Linphone Instant Message Encryption v2.0 (Lime v2.0)

Johan Pascal

December 13, 2017
Draft

Contents

1 Introduction

Linphone Instant Message Encryption(Lime) v2.0 implements the Signal protocol allowing users to privately and asynchronously exchange messages. Detailed specification of the Signal protocol can be found on the Signal website. Lime supports multiple devices per user and multiple users per device.

Lime is designed to be used with Linphone, an open source SIP phone. Lime establishes encrypted sessions and encrypts messages but relies on Linphone to acquire the unique identification string of peer devices and route the messages to their recipients. The use of Lime with other message delivery software is possible but is out of scope of this document.

Lime is written in C++11 and the library uses templates to provide support for Curve25519 and Curve448-based cryptographic algorithms. The library supports one or both curves according to build settings.
A users network(all clients and keys server) must commit to using either Curve25519 or Curve448, but a device may host several users communicating on separate user's networks, using different curves.

Note: Lime v1.0 was based on SCIMP. This document presents Lime v2.0 which is not related to nor compatible with Lime v1.0. In this document the use of the term Lime refers to Lime v2.0

2 Notations

$A||B$ denotes the concatenation of byte sequences A and B

$A\langle value\rangle$ the bytes sequence A size is $value$. For example: $key\langle 32bytes\rangle$ denotes a 32 bytes long buffer called key. Several values may be included in a comma-separated list, indicating that several sizes are possible.

$element\{instances\}$ denotes the number of occurrences of a given $element$. $instances$ may be a number, a range or a comma-separated list of possible values. For example: $key\{4\}$ means 4 $keys$, $key\{0,1\}$ means either 0 or 1 key.

$element[values]$: element value can be one of the values given in a comma-separated list. For example $type[1,2,3]$ means $type$ equals either 1, 2 or 3.

3 Brief introduction to Signal protocol specification documents

3.1 The Double Ratchet Algorithm[1]

The Double Ratchet algorithm is used by two parties to exchange encrypted messages based on a shared secret key. Typically the parties will use some key agreement protocol (such as X3DH[2]) to agree on the shared secret key. Following this, the parties will use the Double Ratchet to send and receive encrypted messages.

The parties derive new keys for every Double Ratchet message so that earlier keys cannot be calculated from later ones. The parties also send Diffie-Hellman public values attached to their messages. The results of Diffie-Hellman calculations are mixed into the derived keys so that later keys cannot be calculated from earlier ones. These properties give some protection to earlier or later encrypted messages in case of a compromise of a party's keys.

3.2 The X3DH Key Agreement Protocol[2]

The "X3DH" (or "Extended Triple Diffie-Hellman") key agreement protocol establishes a shared secret key between two parties who mutually authenticate each other based on public keys. X3DH provides forward secrecy and cryptographic deniability.

X3DH is designed for asynchronous settings where one user ("Bob") is offline but has published some information to a server. Another user ("Alice") wants to use that information to send encrypted data to Bob and also to establish a shared secret key for future communication.

3.3 The Sesame Algorithm[3]

The Sesame algorithm manages message encryption sessions in an asynchronous and multi-device setting. Sesame was designed to manage Double Ratchet sessions[1] created with a X3DH key agreement[2]. However, Sesame is a generic algorithm that works with any session-based message encryption algorithm that meets certain conditions.

4 Major discrepancies between Lime v2.0 and Signal protocol

This section will not go into the details of the Signal protocol specification but will focus only on the points where the Lime v2.0 implementation does not follow the Signal specification documentation. A prior knowledge of these specs is essential to understand the possible effects of such discrepancies.

4.1 Double Ratchet

4.1.1 Group chat management

The group chat mechanism implemented by Whisper Systems in libsignal-protocol-c[10] uses an unspecified (at least in Double Ratchet document[1]) feature, the sender key:

- When accepting membership, a group member creates its sender key and distributes it to all other members using pairwise Double Ratchet sessions.
- Members will then use their sender key to encrypt messages to the group, deriving it using a simple symmetric ratcheting.

This mechanism allows an efficient server-side fan-out but loses the break-in recovery property provided by the Double Ratchet mechanism.

Operating in a multi device environment Lime does not differentiate between single peer recipients and group recipients : every message sent to a peer may be encrypted to several peer devices and several self devices. The implementation principle is:

- Generate a random key and use it to encrypt the message
- Use Double Ratchet sessions to encrypt the random key.
- Send to server a bundle of:

$$\begin{gathered} \textit{DR encrypted random key\{one for each recipient device\}} \\ \| \textit{ Message encrypted using the random key} \end{gathered}$$

- Server fans out the messages to recipients mailboxes posting only the appropriate double ratchet encrypted random key and encrypted message.

The bandwidth and computational power consumption is greater than the Whisper System implementation but all the exchanges are protected by an actual Double Ratchet maintaining the break-in recovery property.

Silent members/devices (lost devices and users quitting the network are good candidates) may result in weakness in the break-in recovery as no Diffie-Hellman ratchet step is ever performed. This can be mitigated by setting a limit to the sending chain length. The sending device would create a new Double Ratchet session fetching keys from X3DH key server if the limit is reached.

Note : The actual implementation generates a 32 bytes random seed derived through HKDF[8] expansion round into a 32 bytes key and a 16 bytes nonce. The DR session encrypt the 32 bytes random seed using AES256-GCM(with 16 bytes authentication tag) producing a 48 bytes output to transmit the key.

4.1.2 AEAD encryption scheme: AES256-GCM

The Double Ratchet specification[1, section 5.2] recommends to use a SIV based AEAD encryption scheme.

The Lime implementation of the Double Ratchet Chain Key derivation is described in 5.2.3 of this document. The message key$\langle 32bytes \rangle$ and initialisation vector $\langle 16bytes \rangle$ are generated, used and destroyed during the encryption process. The direct use of an AES256-GCM as the AEAD encryption scheme is assumed to be secure as the key and IV are not reused.

4.2 X3DH Identity Key signature

The X3DH specification uses ECDH keys only in combination with XEdDSA[4] to provide an EdDSA-compatible signature using its Identity key(Ik) formatted for X25519 or X448 ECDH functions.

Lime performs the same signature and ECDH operations but the identity key(Ik) is generated, stored and transmitted in its EdDSA format and then converted into X25519 or X448 format when ECDH computation is performed on it.

The X3DH $Encode(PK)$ function recommends the usage of a single byte constant to represent the type of curve followed by the encoding specified in [5]. Lime uses direct encoding specified in [5] for its ECDH public keys and [6] for its EdDSA keys.

4.3 Authentication

X3DH specification mentions [2, section 4.1] the necessity of an identity authentication mecanism. libsignal[10] implements key fingerprints comparison to provide it. Lime make use of ZRTP[9] call with oral SAS verification to provide mutual identity authentication. See implementation details in section 5.5

4.4 Optional features not implemented

- Double ratchet with header encryption as in [1, section 4]
- Retry request as in [3, section 4.1]
- Session expiration as in [3, section 4.2] but a related mechanism is implemented: Double Ratchet session expires after encrypting a certain number of messages without performing any Diffie-Hellman ratchet step.

5 Implementation details

5.1 Preliminaries

For clarity, different terms used are defined here:

- device Id: a unique string associated to a device, provided to Lime by Linphone. It shall be the GRUU[7]
- user Id: a unique string defining a user or a group of users, provided to Lime by Linphone. It shall be the sip URI.
- source: the device generating and encrypting a message.
- recipient: the parties targeted to receive and decrypt the message. Multiple devices can be associated to the it so any mention of recipient must specify user Id or device Id to clarify the intent.

5.2 Double Ratchet

5.2.1 Diffie-Hellman

The ECDH function can be either X448 or X25519 described in [5]

5.2.2 KDF_RK

This function implements one round of HKDF[8]. The *salt* is RK and *ikm* is the output of ECDH(DH_out). The info string is a simple char: 0x03. DH_out size depends on ECDH function used, X25519 produces a 32 bytes output, X448 a 56 bytes output.

function KDF_RK(RK$\langle 32 bytes\rangle, DH_out\langle 32, 56 bytes\rangle$)
  $PRK\langle 64bytes\rangle \leftarrow \textsc{HmacSha512}(RK, DH\_out)$
  $RK\|CK \leftarrow \textsc{HmacSha512}(PRK, 0x03\|0x01)$

return $RK\langle 32bytes\rangle, CK\langle 32bytes\rangle$
end function

5.2.3 KDF_CK

Implemented as described in [1, section 5.2]. Message key derivation outputs 48 bytes as it generates the message key($MK\langle 32bytes\rangle$) and the AEAD nonce($IV\langle 16bytes\rangle$) as suggested in [1, section 3.1 - ENCRYPT].

function KDF_CK(CK$\langle 32bytes\rangle$)
$MK\|IV \leftarrow$ HMACSHA512($ChainKey, 0x01$)
$CK \leftarrow$ HMACSHA512($ChainKey, 0x02$)
return $CK\langle 32bytes\rangle, MK\langle 32bytes\rangle, IV\langle 16bytes\rangle$
end function

5.2.4 RatchetEncrypt

The RatchetEncrypt function described in [1, section 3.4] is not directly used to encrypt the message. To provide the group chat(see section 4.1.1) capabilities, an encryption request expects a list of recipient devices(can contain one or more elements).
Each recipient in the list is composed of:

$recipientDeviceId$: the recipient device Id
$DRsession$: an active Double Ratchet session with the recipient device
$DRcipher$: encryption output($header\|ciphertext$) for this recipient device

The message is sent from sender device to one recipient user (with one user Id and one or more device Id associated) but also distributed to others devices registered to the same sender user.
Recipient devices in the list must all be linked to one recipient user Id or the sender user Id.

The encryption request performs:

function MESSAGEENCRYPT($recipientList, plain, sourceDeviceId, recipientUserId$)
▷ Generate a random key and nonce to encrypt the plain
$randomSeed\langle 32bytes\rangle \leftarrow$ RANDOMSOURCE
$info \leftarrow$ "DR Message Key Derivation"
$key\langle 32bytes\rangle\|IV\langle 16bytes\rangle \leftarrow$ HKDFSHA512EXPANSION($randomSeed, info$)
$cipher\|tag\langle 16bytes\rangle \leftarrow$ ENCRYPT($key, IV, plain, sourceDeviceId\|recipientUserId$)

▷ Use Double Ratchet sessions to encrypt the random seed used to encrypt the plain
for all $r \in recipientList$ **do**
$AD \leftarrow tag\|sourceDeviceId\|r.recipientDeviceId$
$r.DRcipher \leftarrow$ RATCHETENCRYPT($r.session, randomSeed, AD$)
end for
return $recipientList, cipher, tag$
end function

with following functions definitions:

function $\textsc{Encrypt}(key\langle 32bytes\rangle, IV\langle 16bytes\rangle, plain, associatedData)$
    return AES256-GCM output$\|$Auth tag(on plain and associatedData)$\langle 16bytes\rangle$
end function

function $\textsc{RatchetEncrypt}(DRsession, plaintext, AD)$
    as described in [1, section 3.4]:
    $CKs, mK, IV \leftarrow$ KDF_CK(CKs)
    $header \leftarrow \textsc{header}(DHs, PN, Ns)$
    $Ns+=1$
    $\textsc{UpdateDRsessionInLocalStorage}$(DRsession)
    return $header, \textsc{Encrypt}(mK, IV, plaintext, AD\|X3DH\ provided\ AD\|header)$
end function

function HKDFSha512Expansion(ikm, info)
    return $\textsc{HmacSha512}(ikm, info\|0x01)$
end function

Notes: HKDFSha512Expansion implements one round of the HKDF expansion defined in [8].
Header function is specified in section 6.1

5.2.5 RatchetDecrypt

The decryption function described in [1, section 3.5] is not directly used to decrypt the message. Instead the decryption matches the two steps of encryption: first decrypt the Double Ratchet message to retrieve the random Key and IV, then decrypt the message itself.

The receiving process described in Sesame specifications[3, section 3.4] is partly implemented in the Double Ratchet decryption process: the message decrypt function accepts a list of Double Ratchet sessions and tries them all until one decrypts correctly the message(or all fail).

function $\textsc{MessageDecrypt}(sourceDeviceId,$
    $recipientDeviceId, recipientUserId,$
    $DRsessionList, DRcipher,$
    $cipher, tag)$

    $AD \leftarrow tag\|sourceDeviceId\|recipientDeviceId$
    for all $DRsession \in DRsessionList$ **do**
        if $randomSeed \leftarrow \textsc{RatchetDecrypt}(DRsession, DRcipher, AD)$ **then**
            $key\langle 32bytes\rangle\|IV\langle 16bytes\rangle \leftarrow \textsc{HKDFSha512Expansion}(randomSeed\langle 32bytes\rangle)$
            $plain \leftarrow \textsc{Decrypt}(key, IV, cipher, sourceDeviceId\|recipientUserId)$
            return $plain$
        end if
    end for
    return fail
end function

function RATCHETDECRYPT($DRsession, header \| cipher \| tag, AD$)
    As described in [1, section 3.5]
    Associated Data given to AEAD is $AD \| X3DHprovidedAD \| header$
    if Success **then**
        UPDATEDRSESSIONINLOCALSTORAGE(DRsession)
    end if
end function

5.3 X3DH

As stated in section 4.2, Lime does not use XEdDSA but manipulates two keys formats: the identity key is stored in EdDSA format(as defined in [6]), while all the other keys are stored in ECDH ready format(as defined is [5]).

5.3.1 DH

Available Diffie-Hellman algorithms are X25519 and X448, the DH computations performed strictly follow the X3DH specifications.

5.3.2 Sig

The signature/verify operation performed is an EdDSA (EdDSA25519 or EdDSA448 are available). The identity key used is stored in EdDSA format so there is no need to use XEdDSA as the X3DH specifications recommend.

5.3.3 KDF

As specified in [2, section 2.2], the HKDF function[8] is used with a zero filled salt buffer. This function is used for shared key derivation but also for shared associated data derivation for implementation convenience.

function KDF($ikm, info$)
    $PRK\langle 64bytes\rangle \leftarrow \textsc{HmacSha512}(ZeroFilledBuffer\langle 64bytes\rangle, ikm)$
    $output\langle 32bytes\rangle \leftarrow \textsc{HmacSha512}(PRK, info \| 0x01)$
    return $output\langle 32bytes\rangle$
end function

5.3.4 Shared Secrets generation

SK is computed as specified in [2, section 3.3].

$SK\langle 32bytes\rangle \leftarrow \mathrm{KDF}(F\langle 32, 57bytes\rangle \| DH1 \| DH2 \| DH2 \| DH4, \text{"Lime"})$

F being a 32(when using curve25519) or 57(when using curve448) bytes 0xFF filled buffer.

Associated Data is computed from identity keys and devices Id as specified in [2, section 3.3].
For a matter of implementation convenience, the actual AD used by the Double Ratchet session is derived from these inputs by the KDF function producing a fixed size buffer as following:

$ADinput \leftarrow initiatorIk \| receiverIk \| initiatorDeviceId \| receiverDeviceId$

$AD\langle 32bytes\rangle \leftarrow \mathrm{KDF}(ADinput, "X3DH\ Associated\ Data")$

initiator being the device who initiates the session (Alice in the X3DH spec) by fetching a keys bundle on the X3DH server and *receiver* being the recipient device of the first message (Bob in the X3DH spec).

5.3.5 Server

An X3DH test server running on nodejs is provided along the Lime library source code. This server is not meant to be used in production and its purpose is for testing only. Open instances of the test server shall be running on https://sip5.linphone.org:25519 (operating with curve 25519) and https://sip5.linphone.org:25520 (operating with curve 448) accepting connection identified as anyone using the credentials:

- username: "alice"
- password: "you see the problem is this"

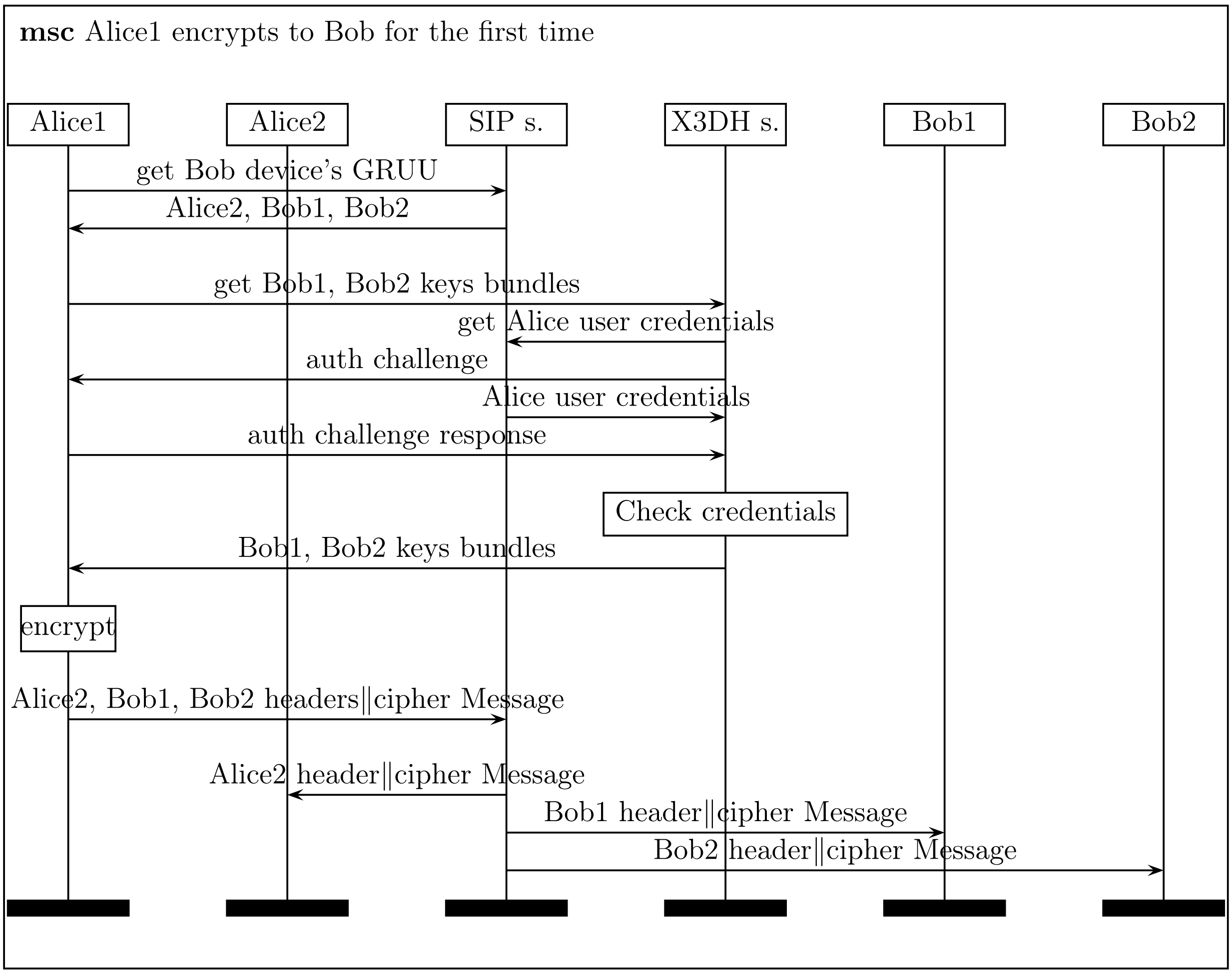
5.4 Sesame

The Sesame part of the system is distributed into different elements:

- Lime is operating in per-device identity keys mode.
- Providing an updated list of Devices Id to match the intended recipients (and sender user other devices) is performed by the linphone ecosystem (SIP and conference server). So the loop between client and server during encryption described in the Sesame spec[3] is not relevant. Lime relies on the SIP or conference server to provide an updated list of recipient devices before the message encryption.
- Encrypt message to multiple recipient device is performed by the Lime Double Ratchet *messageEncrypt* function (see section 5.2.4)
- Support for multiple sessions between devices is performed by Lime Double Ratchet *messageDecrypt* trying multiples sessions, if present, to find one able to decrypt the incoming message.
- User and device identifications are provided by the linphone ecosystem: a user Id is its sip:uri, also used to identify groups. A device Id is its GRUU[7]. The connection to the X3DH server is performed over HTTPS and uses the user authentication associated to the sip user account.
- Mailboxes and message routing is provided by the linphone ecosystem

5.4.1 use case: first encryption, multiple devices

In the following diagram, Alice1 encrypts a message to Bob for the first time. Alice1 must establish Double Ratchet's sessions and for that requests key bundles. It is assumed that Alice2 is known to Alice1 so there is no request for Alice2 key bundle.



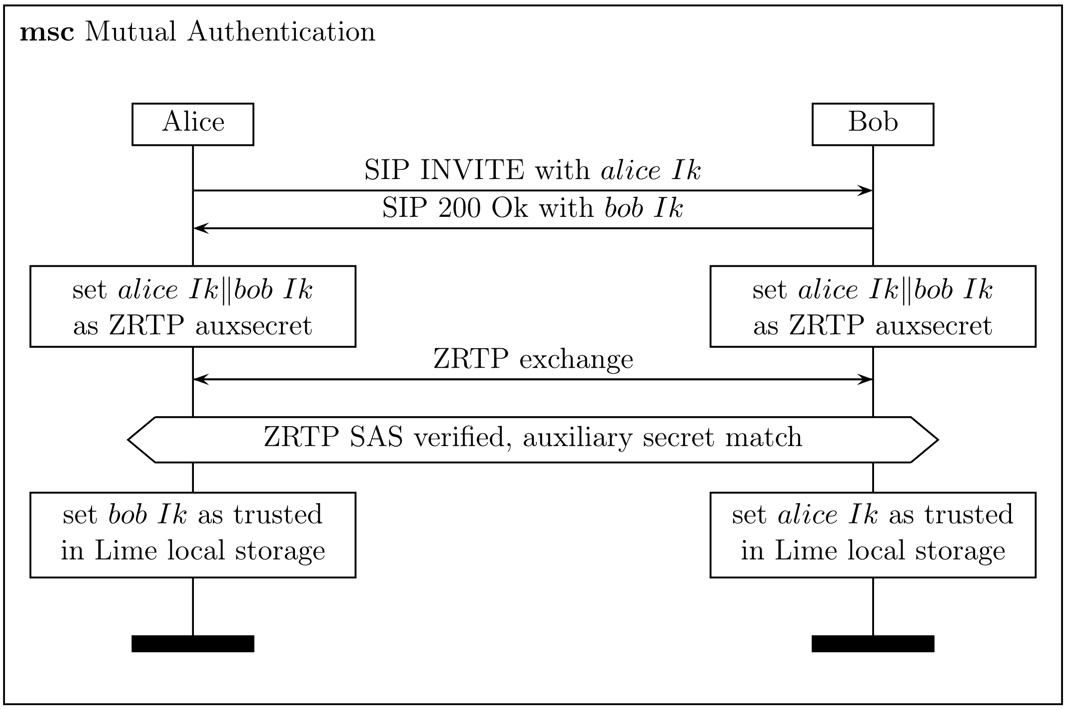
5.5 Mutual authentication

As stated in [2, section 4.1], the parties shall compare their identity public keys otherwise they receive no cryptographic guarantee as to who they are communicating with.

Lime relies on a ZRTP[9] audio call leveraging the MiTM detection offered by the ZRTP short authentication string to authenticate peer identity key. ZRTP auxiliary secret is used to compare both parties identity public keys in the following way:

- parties exchange they identity public key in the signaling channel at call establishment
- parties use $caller\ Ik \| receiver\ Ik$ as ZRTP auxiliary secret
- when ZRTP key exchange is complete, parties check that auxiliary secret are matching and perform vocal SAS comparison(if not performed before)
- if all is Ok, each party store peer Ik as trusted in the Lime local storage. If the peer key is already present, check it matches the one retrieved through the ZRTP channel

In the following diagram *alice Ik* and *bob Ik* refer to the identity public key associated to the particular device of alice and bob involved in the audio call.



5.6 Keys and sessions management

Key lifetime management is the responsibility of the client device only. The X3DH server is not involved in their management. On a regular schedule (once a day is recommended), the devices must run the *update* function to check keys validity, renew and/or delete of outdated ones. Several settings are involved in the *update* operation, they are all defined in *lime_settings.hpp*.

5.6.1 Identity Key

Is valid for all the lifetime of a device.

5.6.2 Signed Pre-key

SPK_lifeTime_days is a constant (default 7 days) defining the key validity period. Once a key is outdated, a new one is generated, signed and uploaded on the X3DH server. Old keys are kept in storage with an invalid status.

SPK_limboTime_days is a constant (default is 30 days) defining the period invalid keys are kept by the device.

5.6.3 One-time Pre-key

These can be use only once, so any usage implies immediate deletion:

- when the server delivers a One-time Pre-key, it immediately deletes it

- when a client makes use of one of its One-time Pre-key(upon reception from peer of an X3DH init message using that key), it immediately deletes it

During *update*, a device requests to the X3DH server the list of its own OPk available on server. The device can upload more keys if there are not enough online and track which keys where delivered by server but not yet used by peers.

The three following constants can be overridden at runtime by parameters passed to the *update* or *create_user* functions

OPK_serverLowLimit is a constant (default 100) defining the lower bound of keys count present on server, during an update, if there is less occurrences of keys on the X3DH server, the client will generate and upload a batch of One-time Pre-keys.

OPK_batchSize is a constant (default 25) defining the number of keys generated and uploaded to the server if an upload is needed

OPK_initialBatchSize is a constant (default 100) defining the number of keys generated and uploaded to the server at user's creation.

During *update*, the client will update the status of One-time Pre-keys in local storage to reflect the information provided by server:

- a key still available on server is left with its original available status
- any available key still in local storage but not anymore on server get the dispatched status

During *update*, the device deletes One-time Pre-keys having the dispatched status for a too long period of time.

OPK_limboTime_days is a constant (default 37 days) defining the period dispatched One-time Pre-keys are kept by the device.

5.6.4 Double Ratchet Sessions

More than one double ratchet session may exists between two devices but only one shall be active. The encryption is always performed by the active session and on reception, the session successfully decrypting the message become the active session. Staled sessions are kept to allow decrypting of delayed or unordered messages.

DRSession_limboTime_days is a constant (default 30 days) defining the period stale sessions are kept by the device.

5.6.5 Skipped message keys

Messages may be out of order on reception, Double Ratchet specifies how skipped intermediate messages keys generated to decrypt a received message shall be stored to allow the decryption of out-of-order messages. After a certain number of message successfully decrypted by a double ratchet session, messages are considered lost and the stored message keys are deleted from local storage.

maxMessagesReceivedAfterSkip is a constant (default 128) linked to a double ratchet receiving chain (a new chain is started within the session each time a Diffie-Hellman ratchet is performed). Each time a skipped message key is stored in this chain, a counter is reset. Each time a message is decrypted by the session, all skipped message key chain counters are increase by one. When the counter reach $maxMessagesReceivedAfterSkip$, the skipped message key chain is deleted.

5.7 Local Storage

The local storage is provided by an sqlite3 database accessed using the SOCI library [13].

5.7.1 Devices tables

Two tables store the devices records. One for local devices defining an Uid used as foreign key in other table to relate to a local device definition, the other one stores peer devices information.

lime_LocalUsers

- Uid: integer primary key
- $UserId$: the device Id provided by linphone, it shall be the GRUU.
- Ik: Identity key, an EdDSA key stored as public key || private key
- $server$: the X3DH server URL to be used by this user
- $curveId$: 0x01 for Curve 25519 or 0x02 for Curve 448. This value must match the X3DH server setting

lime_PeerDevices *Note:* Records in this table are not linked to a local user but shared between them in order to avoid getting multiples records storing the same information.

- Did: integer primary key
- $DeviceId$: the peer device Id, it shall be its GRUU
- Ik: the peer's public EdDSA identity key.
- $verified$: flag: 0 for peer's identity not verified, 1 for peer's identity verified, see this document section 5.5 for usage

5.7.2 X3DH tables

The X3DH dedicated tables store local users Signed Pre-keys and One-time Pre-keys, records are linked to a local user through a foreign key: Uid.

X3DH_SPK *Note:* signature is computed and uploaded to server at key generation but is then not needed so not stored.

- $SPKid$: a random Id (unsigned integer on 31 bits) to identify the key, is public so make it random
- SPK: an ECDH key (stored as public key||private key)
- $timeStamp$: is set to current time when the key status is set to invalid
- $Status$: set to valid(1) at creation and then to invalid(0) when a new key is generated
- Uid: link to $lime\_LocalUsers$: identify which local user own this record

X3DH_OPK

- $OPKid$: a random Id (unsigned integer on 31 bits) to identify the key, is public so make it random
- OPK: an ECDH key (stored as public key||private key)
- $timeStamp$: is set to current time when the key status is set to dispatched
- $Status$: set to online(1) at key generation and to dispatched(0) the key is not found anymore on server by the *update* request
- Uid: link to $lime\_LocalUsers$: identify which local user own this record

5.7.3 Double ratchet tables

The Double Ratchet dedicated tables store all material needed for the Double Ratchet session, including dedicated table for skipped keys. Records are linked to local and peer devices through a foreign keys: Uid and Did.

DR_sessions

- Did: link to $lime\_PeerDevices$: identify peer device for this session
- Uid: link to $lime\_LocalUsers$: identify local device for this session
- $sessionId$: integer primary key
- Ns: index of current sending chain
- Nr: index of current receiving chain
- PN: index of previous sending chain
- DHr: peer's ECDH public key
- DHs: self ECDH key (public||private)
- RK: Diffie-Hellman Ratchet Root key
- CKr: Symmetric Ratchet receiver chain key

- CKs: Symmetric Ratchet sender chain key
- AD: session Associated Data (provided at session creation by X3DH)
- $Status$: active(1) or stale(0), only one session can be active between two devices
- $timeStamp$: is set to current time when the status is set switched from active to stale
- $X3DHInit$: holds the X3DH init message while it is requested to insert it in message header

The two following tables store the skipped message keys, indexed by peer's ECDH public key and receiving chain index

DR_MSk_DHr

- $DHid$: integer primary key
- $sessionId$: link to $DR\_sessions$: identify which session this chain of skipped message keys belongs to
- DHr: peer's ECDH public key associated to this message keys
- $received$: count the messages successfully decrypted after the last insertion of a skipped message key in this chain. Is used to delete old message keys.

DR_MSk_MK

- $DHid$: link to DR_MSk_DHr : identify which receiving chain this message key belongs to.
- Nr: index of skipped message in the receiving chain
- MK: the message key$\langle 32bytes\rangle \| $initial vector$\langle 16bytes\rangle$

5.8 Summary of cryptographic algorithms used

5.8.1 Double Ratchet

- Diffie-Hellman uses either X25519 or X448
- KDF are HKDF[8] based on Sha512
- ENCRYPT is AES256-GCM with a 128bits authentication tag

5.8.2 X3DH

- Diffie-Hellman uses either X25519 or X448
- HKDF uses Sha512
- Signature uses EdDSA25519 or EdDSA448
- EdDSA keys are converted to ECDH keys to perform classic ECDH

5.8.3 Cryptographic libraries

Elliptic curves operations are provided by decaf library[11], version 0.9.4 or above: X25519, X448, EdDSA25519, EdDSA448 and conversion function from ECDH key to EdDSA key format.

Hash(HmacSha512) and encryption(AES256-GCM) are provided by mbedtls library[12]. Version 2.1 or above.

Note : These libraries are not accessed directly but through the bctoolbox abstraction library.

6 Protocol specification

This section describes the details of packets structures.

Notes : Keys are intended as public keys and their size depends on the selected curve indicated in the packets header. Following sizes applies:

- Curve 25519 ECDH : 32 bytes
- Curve 25519 EdDSA : 32 bytes
- Curve 25519 Signature : 64 bytes
- Curve 448 ECDH : 56 bytes
- Curve 448 EdDSA : 57 bytes
- Curve 448 Signature : 114 bytes

Keys are stored and distributed in the formats described in [5] and [6].
Others numeric values(counts, Ids, counters) are unsigned integers in big endian.

6.1 Double Ratchet message

This packets are exchanged between devices. The system running in asynchronous mode, they are sent to and stored by a server and are fetched by final recipient when online. The server in charge of storing/routing the packets shall fan-out to the respective recipients not all the incoming message but only the part addressed to them.

Double Ratchet messages are composed of a header and the cipher message. Sender produces one header per recipient device while the cipher message is common to all of them.
Definitions:

- Protocol Version: 0x01.
- Packet Type: [0x01(no X3DH init in the header), 0x02(an X3DH init is present in the header)]
- Curve Id: [0x01(curve 25519), 0x02(curve 448)]

6.1.1 Header

byte 0	byte 1	byte 2	byte 3
Protocol Version [0x01]	Packet type [0x01,0x02]	Curve Id [0x01,0x02]	
X3DH Init $\langle$variable size$\rangle${0,1} This part is present only if Packet type = 0x02			
Ns		PN	
DHs$\langle 32, 56 bytes\rangle$ ...			
Random Seed encrypted using DR session$\langle 32 bytes\rangle$ ...			
Double Ratchet AEAD authentication tag$\langle 16 bytes\rangle$ ...			

6.1.2 Cipher Message

byte 0	byte 1	byte 2	byte 3
Cipher text produced by AEAD using a derivative of Random Seed <variable size> ...			
AEAD authentication tag$\langle 16 bytes\rangle$ ...			

6.1.3 X3DH init

byte 0	byte 1	byte 2	byte 3
OPk flag [0x00,0x01]			
EdDSA Identity Key$\langle 32, 57 bytes\rangle$ ...			
ECDH Ephemeral Key$\langle 32, 56 bytes\rangle$ ...			
Signed Pre-key Id			
One Time Pre-key Id{0,1} only if OPk flag = 0x01			

6.2 X3DH message

Theses packets are exchanged between devices and X3DH key server.

The packets are sent to server using HTTPS protocol. Clients identify themselves to server by setting their device Id(GRUU) in the HTTPS packet *From* field. Server challenges the client with a nonce and expect a SHA-256 digest of the password of their user account on the SIP server. X3DH server must have access to the SIP register server database to be able to authenticate clients.

X3DH packets are composed of a header and the content:
Protocol Version$\langle 1 byte\rangle \|$ Message Type $\langle 1 byte\rangle \|$ Curve Id $\langle 1 byte\rangle \|$ Packet content. Definitions:

- Protocol Version: 0x01.
- Message Type :
  - *0x01: register User* : a device register its Id and Identity key on X3DH server.
  - *0x02: delete User* : a device delete its Id and Identity key from X3DH server.

- *0x03: post Signed Pre-key* : a device publish a Signed Pre-key on X3DH server.
- *0x04: post One-time Pre-keys* : a device publish a batch of One-time Pre-keys on X3DH server.
- *0x05: get peers key bundles* : a device request key bundles for a list of peer devices.
- *0x06: peers key bundles* : X3DH server respond to device with the list of requested key bundles.
- *0x07: get self One-time Pre-keys* : ask server for self One-time Pre-keys Ids available.
- *0x08: self One-time Pre-keys* : server response with a count and list of all One-time Pre-keys Ids available.
- *0xFF: error* : something went wrong on server side during processing of client message, server respond with details on failure

- Curve Id: [0x01(curve 25519), 0x02(curve 448)]

To device generated messages *register User, delete User, post Signed Pre-key* and *post One-time Pre-key*, X3DH server responds on success with the original message header: Protocol Version || Message type || Curve Id

6.2.1 Register User Packet

<table>
<tr><th>byte 0</th><th>byte 1</th><th>byte 2</th><th>byte 3</th></tr>
<tr><td>Protocol Version [0x01]</td><td>Packet type [0x01]</td><td>Curve Id [0x01,0x02]</td><td rowspan="2"></td></tr>
<tr><td colspan="4">EdDSA Identity Key$\langle 32, 57 bytes\rangle$
...</td></tr>
</table>

6.2.2 Delete User Packet

<table>
<tr><th>byte 0</th><th>byte 1</th><th>byte 2</th><th>byte 3</th></tr>
<tr><td>Protocol Version [0x01]</td><td>Packet type [0x02]</td><td>Curve Id [0x01,0x02]</td><td></td></tr>
</table>

6.2.3 post Signed Pre-key Packet

<table>
<tr><th>byte 0</th><th>byte 1</th><th>byte 2</th><th>byte 3</th></tr>
<tr><td>Protocol Version [0x01]</td><td>Packet type [0x03]</td><td>Curve Id [0x01,0x02]</td><td rowspan="2"></td></tr>
<tr><td colspan="4">ECDH Signed Pre-key$\langle 32, 56 bytes\rangle$
...</td></tr>
</table>

6.2.4 post One-time Pre-key Packet

<table>
<tr><th>byte 0</th><th>byte 1</th><th>byte 2</th><th>byte 3</th></tr>
<tr><td>Protocol Version [0x04]</td><td>Packet type [0x04]</td><td>Curve Id [0x01,0x02]</td><td>keys Count MSB</td></tr>
<tr><td>keys Count LSB</td><td colspan="3">One-time Pre-key bundle$\langle 36, 60 bytes\rangle$\{keys Count\}</td></tr>
<tr><td colspan="4">...</td></tr>
</table>

with One-time Pre-key bundle:

<table>
<tr><th>byte 0</th><th>byte 1</th><th>byte 2</th><th>byte 3</th></tr>
<tr><td colspan="4">ECDH One-Time Pre-key$\langle 32, 56 bytes\rangle$
...</td></tr>
<tr><td colspan="4">One-Time Pre-key Id</td></tr>
</table>

6.2.5 get peers key bundles Packet

byte 0	byte 1	byte 2	byte 3
Protocol Version [0x04]	Packet type [0x05]	Curve Id [0x01,0x02]	request Count MSB
request Count LSB	request{request Count}		
...			

with request:

byte 0	byte 1	byte 2	byte 3
Device Id size		Device Id$\langle$variable size$\rangle$...	
...Device Id$\langle$variable size$\rangle$			

6.2.6 peers key bundles Packet

byte 0	byte 1	byte 2	byte 3
Protocol Version [0x04]	Packet type [0x06]	Curve Id [0x01,0x02]	bundles Count MSB
bundles Count LSB	key Bundle{bundles Count}		
...			

with key Bundle:

byte 0	byte 1	byte 2	byte 3
OPk flag [0x00,0x01]			
EdDSA Identity Key$\langle 32, 57 bytes\rangle$ ...			
ECDH Signed Pre-key$\langle 32, 56 bytes\rangle$ ...			
Signed Pre-key Id			
ECDH Signed Pre-key Signature$\langle 64, 114 bytes\rangle$ ...			
ECDH One-Time Pre-key$\langle 32, 56 bytes\rangle${0,1} only if OPk flag = 0x01 ...			
One-Time Pre-key Id{0,1} only if OPk flag = 0x01			

6.2.7 get Self OPks Packet

byte 0	byte 1	byte 2	byte 3
Protocol Version [0x04]	Packet type [0x07]	Curve Id [0x01,0x02]	OPk Count MSB
OPk Count LSB	OPk Id$\langle 4 bytes\rangle${OPk Count}		
...			

6.2.8 self OPks Packet

byte 0	byte 1	byte 2	byte 3
Protocol Version [0x04]	Packet type [0x08]	Curve Id [0x01,0x02]	

6.2.9 Error Packet

byte 0	byte 1	byte 2	byte 3
Protocol Version [0x01]	Packet type [0xFF]	Curve Id [0x01,0x02]	Error Code[0x00-0x08]
Optional error message of variable size Null terminated ASCII string ...			

With Error codes in:

- 0x00: **bad content type** : HTTPS packet *content-type* is not "x3dh/octet-stream"
- 0x01: **bad curve** : client and server curve mismatch.
- 0x02: **missing Sender Id** : HTTPS packet *from* is not set.
- 0x03: **bad protocol version** : client and server X3DH protocol version number mismatch.
- 0x04: **bad size** : the size of received packet is not the expected one
- 0x05: **user already in** : trying to register a user on X3DH server but it is already in the database
- 0x06: **user not found** : an operation concerning a user couldn't be performed because he was not found in server database.
- 0x07: **db error** : server encounter problem with its database.
- 0x08: **bad request** : malformed peer bundle request.

References

[1] Moxie Marlinspike, Trevor Perrin (editor) : *"The Double Ratchet Algorithm"*, Revision 1, 2016-11-20. https://signal.org/docs/specifications/doubleratchet/

[2] Moxie Marlinspike, Trevor Perrin (editor) : *"The X3DH Key Agreement Protocol"*, Revision 1, 2016-11-04. https://signal.org/docs/specifications/x3dh/

[3] Moxie Marlinspike, Trevor Perrin (editor) : *"The Sesame Algorithm: Session Management for Asynchronous Message Encryption"*, Revision 2, 2017-04-14. https://signal.org/docs/specifications/sesame/

[4] Trevor Perrin (editor) : *"The XEdDSA and VXEdDSA Signature Schemes"*, Revision 1, 2017-10-20. https://signal.org/docs/specifications/xeddsa/

[5] A. Langley, M. Hamburg, and S. Turner, : *"Elliptic Curves for Security."*, Internet Engineering Task Force; RFC 7748 (Informational); IETF, Jan-2016. http://www.ietf.org/rfc/rfc7748.txt

[6] S. Josefsson and I. Liusvaara : *"Edwards-Curve Digital Signature Algorithm (EdDSA)"*, Internet Engineering Task Force; RFC 8032 (Informational); IETF, Jan-2017. https://tools.ietf.org/html/rfc8032

[7] J. Rosenberg :*"Obtaining and Using Globally Routable User Agent URIs (GRUUs) in the Session Initiation Protocol (SIP)"*, Internet Engineering Task Force; RFC 5627 (Standards Track); IETF, Oct-2009. https://tools.ietf.org/html/rfc5627

[8] H. Krawczyk and P. Eronen :*"HMAC-based Extract-and-Expand Key Derivation Function (HKDF)"*, Internet Engineering Task Force; RFC 5869 (Informational); IETF, May-2010. https://tools.ietf.org/html/rfc5869

[9] P. Zimmermann, A. Johnston and J. Callas :*"ZRTP: Media Path Key Agreement for Unicast Secure RTP"*, Internet Engineering Task Force; RFC 6189 (Informational); IETF, April-2011. https://tools.ietf.org/html/rfc6189

[10] Whisper Systems :*"Signal Protocol C Library"*, https://github.com/WhisperSystems/libsignal-protocol-c

[11] Mike Hamburg :*"Ed448-Goldilocks"*, https://sourceforge.net/projects/ed448goldilocks/

[12] ARM mbed :*"mbed TLS"*, https://tls.mbed.org/

[13] SOCI :*"SOCI - The C++ Database Access Library."*, https://github.com/SOCI/soci