Messenger End-to-End Encryption Overview

December 6, 2023
Version 1

Meta

TABLE OF CONTENTS

Messaging Security

1. Introduction

This whitepaper describes Meta’s current designs and intention around how end-to-end encryption will operate. Details of Meta’s implementation may differ in places and will likely evolve over time. This white paper should not be read as making any assurances or commitments to users on Meta’s products or services.

We are publishing this whitepaper to help the security community understand and analyze our approach to end-to-end encryption. It provides a technical explanation of the cryptography underpinning the design of Messenger and Instagram Direct's end-to-end encryption system. They use the same encryption scheme, so for brevity throughout the rest of this doc, we will refer only to Messenger. Please visit Messenger's website at https://about.meta.com/actions/protecting-privacy-and-security/ for more information.

Messenger allows people to exchange messages (including chats, group chats, images, videos, voice messages and files) and make voice and video calls around the world. End-to-end encrypted messages, voice and video calls use the Signal protocol outlined below. See "Defining End-to-End Encryption" for information about which communications are end-to-end encrypted.

The Signal Protocol, developed by the Signal Foundation, is the basis for Messenger's end-to-end encryption. This end-to-end encryption protocol is designed to prevent third parties (including Meta) from having plaintext access to messages or calls. For more information on our guiding values for how we are approaching securely building end-to-end encryption for Messenger and Instagram Direct, which is consistent with how WhatsApp currently operates, see our 2022 white paper Security Principles for Private Messaging.

A person can have multiple devices, each with its own set of encryption keys. If the encryption keys of one device are compromised after the device has been revoked, an attacker cannot use them to decrypt messages sent from or to other devices, even devices registered to the same user.

2. Scope

Not all message threads are currently end-to-end encrypted. Messenger is rolling out end-to-end encryption for 1:1 chats, and beginning to make it optionally available for group chats.

Signal Protocol

Messenger uses Signal Protocol as the basis for its 1:1 sessions. The library used by Messenger is based on the Open Source library, with one of its implementations available at https://github.com/signalapp/libsignal-protocol-java/

The protocol described here, for 1:1 messages, describes the protocol as implemented in the above library.

Terms

Public Key Types

- `Identity Key Pair` – A long-term Curve25519 key pair, generated at registration time.
- `Signed Pre Key` – A medium-term Curve25519 key pair, generated at registration time, signed by the `Identity Key`, and rotated on a periodic timed basis.
- `One-Time Pre Keys` – A queue of Curve25519 key pairs for one time use, generated at registration time, and replenished as needed.

Session Key Types

- `Root Key` – A 32-byte value that is used to create `Chain Keys`.
- `Chain Key` – A 32-byte value that is used to create Message Keys.
- `Message Key` – An 80-byte value that is used to encrypt message contents. 32 bytes are used for an AES-256 key, 32 bytes for a HMAC-SHA256 key, and 16 bytes for an IV.

Cryptographic Operations

- `ECDH` - An Elliptic Curve Diffe-Hellman key agreement.
- `HMAC` - A hash-based message authentication code.
- `SHA256` - A cryptographic hash function.
- `HKDF` - A key derivation function based on HMAC-SHA256.
- `AES` - Advanced Encryption Standard, a symmetric block cipher.
- `CBC` - Cipher Block Chaining - a block cipher mode.

Device Registration

At registration time, a Messenger client transmits its public `Identity Key`, public `Signed Pre Key` (with its signature), and a batch of public `One-Time Pre Keys` to the server.
The Messenger server stores these public keys associated with the user's device specific identifier. This facilitates offline session establishment between two devices when one device is offline.

Initiating One-to-one Session Setup

1. One-to-one Session Setup (Sender)

In order for Messenger users to communicate with each other securely and privately, the device sending a message establishes a pairwise encrypted session with each of the recipient's devices. Additionally, the sending device establishes a pairwise encrypted session with all other devices associated with the sender account.

Once these pairwise encrypted sessions have been established, clients do not need to rebuild new sessions with these devices unless the session state is lost or damaged, which can be caused by an event such as an app reinstall or device change.

Messenger uses this "client-fanout" approach for transmitting messages to multiple devices, where the Messenger client transmits a single message `N` number of times to `N` number of different devices. Each message is individually encrypted using the established pairwise encryption session with each device.

To establish a session using the Signal Protocol:

1. The initiating client ("initiator") requests the public `Identity Key`, public `Signed Pre Key`, and a single public `One-Time Pre Key` for each device of the recipient and each additional device of the initiating user (excluding the initiator).
2. The server returns the requested public key values. A `One-Time Pre Key` is only used once, so it is removed from server storage after being requested. If the recipient's latest batch of `One-Time Pre Keys` has been consumed and the recipient has not replenished them, no `One-Time Pre Key` will be returned.

After fetching the keys from server, the initiator starts to establish the encryption session with each individual device using the X3DH protocol:

1. The initiator saves the recipient's `Identity Key` as $I_{recipient}$, the `Signed Pre Key` as `S_recipient`, and the `One-Time Pre Key` as $O_{recipient}$.
2. The initiator generates an ephemeral Curve25519 key pair, $E_{initiator}$.

3. The initiator loads its own Identity Key as $I_{initiator}$.
4. The initiator calculates a master secret as `master_secret = ECDH(`$I_{initiator}$, $S_{recipient}$`) || ECDH(`$E_{initiator}$, $I_{recipient}$`) || ECDH(`$E_{initiator}$, $S_{recipient}$`) || ECDH(`$E_{initiator}$, $O_{recipient}$`)`. If there is no `One Time Pre Key`, the final ECDH is omitted.
5. The initiator uses HKDF to create a `Root Key` and `Chain Keys` from the `master_secret`.

2. One-to-oneSession Setup (Recipient)

After building a long-running encryption session, the initiator can immediately start sending messages to the recipient, even if the recipient is offline. Until the recipient responds, the initiator includes the information (in the header of all messages sent) that the recipient requires to build a corresponding session. This includes the initiator's public components of $E_{initiator}$, $I_{initiator}$, and $O_{recipient}$ (if used).

When the recipient receives a message that includes session setup information:

1. The recipient calculates the corresponding `master_secret` using its own private keys and the public keys advertised in the header of the incoming message.
2. The recipient deletes the `One-Time Pre Key` used by the initiator.
3. The initiator uses HKDF to derive a corresponding `Root Key` and `Chain Keys` from the `master_secret`.

Exchanging One-to-one Messages

Once a session has been established, clients exchange messages that are protected with a `Message Key` using `AES-256` in CBC mode. The client uses client-fanout for all the exchanged messages, which means each message is encrypted for each device with the corresponding pairwise session.

The `Message Key` changes for each message transmitted, and is ephemeral, such that the `Message Key` used to encrypt a message cannot be reconstructed from the session state after a message has been transmitted or received. The `Message Key` is derived from a sender's Chain Key that "ratchets" forward with every message sent.

Additionally, a new `ECDH` agreement is performed with each message round-trip to create a new chain key. This provides forward secrecy through the combination of both an immediate "hash ratchet" and a round trip "DH ratchet." This means that, due to the ephemeral nature of these cryptographic keys, even in a situation where the current encryption keys from a user's device are physically compromised, they should not be usable for decrypting previously transmitted messages.

1. Calculating a Message Key from a Chain Key

Each time a new `Message Key` is needed by a message sender, it is calculated as:

1. `Message Key = HMAC-SHA-256(Chain Key, 0x01)`.
2. The `Chain Key` is then updated as `Chain Key = HMAC-SHA-256(Chain Key, 0x02)`.

This causes the `Chain Key` to "ratchet" forward, and also means that a stored `Message Key` can't be used to derive current or past values of the `Chain Key`.

2. Calculating a Chain Key from a Root Key

Each time a message is transmitted, an ephemeral Curve25519 public key is advertised along with it. Once a response is received, a new `Chain Key` and `Root Key` are calculated as:

1. `ephemeral_secret = ECDH(Ephemeral_sender, Ephemeral_recipient)`.

2. `Chain Key, Root Key = HKDF(Root Key, ephemeral_secret)`.

A chain is only ever used to send messages from one user, so message keys are not reused. Because of the way `Message Keys` and `Chain Keys` are calculated, `Message Keys` can be cached - meaning that messages can arrive delayed, out of order, or can be lost entirely without any problems.

Group messages

Messenger leverages "Server-side fanout" messaging for a majority of group messaging use-cases.
Group sessions are defined as sessions supporting efficient one-to-many communication.

The protocol here describes the “Sender Keys” variant of the Signal Protocol, referred to as “GroupCipher” within Signal’s reference implementation.

1. Group Session Setup

End-to-end encryption of messages sent in Messenger groups utilizes the established one-to-one encrypted sessions, as previously described in the "Initiating One-to-one Session Setup" section, to distribute the "Sender Key" component of the Signal Messaging Protocol. When sending a message to a group session for the first time, and periodically thereafter, a "Sender Key" is generated and distributed to each member device of the session, using the pairwise encrypted sessions.

The first time a Messenger group member sends a message to a group:

1. The sender generates a random 32-byte `Chain Key`.
2. The sender generates a random Curve25519 `Signature Key` key pair.
3. The sender combines the 32-byte `Chain Key` and the public key from the `Signature Key` into a `Sender Key` message.
4. The sender individually encrypts the `Sender Key` to each member of the group, and exchanges it as a normal "one-to-one" message.

2. Exchanging Group Messages

Once a session group session is established. Messages can be sent to those users as follows:

1. The sender derives a `Message Key` from the `Chain Key`, and updates the `Chain Key`.
2. The sender encrypts the message using `AES-256` in CBC mode.
3. The sender signs the ciphertext using the `Signature Key`.

4. The sender transmits the single ciphertext message to the server, which does server-side fan-out to all group participants.

The "hash ratchet" of the message sender's `Chain Key` provides forward secrecy. `Sender Key`s are distributed to new group members as they join, allowing them to decrypt future messages while being unable to decrypt past messages. Whenever a group member leaves, or a device associated with a group member is removed, all group participants clear their `Sender Key` and start over at "Group Session setup.”

Message Retries

Occasionally, a recipient device may fail to decrypt a message, for example, if a ciphertext is corrupted in transit. We have added a retry mechanism on top of the Signal protocol to securely re-attempt message delivery.

When a recipient device fails to decrypt a message, it will send a “retry receipt” back to the original sender, specifying the ID of the message which failed. In order to track which retry receipts may be honored, the sender keeps track of the following things on each message send in an “encrypt journal”:

1. The list of intended recipient devices. This includes other devices of the sender, as well as devices of other users.
2. The “identity version” of each device. The identity version refers to the number of identity changes the local device has processed. Each time a local device processes a new identity key for a remote device, the local device increments and locally assigns a new identity version for the remote device. The current identity version (at the time of message send) is saved in the encrypt journal.
3. The original timestamp of the message send.
4. The number of times a retry attempt has been honored for a specific device and message.

When the sender receives a delivery receipt from a device for a specific message, it will delete the entry for that message and device from the encrypt journal. This ensures that any future retry receipts claiming to be from that device for that message will not be honored. The entry will remain in the encrypt journal until a delivery receipt is received or until 30 days (after which retries cannot be honored).

The sender will honor the retry (i.e. re-encrypt and re-send the message) under the following conditions, in order to uphold privacy and security expectations:

1. The recipient device’s identity key has not changed. This is tracked using the aforementioned “identity version”. If the identity key has changed (determined by whether the current identity version for the device is higher than the version saved in the encrypt journal at send time), this entry in the encrypt journal will be treated as ineligible for any future retries.
2. The message is less than 30 days old. On the server registration side, we expire devices after 30 days offline. This enforces the same expectation on the client side.

3. There have been fewer than 5 previous retry attempts.
4. The message has not previously been delivered successfully to the device requesting the retry (i.e. sender has not received a delivery receipt).

Sender Side Backfill

Unlike most messages, in cases of control messages like “remove for me” and “unsend”, senders will encrypt the messages to devices they learn of at send-time. This is to ensure that these commands, which themselves serve the user’s privacy, can be respected across all relevant devices. These control messages do not themselves contain message content.

As described above, in Messenger, the sender client must specify all the destination devices at the sending time for a message. Any device which is not listed at the sending time will not be able to receive the encrypted message. Each client maintains a list of devices for Messenger accounts the user communicates with, as well as all other devices associated with its own account, and uses this list to specify the destination devices at send-time.

However, when sending a message, it is possible for a client to miss valid devices if its current device list is out-of-date. The mechanism "Sender Side Backfill" is designed so that, in the specific set of cases where it is used, these missed devices may recover from permanently missing the control message. When Messenger receives the encrypted message from the sender, it compares the hash of all the destination devices listed by the sender to the hash of server-side device records of these accounts. If there is a mismatch between two hash values, the server will notify the sender to update the devices list for itself and all the recipient accounts.

The sender client will establish pairwise sessions with those devices using the same method described in the "Initiating One-to-one Session Setup" section, encrypt and resend the original message to these new devices. Users can be made aware of the new devices, as described under “Verifying Keys” below.

Transmitting Media and Other Attachments

Large attachments of any type (video, audio, images, or files) are also end-to-end encrypted:

1. The Messenger user's device sending a message ("sender") generates an ephemeral 32 byte `media_key` From the `media_key`, it derives a 32-byte `media_encryption_key`, a 16-byte `media_initialization_vector`, and a 32-byte `media_hmac_key`.
2. The sender encrypts the attachment with the `media_encryption_key` and `media_initialization_vector` via AES-CBC mode, then appends an `HMAC-SHA-256` of the ciphertext.
3. The sender uploads the encrypted attachment to a blob store via HTTPS.
4. The sender transmits a normal encrypted message to the recipient that contains the `media_key`, a `SHA-256` hash of the encrypted blob, and a pointer to the blob in the blob store.
5. All receiving devices decrypt the message, retrieve the encrypted blob from the blob store, generate the `SHA-256` hash of the encrypted blob, verify the `HMAC-SHA-256`, and decrypt the attachment.

Abuse Reporting

A participant in an end-to-end encrypted conversation may voluntarily notify Meta of unwanted message content. Meta uses such reports to identify users who violate its Community Standards.

The ability to report message content does not represent a relaxation of end-to-end encryption guarantees. Meta will never have access to end-to-end encrypted messages unless a participant voluntarily reports the conversation.

End-to-end encrypted messages include a mechanism outlined below to ensure reports are reliable and authentic, while maintaining end-to-end security.

1. Franking

The franking mechanism must satisfy three main guarantees: *authenticity*, *confidentiality* and *third-party deniability*. The authenticity property ensures that if a user submits a report then the messages in the report must have legitimately originated from the sending device. The confidentiality property ensures that no outside party — including Meta — should learn the content of an end-to-end encrypted message unless a participant voluntarily shares that information. Finally, the third-party deniability property ensures that no party outside of Meta can cryptographically determine the validity of a report.

Franking Tag

Authenticity for messages in a secret conversation is provided by the Franking tag T_F. Senders must send the Franking tag along with each encrypted message. To compute T_F, the sender first generates a 256-bit random nonce N_F. N_F is added to the unencrypted message being transmitted. Next, the entire data structure is serialized into a string M, and T_F is computed as:

$$T_F = \text{HMAC-SHA256}(N_F, M)$$

N_F remains within the serialized, encrypted data sent to the recipient. T_F is transmitted to Meta along with each encrypted message.

Franking Messages

When Meta receives T_F, it uses a Meta key K_F to compute the Reporting tag R_F over T_F and conversation context (e.g., sender and recipient identifiers, timestamp) as:

$$R_F = \text{HMAC-SHA256}(K_F, T_F \mid\mid \text{context})$$

Both T_F and R_F are sent to the recipient along with the encrypted message. R_F is also sent in an ack response to the sender. The recipient decrypts the ciphertext, parses the resulting plaintext to obtain N_F, and verifies the structure of T_F prior to displaying the message. The sender and recipient store the message M, N_F, T_F, R_F and context in their message storage.

2. Reporting Abuse

To report abuse, the recipient of a message submits to Meta the full serialized message plaintext, T_F, R_F, N_F, and `context`. Upon receiving this message Meta first recomputes T_F and then validates R_F using the provided information as well as its internal key K_F.

Security and Privacy

The authenticity properties of the franking mechanism are based on reasonable assumptions about the collision resistance of the SHA256 hash function and the unforgeability of HMAC-SHA256.

Authenticity

In order to “forge” invalid content M', a user must either (a) produce a forged HMAC tag under Meta’s key K_F, or (b), identify a collision N_F', M', $\text{context}'$ such that the HMAC of these values is equal to the HMAC of a different valid message M sent through Meta.

Confidentiality

Similarly, under reasonable assumptions about HMAC-SHA256, the resulting tag reveals no information about the message to Meta or to eavesdroppers.

Third-party deniability

The guarantee holds under the assumption that HMAC-SHA256 is a pseudorandom function and that K_F is never publicly revealed. Meta rotates K_F periodically.

Server-Side Message Storage

Prior to end-to-end encryption, Messenger users relied on server-side storage of their message history so they could access their message history from anywhere while minimizing the use of scarce client-side storage. To preserve this capability, while maintaining E2EE safeguards, we designed a new server-side storage protocol that people can use for their message history.

Further details in its dedicated whitepaper.

Calling

In order to end-to-end encrypt call content, each participant in the call establishes a pairwise crypto session with every other call participant. These sessions are distinct from the sessions established for sending end-to-end encrypted messages, are ephemeral, and their lifecycle is closely tied to the lifecycle of the call (i.e. the sessions are only stored in memory and are purged at the end of the call).

The crypto sessions are negotiated using the Signal protocol mentioned above. However, one major distinction to mention is that sharing the `Identity Key`, `Signed Pre Key`, and `One-Time Pre Key` does not rely on server storage. Instead the keys, packaged as what we call a `Pre Key Bundle`, are distributed as part of the payloads exchange to establish the webRTC connections. These ephemeral crypto sessions are then used to exchange shared secrets from which keys to encrypt the media content are derived. The use of ephemeral Signal sessions, instead of integrating with those used for chat, is intended to improve reliability and performance; as it avoids the possibility of attempting a call on an older corrupted session that will result in a failed connection.

1. One-to-One Calling

For one-to-one calls the audio and video streams are transmitted over a webRTC connection established between the two end-user devices. As such, Messenger achieves end-to-end encryption for call media by using a key, known only to the 2 devices participating in the call, to encrypt the SRTP packets.

The flow to establish the secure encryption sessions and transmit the shared secret is depicted below:

```
Definitions:
 cpk -- callee key pair (identity key and ephemeral key)
 pkb -- pre-key bundle

UserA (caller)                                          UserB (callee)
|                                                                    |
|                                                                    |
| (user initiates call)                                              |
|                                                                        |
| generateRandomData() -> nonce                                          |
| generatePkb() -> (pkb_A_priv, pkb_A_pub, nonce)                        |
|                                                                        |
|                                                                        |
|--------------------> Offer SDP, contains pkb_A_pub ------------------->|
|                                                                        |
|                                                                        |
|                                                          (answer call) |
|                                                                        |
|                     establishSession(cpk_B_priv, pkb_A_pub) -> session |
|                                           generateSharedSecret() -> ss |
|                         encrypt(session, ss) -> (ciphertext, cpkB_pub) |
|                                                                        |
|                                                                        |
|<------------- Answer SDP, contains cpkB_pub, ciphertext <--------------|
|                                                                        |
|                                                                        |
| establishSession(cpk_B_pub, pkb_A_priv) -> session                     |
| decrypt(session, ciphertext) -> ss                                     |
|                                                                        |
|                                                                        |
```

Both participants then use HKDF to derive the SRTP master encryption keys from the `shared_secret`, the `nonce`, as well as the public `Identity Keys` and user ids of both call participants.

2. Group Calling

Data Encryption

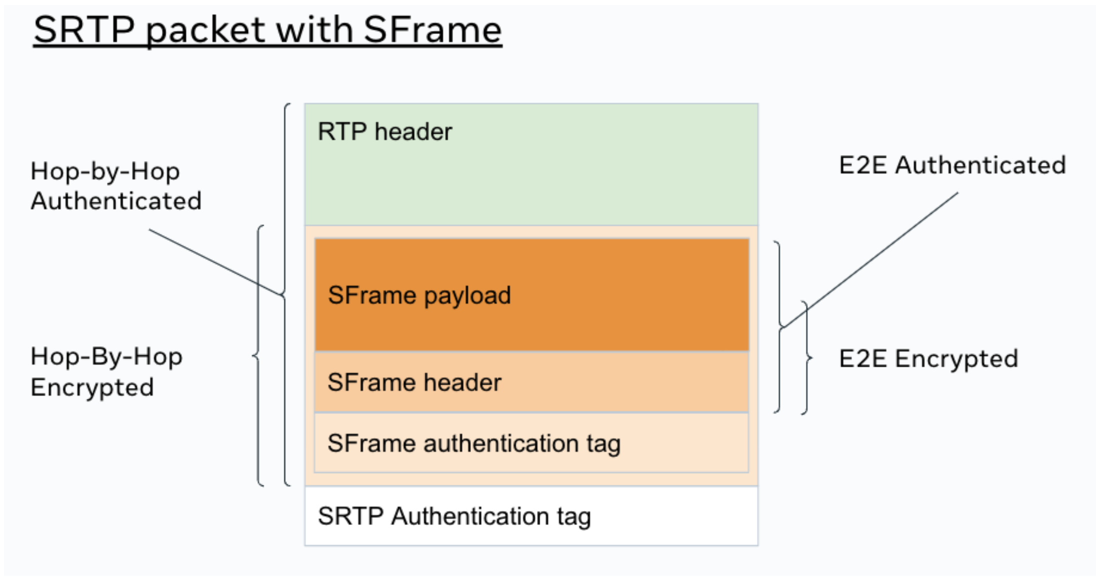
For group calls, audio and video streams are routed through a selective forwarding unit (SFU). Each participant opens a webRTC connection with the SFU server. This means that SRTP encryption keys are negotiated between the user device and the server. Therefore, SRTP encryption is not sufficient to provide end-to-end encryption for group calls.

For group calls, Messenger employs a second layer of encryption to provide end-to-end encryption. The app implements the Secure Frame (SFrame) encryption and authentication mechanism[1], to end-to-end encrypt the content of the audio and video frames while still

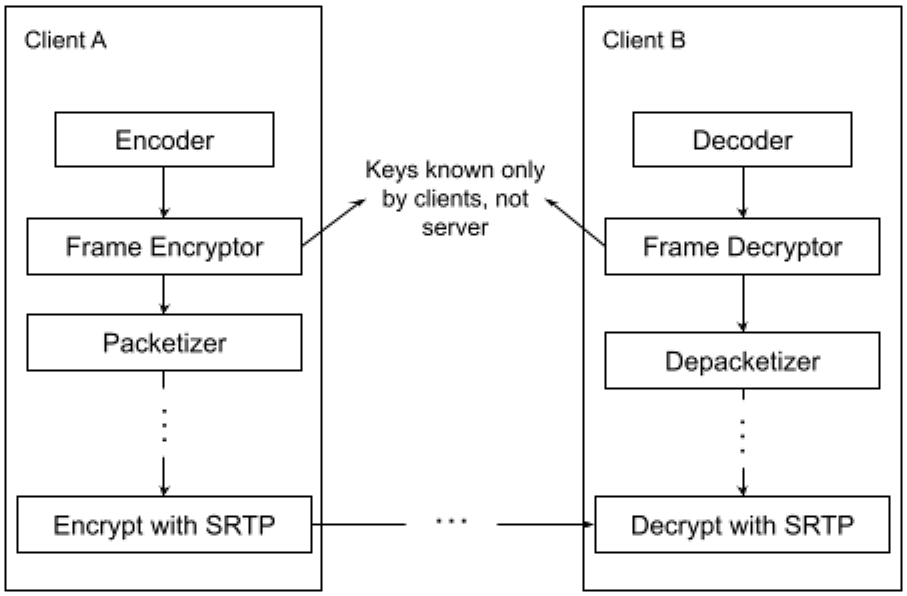
[1] https://datatracker.ietf.org/doc/html/draft-omara-sframe-00

allowing the SFU access to the unencrypted media frame headers. Messenger uses authenticated symmetric encryption with the AES128_GCM64_HKDFSHA256 algorithm for SFrame encryption.

The diagram below presents a high level breakdown of the data blocks and their encryption and authentication properties for an SRTP packet with an SFrame payload:



Encrypting the audio and video frames is achieved by leveraging the webRTC APIs ([SetFrameEncrytor](), [SetFrameDecryptor]()) that allows Messenger to inject frame encryptors and frame decryptors into the frame processing pipeline. The following diagram depicts the client frame processing pipeline with encryptors and decryptors:



Key Management

Messenger uses the ‘sender key’ protocol to derive the keys used to end-to-end encrypt the audio and video stream. Each call participant generates a 128-bit key on call start and configures the frame encryptor attached to the rtp_sender(s) with that key. It then distributes this key to each call participant using the Signal session it established with them. Each participant sends out **n-1** key messages if **n** is the total number of call participants. In turn as it receives key messages from other participants it configures the frame decryptor attached to the rtp_receiver(s) processing the media streams from that participant with the received key.

When a new participant joins the call each existing participant `ratchets` all encryption keys (the key they use to encrypt their local media stream and the keys they received from the other participants) by performing a non-reversible HKDF operation. Each participant then establishes a pairwise Signal session with the new participant, and sends it the newly ratcheted key they are using to encrypt their local media stream. This mechanism provides forward secrecy for the call, ensuring that the new participant will not be able to decrypt any media frames that were sent before it joined the call. .

When a participant leaves the call each remaining participant generates a new `sender key` and distributes it to the other call participants using the pairwise Signal session. This mechanism maintains secrecy for the call ensuring that the former participant will not be able to decrypt any media frames that are sent after they left the call.

Verifying Keys

Messenger users additionally have the option to verify the keys of their devices and the devices of the users with which they are communicating in end-to-end encrypted chats, so that they are able to confirm that an unauthorized third party (or Messenger) has not initiated a middleperson or split-view attack. Each device's sessions are independent from one another, and as a result all devices of the same user must be verified independently. Verification can be done by comparing the list of devices present on each account, and the Identity key associated with them.

Users can also enable alerts on added, removed, or changed keys for themselves and for their chat-partners in settings.

Transport Security

Communication between Messenger clients and Messenger’s backend and chat servers uses Transport Security. Much of this communication uses standards such as TLS and QUIC. Some communication to Messenger’s chat servers is layered within a separate encrypted channel using Noise Pipes with Curve25519, AES-GCM, and SHA-256 from the Noise Protocol Framework for long running interactive connections.

Displaying End-to-End Encryption Status

Across all our services, Messenger makes the end-to-end encryption status of a chat clear. The user can also double check the encryption status within the thread details for a chat thread or the call details for a call. This section also allows the user to verify the device keys of the other participants in a chat. Users may also enable security alerts for their contacts’ key changes from their Privacy & Security Settings.

Furthermore, Messenger uses additional UI elements to improve the security of voice and video calls. The app displays a notification to alert the user when a participant in the call is using a device for the first time when communicating with the user. The user is also able to pull up and review the identity keys for each participant present in the call from the call UI.