Pryvate Instant Message Encryption v3.0 (Pime v3.0)

Johan Pascal

March 31st, 2020
Draft

Contents

1 Introduction

Pryvate Instant Message Encryption (Pime) 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. Pime supports multiple devices per user and multiple users per device.

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

Pime 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 users' networks; using different curves.

Note: Pime v1.0 was based on SCIMP. This document presents Pime v2.0, which is neither related to nor compatible with Pime v1.0. In this document the use of the term Pime refers to Pime v2.0.

2 Notations

A B denotes the concatenation of byte sequences A and B

A(value) the bytes sequence A size is value. For example, key(32bytes) 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

"The Double Ratchet algorithm[1] 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

"'X3DH'(or 'Extended Triple Diffie-Hellman')[2] 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

"The Sesame algorithm[3] 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 Pime 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 Pime v2.0 implementation does not follow the Signal specification documentation[1][2][3]. 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, which:

1. When accepting membership, a group member creates its sender key and distributes it to all other members using pairwise Double Ratchet sessions; then
2. Members use their sender key to encrypt messages to the group, deriving it by 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, Pime provides the following mechanism to save bandwith when sending message to multiple devices:

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

   DR encrypted random key{one for each recipient device}
   Message encrypted using the random key

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

This mechanism is optional and the default behavior of the library is to use it when it saves upload bandwidth, using a regular encryption in the Double Ratchet message otherwise.

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 is 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] into a 32 bytes key and a 16 bytes nonce. The DR session encrypts 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 the use of a SIV based AEAD encryption scheme.

The Pime implementation of the Double Ratchet Chain Key derivation is described in 5.3.3 of this document. The message key(32bytes) and initialisation vector (16bytes) 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.

Pime 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 an 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]. Pime uses direct encoding specified in [5] for its ECDH public keys and [6] for its EdDSA keys but the type of curve is present in the messages header.

4.3 Authentication

X3DH specification mentions [2, section 4.1] the necessity of an identity authentication mechanism and libsignal[10] implements a key fingerprints comparison to provide it. Pime makes use of a ZRTP[9] call with an oral SAS verification to provide mutual identity authentication. See implementation details in section 5.6

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: A 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, the different terms used in this document are defined here:

- device Id: a unique string associated to a device, provided to Pime by Pryvate. It shall be the GRUU[7]
- user Id: a unique string defining a user or a group of users, provided to Pime by Pryvate. 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 HKDF

The HKDF function, as described in RFC5869 [8] is used in both X3DH and Double Ratchet. Pime uses an implementation of HKDF based on SHA512. Its prototype in the pseudo-code is as follow, all inputs and output have variable size. salt is optionnal and the function may be used without(set to null in the pseudo-code). The size of the generated output key material, okm, is arbitrary and depends only on request not on input or hash algorithm used.

function HKDFSha512(salt, ikm, info)
return okm
end function

5.3 Double Ratchet

5.3.1 Diffie-Hellman

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

5.3.2 KDF_RK

As recommended in [1, section 5.2], this function uses HKDF[8] based on SHA512. The salt is RK and ikm is the output of ECDH(DH_out). The info string is "DR Root Chain Key Derivation". DH_out size depends on ECDH function used, X25519 produces a 32 bytes output, X448 a 56 bytes output.

function KDF_RK(RK(32bytes), DH_out(32, 56bytes))
info ← "DR Root Chain Key Derivation"
RK(32bytes) CK(32bytes) ← HKDFSha512(RK, DH_out, info)
return RK(32bytes), CK(32bytes)
end function

5.3.3 KDF_CK

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

function KDF_CK(CK(32bytes))
MK IV ← HmacSha512(ChainKey, 0x01)
CK ← HmacSha512(ChainKey, 0x02)
return CK(32bytes), MK(32bytes), IV(16bytes)
end function

5.3.4 RatchetEncrypt

The RatchetEncrypt function described in [1, section 3.4] is not directly used to encrypt the message. Instead, to provide the group chat (see section 4.1.1) capabilities, an encryption request must include 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
DRmessage: encryption output (Double Ratchet Message) for this recipient device
peerDeviceStatus: an ouput giving a status on the recipient: unknown(till now thus), untrusted or trusted

The ouput may be completed by a Cipher Message holding the encrypted plain text according to the selected encryption policy,

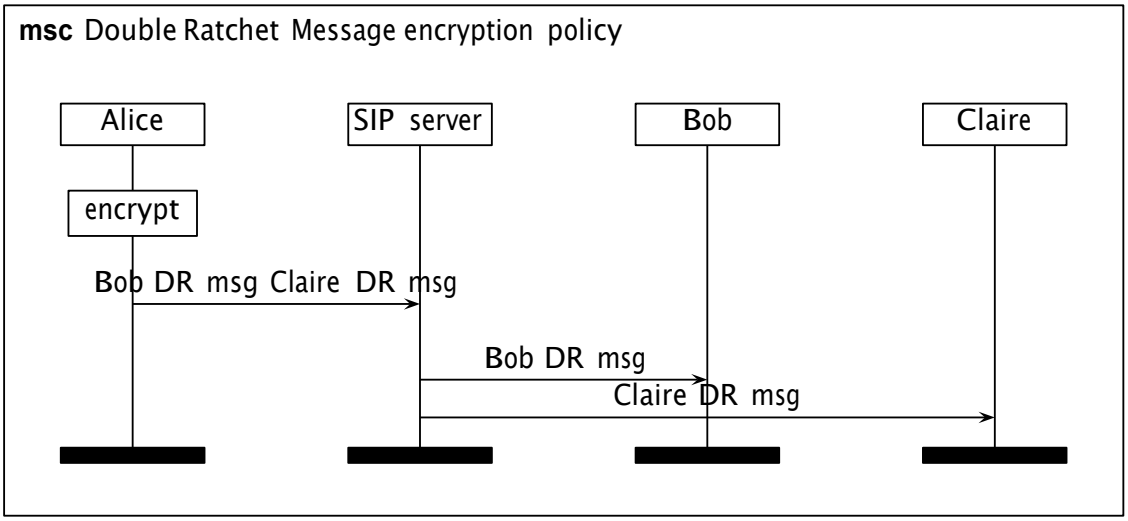
The message is sent from the sender device to one recipient user (with one user Id and one or more associated device Id) but also distributed to other devices registered to the same sender user. Recipient devices in the list must all be linked to this, unique, recipient user Id or to the sender user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

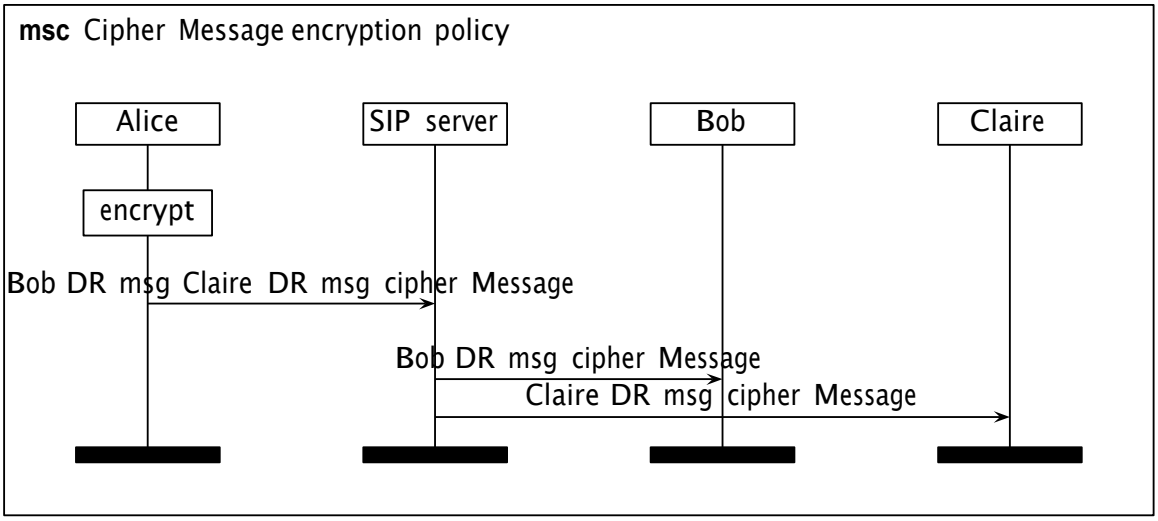
- Alice, Bob and Claire are users Id. Each of them have several (nA, nB, nC) associated devices with devices Id Alice.1, Alice.2, .., Alice.nA
- Alice, Bob and Claire are members of a group with user Id Group
- If Alice.1 sends a message to Bob, the inputs for the RatchetEncrypt function must include Bob as recipient user and Bob.1, .., Bob.nB, Alice.2, .., Alice.nA as list of recipient devices.
- If Alice.1 sends a message to Group, the inputs for the RatchetEncrypt function must include Group as recipient user and Bob.1, .., Bob.nB, Alice.2, .., Alice.nA, Claire.1, .., Claire.nC as list of recipient devices.
- The Pime library does not perform any check on the link between user Id and device Id and will not generate any error if the RatchetEncrypt arguments are Bob as recipient user and Bob.1, .., Bob.nB, Alice.2, .., Alice.nA, Claire.1 as list of recipient devices. The error would instead be detected by Claire.1 during decryption. See 5.3.6 for details on the use of Associated Data to detect mismatching association of user Id and device Id.

Encryption policy : As stated in section 4.1.1, the plain message can be:

- encrypted directly in the Double Ratchet messages.(Double Ratchet Message encryption policy)
- encrypted by a random key in a common cipher message, the random key being encrypted into the Double Ratchet messages.(Cipher Message encryption policy)

The two policies are represented on the following diagrams. It is assumed that the server will dispatch only the requested parts to recipients and not the whole upload. Double Ratchet sessions establishment are not shown on the diagram but are assumed to be already completed between all participants. All participants have one device only.





Selection of the encryption policy according to policy parameter, recipientLists and plain text characteristics. The policy parameter is given at runtime by caller and default to optimize Upload Size if omitted. Possible values of this parameter are:

- Double Ratchet Message: the plain text is encrypted and embeded in the Double Ratchet message.
- cipher Message: the plain text is encrypted in a cipher message with a random key, itself encrypted in the DR message.
- optimize Upload Size: for each message, select the mode which minimize the upload size. This is the default policy.
- optimize Global Bandwidth: for each message, select the mode which minimize upload + download size.

Note : the optimize modes do not take in consideration the multipart boundary added by the presence of an extra part holding the cipher Message.

```
function MessageEncrypt(recipientList, plain, sourceDeviceId, recipientUserId, policy)
    switch policy do
        case DoubleRachetMessage
            DRMessageEncrypt(recipientList, plain, sourceDeviceId, recipientUserId)
        case cipherMessage
            cipherMessageEncrypt(recipientList, plain, sourceDeviceId, recipientUserId)
        case optimize Upload Size
            n ← number of recipients in the recipientList
            DRM essageSize ← n × plain size
            cipherMessageSize ← (plain size+authTag size)+n×(randomSeed size)
            if DRmessageSize ≤ cipherMessageSize then
                DRMessageEncrypt(recipientList, plain, sourceDeviceId, recipientUserId)
            else
                cipherMessageEncrypt(recipientList, plain, sourceDeviceId, recipientUserId)
```

```
            end if
        case optimize Global Bandwidth
            n ← number of recipients in the recipientList
            DRMessageSize ← 2 × n × plain size
            cipherMessageSize ← (plain size + authTag size)
                                +n×(2×randomSeed size+plain size+authTag size)
            if DRmessageSize ≤ cipherMessageSize then
                DRMessageEncrypt(recipientList, plain, sourceDeviceId, recipientUserId)
            else
                cipherMessageEncrypt(recipientList, plain, sourceDeviceId, recipientUserId)
            end if
end function
```

with following functions definitions:

```
function DRMessageEncrypt(recipientList, plain, sourceDeviceId, recipientUserId)
▷ Encrypts the plain in the Double Ratchet message
    for all r ∈ recipientList do
        AD ← recipientUserId sourceDeviceId r.recipientDeviceId
        r.DRmessage ← RatchetEncrypt(r.session, plain, AD)
    end for
    return recipientList
end function

function cipherMessageEncrypt(recipientList, plain, sourceDeviceId, recipientUserId)
▷ Generate a random key and nonce to encrypt the plain
    randomSeed(32bytes) ← RandomSource
    info ← "DR Message Key Derivation"
    key(32bytes) IV (16bytes) ← HKDFSha512(null, randomSeed, info)
    cipherMessage(plainsize+16bytes) ← Encrypt(key, IV, plain, sourceDeviceId recipientUserId)

▷ Use Double Ratchet sessions to encrypt the random seed used to encrypt the plain
    for all r ∈ recipientList do
        AD ← tag sourceDeviceId r.recipientDeviceId
        r.DRmessage ← RatchetEncrypt(r.session, randomSeed, AD)
    end for
    return recipientList, cipherMessage
end function

function RatchetEncrypt(DRsession, plaintext, AD)
     as described in [1, section 3.4]:
    CKs, mK, IV ← KDF_CK(CKs)
    header ← header(DHs, PN, Ns)
    Ns+ = 1
    UpdateDRsessionInLocalStorage(DRsession)
    return header Encrypt(mK, IV, plaintext, AD X3DH provided AD header)
end function
```

```
function Encrypt(key(32bytes), IV (16bytes), plain, associatedData)
    return AES256-GCM output Auth tag (on plain and associatedData)(16bytes)
end function
```

Header function is specified in section 6.1.1

5.3.5 RatchetDecrypt

The decryption function described in [1, section 3.5] is not directly used to decrypt the message. Pime first assess the presence of a cipher message and depending on it use directly the Double Ratchet or perform 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).

The decryption returns the peer device's status(unknown, unsafe, untrusted or trusted) in case of success or fail in case of failure.

```
function MessageDecrypt(sourceDeviceId,
                        recipientDeviceId, recipientUserId,
                        DRsessionList, DRmessage, cipherMessage)
    if cipherMessage∃ then
        return cipherMessageDecrypt(sourceDeviceId, recipientDeviceId,
                        recipientDeviceId, recipientUserId
                        DRsessionList, DRmessage, cipherMessage)
    else
        return DRMessageDecrypt(sourceDeviceId, recipientDeviceId,
                        recipientDeviceId, recipientUserId
                        DRsessionList, DRmessage)
    end if
end function

function DRMessageDecrypt(sourceDeviceId,
                        recipientDeviceId, recipientUserId,
                        DRsessionList, DRmessage)

    AD ← recipientUserId sourceDeviceId recipientDeviceId
    for all DRsession ∈ DRsessionList do
        if plain ← RatchetDecrypt(DRsession, DRmessage, AD) then
            return plain
        end if
    end for
    return fail
```

```
end function

function cipherMessageDecrypt(sourceDeviceId, recipientDeviceId,
                              recipientUserId, DRsessionList, DRmessage,
                              cipherMessage)

    AD ← tag sourceDeviceId recipientDeviceId
    for all DRsession ∈ DRsessionList do
        if randomSeed ← RatchetDecrypt(DRsession, DRmessage, AD) then
            info ← "DR Message Key Derivation"
            key(32bytes) IV (16bytes) ← HKDFSha512(null, randomSeed(32bytes), info)
            return AEADDecrypt&auth(key, IV, cipher, tag, sourceDeviceId recipientUserId)
        end if
    end for
    return fail
end function
function RatchetDecrypt(DRsession, header payload 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
```

5.3.6 Associated Data

The double ratchet encryption and decryption AEAD scheme uses Associated Data as recommended by X3DH and Double Ratchet specifications[2, section 3.3], [1, section 3.4]. The Associated Data authenticated is composed of:

Cipher Message encryption policy

Message Tag(16bytes) Source deviceId Recipient deviceId X3DHAD(32bytes) DR Header

Double Ratchet Message encryption policy

Recipient UserId Source deviceId Recipient deviceId X3DHAD(32bytes) DR Header

- Message Tag: AEAD authentication tag computed on plaintext and the associated data given to AEAD in cipher Message mode: Source deviceId Recipient UserId.
- Recipient UserId: The inclusion of Recipient UserId allows the recipient device to verify the original intended recipient user. The Recipient UserId is provided to the recipient device along the message by the routing protocol as it may not be the UserId linked to the recipient device but a group user Id.
- Source deviceId and Recipient deviceId: Enforce identification of source and recipient device.
- X3DH AD: Associated data computed at session creation by the X3DH protocol, based on both parties Identity keys and devices Id. See 5.4.3 for details. It is present in the device local storage from the X3DH initialisation completion.

- DR Header: as specified in [1, section 3.4]. See 6.1.1 for details.

5.4 X3DH

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

5.4.1 DH

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

5.4.2 Sig

The signature/verify operation performed is an EdDSA (both EdDSA25519 and EdDSA448 are available). The identity key used is stored in EdDSA format so there is no need to use XEdDSA contrary to the X3DH specifications [2, section 2.2].

5.4.3 Shared Secrets generation

SK is computed as specified in [2, section 3.3 and 2.2]. The salt used for the HKDF function is a zero filled buffer the size of the hash function used, the info parameter is "Pime".

ZeroBuffer(SHA512outputsize(64bytes)) ← 0
SK(32bytes) ← HKDFSha512(ZeroBuffer, F(32, 57bytes) DH1 DH2 DH3 DH4,"Pime")

F is 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 implementation convenience, the actual AD used by the Double Ratchet session is derived from these inputs by the HKDF function producing a fixed size buffer as following:

ZeroBuffer(SHA512outputsize(64bytes)) ← 0
ADinput ← initiatorIk receiverIk initiatorDeviceId receiverDeviceId
AD(32bytes) ← HKDFSha512(ZeroBuffer, 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).

5.4.4 X3DH test server

PHP : An X3DH test server running on nginx/mysql/php docker container is provided with the Pime library source code. This server is not meant to be used in production and its purpose is for testing only. This server lacks user authentication layer, which in real use case is provided by the Pryvate ecosystem.

Nodejs : An X3DH test server running on nodejs is provided with the Pime library source code. This server is not meant to be used in production and its purpose is for testing only. This server lacks user authentication layer, which in real use case is provided by the Pryvate ecosystem.

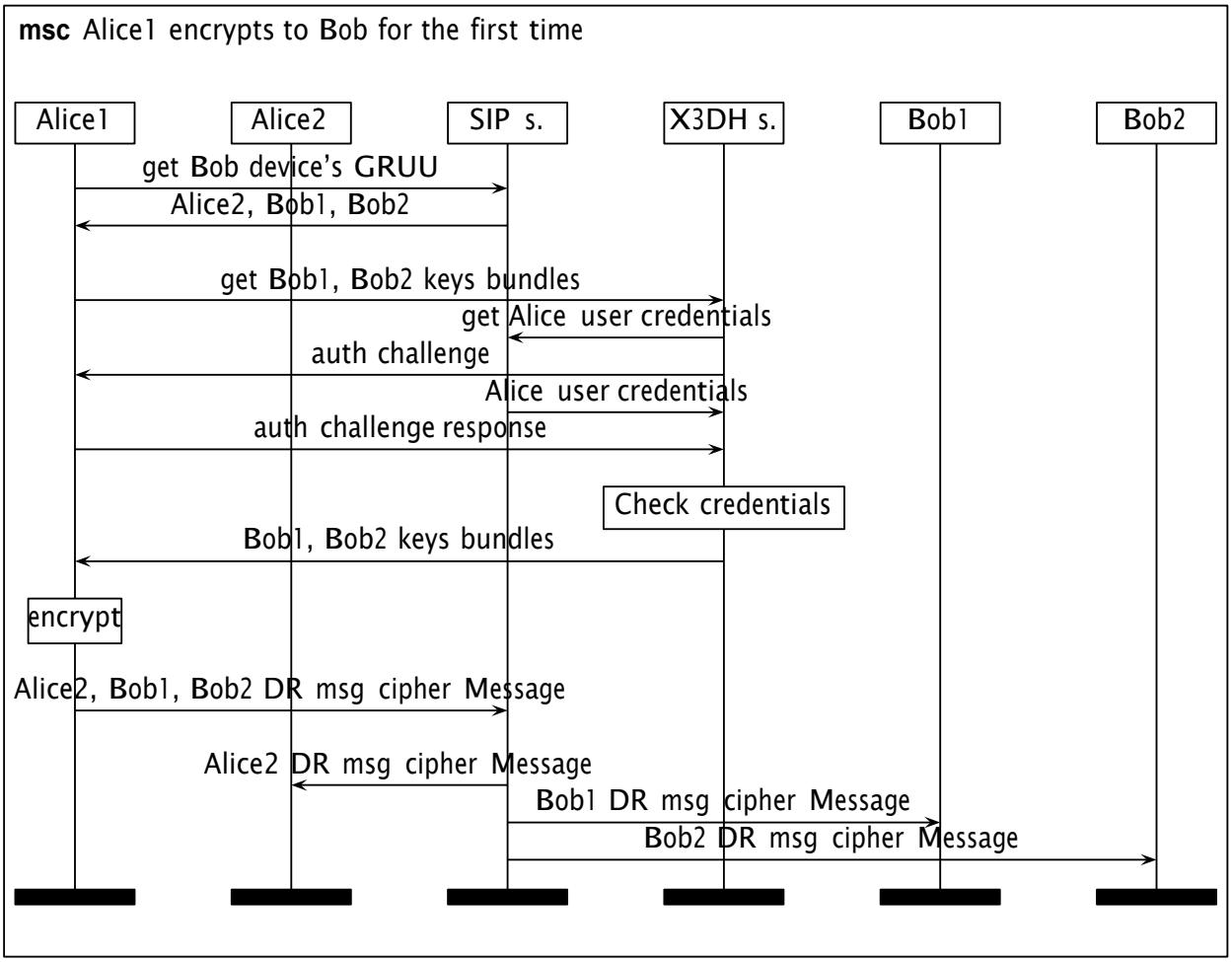
5.5 Sesame

The Sesame requirements are fulfilled as follow:

- Pime 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 Pryvate ecosystem (SIP and conference server). So the loop between client and server during encryption described in the Sesame spec[3] is not relevant. Pime 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 Pime Double Ratchet messageEncrypt function (see section 5.3.4).
- Support for multiple sessions between devices is performed by Pime 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 Pryvate 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 are provided by the Pryvate ecosystem

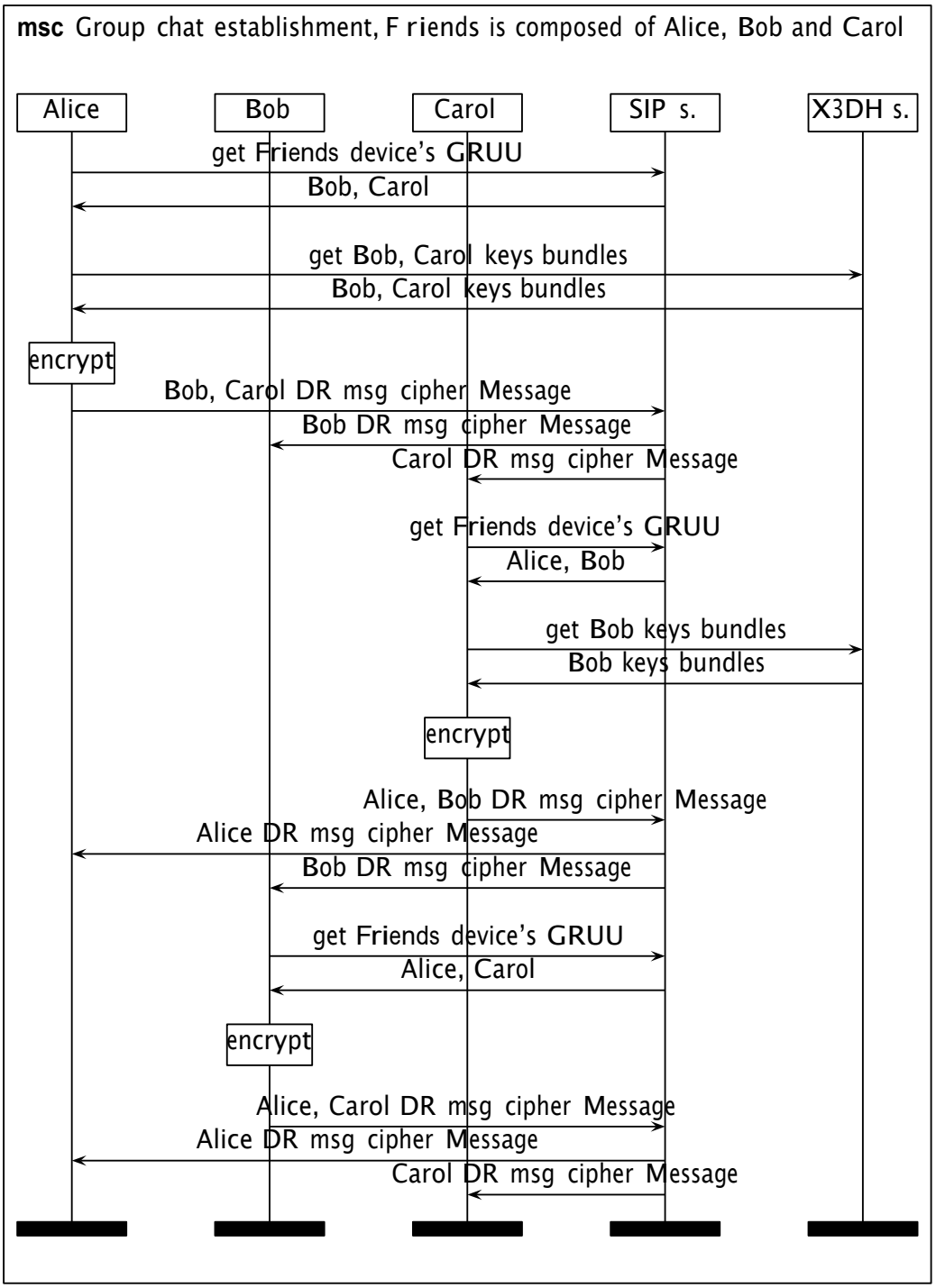
5.5.1 Scenario 1: first encryption, multiple devices

Alice1 encrypts a message to Bob for the first time. Alice1 must establish Double Ratchet sessions and, for that, requests key bundles. It is assumed that Alice2 is known to Alice1; so there is no request for an Alice2 key bundle. The cipher message encryption policy is used.



5.5.2 Scenario 2: group chat

Alice sends a first message to a group called Friends composed of Alice, Bob and Carol. Alice's message is dispatched and then Carol posts a message to the group. Carol's message is dispatched and finally Bob sends a message to the group. It is assumed that users did not exchanged any message prior and that they have one device each. User authentication messages to and from X3DH server are not shown for better readability but the users authentication by X3DH server and X3DH server authentication by users must take place. The cipher message encryption policy is used.



5.6 Mutual authentication and peer device status

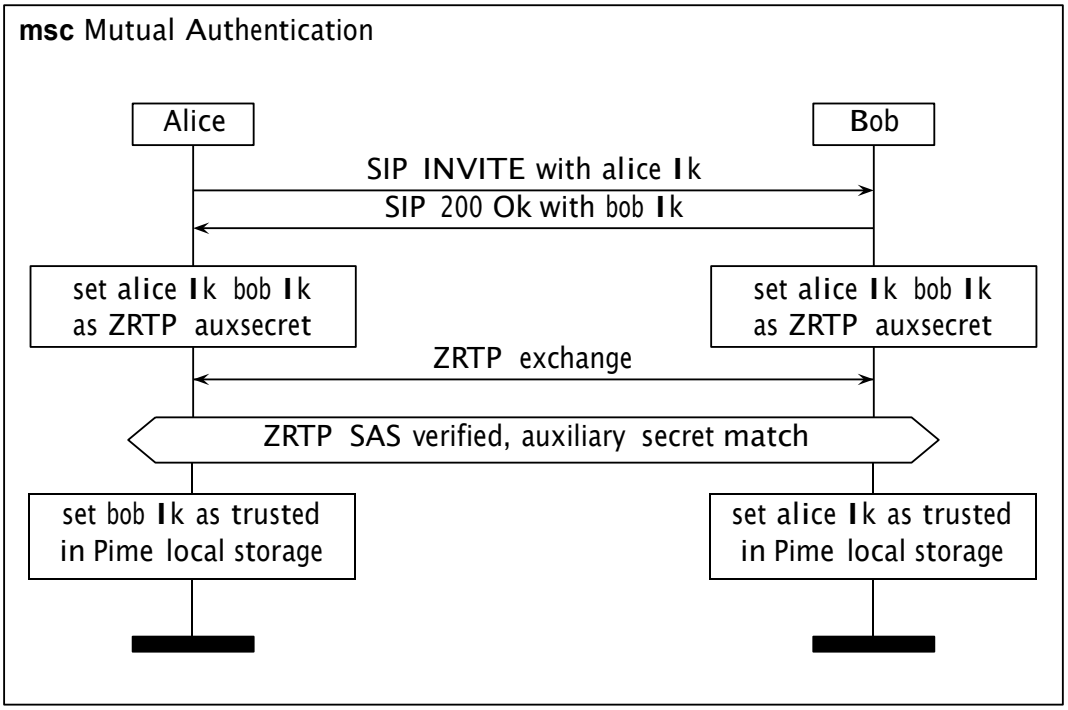
As stated in [2, section 4.1], the parties shall compare their identity public keys otherwise they receive no cryptographic guarantee as to whom they are communicating with. Each peer device has a status available after any encryption or decryption operation which can be:

- unknown: we had no information about this device in the local storage(before the last encryption or decryption), this status spots a newly encountered device and shall be clearly made available to the end user.
- untrusted: it's is not the first interaction with this device, but we never established mutual authentication
- trusted: we already performed the mutual authentication ritual with this peer device.
- unsafe: we know this device, it has been tagged as unsafe by the application(Pryvate).

Pime provides an API to set/get peer devices identity keys and trust level indexed by its device Id. Pryvate uses a ZRTP[9] audio call leveraging the MiTM detection offered by the ZRTP short authentication string to authenticate the peer identity key. ZRTP auxiliary secret is used to compare both parties' identity public keys in the following way:

- parties exchange their identity public keys 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 the auxiliary secret is matching and perform a vocal SAS comparison (if not performed before); and
- if the verification succeeds, each party sets the peer Ik status as trusted in the Pime local storage. If the peer key is already present in the Pime local storage, Pime verifies that it matches the one obtained through the ZRTP channel.

In the following diagram alice Ik and bob Ik refer to the identity public key associated to the particular devices used by Alice and Bob to perform the ZRTP audio call.



5.7 Keys and sessions management

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

5.7.1 Identity Key

Is valid for the lifetime of a device.

5.7.2 Signed Pre-key

SPK_lifeTime_days is a constant (7 days default) 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 so valid but delayed X3DH initiation messages using this signed pre-key can still be processed.

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

5.7.3 One-time Pre-key

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

- when the server delivers a One-time Pre-key, it immediately deletes it; and
- 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 from the X3DH server the list of its own OPk available on the server. The device can upload more keys if there are not enough online and track which keys where delivered by the server but not yet used by comparing the server's OPk list and the OPk actually in local storage.

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

OPK_serverLowLimit is a constant (100 default) defining the lower bound of keys count present on server. During an update, if there are fewer 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 (25 default) defining the number of keys generated and uploaded to the server if an upload is needed.

OPK_initialBatchSize is a constant (100 default) defining the number of keys generated and uploaded to the server at the registration of a new user device.

During update, the client will update the status of One-time Pre-keys in local storage to reflect the information provided by the server. Any key still in local storage but no longer on the server is assigned the dispatched status.

During update, the device deletes One-time Pre-keys having the dispatched status for a longer than pre-determined period of time.

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

5.7.4 Double Ratchet Sessions

More than one double ratchet session may exist 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 becomes the active session. Stalled sessions are kept for a pre-determined period of time to allow decrypting of delayed or unordered messages:

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

In case a peer device is silent, the double ratchet session will never perform a Diffie-Hellman ratchet but only symmetric ratchet steps. To mitigate this problem, a predefined limit on the number of messages encrypted without performing Diffie-Hellman is set (effectively being a limit on the length of the sending chain, each Diffie-Hellman ratchet reset the sending chain counter):

maxSendingChain is a constant (1000 default) defining the maximum length of a sending chain. When reached, the Double Ratchet session status is stalled, forcing the sender device to create a new session; fetching a new key bundle from the X3DH server in order to keep on sending messages.

5.7.5 Skipped message keys

As 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 pre-determined number of messages successfully decrypted by a double ratchet session, skipped messages are considered lost and their stored message keys are deleted from local storage:

maxMessagesReceivedAfterSkip is a constant (128 default) 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, the 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 reaches maxMessagesReceivedAfterSkip, the skipped message key chain is deleted.

5.8 Local Storage

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

5.8.1 Devices tables

Pime_LocalUsers stores data relative to local devices.

- Uid: integer primary key.
- UserId: the device Id provided by Pryvate, 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.

Pime_PeerDevices Note: Records in this table are not linked to a local user but shared among local users in order to avoid storing multiple records containing 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.
- Status: status flag:
  - 0 for untrusted: peer's identity is not verified(default value)

- 1 for trusted: peer's identity was already verified
- 2 for unsafe: peer's device has been flagged as unsafe

see this document section 5.6 for usage.

5.8.2 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 the server at key generation but is then not needed, so not stored locally.

- SPKid: a random Id (unsigned integer on 31 bits) to identify the key. This value being public, it is not a sequence but a random number.
- 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 Pime_LocalUsers: identifies which local user owns this record.

X3DH_OPK

- OPKid: a random Id (unsigned integer on 31 bits) to identify the key. This value being public, it is not a sequence but a random number.
- 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 then to dispatched (0) when the key is not found anymore on the X3DH server by the update request.
- Uid: link to Pime_LocalUsers: identify which local user owns this record.

5.8.3 Double ratchet tables

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

DR_sessions

- Did: link to Pime_PeerDevices: identify peer device for this session.
- Uid: link to Pime_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 stores key chain information: peer's ECDH public keys.

- DHid: integer primary key
- sessionId: link to DR_sessions: identifies to which session this chain of skipped message keys belongs.
- DHr: peer's ECDH public key associated to this message key chain.
- received: counts 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 is the actual storage of message keys.

- DHid: link to DR_MSk_DHr: identifies to which receiving chain this message key belongs.
- Nr: index of the skipped message in the receiving chain.
- MK: the message key(32bytes) initial vector(16bytes).

5.9 Summary of cryptographic algorithms used

5.9.1 Double Ratchet

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

5.9.2 X3DH

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

5.9.3 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 EdDSA key to ECDH 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 messages structures.

Notes : Keys are intended as public keys and their size depends on the selected curve indicated in the messages header. The following sizes apply:

- 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

These messages are exchanged among devices. The system runs in asynchronous mode, and messages are sent to and stored by a server and are fetched by final recipients when online. The server in charge of storing/routing the messages 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 header and payload. The payload is the AEAD output (cipher text and authentication tag) of either a random seed used to encrypt the plain message or the plain message itself according to selected encryption policy. The sender produces one Double Ratchet message per recipient device.
Definitions:

- Protocol Version: 0x01.
- Message Type is a byte with following bit mapping:
  - bit 7 to 2: not used.
  - bit 1: Payload encryption flag:
    - 1: payload in the DR message
    - 0: payload in a cipher message, DR holds the random seed
  - bit 0: X3DH init flag:
    - 1: (X3DH init in the header)
    - 0: (no X3DH init 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]	Message type	Curve Id [0x01,0x02]	
X3DH Init (variable size){0,1} This part is present only if Message type X3DH init flag is set			
Ns		PN	
DHs(32, 56bytes) ...			

6.1.2 Payload in cipher message encryption policy

byte 0	byte 1	byte 2	byte 3
Random Seed encrypted using DR session(32bytes) ...			
Double Ratchet AEAD authentication tag(16bytes) ...			

6.1.3 Payload in Double Ratchet message encryption policy

byte 0	byte 1	byte 2	byte 3
plaintext encrypted using DR session(variable size, same as plaintext) ...			
Double Ratchet AEAD authentication tag(16bytes) ...			

6.1.4 X3DH init

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

6.2 Cipher Message

. The cipher message is produced only when selecting the cipher message encryption policy. The sender produces one cipher message common to all recipients. When present, the cipher message is dispatched along the Double Ratchet messages.(see 5.3.4 for details)

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

6.3 X3DH message

Theses messages are exchanged between devices and the X3DH key server.

The messages are sent to the server using the HTTPS protocol. Clients identify themselves to the server by setting their device Id (GRUU) in the HTTPS packet From field. Server challenges the client with a nonce and expects a 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. Communications between clients and X3DH server are assumed to be secure and the details of this assumption are out of the scope of this document.

X3DH messages are composed of a header and the content:
Protocol Version(1byte) Message Type (1byte) Curve Id (1byte) Message content.
Definitions:

- Protocol Version: 0x01.
- Message Type:
  - 0x01: register User: a device registers its Id and Identity key on X3DH server.
  - 0x02: delete User: a device deletes its Id and Identity key from X3DH server.
  - 0x03: post Signed Pre-key: a device publishes a Signed Pre-key on X3DH server.
  - 0x04: post One-time Pre-keys: a device publishes a batch of One-time Pre-keys on X3DH server.
  - 0x05: get peers key bundles: a device requests key bundles for a list of peer devices.

- 0x06: peers key bundles: X3DH server responds 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, on success, the X3DH server responds with the original message header:
Protocol Version Message type Curve Id

6.3.1 Register User Message

byte 0	byte 1	byte 2	byte 3
Protocol Version [0x01]	Message type [0x01]	Curve Id [0x01,0x02]	
EdDSA Identity Key(32, 57bytes) ...			

6.3.2 Delete User Message

byte 0	byte 1	byte 2	byte 3
Protocol Version [0x01]	Message type [0x02]	Curve Id [0x01,0x02]	

6.3.3 post Signed Pre-key Message

byte 0	byte 1	byte 2	byte 3
Protocol Version [0x01]	Message type [0x03]	Curve Id [0x01,0x02]	
ECDH Signed Pre-key(32, 56bytes) ...			

6.3.4 post One-time Pre-key Message

byte 0	byte 1	byte 2	byte 3
Protocol Version [0x01]	Message type [0x04]	Curve Id [0x01,0x02]	keys Count MSB
keys Count LSB	One-time Pre-key bundle(36, 60bytes){keys Count} ...		

with One-time Pre-key bundle:

byte 0	byte 1	byte 2	byte 3
ECDH One-Time Pre-key(32, 56bytes) ...			
One-Time Pre-key Id			

6.3.5 get peers key bundles Message

byte 0	byte 1	byte 2	byte 3
Protocol Version [0x01]	Message 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(variable size)...	
...Device Id(variable size)			

6.3.6 peers key bundles Message

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

with key Bundle(if a the device has published keys on the server):

byte 0	byte 1	byte 2	byte 3
Device Id size		Device Id(variable size)...	
...Device Id(variable size)			
bundle flag [0x00,0x01]			
EdDSA Identity Key(32, 57bytes)			
...			
ECDH Signed Pre-key(32, 56bytes)			
...			
Signed Pre-key Id			
ECDH Signed Pre-key Signature(64, 114bytes)			
...			
ECDH One-Time Pre-key(32, 56bytes){0,1} only if bundle flag = 0x01			
...			
One-Time Pre-key Id{0,1} only if bundle flag = 0x01			

or key Bundle(if a the device has not published keys on the server):

byte 0	byte 1	byte 2	byte 3
Device Id size		Device Id(variable size)...	
...Device Id(variable size)			
bundle flag [0x02]			

6.3.7 get Self OPks Message

byte 0	byte 1	byte 2	byte 3
Protocol Version [0x01]	Message type [0x07]	Curve Id [0x01,0x02]	OPk Count MSB
OPk Count LSB	OPk Id(4bytes){OPk Count}		
...			

6.3.8 self OPks Message

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

6.3.9 Error Message

byte 0	byte 1	byte 2	byte 3
Protocol Version [0x01]	Message 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 Message 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 could not be performed because the user was not found in server database.
- 0x07: **db error**: server encountered problem with its database.
- 0x08: **bad request**: malformed peer bundle request.

7 IPR

8 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