Threema.

Cryptography Whitepaper

Threema uses modern cryptography based on open source components that strike an optimal balance between security, performance and message size. In this whitepaper, the algorithms and design decisions behind the cryptography in Threema are explained.

VERSION: FEBRUARY 05, 2024

Contents

Inhalt

Overview **4**

Open Source **5**

End-to-End Encryption **5**
Key Generation and Registration 5
Key Distribution and Trust 6
Message Encryption 7
Group Messaging 8
Key Backup 9

Client-Server Protocol Description **10**
Chat Server Protocol 10
Directory Access Protocol 11
Media Access Protocol 11

Cryptography Details **12**
Key Lengths 12
Random Number Generation 13
Perfect Forward Secrecy 14
Padding 15
Repudiability 15
Replay Prevention 15

Local Data Encryption **16**
iOS 16
Android 16

Key Storage **17**
iOS 17
Android 17

Push Notifications **18**
iOS 18
Android 18

Address Book Synchronization **18**
Linking 19

ID Revocation **20**
An Example 20

Profile Pictures **20**

Web Client / Desktop App **21**
Architecture 21
Connection Buildup 22
WebRTC Signaling 23
WebRTC Connection Buildup 23
Trusted Keys / Stored Sessions 24
Push Service 24
Self Hosting 25

Threema Calls **25**
WebRTC Encryption 25
1:1 Calls 26
Group Calls 27

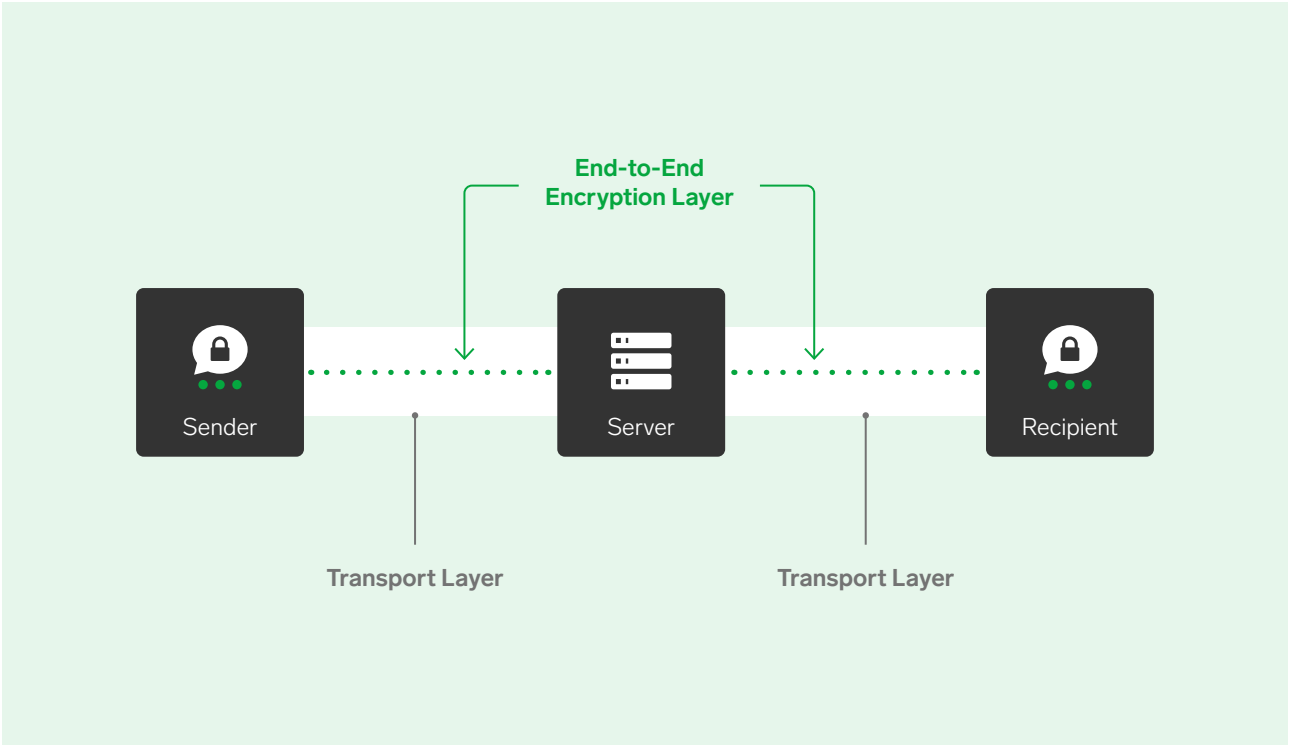
Threema Safe **29**
Overview 29
Backup Format 30
Encryption 30
Upload/Storage 30
Backup Intervals 31
Restore/Decryption 31
Running a Custom Threema Safe Server 31

Overview

Threema uses two different encryption layers to protect messages between the sender and the recipient.

- **End-to-end encryption layer:** this layer is between the sender and the recipient.
- **Transport layer:** each end-to-end encrypted message is encrypted again for transport between the client and the server, in order to protect the header information.

These layers and other important aspects of the cryptography in Threema are described in detail in the following chapters. The crucial part is that the end-to-end encryption layer passes through the server uninterrupted; the server cannot remove the inner encryption layer.



Open Source

The Threema apps for iOS and Android as well as all components of the web client are published under open source licenses:

- iOS app: https://github.com/threema-ch/threema-ios
- Android app: https://github.com/threema-ch/threema-android
- Web client: https://github.com/threema-ch/threema-web
- SaltyRTC server: https://github.com/saltyrtc/saltyrtc-server-python
- Push relay: https://github.com/threema-ch/push-relay

End-to-End Encryption

In Threema, all messages (whether they are simple text messages, or contain media like images, videos or audio recordings) are end-to-end encrypted. For this purpose, each Threema user has a unique asymmetric key pair consisting of a public and a private key based on Elliptic Curve Cryptography. These two keys are mathematically related, but it is not technically feasible to calculate the private key given only the public key.

Key Generation and Registration

When a Threema user sets up the app for the first time, the following process is performed:

1. The app generates a new key pair by choosing a private key at random[1], storing it securely on the device, and calculating the corresponding public key over the Elliptic Curve (Curve25519).
2. The app sends the public key to the server.
3. The server stores the public key and assigns a new random Threema ID, consisting of 8 characters out of A-Z/0-9.
4. The app stores the received Threema ID along with the public and private key in secure storage on the device.

1 including entropy generated by user interaction; see section "Random Number Generation" for details

Key Distribution and Trust

The public key of each user is stored on the directory server, along with its permanently assigned Threema ID. Any user may obtain the public key for a given Threema ID by querying the directory server. The following input values can be used for this query:

- a full 8-character Threema ID
- a hash of a E.164 mobile phone number that is linked with a Threema ID (see section "Address Book Synchronization" for details on the hashing)
- a hash of an email address that is linked with a Threema ID

The **Threema app** maintains a "verification level" indicator for each contact that it has stored. The following levels are possible:

🔴 ⚪ ⚪ **Red (level 1):**The public key has been obtained from the server because a message has been received from this contact for the first time, or the ID has been entered manually. No matching contact was found in the user's address book (by phone number or email), and therefore the user cannot be sure that the person is who they claim to be in their messages.

🟠 🟠 ⚪ **Orange (level 2):** The ID has been matched with a contact in the user's address book (by phone number or email). Since the server verifies phone numbers and email addresses, the user can be reasonably sure that the person is who they claim to be.

🟢 🟢 🟢 **Green (level 3):** The user has personally verified the ID and public key of the contact by scanning their QR code. Assuming the contact's device has not been hijacked, the user can be very sure that messages from this contact were really written by the person that they indicate.

In the **Threema Work app**, the following levels are possible in addition to the ones mentioned above:

🔵 🔵 ⚪ **Blue (level 2):** The ID is a Threema Work internal contact that has not been personally verified by scanning the QR code.

🔵 🔵 🔵 **Blue (level 3):** The ID is a Threema Work internal contact that has been personally verified by scanning the QR code.

To upgrade a contact from the red or orange to the green level (or from the the second blue to the third blue level), the user must scan that contact's public QR code. This QR code is displayed in the "My ID" section of the app, and uses the following format:
`3mid:<identity>,<publicKeyHex>`
where `<identity>` is the 8-character Threema ID, and `<publicKeyHex>` is the hexadecimal (lowercase) representation of the user's 32 byte public key. The app verifies that the scanned public key matches the one that was returned by the directory server.

Message Encryption

Threema uses the "Box" model of the NaCl Networking and Cryptography Library to encrypt and authenticate messages. For the purpose of this description, assume that Alice wants to send a message to Bob. Encryption of the message using NaCl works as follows:

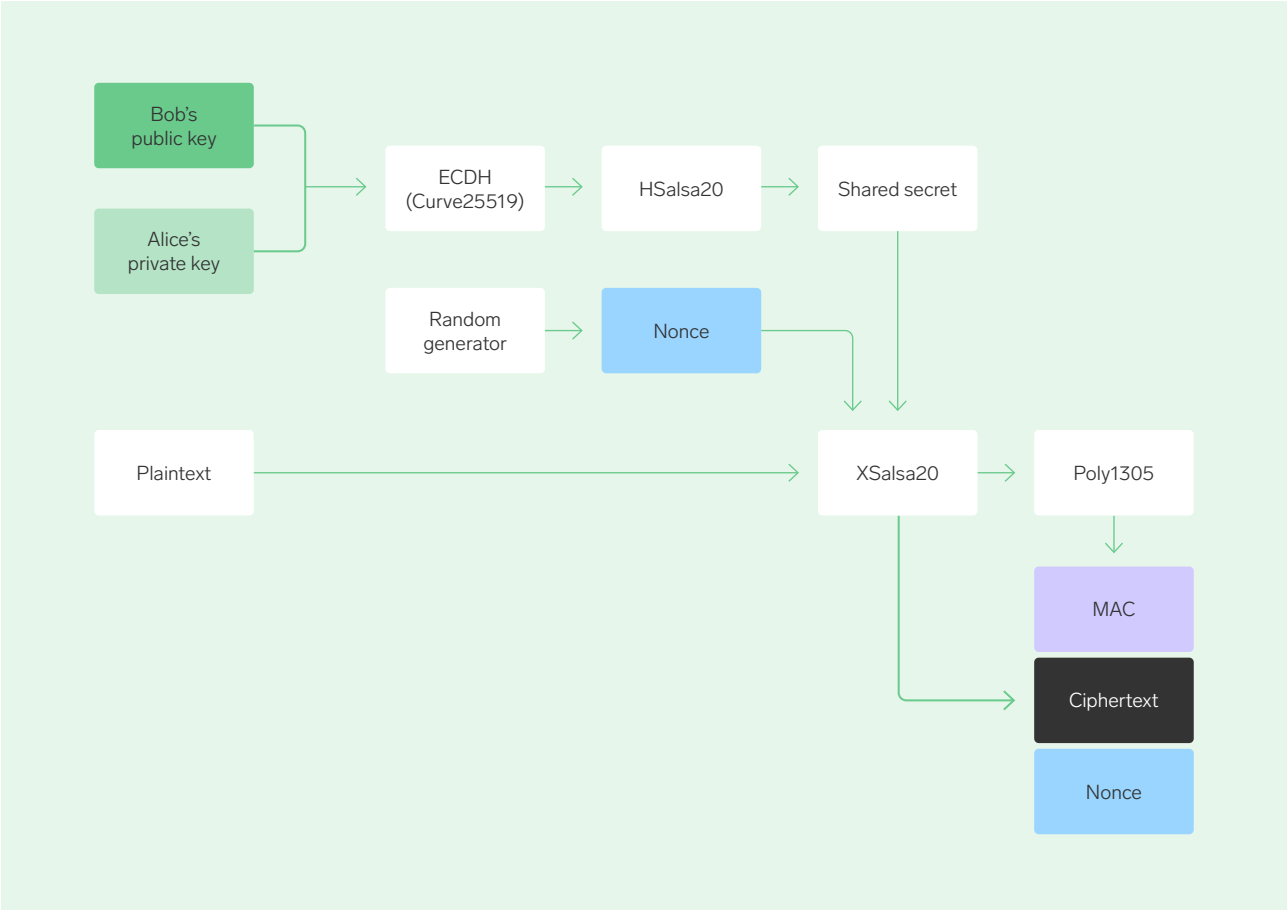
Preconditions

1. Alice and Bob have each generated a key pair consisting of a public and a private key.
2. Alice has obtained the public key from Bob over an authenticated channel.

Procedure to Encrypt a Message

1. Alice uses Elliptic Curve Diffie-Hellman (ECDH) over the curve Curve25519 and hashes the result with HSalsa20 to derive a shared secret from her own private key and Bob's public key.
2. Note that due to the properties of the elliptic curve, this shared secret is the same if the keys are swapped (i.e. if Bob's private key and Alice's public key are used instead).
3. Alice generates a random nonce.
4. Alice uses the XSalsa20 stream cipher with the shared secret as the key and the random nonce to encrypt the plaintext.
5. Alice uses Poly1305 to compute a Message Authentication Code (MAC), and prepends it to the ciphertext. A portion of the key stream from XSalsa20 is used to form the MAC key.
6. Alice sends the MAC, ciphertext and nonce to Bob.

By reversing the above steps and using his own private key and Alice's public key, Bob can decrypt the message and verify its authenticity.



For further details, see Cryptography in NaCl.

For details on the Perfect Forward Secrecy implementation, see the section "Cryptography Details" below.

Group Messaging

In Threema, groups are managed without any involvement of the servers. That is, the servers do not know which groups exist and which users are members of which groups. When a user sends a message to a group, it is individually encrypted and sent to each other group member. This may appear wasteful, but given typical message sizes of 100-300 bytes, the extra traffic is insignificant. Media files (images, video, audio) are encrypted with a random symmetric key and uploaded only once. The same key, along with a reference to the uploaded file, is then distributed to all members of the group.

Key Backup

The user can back up their private key so that they are able to move their Threema ID to another device, or to restore it in case of loss/damage to the device. The app automatically reminds the user to do so, as without a backup, there is no way to recover a lost private key.

To generate a backup, the user must first choose a password (min. 8 characters), which is used to encrypt the private key.

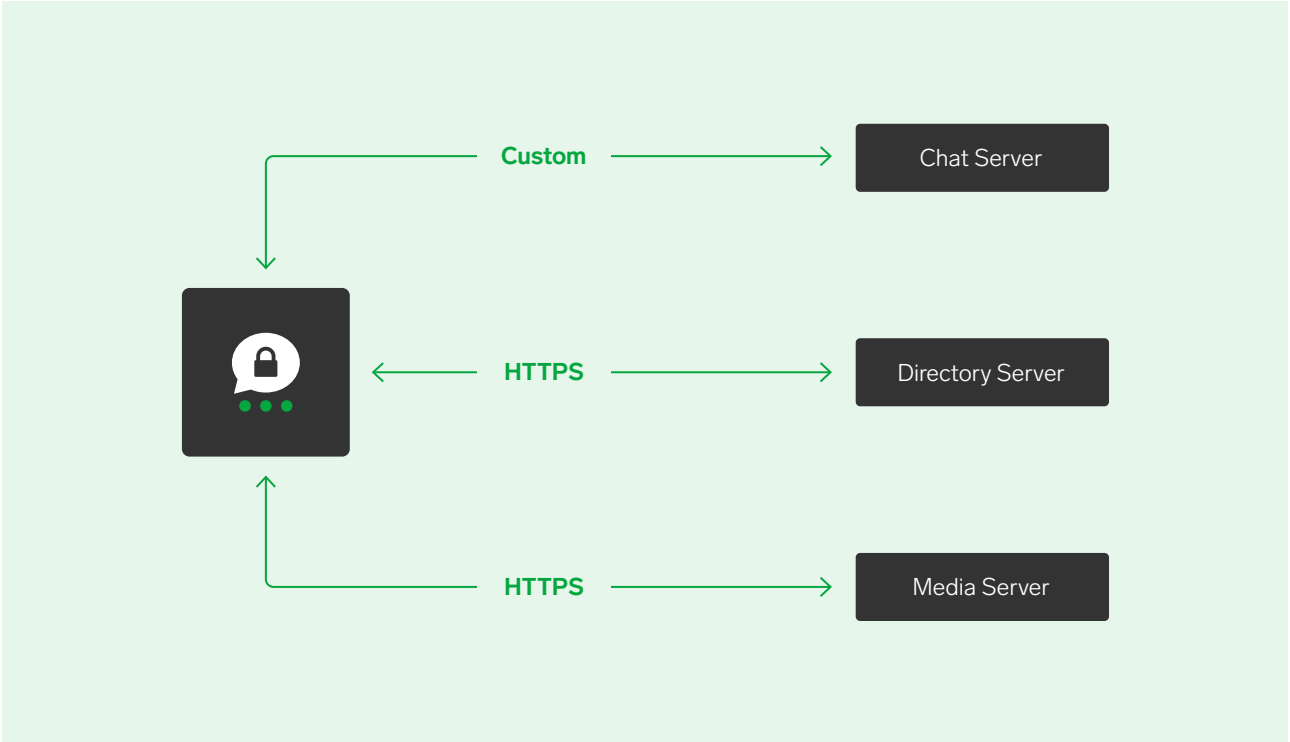
The backup data is generated as follows:

1. Calculate the SHA-256 hash of the following binary string: `<identity><private key>`
   where `<identity>` is the 8 character Threema ID, and `<private key>` is the 32 byte private key.
   Keep only the first two bytes of the resulting hash. It is used during restoration to verify with reasonable confidence that the provided password was correct.
2. Choose a random 64 bit salt.
3. Derive a 256-bit encryption key from the given password and the random salt using PBKDF2 with HMAC-SHA256 and 100000 iterations:
   key_{enc} = PBKDF2(HMAC-SHA256, password, salt, 100000, 32)
4. Use the XSalsa20 stream cipher with key_{enc} and an all-zero nonce to encrypt a binary string of the following format:
   `<identity><private key><hash>`
   where `<hash>` is the two byte truncated hash calculated in step 1. Note that this can be done using the crypto_stream function of NaCl.
5. Prepend the salt to the encrypted string.
6. Base32 encode the result.
7. Split the Base32 encoded string into groups of four characters and separate the groups with dashes. The result will look like this:
   `XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-`
   `XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX`

The encrypted backup data consists of 80 characters (A-Z, 2-7). It is also displayed in QR code form so that the user can more easily transfer it to another device.

Client-Server Protocol Description

Threema communicates with three different types of servers. To access the directory and to download/upload encrypted media files, HTTPS is used. To transport the actual chat messages, a custom protocol built on TCP is used.



Chat Server Protocol

The chat server protocol is used to transport incoming and outgoing messages between the client (app) and the Threema servers. It is a custom binary protocol that uses the NaCl library to secure the connection on the application layer. Its properties are:

- Provides Perfect Forward Secrecy
  - New ephemeral keys are generated for each connection
- Optimized for minimal overhead
- User is authenticated using their public key during connection setup
  - Ensures that a user can only log in if they are in possession of the private key for the Threema ID

Directory Access Protocol

The directory access protocol is used for the following purposes:

- Creating new Threema IDs
- Fetching the public key of another user
- Linking email addresses and mobile phone numbers
- Matching address book contacts (hashes of phone numbers and email addresses) to Threema IDs

Requests are authenticated with a challenge/response protocol based on the user's Threema ID and key pair.

HTTPS (HTTP with TLS) is used as the transport protocol. Strong TLS cipher suites with forward secrecy (ECDHE/DHE) and TLS v1.3 are supported. In order to preclude man-in-the-middle (MITM) attacks even if the system's trusted CA store has been tampered with, or if a trusted CA has illegally issued a certificate for a Threema domain name, the app uses public-key pinning with hardcoded pins to only accept specific, Threema-owned server certificates.

Media Access Protocol

The media servers are used for temporary storage of large media data (e.g. images, videos, audio recordings). Such media is not sent directly via the chat server protocol. Instead, the following procedure is used:

1. The sender encrypts the media file with a random 256-bit symmetric key using XSalsa20 and adds a Poly1305 authenticator.
2. The sender uploads the encrypted media file to a media server via HTTPS.
3. The media server assigns a unique ID for this upload and returns it to the sender.
4. The sender sends an end-to-end encrypted message to the recipient, which contains the media ID and the symmetric key.
5. The recipient receives and decrypts the end-to-end encrypted message, obtaining the media ID and the symmetric key.
6. The recipient uses the media ID to download the encrypted media file from the media server.
7. The recipient decrypts the media file using the symmetric key.
8. The recipient signals the media server to delete the file, unless it was sent in a group conversation, in which case it remains on the media server for the maximum message lifetime (14 days), after which it is deleted automatically in any case.

The media servers use the same TLS configuration (cipher suites with forward secrecy, TLS v1.3) as the directory servers, and the app uses public-key pinning when accessing them.

Cryptography Details

As mentioned earlier, Threema uses the NaCl Networking and Cryptography Library for both the end-to-end encryption, and to secure the chat server protocol at the transport level. This library uses a collection of algorithms to provide a simplified interface for protecting a message using what the authors call "public-key authenticated encryption" against eavesdropping, spoofing and tampering. By default, and as implemented in Threema, it uses the following algorithms:

- Key derivation: Elliptic Curve Diffie-Hellman (ECDH) over the curve Curve25519
- Symmetric encryption: XSalsa20
- Authentication and integrity protection: Poly1305-AES

It is worth mentioning that the ECC curve used by NaCl (and thus by Threema) is **not one of the NIST-recommended curves** that have been suspected of containing deliberately selected weakening constants in their specification. For more details, see Cryptography in NaCl.

Key Lengths

The asymmetric keys used in Threema have a length of 256 bits, and their effective ECC strength is 255 bits.

The shared secrets, which are used as symmetric keys for end-to-end message encryption (derived from the sender's private key and the recipient's public key using ECDH, and combined with a 192 bit nonce), have a length of 256 bits.

The random symmetric keys used for media encryption are also 256 bits long.

The message authentication code (MAC) that is added to each message to detect tampering and forgery has a length of 128 bits.

Discussion of Reasonable Key Lengths

According to NIST Special Publication 800-57 (page 53), the security level of ECC based encryption at 255 bits can be compared to RSA at roughly 2048 to 3072 bits, or a symmetric security level of ~128 bits. The possibility of a successful brute force attack on a key with a 128 bit security level is considered extremely unlikely using current technology and knowledge, according to the judgment of reputable security researchers. A revolutionary breakthrough in mathematics or quantum computing would most possibly render keys breakable anyway, whether they are 128 or 256 bits in length.

- https://cr.yp.to/talks/2005.09.19/slides.pdf
  (Daniel J. Bernstein, author of Curve25519)
- https://www.imperialviolet.org/2014/05/25/strengthmatching.html
  (Adam Langley, Google security researcher)

Random Number Generation

Threema uses random numbers for the following purposes, listed by descending order of randomness quality required:

- Private key generation
- Symmetric encryption of media files
- Nonces
- Backup encryption salt
- Padding amount determination

Nonces and salts must never repeat, but they are not required to be hard to guess.

To obtain random numbers, Threema uses the system-provided random number generator (RNG) intended for cryptographically-secure purposes provided by the device's operating system. The exact implementation varies among operating systems:

Platform	Facility	Implementation
iOS	/dev/urandom	Fortuna
Android	/dev/urandom[2]	Linux PRNG

User-generated Entropy for Private Key Generation

Due to the requirement for very high quality randomness when generating the long-term private key, the user is prompted to generate additional entropy by moving a finger on the screen. The movements (consisting of coordinate values and high-resolution timestamps) are continuously collected and hashed for several seconds. The resulting entropy is then mixed (XOR) with entropy obtained from the system's RNG.

2 To avoid the flawed SecureRandom implementation in some older Android versions, Threema uses its own implementation that directly accesses /dev/urandom.

Perfect Forward Secrecy

Recent versions of the Threema app use the “Ibex” protocol that enables Perfect Forward Secrecy (PFS) for 1:1 messages on the end-to-end encryption layer through mutual agreement of ephemeral keys using ECDH. The app will actively try to negotiate a PFS session with the other party as soon as a message is being sent or received. The Ibex protocol provides provable security, see its security analysis for details.

Key Negotiation

The first party to send a message with PFS enabled starts a PFS session negotiation by sending the public key part of a new ephemeral Curve25519 key pair. The other, responding party then also generates an ephemeral key pair and sends the public key back to the initiator. Since full PFS requires both parties to have completed the key exchange (i.e. contributed a public key), there are two different modes for a PFS session:

- unilateral
  - Can be used immediately (even before the other party has responded to the negotiation).
  - Past messages are protected in the event of a compromise of the sender’s, but not the recipient’s long-term private key.
- complete
  - Can be used after the other party has responded to the negotiation request with their public key.
  - Past messages are protected in the event of a compromise of either party’s long-term private key.

Once a session has been established, it will be used for messages in either direction.

Key Derivation / Ratcheting

Mutual shared secrets are established using ECDH over the ephemeral keys of both parties, and are then used as the initial chain keys in a hash chain ("ratchet"). For each message, a new chain key is derived from the previous chain key using BLAKE2b as a KDF, and the previous chain key is then erased. Separate chain keys for each direction are derived by including the participant's Threema ID in the salt. From the chain key, an actual message encryption key is derived by applying BLAKE2b as a KDF again, making knowledge of a message encryption key useless for calculating future chain or encryption keys.

Both sides maintain a counter of the number of times the ratchet has been turned, and include this in messages to the other party. This allows the other party to detect lost messages, notify the user and adjust their ratchet state accordingly.

Key Storage

Session keys are stored in an SQLite (iOS) / SQLCipher (Android) database, with journaling disabled and PRAGMA secure_delete enabled, ensuring that keys are erased securely.

Error Handling

If a party cannot decrypt an incoming PFS message because it does not have the corresponding key material, it signals this to the other party. The user who sent the message is notified about this, and can decide whether it is safe to resend the message or not. Both participants are also notified if a new session is negotiated, or if a message without PFS has been received unexpectedly.

Threema Calls

Threema Calls perform an interactive codec and media path negotiation. As part of this, ephemeral keys are negotiated between the participants of the call. Thus Threema Calls have always provided PFS on the end-to-end layer, regardless of the per-contact setting that applies to 1:1 messages.

Padding

In order to thwart attempts to guess the content of short messages by looking at the amount of data, Threema adds a random amount of PKCS#7 padding to each message before end-to-end encryption.

Repudiability

In general, cryptographically signed messages also provide non-repudiation; i.e. the sender cannot deny having sent the message after it has been received. The NaCl library's box model uses so-called public-key authenticators instead, which guarantee repudiability (see https://nacl.cr.yp.to/box.html, "Security model"). Any recipient can forge a message that will look just like it was actually generated by the purported sender, so the recipient cannot convince a third party that the message was really generated by the sender and not forged by the recipient. However, the recipient is still protected against forgeries by third parties. The reason is that in order to perform such a forgery, the private key of the recipient is needed. Since the recipient will know whether or not they have used their own private key for such a forgery, they can be sure that no third party could have forged the message.

Replay Prevention

The Threema app remembers the nonce of every message that has been sent in the past, and rejects messages with duplicate nonces. Since the server cannot successfully modify

the nonce of a message without knowing the private key of one of the parties involved in the communication, this prevents a malicious server from replaying/duplicating previously sent messages. This is only relevant for messages that don't use PFS, as PFS already prevents such replays.

Local Data Encryption

The Threema app stores local data (such as the history of incoming and outgoing messages, and the contact list) in encrypted form on the device. The way in which this data is encrypted varies among platforms.

iOS

On iOS, Threema stores local data in a Core Data database, which is backed by files in the app's private data directory. The iOS sandbox model ensures that no other apps can access this data directory. All files stored by Threema are protected using the iOS Data Protection feature using the Data Protection class `NSFileProtectionCompleteUntilFirstUserAuthentication`. The encryption key used to protect the files is derived from the device's UID key and the user's passcode. A longer passcode will provide better security (on devices equipped with a Touch/Face ID sensor, longer passcodes can be used without inconvenience as the passcode normally only needs to be entered after a reboot).

Note that the private key for the Threema ID is stored separately in the iOS Keychain.

Android

On Android, Threema stores local data in an SQLite database within the app's private data directory. Android ensures that no other apps can access this data directory (as long as the device is not rooted). The database itself is protected by SQLCipher with AES-256, and any media files (which are stored separately in the app's private directory within the file system) are also encrypted using AES. The key is randomly generated when the database is first created, and can be protected with a user-supplied passphrase (which is of course necessary if the user wishes to take advantage of this encryption). The passphrase is never written to permanent storage and therefore needs to be entered whenever the app process restarts (e.g. after a low-memory situation, or after the device has rebooted). Alternatively (or additionally), the user may enable full-device encryption if supported by the device and Android version.

When a passphrase is used, the master key (32 bytes) is protected before being stored in the following way:

```
protected-master-key =
  master-key XOR
  scrypt(P = <passphrase>, S = <random-stored-on-device>, N =
65536, r = 8, p = 1, dkLen = 32)
```

The scrypt parameters have been chosen to provide a reasonable trade-off between calculation time, memory usage and security without precluding the use on older/lower-end devices.

Key Storage

On the user's mobile device, the private key is stored in such a way as to prevent access by other apps on the same device, or by unauthorized users. The procedure differs among platforms.

iOS

- The private key is stored in the iOS Keychain with the attribute `kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly`.
- Only the Threema app can access this keychain entry.
- While the entire iOS keychain is included in backups via iTunes or iCloud, the keychain attribute causes the keychain entry of Threema to be encrypted with a device-specific key ("UID key") that cannot be read directly and is not known to Apple. Therefore, the keychain entry is only usable if the iTunes/iCloud backup is restored to the same device.
- When the Threema app is deleted and reinstalled, the keychain entry persists and the Threema ID is not lost.

Android

- The private key is stored in a file in the app's private data directory.
- Other apps cannot access this file.
- AES-256-CBC encryption is applied to the private key before it's written to the file. The key for this encryption is stored separately and can be protected using a passphrase (see section "Local Data Encryption" for details).

Push Notifications

Threema uses the standard push notification service provided by the operating system. This is necessary to ensure reliable delivery of push notifications, as mobile operating systems place high importance on conserving battery power, and thus severely restrict background operation of apps (such as would be needed to maintain a separate, app-specific persistent connection for push notifications). Channelizing push notifications through the operating system's standard mechanism thus helps conserve battery power, and on iOS, it is the only means possible and accepted in the App Store.

iOS

The Threema-specific APNS payload consists of a JSON/Base64 encoded NaCl "crypto_secretbox" ciphertext and corresponding nonce. The symmetric key used by the Threema chat server to encrypt this payload is chosen at random by the client, and securely transmitted to the server beforehand using the chat server protocol described above. The symmetric key is not known to Apple. When a push notification arrives, the Threema app decrypts the APNS payload using the symmetric key, and thus recovers the metadata (sender, message ID) of the corresponding message. With this information, the app can obtain the actual encrypted message by connecting to the chat server, and then decrypt the message and display a preview (if enabled) and the sender's contact name in the local notification.

Android

The FCM payload is empty; that is, the FCM message only serves to wake up the Threema app in the background. It then connects to the chat server to retrieve any pending messages, decrypts them, and shows local notifications for them.

Address Book Synchronization

Threema optionally lets the user discover other Threema contacts by synchronizing with the phone's address book. If the user chooses to do so, the following information is uploaded through a TLS connection to the directory server:

- HMAC-SHA256 hash of each email address found in the phone's address book
  Key: `0x30a5500fed9701fa6defdb610841900febb8e430881f7a`
  `d816826264ec09bad7`

- HMAC-SHA256 hash of the E.164 normalized form of each phone number found in the phone's address book
  Key: `0x85adf8226953f3d96cfd5d09bf29555eb955fcd8aa5ec4f9f cd869e258370723`

The directory server then compares the list of hashes from the user with the known email/phone hashes of Threema IDs that have been linked with an email address and/or phone number. Any matches are returned by the server as a tuple (Threema ID, hash). Only those hashes that have already been submitted by the user are returned (i.e. if the user submits an email hash which then matches a Threema ID, the server will only return the email hash of that ID and not the linked phone number's hash, even if one exists). After returning the matches to the client, the directory server discards the submitted hashes.

Use of HMAC Keys
The reason why HMAC-SHA256 is used instead of plain SHA256 is not for obfuscation, but as a best practice to ensure that hashes generated by Threema are unique and do not match those of any other application. Obviously the keys need to be the same for all users, as random salting (such as is used when hashing passwords for storage) cannot be used here because the hashes of all users must agree so that matching contacts can be found.

A Word About Hashing Phone Numbers
Due to the relatively low number of possible phone number combinations, it is theoretically possible to crack hashes of phone numbers by trying all possibilities. This is due to the nature of hashes and phone numbers and cannot be solved differently in a reasonably efficient way (using salts like for hashing passwords does not work for this kind of data matching). Therefore, the servers treat phone number hashes with the same care as if they were raw/unhashed phone numbers. Specifically, they never store hashes uploaded during synchronization in persistent storage, but instead discard them as soon as the list of matching IDs has been returned to the client.

Linking

If a user chooses to link their Threema ID to an email address or a phone number, the directory server verifies that the user actually owns the email address or phone number in question.

- For email addresses, a verification email with a hyperlink is sent to the user. The user must open the hyperlink and confirm in the browser before the ID link is established.
- For phone numbers, the directory server sends an SMS message with a random 8 digit code. The user must enter that code in the app, which sends it back to the server to verify the phone number.
  - If the user cannot receive the SMS message, they may choose to receive an automated phone call in which the code is read out.

ID Revocation

Threema users may revoke their ID at any time by going to a website (https://myid.threema.ch/revoke) and entering their Threema ID and a revocation password that they have set beforehand. Revoked IDs can no longer log in, and they will be shown in strikethrough text on (or, depending on the users' settings, disappear from) the contact lists of other users within 24 hours. Messages cannot be sent to revoked IDs.

Revocation passwords are hashed with SHA256 on the client side, but only the first four bytes are sent to and stored/compared on the directory server. This mitigates the possibility of brute-force hash cracking on the server side to recover the user's password. Obviously, the downside is that the stored key has significantly less entropy than a reasonable user password and a large number of possible passwords correspond to each key. However, the server can easily limit the number of key revocation attempts over a period of time.

An Example

A user has set an eight character revocation password with a combination of characters consisting of the lowercase latin alphabet (a-z) and digits. This is a pretty weak password by modern standards, but already results in $36^8 \cong 2^{41.36}$ possible combinations.

Therefore, a given key can correspond with over 600 different passwords of this length and character set. On the other hand, the probability of randomly choosing the correct hash value without knowing anything about the password is $2^{-32} \cong 0.000000023\%$. Assuming that a user gets three attempts per day, the probability of finding the correct hash value within 10 years is $\cong 0.000255\%$.

Profile Pictures

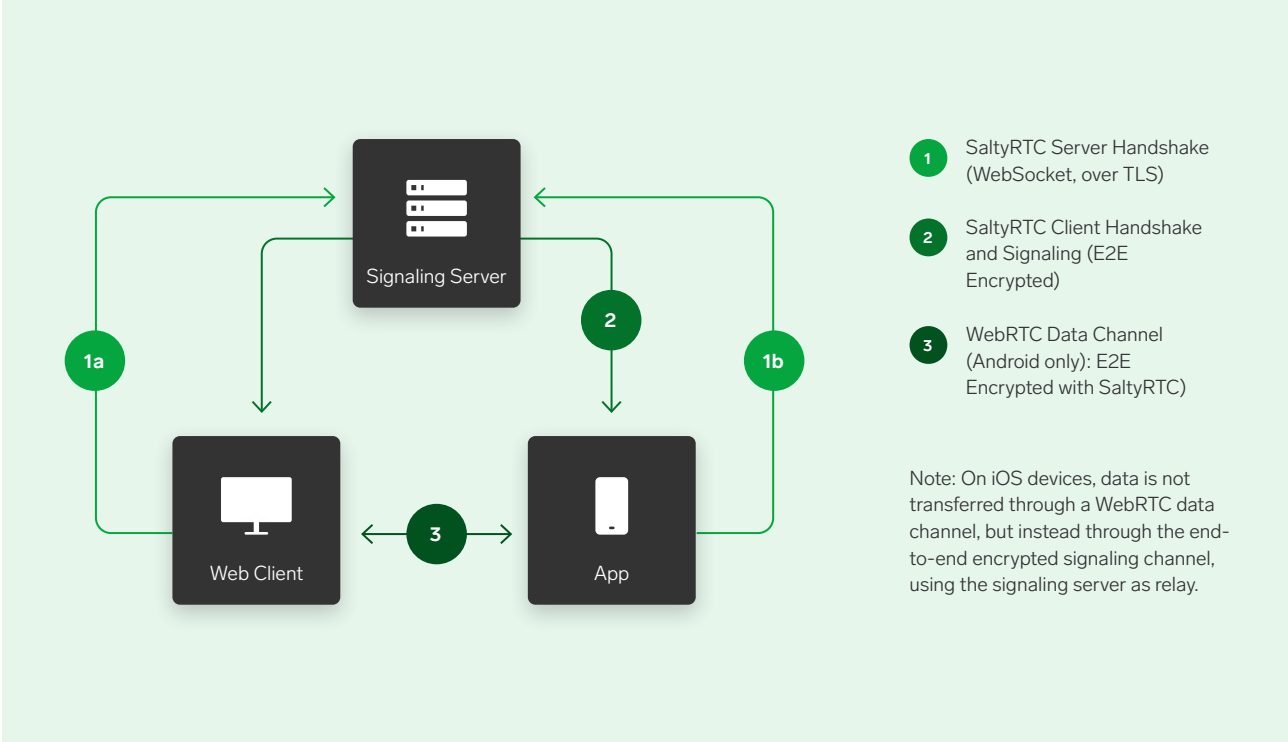
It is possible for a user to select a photo to be used as his/her "profile picture". The user can choose within the app whether they want to share the profile picture with all their contacts, with only those contacts selected from a list, or with nobody. If they chose to share the profile picture and send a message to one of the designated recipients, the newly set profile picture is encrypted with a random symmetric key within the app, and then uploaded to the media server like a regular image message (see Media Access Protocol). A reference to the encrypted picture on the media server, along with the symmetric key, is subsequently sent as an end-to-end encrypted message in the background whenever the user sends a regular message to one of the designated contacts who has not received the current profile picture yet. As the media server deletes files after two weeks, the profile picture is uploaded again once a week (if necessary), using a new symmetric key each time.

Web Client / Desktop App

To use Threema from a desktop computer, a web client and a desktop app are available. The desktop app is simply a bundle of the Electron framework and the web client's source code. One slight advantage of the desktop app over the web client is the fact that the code isn't loaded from a server each session but permanently stored on the user's end, which means that it would be harder to alter it without the user noticing it. Other than that, the desktop app and the web client are essentialy the same. In what follows, only the web client is mentioned for brevity's sake (but everything a fortiori holds true for the desktop app).
The Threema web client communicates directly with the Threema app over WebRTC data channels (Android) and WebSockets (iOS). In addition to the standard WebRTC channel encryption (Android) and WebSocket TLS transport encryption (iOS), all packets are end-to-end encrypted using NaCl. The WebRTC signaling is also implemented with end-to-end encryption using the SaltyRTC protocol.

The following paragraphs explain the general architecture of the Threema web client implementation as well as some aspects of the signaling protocol. If you want to learn more about the WebRTC signaling protocol, please refer to the main specification as well as the WebRTC task specification.

Architecture



The web client and the app form a client-server relationship. All data about new events and old conversations is exchanged directly between the two through an end-to-end encrypted channel.

When the web client is started, it requests the initial data (conversations, contacts, avatars, etc.) from the app. The app responds with the requested data.
To send a message to a contact, the web client dispatches a message request to the app, which will then send the message to the recipient. When a new message arrives, the app notifies the web client. No messages are exchanged directly between the web client and the Threema chat server. The user's private key never leaves the device.

Connection Buildup

The full connection buildup happens in three stages.

1. SaltyRTC Server Handshake
First, the web client generates a new permanent key-pair and a random authentication token. It then connects to the SaltyRTC signaling server as initiator. The hex-encoded public permanent key is used as the WebSocket path.

The public permanent key of the web client as well as the authentication token and the protocol version are transferred to the app through a QR code. The app first generates its own permanent key-pair and then connects to the same WebSocket path as responder.

Both peers conduct the server handshake according to the SaltyRTC protocol.

2. SaltyRTC Client Handshake and Signaling
As soon as both peers have successfully finished the server handshake, they can start exchanging client handshake messages. Because the server has established a trusted connection with both the initiator and the responder, it can relay encrypted messages without knowing the content. This is how the session keys and the signaling data are transferred.

The two peers each generate a random session key-pair independent of the permanent key-pair. The public session keys are exchanged. Additionally, the peers exchange information on how they intend to communicate after the client handshake (the so-called SaltyRTC Task):

- Android devices request to communicate over WebRTC data channels
- iOS devices request to communicate over the signaling channel

The transfer of data between browser and app differs between Android and iOS, due to platform constraints:

2.1 Android
After the client handshake is successful and both peers know each other's public session key, they initiate a WebRTC PeerConnection. The necessary signaling information to build up the connection (like offer, answer and ICE candidate messages) are exchanged via the WebSocket connection, encrypted with the session keys.

As soon as the WebRTC PeerConnection is established, the two peers initiate the handover of the signaling channel. Future signaling messages (like additional ICE candidates when the network configuration changes) are exchanged over a secure signaling data channel. Once the handover is complete, the WebSocket connection to the server is closed.

The two peers now open a second WebRTC data channel in order to exchange application data. All packets transferred through that data channel are encrypted using the session keys.

2.2 iOS
After the client handshake is successful and both peers know each other's public session key, they can immediately start exchanging application data over the established WebSocket signaling connection. All packets transferred through that channel are encrypted using the session keys.

3. Data Channel (Android Only)
All required web client data (conversations, contacts, messages, etc.) can now be freely exchanged through the encrypted WebRTC data channel.

WebRTC Signaling

The direct communication channel between the app and the Android web client is established using a WebRTC PeerConnection. In order to establish such a peer-to-peer connection, a signaling channel is inevitable. Common signaling server implementations often use WebSockets without any end-to-end encryption (often even without transport encryption), meaning that the server can read all (potentially sensitive) network information of the peers connecting. There is also the risk of a server manipulating the data being transmitted, opening up possibilities of MITM attacks.

In order to mitigate this risk as well as minimizing metadata exposure in general, we participated in the design and implementation of the SaltyRTC protocol, which offers end-to-end encryption of signaling data and does not require the clients to trust the server at all.

WebRTC Connection Buildup

Because networks in the real world don't always make it possible to establish a direct connection between two peers due to obstacles like firewalls, NATs, CGNs and the like, WebRTC connection buildup may have to resort to mechanisms like STUN and TURN. STUN

(Session Traversal Utilities for NAT) is a protocol that allows to find each other's public IP despite the existence of NATs. TURN (Traversal Using Relays around NAT) is a protocol that provides relaying of data packets in case the connection buildup using STUN alone fails. Note, however, that even though a TURN server relays packets between two peers, it cannot know anything about the content of these packets as they are both end-to-end encrypted by WebRTC using DTLS and end-to-end encrypted by SaltyRTC.

Threema runs its own STUN and TURN servers.

Trusted Keys / Stored Sessions

In order to prevent having to scan the QR code each time a connection needs to be established, the public permanent key of the peer can optionally be stored as a trusted key.

To be able to securely store this information in the browser, users must provide a password if they want to be able to restore the session at a later point in time. The public permanent key of the app as well as the private permanent key of the web client are then encrypted with the provided user password using authenticated NaCl secret key encryption (XSalsa20 + Poly1305) and stored in local browser storage.

On the app side, the public permanent key of the web client and the private permanent key of the app are stored in the encrypted app database.

When reconnecting to an existing session, instead of creating a new permanent key-pair, the peers restore the trusted key-pair before initiating the server and client handshake. Note, however, that new session keys are still created, thus offering perfect forward secrecy on a session level.

Push Service

On Android devices, if Google Play Services are installed on the device of the user, the FCM push token (an opaque string provided by Google) is transferred to the web client together with the initial data.

On iOS devices, the APNs push token (an opaque string provided by Apple) is transferred to the web client together with the initial data.

If the user requests to persist a session, the token is then stored in the local browser storage together with the trusted keys, encrypted with a key derived from the user-provided password.

When reconnecting a web client session (e.g. when the user requests to restore a previous session, or when automatically reconnecting due to connection loss) and if a push token is available, the browser sends this token via HTTPS to a push relay server provided by Threema,

together with the SHA256 hash of the public permanent key of the initiator. The push relay server then sends the hash to the app as a FCM/APNs push notification. When the app receives such a notification, it first checks whether the web client is enabled. If it is enabled and if a web client session with the specified public key exists, that session is started and connects to the SaltyRTC server for the handshake and reconnection procedure.

Self Hosting

All components of the web client are published under open source licenses (see the beginning of this document for repository links).

The URL and public permanent key of the SaltyRTC server to be used are transferred through the QR code and stored together with the session information. This allows the user to entirely bypass the web server and SaltyRTC server operated by Threema, if desired. Instructions on how to set up a self-hosted Threema Web instance can be found in the Threema Web repository.

The push relay cannot practically be run by an end user, since the FCM API key and the APNs certificates are not public, but the source code is provided on GitHub for review purposes.

Threema Calls

Threema Calls are based on WebRTC, an open IETF Standard. WebRTC uses the ICE, STUN and TURN protocols to establish a secure peer-to-peer connection.

WebRTC Encryption

Media streams are encrypted with the SRTP protocol, with DTLS-SRTP being used for the key exchange. DTLS version 1.2 is enforced.

The DTLS ciphersuites offered by the Threema app are (in that order):

```
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9)
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
```

The SRTP ciphersuites offered by the Threema app are (in that order):

```
SRTP_AEAD_AES_256_GCM (0x0008)
SRTP_AEAD_AES_128_GCM (0x0007)
SRTP_AES128_CM_HMAC_SHA1_80 (0x0001)
```

1:1 Calls

Threema Calls started in a 1:1 chat establish a 1:1 call directly between the two parties. Besides potential relaying, there is no server in between.

Signaling

Signaling data (WebRTC offer, answer and candidates SDP) is sent through Threema messages, thus providing the same end-to-end security and trust level as a regular Threema conversation. If you have verified your contact, you can also trust your Threema Calls with that contact.

Encryption

The certificates used for the DTLS session are cryptographically linked to the keys used for Threema's end-to-end encryption by means of including the certificate fingerprints in signaling messages

RTP header extensions are only used if offered in an encrypted form.

Encoding

Audio data is encoded using the Opus codec[3] at 48 kHz. To avoid potential privacy and security issues associated with variable bitrate audio codecs4, 1:1 Calls always enforce constant bitrate to encode the audio stream.
Video data is encoded by VP9, VP8 or H.264.

Privacy / IP exposure

WebRTC connections can be established in two ways: Either a direct connection between the two devices is used (using STUN to help with NAT traversal), or one or both sides use a TURN server to relay the encrypted packets.

Relaying has the advantage of hiding the public IP address from the call partner, thus providing added privacy when the peer is not trusted. On the other hand it exposes some metadata towards the TURN server, and the connection quality (especially latency) may be worse than in a direct call. By default, Threema enforces relaying through TURN for all unverified contacts (with a single red dot as verification status). Optionally, users can enforce relaying for all calls through the "Always relay calls" option in the "Privacy" settings.

Group Calls

Threema Calls started in a group chat establish a group call. Group Calls always flow via a server, the Selective Forwarding Unit (SFU), to selectively forward media streams to participants and coordinate participants joining and leaving. The SFU also relays arbitrary messages between participants. The SFU cannot inspect any message between participants, not even during the handshake phase.

Creating/Joining

When creating a call, the client generates a random Group Call Key (GCK). When joining a call, the client will receive a GCK from another member of the group as part of a Threema message informing all members that a Group Call has been started. The following properties are then derived from GCK using BLAKE2b:

- Group Call Handshake Key (GCHK)
- Group Call State Key (GCSK)
- Group Call ID which is cryptographically tied to the group and the call by including the following properties in its derivation:
  - Group Creator ID
  - Group ID
  - GCK

The flow to create/join a call consists of making an HTTPS `join` request to the SFU with the Group Call ID. This is the rendezvous path for all group members wanting to participate in the call.
The `join` request/response flow via HTTPS also exchanges the certificate fingerprints of the certificates used in the DTLS session that is being established by WebRTC between client and SFU.

Participant Authentication and Encryption

Once a client joined a call and established a WebRTC connection to the SFU, it will generate a random Participant Call Media Key (PCMK) for media transmission. Furthermore, a handshake with each other participant will be fulfilled to establish an ephemeral key between each participant pair in the following way:

1. Send a `hello` message to the other participant, encrypted by GCHK. This message includes the client's Threema ID, nickname, the public part of a random Participant Call Key (PCK) and a random challenge.
2. The remote participant will respond with a `hello` message with the same properties.
3. The remote participant will send an `auth` message. This message is first encrypted by a key derived via BLAKE2b from the long-term shared secret between the two Threema

IDs, and then encrypted by the shared secret derived from the two participant's PCK. This message includes the challenge response and the current PCMK iteration.

4. The client will also respond with an `auth` message with the same properties.

Note that this handshake and any further communication between participants is relayed by the SFU and not via Threema messages.

Perfect Forward Secrecy of Media

When a new participant joins, all existing participants of the call will advance the ratchet of PCMK by applying a BLAKE2b key derivation. The resulting PCMK will then be distributed to the new participant.

When a participant leaves, all other participants of the call will redistribute a new random PCMK to the remaining participants. The new PCMK will be applied after a brief delay to ensure all participants have received the PCMK before applying it to encrypt media frames.

Media Encryption

All media frames are (additionally to the SRTP encryption towards the SFU) end-to-end encrypted with AES-256-GCM by the current iteration of the Participant Call Media Frame Key (PCMFK) derived from PCMK via BLAKE2b. A counter is being used as IV/nonce.

In case of a VP8 video frame, the first 3 (not a key frame) or 10 (key frame) bytes are not additionally end-to-end encrypted but are authenticated by the additional data section of AES-256-GCM. This is necessary because the SFU needs to be able to parse the VP8 header of the frame.

RTP header extensions are not encrypted for now and the set of header extensions is therefore reduced to not include sensitive metadata.

Encoding

Audio data is encoded using the Opus codec[3] at 48 kHz. Opus Discontinued Transmission (DTX) mode is being used to minimise the amount of data transmitted between SFU and participants.

Video data is encoded by VP8.

Threema Safe

Overview

Threema Safe is an encrypted server-based backup solution that encompasses the following user-specific data:

- Threema ID
- Private key
- Profile information
  - Nickname
  - Profile photo
  - Linked email address and phone number
- Contact list
  - Contact Threema ID
  - Public key
  - Name
  - Verification level
- Group definitions
  - Group ID
  - Creator
  - Group name
  - List of members
- Distribution lists
  - Name
  - List of members
- App settings

Messages and media data are not part of a Threema Safe backup.
The rationale behind Threema Safe is to give users a secure backup option that allows them to restore their Threema ID and important related data, or to move it to another device, using nothing but the knowledge of their Threema ID and the Threema Safe password that they have chosen.

The use of Threema Safe is optional; the user can choose to enable or disable it at any time. By default, encrypted backups are stored on Threema servers, but the user can use their own server as a backup store instead. A Threema Safe backup server cannot tell which backup belongs to which Threema user by looking at the uploaded data.

Backup Format

The Threema app compiles the information mentioned in the previous section as a JSON document (binary values like the private key or profile photo are encoded in Base64). This JSON document is then encrypted as follows. The resulting encrypted backup typically has a size of only a few dozen kilobytes (depending mostly on the profile photo size, number of contacts and groups).

Encryption

In order to set up Threema Safe, the user must choose a password consisting of at least 8 characters. The app checks whether the password is on a list of the most frequently used passwords, and warns the user if that is the case.

From the password and the Threema ID of the user, a `Threema Safe Master Key` is derived as follows:

- `threemaSafeMasterKey = scrypt(P = Password, S = Threema ID, N = 65536, r = 8, p = 1, dkLen = 64)`

The scrypt parameters have been chosen to provide a reasonable trade-off between calculation time, memory usage and security without precluding the use on older/lower-end devices.

The scrypt output is then split into two parts of 32 bytes each as follows:

- `threemaSafeBackupId = threemaSafeMasterKey[0..31]`
- `threemaSafeEncryptionKey = threemaSafeMasterKey[32..63]`

The Threema Safe backup data in the form of a UTF-8 encoded JSON string `(plaintext)` is then encrypted as follows:

1. Generate random 24 byte nonce: `nonce = randombytes(24)`
2. `threemaSafeEncryptedBackup = nonce || crypto_secretbox(plaintext, threemaSafeEncryptionKey, nonce)`
   `|| = concatenation`
   The crypto_secretbox operation is described in the NaCl documentation.

Upload/Storage

The encrypted data (`threemaSafeEncryptedBackup`) is uploaded to, or downloaded from, the Threema Safe server using simple WebDAV-compatible HTTPS requests (see below for details). The `threemaSafeBackupId` in lowercase hex encoding is used as a filename. If the user disables Threema Safe, a DELETE request is issued by the app.

Backup Intervals

When Threema Safe is enabled and a backup password is set, the Threema app generates a backup once per day when the app is in use. If the backup data has changed since the last upload, or if more than half of the server-specified backup retention time has passed, then the backup is uploaded to the server, replacing any existing backup with the same ID that may already exist on the server.

Restore/Decryption

1. User enters their Threema ID and password in the setup screen.
2. Derive the master key as described above.
3. Fetch `threemaSafeEncryptedBackup` from the server using the `threemaSafeBackupId`.
4. `nonce = threemaSafeEncryptedBackup[0..23]`
5. `plaintext = crypto_secretbox_open(threemaSafeEncryptedBackup[24..], nonce, threemaSafeEncryptionKey)`

The crypto_secretbox operation is described in the NaCl documentation.

The decrypted JSON document (`plaintext`) is then used to restore the Threema ID and other information.

Running a Custom Threema Safe Server

When setting up Threema Safe, a custom backup server URL may be specified by the user. This URL must start with https:// and may include username/password information (e.g. https://username:password@my.server.example/path). The server must use a TLS certificate trusted by the operating system.

The server must respond to the following requests (paths relative to the backup URL):

Request Server Configuration

```
GET config
Accept: application/json
```

Response Codes

Code	Description
200 OK	Response body contains the configuration, see example below.
400 Bad Request	Validation failed.

Example Response

```
{
    "maxBackupBytes": 524288,
    "retentionDays": 180
}
```

The app uses `maxBackupBytes` to determine whether a backup is too big to be uploaded (in which case the user is alerted). `retentionDays` is used to determine backup expiration: if `retentionDays`/2 days have elapsed since the last backup, the backup is uploaded again even if the backup data has not changed, to keep the backup from expiring.

Create/Update Backup

```
PUT backups/<threemaSafeBackupId>
Content-Type: application/octet-stream
```

The `threemaSafeEncryptedBackup` is sent along unmodified in the PUT body.

Validation

- The `threemaSafeBackupId` **must** be a 64 character lowercase hex string
- The `Content-Type` header **must** be set to `application/octet-stream`

Response Codes

Code	Description
200 OK	The backup was created or updated (for servers that do not differentiate between the two).
201 Created	The backup was created.
204 No Content	The backup was updated.
400 Bad Request	Validation failed.
413 Payload Too Large	Message body is too large.
429 Too Many Request	Rate limit reached.

Fetch Backup

```
GET backups/<threemaSafeBackupId>
Accept: application/octet-stream
```

Validation

- The `threemaSafeBackupId` **must** be a 64 character lowercase hex string
- The `Accept` header **must** be set to `application/octet-stream`

Response Codes

Code	Description
200 OK	Response body contains the `threemaSafeEncryptedBackup`.
400 Bad Request	Validation failed.
404 Not Found	No backup was found at the specified URL.
429 Too Many Requests	Rate limit reached.

Delete Backup

`DELETE backups/<threemaSafeBackupId>`

Validation

- The `threemaSafeBackupId` **must** be a 64 character lowercase hex string

Response Codes

Code	Description
200 OK	The backup was deleted.
204 No Content	The backup was deleted.
400 Bad Request	Validation failed.
404 Not Found	No backup was found at the specified URL.
429 Too Many Requests	Rate limit reached.

Security Recommendations

The following recommendations are not required for automated backups to work, but they improve security. Implementing these recommendations is especially important when offering a public service.

Throttling

To avoid misuse, throttling/rate limiting **should** be enforced in the server. If the limit has been reached, requests **must** be responded to with a HTTP 429 (Too Many Requests) status code.

If the server software itself does not support rate limiting, this can be implemented by using a Proxy like Nginx or HAProxy.

Data Retention

Backups **should** be automatically deleted after a certain amount of time, as specified in the `retentionDays` configuration key.