, available on github [2]. The implementation covers registration and messaging (but not out-of-order message decryption).

4.1 Implementation details

Our implementation relies on the Java implementation of Signal, available at https://github.com/signalapp/libsignal-protocol-java. We used low-level libsignal functionalities but had to re-implement its more abstract layers, to fit with MARSHAL. The libsignal library uses interfaces in order to store and recover keys and session state data; it allows us to abstract the central server and simulate exchanges between two parties within the same process.

Cryptographic details. Our implementation uses similar elliptic curve material. For signatures we used XEd25519 on Curve25519. We use similar authenticated encryption algorithms relying on AES with 128-bit keys. However, while Signal uses CBC mode and an hmac256-MAC, we prefer to use AES-GCM with a 12-byte IV and a 16-byte tag.

4.2 Implementation results

All the results indicated in this section are the mean result over 1000 executions of each given test. See Figure 5 for the results.

Test	Signal	MARSHAL
Session Setup	3.856	6.924
Message $(1, y)$	1.284	5.082
Message (ℓ, y)	0.06	1.512

Figure 5: Average runtime for each test in ms.

Session setup. The first differences between Signal and MARSHAL occur during registration and session setup. For the setup, Alice and Bob need to generate two new elements: a signature key and a Diffie-Hellman public value. While the master secret of both sessions is similar, the first chain and message keys are generated differently in Signal and MARSHAL. Our Session-Setup tests covers: key-generation steps of both Alice and Bob, publishing a PreKey-Bundle, initiating a session that uses that bundle, including the encryption and decryption of a first message.

We notice in Table 5 that MARSHAL takes twice as long as Signal for this test. This is partly because we also require the use of signatures. Note that the signature scheme used is the same as Signal; however, a faster signature scheme could significantly improve performance. We also note that, while the relative increase in runtime is significant, the absolute measurement is still low.

Message $(1, y)$. This test focuses on the first message of a new chain. The test covers the initialization of new chain randomness, the derivation of chain- and message-keys, the encryption, and decryption of that first message. Note that some of these operations are partially included in the previous test's result.

The results presented in Figure 5 indicate a much larger runtime for MARSHAL than for Signal. This is due to the generation of one additional Diffie-Hellman element, as well as the computation of two signatures (which also require verification). Still, the absolute value of the runtime remains small.

Message (ℓ, y). This test compares the runtime when the same speaker adds a new message. The test includes key-derivation, encryption, and subsequent decryption. We notice the same trend in terms of increase in runtime; however, the comparative increase is much higher, because instead of Signal's symmetric ratchet we have asymmetric ratcheting and signature computations.

Trends in longer message chains. One of the characteristics of MARSHAL is that in longer message chains, a higher number of Diffie-Hellman elements are generated and used as meta-data for each message. We thus run our third test – Message (ℓ, y) a variable number of times, and measure the total required time from the beginning of the test, to the last decryption of the last message. It is perhaps surprising that our time is linear: the comparative cost associated with sending and processing multiple DH values (rather than a single one) is very small comparative to the cost of the asymmetric ratchet operation which we require at each stage.

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 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 (*e.g.*, signature or identity keys). This data should therefore be stored separately from ephemeral data, in a secure component.

Finally, we have implemented our protocol to evaluate the practical cost of our modifications. Our implementation does not require fundamental changes to the basic cryptographic primitives used in Signal. In addition, experiments show that significant benefits to post-compromise security our protocol brings do not come at a too-significant cost, the runtimes of our ratchets and message-exchanges remaining under 10 ms mark in Java implementation.

REFERENCES

[1] 2021. Full version. https://drive.google.com/file/d/1KlJmyn-5LDEzOKCY5C3erEAfaRl0X-m1/view?usp=sharing.

[2] 2021. Implementation of MARSHAL. https://github.com/anonym-123/marshal.

[3] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. 2019. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. In *EUROCRYPT*. https://doi.org/10.1007/978-3-030-17653-2_5

[4] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. 2017. Ratcheted Encryption and Key Exchange: The Security of Messaging. In *CRYPTO*. https://doi.org/10.1007/978-3-319-63697-9_21

[5] Olivier Blazy, Angèle Bossuat, Xavier Bultel, Pierre-Alain Fouque, Cristina Onete, and Elena Pagnin. 2019. SAID: Reshaping Signal into an Identity-Based Asynchronous Messaging Protocol with Authenticated Ratcheting. (2019).

[6] Nikita Borisov, Ian Goldberg, and Eric Brewer. 2004. Off-the-record Communication, or, Why Not to Use PGP *(WPES '04)*. ACM, New York, NY, USA. https://doi.org/10.1145/1029179.1029200

[7] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. 2019. SEEMless: Secure End-to-End Encrypted Messaging with less</> Trust. In *Proceedings of ACMCCS*. ACM, 1639–1656.

[8] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 2017. A Formal Security Analysis of the Signal Messaging Protocol. *EuroS&P* (2017). https://doi.org/10.1109/EuroSP.2017.27

[9] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. 2016. On Post-compromise Security. In *CSF*. https://doi.org/10.1109/CSF.2016.19

[10] F. Betül Durak and Serge Vaudenay. 2019. Bidirectional Asynchronous Ratcheted Key Agreement with Linear Complexity. In *IWSEC*. https://doi.org/10.1007/978-3-030-26834-3_20

[11] Joseph Jaeger and Igors Stepanovs. 2018. Optimal Channel Security Against Fine-Grained State Compromise: The Safety of Messaging. In *CRYPTO*. Springer. https://doi.org/10.1007/978-3-319-96884-1_2

[12] Daniel Jost, Ueli Maurer, and Marta Mularczyk. 2019. Efficient Ratcheting: Almost-Optimal Guarantees for Secure Messaging. In *EUROCRYPT*. Springer. https://doi.org/10.1007/978-3-030-17653-2_6

[13] M. Marlinspike and T. Perrin. 2016. The double ratchet algorithm. https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[14] Moxie Marlinspike and Trevor Perrin. 2016. The X3DH Key Agreement Protocol. *Signal* (2016), 11. https://www.whispersystems.org/docs/specifications/x3dh/

[15] Tatsuaki Okamoto and David Pointcheval. 2001. The gap-problems: A new class of problems for the security of cryptographic schemes. *Lecture Notes in Computer Science* (2001).

[16] Bertram Poettering and Paul Rösler. 2018. Towards Bidirectional Ratcheted Key Exchange. In *CRYPTO*, Vol. 10991. https://doi.org/10.1007/978-3-319-96884-1_1