Analysis of the Signal Protocol

Author: **Tomé Carvalho - 97939**

Date: **27/06/2022**

1. Summary

The purpose of this assignment is to stimulate a reflection and detailed analysis around a relevant topic, with focus on its analytic and quantitative aspects.

The topic chosen is the non-federated open-source cryptographic protocol used for end-to-end encryption in the Signal instant messaging and voice calls application, also implemented in other applications, most notably WhatsApp.[1] The goal is to analyze the steps of this protocol, which makes this encryption possible.

2. Framework

The Signal Protocol, previously known as TextSecure Protocol, was developed by Open Whisper Systems in 2013, a software development group that introduced it in the open-source TextSecure IM application. Signal is the successor of this application and the RedPhone encrypted voice calling application[2]. Aside from Signal itself, various other applications have implemented the protocol, including several closed-source applications, such as WhatsApp, the most popular messaging app, with over 5 billion downloads on Google Play Store[3], Google Messages, Facebook Messenger and Skype. However, the last three only implement it for some of the applications' use cases (Messages only for one-to-one conversations, Messenger and Skype only for secret/private conversations).

Signal, despite not being as popular as WhatsApp (Signal's 100 million Play Store downloads pale in comparison to WhatsApp's 5 billion), still features the most relevant implementation to study, given its open-source nature, in contrast to WhatsApp, which is closed-source and also owned by Meta, a conglomerate infamous for privacy violations, thus leading to skepticism about its implementation. Signal's source code is available publicly on GitHub.[4] Additionally, Signal has been endorsed by individuals relevant to web security, such as whistleblower Edward Snowden, a former computer intelligence consultant who leaked highly classified information from the National Security Agency.[5]

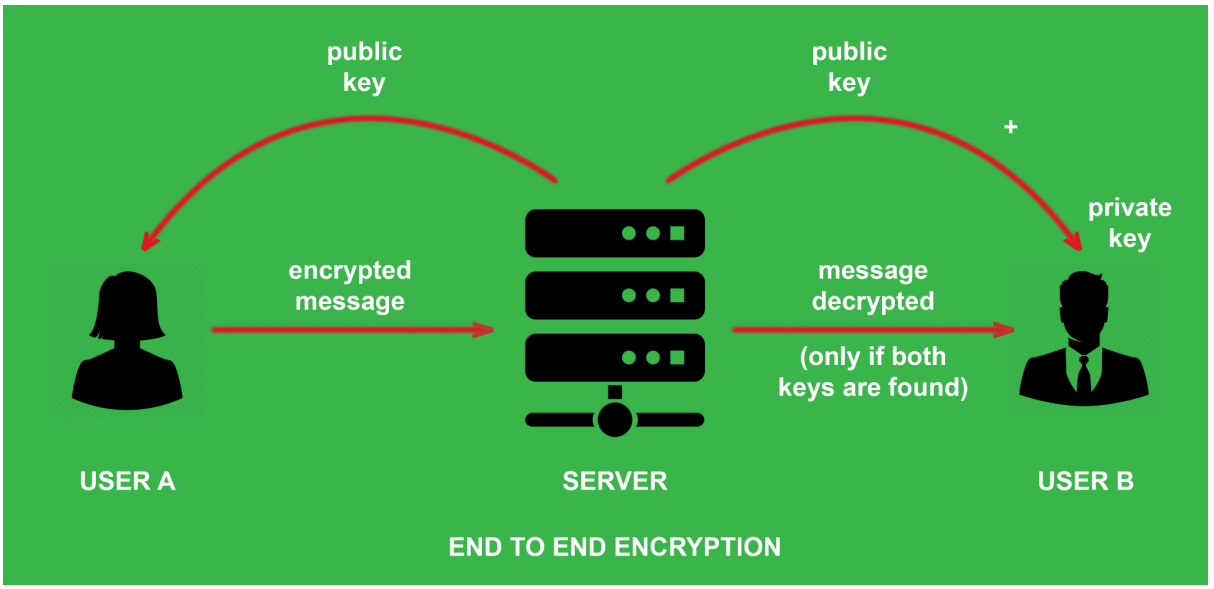


Fig. 1: End-to-end encryption (E2EE) diagram[6]

The purpose of E2EE is to prevent third parties from accessing data as it is transferred between devices. The data is encrypted on the sender's device, and it is only decryptable by the intended recipient. This ensures that messages cannot be read by internet service providers, messaging applications, attackers intercepting the traffic, etc.

3. Signal Protocol

3.1. Overview

The Signal Protocol is a multi-stage authenticated key exchange protocol. Its main steps are:

1. **Registration**
   When installing the application, Alice and Bob register their identity with a key distribution server, uploading keys of various durations: long-term, medium-term and ephemeral (very short).
2. **Session setup**
   Through the aforementioned key distribution server, Alice requests and receives a set of Bob's public keys. These are used to set up a long-lived messaging session and establish initial symmetric encryption keys, providing mutual authentication. This step is referred to as "X3DH" (or "Extended Triple Diffie-Hellman"). It provides forward secrecy and cryptographic deniability.
3. **Synchronous messaging (Asymmetric-ratchet updates)**

If Alice wants to send Bob a message, and has just received a message from him, she exchanges Diffie-Hellman (DH) values with him, originating new shared secrets used to initiate new chains of message keys. Each DH operation is considered a stage of the “asymmetric ratchet”.

4. **Asynchronous messaging (Symmetric-ratchet)**
   If Alice wants to send Bob a message, but she was the last one to send a message, she derives a new symmetric encryption key from her previous state using a pseudorandom function (PRF). Each PRF operation is considered a stage of the “symmetric ratchet”.

3.2. Signal Protocol Stages

3.2.1. Registration

Upon installation, and periodically afterwards, agents generate a number of cryptographic keys and register themselves with a key distribution server (KDS). Each agent P generates the following DH private keys:

1. $\boldsymbol{ik^P}$**:** long-term identity key
2. $\boldsymbol{prek^P}$**:** medium-term signed prekey
3. Multiple ***eprek***: short-term one-time prekeys

Afterwards, the public keys corresponding to these values are uploaded to the KDS, along with a signature on *prek* using *ik*. Collectively, these are referred to as the “pre-key bundle”.

3.2.2. Session Setup

In this stage, public keys are exchanged and used to initialize shared secrets. The key exchange protocol used is the X3DH one-round DH protocol, which includes the exchange of various DH public keys, computation of various DH shared secrets and, finally, the application of a key derivation function (KDF).

It is important to note that, for asynchronicity, Signal makes use of prekeys: initial protocol messages stored at an intermediate server, which allow agents to establish sessions with offline users through the retrieval of their cached messages (in the form of a DH ephemeral public key).

Additionally, agents publish a “medium-term” key, shared between multiple peers. Even if the one-time ephemeral keys stored at the server are exhausted, the session will proceed, using only a medium-term key. Consequently, session setup consists of two steps:

1. Alice obtains ephemeral values from Bob (typically through a KDS)

2. Alice considers the received values the first message of a Signal key exchange, completing the exchange in order to derive a master secret.

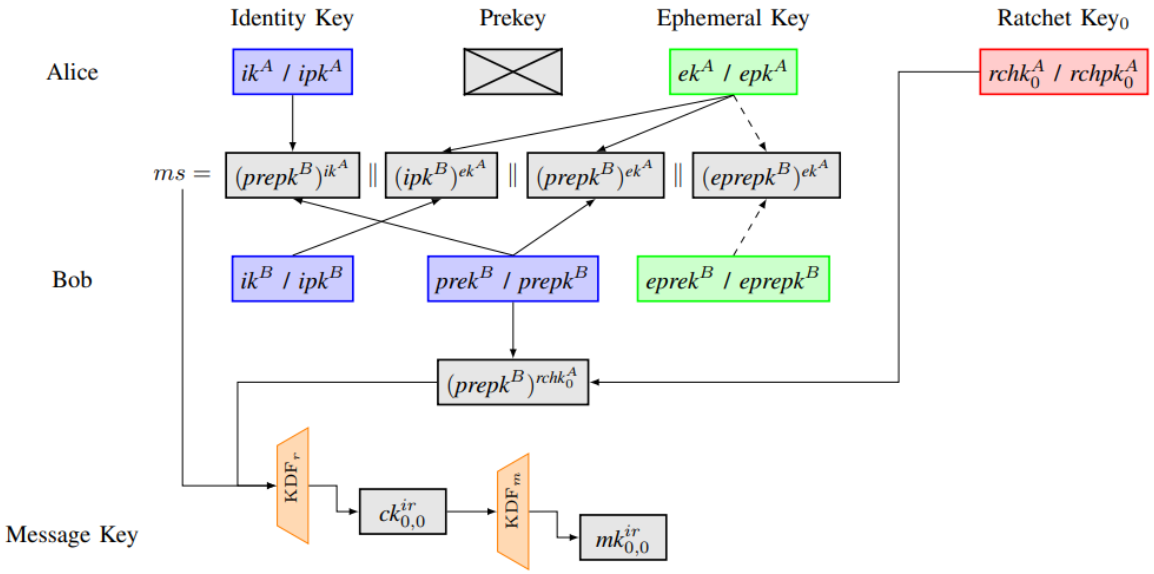


Fig. 2: Key schedule diagram for the initial X3DH key exchange. “Sending” and “receiving” refer to Alice’s point of view. Blue denotes values used in multiple exchanges, green values used only in a single one, and red values that are also used in the Double Ratchet protocol. In this example, Alice uses the standard cryptographic data from Bob’s pre-key bundle and a one-time ephemeral pre-key to compute the first root key, the first sending chain key and the first message key.

3.2.2.1. Receiving ephemerals

The typical way for Alice to receive Bob’s session-specific data is for her to query a semi-trusted server for pre-computed values (PreKeyBundle).

When she requests his identity information, she receives his identity public key, his current signed pre-key and a one-time prekey if available. Signed pre-keys are stored for the medium term, and thus shared between everyone sending messages to Bob. The server deletes one-time keys upon transmission. Alice’s initial message contains pre-key identifiers so Bob is able to learn which ones were used.

3.2.2.2. Computing shared secrets

Fig. 2 shows the key schedule for the initial handshake in detail.

Alice receives the above values, then generates her own ephemeral key ek^A, computing a session key by performing group exponentiations. Afterwards, she concatenates the resulting shared secrets and applies a KDF to them, deriving an initial root key *rk1* and sending chain key $ck^{ir}0,0$. For modeling purposes, the diagram also includes Alice generating her initial sending message $mk^{ir}0,0$ (this stage’s session key output) and the next sending chain key $ck^{ri}1,0$. At last, she generates a new ephemeral DH key $rchk^A0$, known as her ratchet key.

In order for Bob to complete the key exchange, he needs to receive her public ephemeral key epk^A and public ratchet key $rchpk^A0$. In the protocol, these values are attached to all the messages Alice sends until she receives a message from Bob, as from such a message it is possible for her to conclude that Bob has received epk^A and $rchpk^A0$. To

disentangle the model's stages, Alice appears to send epk^A and $rchpk^A_0$ in a separate message. Thus, once the session-construction is finished, Alice and Bob will have derived their root and chain keys.

Upon receiving epk^A and $rchpk^A_0$, Bob checks that he currently knows the private keys that correspond to the identity, signed pre-key and one-time pre-key that Alice used. If so, he performs the receiver algorithm for the key exchanged, deriving the same root key rk1 and chain key (recorded by him as $ck^{ir}_{0,0}$). Once again, for modeling purposes, Bob appears to generate his initial receiving message key $mk^{ir}_{0,0}$ (this stage's session key output) and the next receiving chain key $ck^{ir}_{0,1}$.

3.2.3. Symmetric-Ratchet

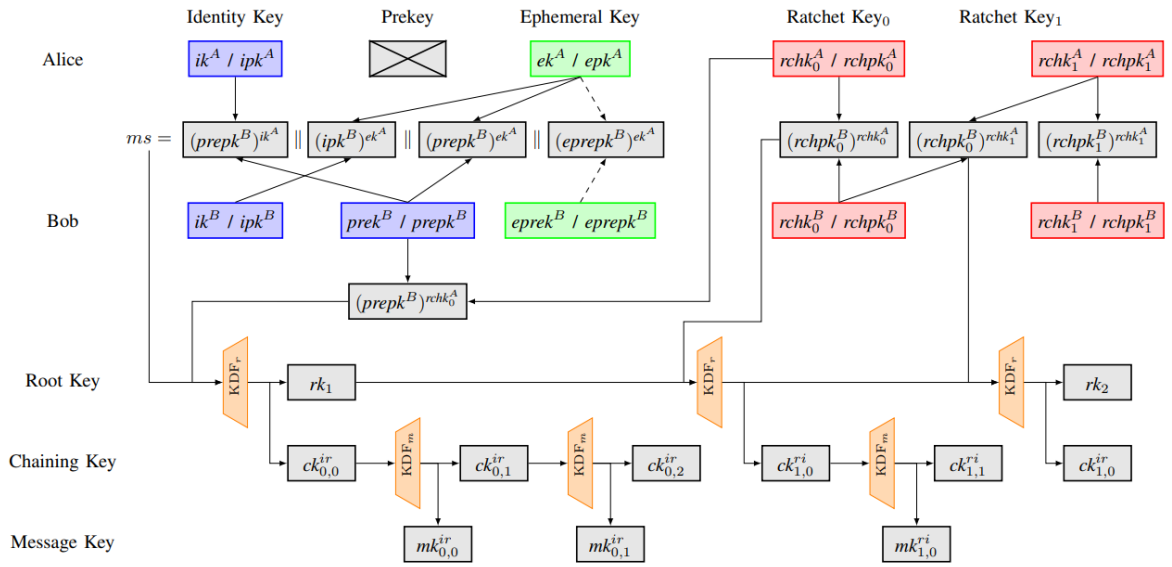


Fig. 3: Key schedule diagram for an example session. Sending" and "receiving" refer to Alice's point of view. Blue denotes values used in multiple exchanges, green values used only in a single one, and red values that are also used in the Double Ratchet protocol. This example captures an initial X3DH key exchange, where Alice uses the standard cryptographic data from Bob's pre-key bundle and a one-time ephemeral pre-key $eprepk^B$ to compute the first rook key rk_1, the first sending chain key $ck^{ir}_{0,1}$ and the first message key $mk^{ir}_{0,0}$. Additionally, Alice sends another message before receiving one from Bob, requiring a symmetric ratchet of the first sending chain so the next message key $mk^{ir}_{0,1}$ can be derived. Next, Bob sends a new ratchet public key $rchpk^B_0$, which is combined with Alice's ratchet key $rchpk^A_0$ to compute the first receiving chain key $ck^{ri}_{1,0}$, used to derive the next one, $ck^{ri}_{1,1}$ and the first receiving message key $mk^{ir}_{1,0}$. At last, Alice sends a new ratchet key $rchpk^A_1$ that is combined with Bob's previous ratchet public key $rchpk^B_0$ to compute the next root key rk_2.

In this stage, two sequences of symmetric keys will be derived using a PRF chain, one of them for sending and the other one for receiving. The symmetric chains may be advanced for one of two reasons:

1. Alice wants to send a new message
2. Alice wants to decrypt a message that she has just received

If she wishes to send a message, she takes her current sending chain key $ck^{ir}_{x,y}$ and applies the message KDF, KDF_m, in order to derive two new keys: an updated sending chain key $ck^{ir}_{x,(y+1)}$ and a sending message key $mk^{ir}_{x,y}$. The sending message key is used to encrypt her outgoing message. Afterwards, it is deleted, along with the old sending chain key. This process may be arbitrarily repeated.

Fig. 3 shows an example of the symmetric ratchet, at the bottom. On the lines labeled “chaining key” and “message key”, a symmetric ratchet stage, [sym-ir:0,1] is shown, which computes $mk^{ir}_{0,1}$.

In the second case, when Alice receives an encrypted message, she checks the accompanying ratchet public key to confirm that she has not processed it yet. If not, she performs an asymmetric ratchet update, explained in the next section. In any case, she proceeds to read the metadata included in the message header to determine the index of the message in the receiving chain, advancing it as many steps as necessary to derive the required receiving message key. By construction, Alice’s receiving message key’s will be equal to Bob’s sending keys. In contrast to the case where she sends a message, Alice cannot delete receiving message keys immediately. She must wait to receive a message encrypted under each one. If that were not the case, out-of-order messages would not be decryptable as their keys would have been deleted).

In consequence, Alice is able to retain any particular receiving chain for as long as she desires. Additionally, along any chain, chain keys can be ratcheted forward in order to generate message keys in such a way that they are independent, meaning that retaining them while waiting for late messages to arrive does not compromise other messages. To exemplify, Alice can produce the message key for delayed message 2, while still being able to symmetrically ratchet forward to decrypt received messages 3, 4, etc., knowing that retaining message key 2 while waiting for it to arrive will not endanger other message keys along the chain. The core concepts of the root key producing chain keys and the latter producing message keys means that messages can arrive in arbitrary order while Alice and Bob continue to ratchet forward.

Signal’s open-source implementation has a hard-coded 2000 messages or 5 asymmetric updates limit, after which the old keys are deleted regardless of whether they have yet been used.

3.2.4. Asymmetric-Ratchet

This is the final top-level stage of the protocol, the asymmetric-ratchet update. Alice and Bob take turns generating and sending new DH ratchet public keys, using them to derive new shared secrets. This occurs in a ping-pong fashion. That is, Bob continues to use the same ratchet key until he receives a new ratchet key from Alice. These newly generated keys are denoted by $rchpk^i_x$ in Fig. 4, where *i* is the identity of the sending party (A

for Alice, B for Bob) and *x* the numbered asymmetric stage. They are accumulated in the asymmetric ratchet chain, used in the initialization of new symmetric message chains.

When Alice receives a message from Bob, it may include a new ratchet public key $rchpk^{B}_{x-1}$. In this case, Alice will enter her next asymmetric ratchet stage [asym-ir:x], having already stored a previously generated private ratchet key $rchk^{A}_{x-1}$. Before decrypting the message, Alice must update her asymmetric ratchet, a process that comprises two steps:

- ([asym-ri:x]) Alice derives three secret values: an intermediate secret value *tmp*, a receiving chain key $ck^{ri}_{x,1}$ and a receiving message key $mk^{ri}_{x,0}$. She computes a DH shared secret (between the newly-received ratchet public key $rchpk^{B}_{x-1}$ and her old ratchet private key $rchk^{A}_{x-1}$), combining it with the previously computed root chain key rk_x in order to derive an intermediate secret value *tmp* and a new receiving chain key $ck^{ri}_{x,0}$ (by applying KDF_r). Next, she uses this new chain key as input to KDF_m to compute the next receiving chain key $ck^{ri}_{x,1}$ and receiving message key $mk^{ri}_{x,0}$. She generates a new DH ratchet private key $rchk^{A}_{x}$.
- ([asym-ir:x]) Alice computes a second DH shared secret (between the received ratchet public key $rchpk^{B}_{x-1}$ and her new ratchet private key $rchk^{A}_{x}$), combining this with the intermediate secret value *tmp* through the application of KDF_r in order to compute the next root chain key rk_{x+1} and a new sending chain key $ck^{ir}_{x,0}$. She then uses this new sending chain key as input to KDF_m to generate the next sending chain key $ck^{ir}_{x,1}$ and receiving message key $mk^{ir}_{x,0}$.

The message keys in the two steps have slight differences in security properties, the reason as to why they are depicted in Fig. 4 as belonging to distinct stages ([asym-ri:x] and [asym-ir:x]).

After these two steps, Alice sends her new ratchet public key $rchpk^{A}_{x}$ along with future messages to Bob. The process continues indefinitely.

Bob performs the corresponding operations shown in Fig. 4 in order to compute the same DH shared secrets and the corresponding root, chain and message keys. Symmetric updates can be triggered either by the session initiator or the session responder (Alice and Bob, respectively), thus they could be as in Fig. 4(c) or its horizontal flip. However, asymmetric updates are only triggered when the initiator (Alice) receives a new ratchet key from the responder (Bob): Fig.4(d) cannot be horizontally flipped.

3.2.5. Memory contents

The Signal protocol is stateful. This means that multiple values are present in Alice's memory at any time. Her global state, which is shared between all of her sessions, contains four collections of values: identity keys, signed pre-keys, ephemeral keys and a session list.

Each session in the aforementioned list contains the keys used by the protocol. To specify, a session always contains: agent's identity private key, peer's identity public key, current root and sending chain key, current ratchet key. Additionally, it also contains some receiving chain and message keys that correspond to out-of-order messages that have not yet been received from the peer.

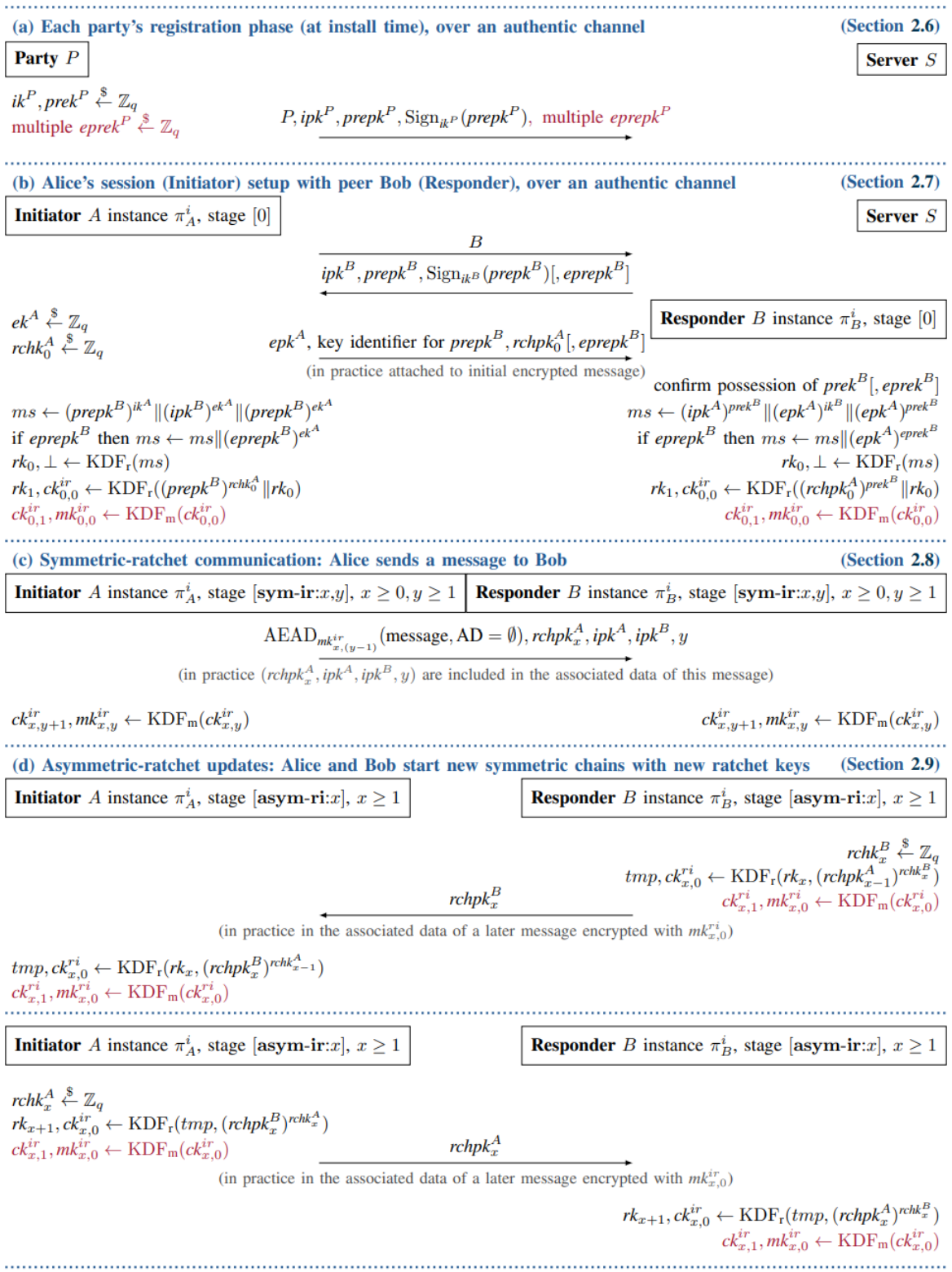


Fig. 4: Flow of the Signal Protocol, including key pre-registration. Local actions are depicted in the left and right columns, and messages flow between them. Only one step of the ratchets is shown, but they can be iterated arbitrarily. Each stage derives message keys with the same index as the stage number, and chaining/root keys with the index for the next

stage; the latter is passed as state from one stage to the next. State info *st* in asymmetric stages is the root key used in the key derivation, and for symmetric it is the chain key used in key derivation. Symmetric stages start at y = 1 and increment. When an actor sends consecutive messages, the first message is a DH ratchet and the next messages use the symmetric ratchet. When an actor replies, they always DH ratchet first; they never carry on the symmetric ratchet.

4. Limitations and potential issues of Signal

While the Signal Protocol has been verified to be secure and Signal remains regarded as the strongest contender for the most secure and private instant messaging application, it is not flawless. There are some caveats that must be taken into account[10].

The most notable one is that, at the time of writing, it requires the use of phone numbers, which many users are reluctant to share. This contrasts with, for example, Telegram, another instant messenger. The latter allows users to choose a public username that can be used by others to contact them without revealing the users' phone numbers if that is not their desire. However, Signal's developers have promised to implement this feature and its development is evidenced on Signal's GitHub repositories.

The second one is Signal's reliance on centralized technology. Thus, server issues (like user count surges and DDoS attacks) disable the entire network. A decentralized architecture could prevent this, but it would bring additional technical challenges.

The third limitation is the lack of cloud data sync. Signal does not rely on a cloud-based sync feature, the data resides locally. Upon logging in to a new device, past conversations cannot be retrieved without restoring a backup. Backups cannot be automatically created and stored in the cloud, the user must manually generate encrypted backups locally and handle their storage according to their needs. For comparison, WhatsApp provides an automatic cloud backup feature that periodically creates encrypted backups and uploads them to a linked Google Drive.

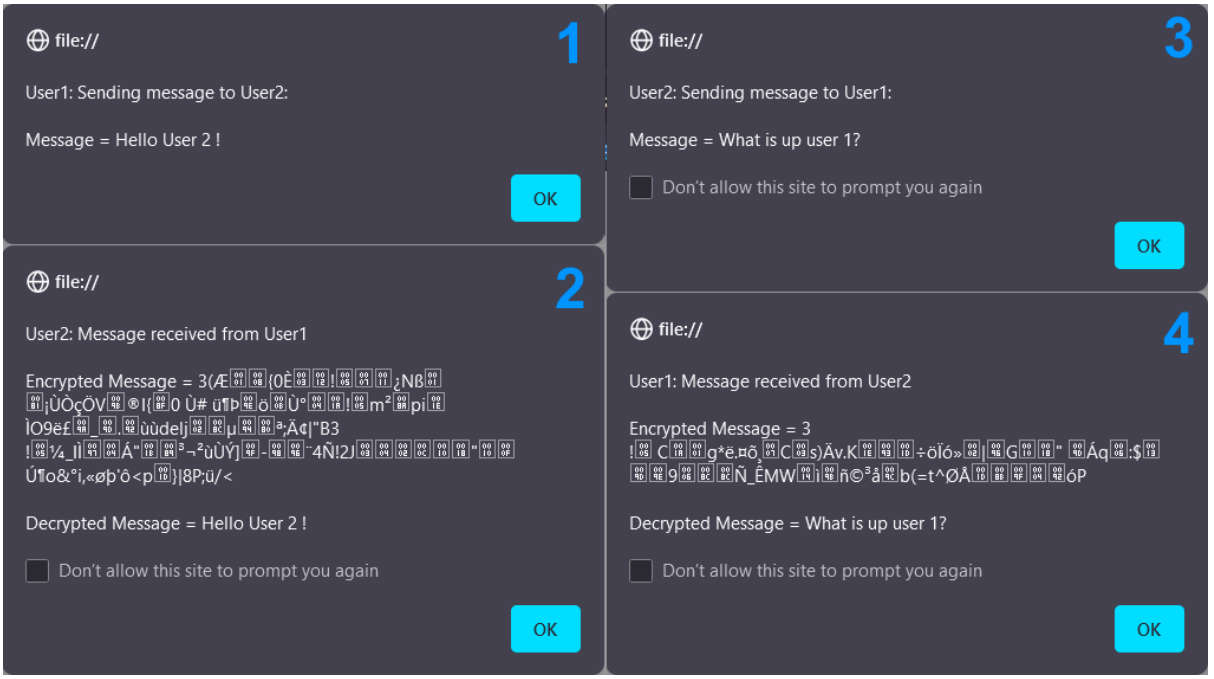
As a concluding remark, it's important to highlight that, despite these limitations, using Signal does not necessarily mean trading features for security and privacy. Aside from user experience features (such as group chats, emoji reactions, calls, etc.), one of its biggest advantages is that it has clients available in a multitude of platforms: Android, iOS, Windows, Mac and GNU/Linux.

5. Demonstration of the Signal Protocol

A simple demonstration of the Signal protocol was written by the user *Jamie-Matthews* on GitHub[9]. It uses a JavaScript browser implementation of the protocol, which has now been deprecated in favor of a TypeScript API. Nevertheless, it remains useful to demonstrate the protocol.

It contains JavaScript files from Signal's repositories: *helpers.js* and *libsignal-protocol.js* and two other ones developed for the demonstration. *InMemorySignalProtocolStore.js* is a local store implementation.

By opening the *SignalDemo.html* file, which calls the aforementioned scripts, in a web browser, the following alert messages are shown:



It creates two users and sends messages between them. We can see their encrypted and decrypted forms. Since the receivers' private keys are needed to decrypt the messages encrypted with their public keys by the senders, the messages can only be decrypted by the intended receiver.

References

[1] ‘Signal Protocol’
Available online (25/06/2022):
https://en.wikipedia.org/wiki/Signal_Protocol

[2] ‘Signal (software)’
Available online (25/06/2022):
https://en.wikipedia.org/wiki/Signal_(software)

[3] ‘Messaging App Revenue and Usage Statistics (2022)’
Available online (25/06/2022):
https://www.businessofapps.com/data/messaging-app-market/#:~:text=Over%20three%20billion%20people%20have,the%20most%20popular%20app%20types.

[4] ‘Signal (GitHub)’
Available online (25/06/2022):
https://github.com/signalapp

[5] ‘Edward Snowden’
Available online (25/06/2022):
https://en.wikipedia.org/wiki/Edward_Snowden

[6] ‘End to End Encryption (E2EE) in Computer Networks’
Available online (25/06/2022):
https://www.geeksforgeeks.org/end-to-end-encryption-e2ee-in-computer-networks/

[7] ‘Signal: Technical information (documentation)’
Available online (25/06/2022):
https://signal.org/docs/

[8] ‘A Formal Security Analysis of the Signal Messaging Protocol’
Available online (26/06/2022):
K. Cohn-Gordon, C. Cremers, B. Dowling. , L. Garratt, D. Stebila. A Formal Security Analysis of the Signal Messaging Protocol. Extended Version, July 2019

[9] ‘Signal-Protocol-Demo’
Available online (26/06/2022)
https://github.com/Jamie-Matthews/Signal-Protocol-Demo

[10] ‘Is Signal the Ultimate Secure Messaging App?’
Available online (27/06/2022)
https://www.makeuseof.com/signal-ultimate-messaging-app/