No Prying – Secure Messaging Protocol
Security Level 2 – Design Specification

Overview

This protocol provides end-to-end encrypted messaging between two parties using exclusively symmetric cryptographic primitives. There is no asymmetric key exchange, no public key infrastructure (other than the inherited HTTPS transport layer if using public servers), and no mathematically complex key establishment protocol. All shared secrets are derived from a single out-of-band voice exchange and never transmitted digitally.

The design is intentionally minimal. Every component removed reduces implementation surface, reduces audit complexity, and eliminates a category of potential failure. What remains is a small set of well-understood, well-audited primitives — HKDF and AEAD-AES-256-GCM-SIV — applied in a straightforward construction.

In addition, the system does not require user accounts; instead, it relies upon an initial setup and ever-revolving mailbox addresses into which encrypted messages are delivered. The protocol incorporates post-attack self-healing (see Section 3 for details)

Contents

1. Initial Key Bootstrap (Out-of-Band)

1.1 Word List Generation

Consider two people who want to communicate securely: let's call them Alice and Bob. Each independently generates a cryptographically secure 128-bit random value within the application. Each value is encoded as 12 words using the BIP-39 word list, giving each party a human-readable representation of their entropy contribution. The concatenation is protocol-specific and must not be treated as a standard 24-word BIP-39 mnemonic.

Each party communicates their 12 words to the other, through an out-of-band channel, such as in order of preference:

in-person meeting,
split the words into sections and deliver separately through different medium,
tamper-evident physical delivery,
authenticated voice call

It is important that the communication is not intercepted or recorded, consider the channel used. Anyone who hears, records, photographs, logs, or captures the two 12-word lists has the full root secret and can decrypt future traffic until re-key.

The application can assist entering the words, after typing the first 2 characters a list of matching words from BIP-39 is presented for quicker selection. The BIP-39 checksum is used as the first line of defence against incorrect input.

One party is designated as Entity1, the other Entity2; whichever side generated the 12-word list with the lower value is Entity1. The decoded 16-byte values are compared numerically (byte[0] to byte[15]).

In the highly unlikely event that both word lists are identical, the process is restarted. In the following example, Alice is Entity1 and Bob Entity2.

The result is a shared 24-word list: Entity1's 12 words followed by Entity2's 12 words, known in full to both parties and to nobody else.

1.2 Master Key Derivation

The two 12 words are decoded and the 4-bit checksum is discarded, yielding the original 128 bits of entropy. The decoded entity1_bytes concatenated with decoded entity2_bytes represent 256 bits (32 bytes) of entropy. HKDF is applied to produce a 256-bit master key:

```
master = HKDF-SHA256-Extract([SALT] "NoPrying/L2/master", [IKM] entity1_bytes || entity2_bytes)
```

1.3 Bootstrap Verification Fingerprint

Each computes a short verification code, verification_code is taken from:

```
verification_bytes = HKDF-SHA256-Expand( [PRK] master, [INFO] "NoPrying/L2/verification", [L] 8)
verification_code = uint64_be(verification_bytes) mod 1,000,000,000
```

9 digits are created by taking 8 bytes output, interpreted as a big-endian unsigned 64-bit integer, reduced modulo 10^{9}, and front zero-padded to 9 digits. Present as xxx-xxx-xxx for readability. This introduces modulo bias, which is acceptable for a human verification fingerprint.

Both parties verify the nine digits match, confirming that both parties derived the same master key and that the word exchange was not corrupted or impersonated. A mismatch means the session is abandoned.

If an attacker silently records the voice call without altering it, both parties derive the correct master key and matching verification codes. The attacker also derives the same master key. The verification fingerprint only detects active substitution, not passive eavesdropping.

1.4 Single Entity Configuration

It is also possible for one person to set up the entire connection. They would generate two 128-bit random values, keeping the lower-valued list for themselves as Entity1. Two 12-word lists are created via BIP-39, and are passed to the other party.

There is less security in one entity creating both word lists, and passing them together; the channel used to pass two 12-word lists should be intercept-free as it contains all information to access the channel.

In addition, the initial random data came from one source, rather than two.

2. Key Derivation

All keys are derived from the master key before it is destroyed.

For initial two-party session setup, chain keys are derived locally and are not transmitted through the server. Linked device addition (through QR) and group invitations are explicit exceptions and must be treated as sensitive key-transfer operations.

2.1 Initial Message Chain Keys

Both parties create:

```
E1_to_E2_msg_chain = HKDF-SHA256-Expand ( [PRK] master, [INFO] "NoPrying/L2/pairwise/e1/chain", [L] 32)
E2_to_E1_msg_chain = HKDF-SHA256-Expand ( [PRK] master, [INFO] "NoPrying/L2/pairwise/e2/chain", [L] 32)
```

2.2 Mailboxes

This encryption method does not use static mailboxes; instead, each message is delivered into a unique per message mailbox. Each side creates:

```
E1_to_E2_mailbox_chain = HKDF-SHA256-Expand ( [PRK] master, [INFO] "NoPrying/L2/pairwise/e1/mailbox", [L] 32)
E2_to_E1_mailbox_chain = HKDF-SHA256-Expand ( [PRK] master, [INFO] "NoPrying/L2/pairwise/e2/mailbox", [L] 32)
```

Entity1 uses E1_to_E2_mailbox_chain derived address to send a message, and Entity2 uses the same to receive messages from Entity1. Entity2 sends using E2_to_E1_mailbox_chain derived address.

Minimum metadata is used to facilitate mailboxes, such as time a message was composed, in UTC (encrypted). In addition, as per Section 11 notifications about a receiver are stored, last active time and if a message has been picked up.

2.3 Device IDs

The very first time the application runs, it asks how to setup this device, either link as an extra device to an existing device, or create a fresh entity to communication with remote parties. If the latter is chosen a local Device ID, 16-bytes of CSPRNG and asks the user for their identifier (name), and a device identifier (Phone, PC, etc). This is used when multiple devices are added to an existing device, an identifier is required so that said device could be removed later on:

```
Local_Device_ID = CSPRNG(16)
```

On creating a bootstrap channel the very first message sent, by each side is a notify id message informing the other side of their Device_ID which is then assigned to Remote_Device_ID.

2.4 Lightweight Connection Repair

A quick repair system should exist for entities which find themselves with a broken communication channel, which could be caused by a stale state from a restored backup, or server issues. A recovery key is created,

which is the same on both sides:

```
Lightweight_Recovery = HKDF-SHA256-Expand ( [PRK] master, [INFO] "NoPrying/L2/lightweight-recovery", [L] 32))
```

2.5 Secret Destruction

After deriving the initial chain keys, mailbox and Device IDs; the master key, both word lists, and all intermediate values are securely erased from memory. They are never persisted to storage.

All that remains in secure storage is:

```
E1_to_E2_msg_chain,
E2_to_E1_msg_chain,
E1_to_E2_mailbox_chain,
E2_to_E1_mailbox_chain,
Remote_Device_ID

Device_ID    [one value for the whole application for these variables]
User Name
Device Identifier

Lightweight_Recovery
```

Later on, as mailboxes advance, Entity1 also keeps the previous sent mailbox_address (and Entity2 keeps theirs), so it can check when a message has been picked up by the other entity (if tracking option is not disabled).

Note if implementing other additional levels, such as Level 4, the master key would be used before destruction to initialize any required keys for L4.

3. Message Key Ratchet

3.1 Per-Message Key Advancement, sending

When Entity1 (Alice) creates a message, the process of sending is:

1. Create message key whilst advancing chain:

   ```
   message_step = HKDF-SHA256-Expand ( [PRK] E1_to_E2_msg_chain, [INFO] "NoPrying/L2/pairwise/e1/message-step", [L] 76)
   next_message_chain = message_step[0:32]
   L2_message_key = message_step[32:64]
   gcmsiv_nonce = message_step[64:76]

   mailbox_step = HKDF-SHA256-Expand ( [PRK] E1_to_E2_mailbox_chain, [INFO] "NoPrying/L2/pairwise/e1/mailbox-step", [L] 64)
   next_mailbox_chain = mailbox_step[0:32]
   mailbox_address = mailbox_step[32:64]

   ratchet_secret = CSPRNG(32 bytes)
   ```

2. Encrypt message using L2_message_key and deliver to mailbox_address, using the nonce gcmsiv_nonce, and supplying ratchet_secret see Section 4 for encryption details.

3. After successful send, advance mailbox and message chain, and folding in ratchet_secret:

   ```
   message_mix_prk = HKDF-SHA256-Extract ( [SALT] next_message_chain, [IKM] ratchet_secret)
   E1_to_E2_msg_chain = HKDF-SHA256-Expand ( [PRK] message_mix_prk, [INFO] "NoPrying/L2/pairwise/e1/message-chain-inject", [L] 32)

   mailbox_mix_prk = HKDF-SHA256-Extract ( [SALT] next_mailbox_chain, [IKM] ratchet_secret)
   E1_to_E2_mailbox_chain = HKDF-SHA256-Expand ( [PRK] mailbox_mix_prk, [INFO] "NoPrying/L2/pairwise/e1/mailbox-chain-inject", [L] 32)
   ```

The sender stores the encrypted envelope locally and retransmits it unchanged on retry (see Section 4.6)

3.2 Receiving and Per-Message Key Advancement

Entity2 (Bob) polls mailbox_address, the mailbox chain can be expanded to pull just the mailbox address each poll (which would be no sooner than 30 seconds apart), the other variables are not extracted. If the box is full, receives an encrypted message.

1. Advance message chain, and get new message decoding key:

   ```
   message_step = HKDF-SHA256-Expand ( [PRK] E1_to_E2_msg_chain, [INFO] "NoPrying/L2/pairwise/e1/message-step", [L] 76)
   next_message_chain = message_step[0:32]
   L2_message_key = message_step[32:64]
   ```

gcmsiv_nonce = message_step[64:76]

mailbox_step = HKDF-SHA256-Expand ([PRK] E1_to_E2_mailbox_chain, [INFO] "NoPrying/L2/pairwise/e1/mailbox-step", [L] 64)
next_mailbox_chain = mailbox_step[0:32]
mailbox_address = mailbox_step[32:64]

2. Decrypt message using L2_message_key, see Section 4, this also extracts ratchet_secret.

3. Advance mailbox address and chain:

message_mix_prk = HKDF-SHA256-Extract ([SALT] next_message_chain, [IKM] ratchet_secret)
E1_to_E2_msg_chain = HKDF-SHA256-Expand ([PRK] message_mix_prk, [INFO] "NoPrying/L2/pairwise/e1/message-chain-inject", [L] 32)

mailbox_mix_prk = HKDF-SHA256-Extract ([SALT] next_mailbox_chain, [IKM] ratchet_secret)
E1_to_E2_mailbox_chain = HKDF-SHA256-Expand ([PRK] mailbox_mix_prk, [INFO] "NoPrying/L2/pairwise/e1/mailbox-chain-inject", [L] 32)

L2_message_key is used once to encrypt, then securely overwritten and destroyed. The server will only enact a purge after the receiver indicates they have the message and the allowed download_allowance is exhausted.

No state negotiation or direction-change signalling is required. The ratchet is self-consistent by construction.

For Entity2 (Bob) to send a message, his E2_to_E1_msg_chain and E2_to_E1_mailbox_chain are used, along with [INFO] "NoPrying/L2/pairwise/e2/message-step" and [INFO] "NoPrying/L2/pairwise/e2/mailbox-step", and Alice would use those same variables to decode. Also "NoPrying/L2/pairwise/e2/message-chain-inject" and "NoPrying/L2/pairwise/e2/mailbox-chain-inject" would be used to forward the ratchet_secret into the chains.

3.3 Forward Secrecy

Each message is encrypted with a unique key derived by ratcheting forward from the previous key. Once a message has been sent and the key advanced, the previous key is destroyed and cannot be recovered. An attacker who obtains the current key state cannot re-wind keys to decrypt previously sent messages.

3.4 Self-Healing

If the attacker obtains keys to any side of a communication channel, through an exploit, or reading from the actual device, then later on loses access to the device; self-healing is a process to exclude them automatically. NoPrying symmetric self-healing works by introducing a ratchet_secret during sending of a message. These are the specifics of the self-healing:

- To continue to access messages after a ratchet_secret the attacker would have to read a message before the intended recipient, the server only allows a set download_allowance. Doing so allows the discovery of said action, if reading from the server, when the real recipient later tries to read.
- If the real recipient reads the message first, the server will have purged the message (including new ratchet_secret), the attacker will not have the information to move the ratchet forward.

- Without access to the actual message through normal channels, attacker would then either have to exploit the server (to read messages directly, and avoid the canary trap), or read decrypt network traffic between client and server, at the transport layer. Which could mean having to decrypt HTTPS, or any employed VPN.

The self-healing provides recovery only against a snapshot attacker who loses endpoint access and does not obtain the next valid ciphertext in that same direction before the legitimate recipient collects it, however normal access through an uncompromised server, this action is likely to be highlighted through the canary protection.

Self-healing is in the direction of the message send, both directions would have to send a message to heal both chains, if both sides were compromised.

It should be noted that an attacker could take over a channel, by writing their own message, which would exclude the real contact (as they fold in a ratchet_secret). This would be detectable by the real user when try to write to a mailbox and it is occupied.

3.5 Lightweight Recovery

A message chain can be broken between two parties, through a restore from backup (client or server), or simply remaining offline until the message was purged. A broken chain means both sides are out of sync and cannot recover without taking action through a lightweight recovery process.

The process, an option will exist for a given contact to initiate the lightweight recovery, both sides need to initiate it. Before initializing, both entities should try to read as many waiting messages as possible.

Entity1 generates a random 6-word bip39 word phrase, Entity2 also generates a random 6-word phrase, which are passed OOB to the other party. The decoded entropy from the two 6-word phrases is combined as entity1_bytes || entity2_bytes.

```
recovery_mix_prk = HKDF-SHA256-Extract ( [SALT] Lightweight_Recovery, [IKM] entity1_bytes || entity2_bytes)
```

A verification code is generated:

```
verification_bytes = HKDF-SHA256-Expand ( [PRK] recovery_mix_prk, [INFO] "NoPrying/L2/lw-verification", [L] 8)
verification_code = uint64_be(verification_bytes) mod 1,000,000,000
```

9 digits are created by taking 8 bytes output, interpreted as a big-endian unsigned 64-bit integer, reduced modulo 10^9, and front zero-padded to 9 digits. Present as xxx-xxx-xxx for readability. This introduces modulo bias, which is acceptable for a human verification fingerprint. When verified the app creates:

```
recovery_step = HKDF-SHA256-Expand ( [PRK] recovery_mix_prk, [INFO] "NoPrying/L2/lightweight-initialize", [L] 64)
recovery_mailbox = recovery_step[0:32]
recovery_message_key = recovery_step[32:64]
```

A lightweight recovery is generated by Entity1 and delivered to recovery_mailbox encrypted with recovery_message_key, as per section 4.3, with normal padding. Using a message type: 11 Lightweight Recovery. ratchet_secret new random entropy is the element which is required to pass from Entity1 to Entity2, this is extracted and used by both sides to recreate new message chains:

```
recovery_mix_prk = HKDF-SHA256-Extract ( [SALT] recovery_step, [IKM] ratchet_secret)
E1_to_E2_msg_chain = HKDF-SHA256-Expand ( [PRK] recovery_mix_prk, [INFO] "NoPrying/L2/lw-recovery/e1/message-chain-inject", [L] 32)
E2_to_E1_msg_chain = HKDF-SHA256-Expand ( [PRK] recovery_mix_prk, [INFO] "NoPrying/L2/lw-recovery/e2/message-chain-inject", [L] 32)
E1_to_E2_mailbox_chain = HKDF-SHA256-Expand ( [PRK] recovery_mix_prk, [INFO] "NoPrying/L2/lw-recovery/e1/mailbox-chain-inject", [L] 32)
E2_to_E1_mailbox_chain = HKDF-SHA256-Expand ( [PRK] recovery_mix_prk, [INFO] "NoPrying/L2/lw-recovery/e2/mailbox-chain-inject", [L] 32)
```

The message is also protected with a one-time-collection, then purged from server. The chains are persisted and used going forward.

Notes: should verification_code fail consistently, the user should be shown the possibilities which include:

- corruption of the original Lightweight_Recovery value, in such a rare case the channel and contact should be removed and the contact re-added through a full bootstrap.
- Intercept and MITM substitution
- User input error

4. Message Encryption / Decryption

4.1 Algorithm

All messages are encrypted with AEAD-AES-256-GCM-SIV. Instead of using a persistent key, the AES algorithm uses the ephemeral 32-byte L2_message_key freshly derived in the ratchet step (Section 3.1).

GCM-SIV provides both confidentiality and authenticated integrity and prevents MITM bit flipping. Any tampering with the ciphertext causes decryption failure and the message is discarded.

Immediately after the encryption (or decryption) operation is completed, the L2_message_key is securely erased from memory. It is never written to secure storage, general storage, or retained for the next message. The only persistent elements retained for the next message are the new chain keys (message and mailbox).

4.2 GCM-SIV Nonce Management

Each message is encrypted with a derived unique 12-byte (96-bit) gcmsiv_nonce, provided during chain expand in Section 3, ensuring uniqueness.

If a sender is retrying a failed send, the same nonce should be used, so that the server can detect if an identical message was already received on the last try.

A device which restored from a backup, could potentially generate a new message with an already used nonce, however this is mitigated with the GCM-SIV protocol.

Nonce is all zeros for message types: 10 Linked Binary Object 11 Lightweight Recovery

4.3 Pre-L2 Encryption Bundle Makeup

Before encryption, the pre encryption bundle is constructed as:

```
L2_pre_encryption_bundle = message_type || length_prefix || composed time || ratchet_secret || message_payload
```

Where:

message_type uint8 identifying:

0 normal message,
1 notify id,
2 multi device comms channel creation,

3 security notification (such as new linked device added),
4 remove linked device,
5 new group member add,
6 remove group member,
7 group message,
8 ratchet forward,
9 L4_init_payload key transfer (L4),
10 Linked Binary Object.

length_prefix	uint32 big-endian (BE) length of message_payload in bytes (excluding any padding)
composed time	int64 (BE) UTC composed time in milliseconds since 1970 (Unix time)
ratchet_secret	32 bytes, generated in Section 3. Note message_type of 10 does not use ratchet_secret (is all zeros).
message_payload	Depends on message_type the following are defined (see section 4.4).

4.4 Message Payloads

Each message payload depends on the message_type:

0 normal message / 10 Linked Binary Object

Contains message components, either singular or multiple repeated, in the form:

- Message_identifier Uint32 (BE)
- len16(MIME_type)
- MIME_type ASCII identifier, such as: text/plain; charset=UTF-8
- data_size Uint32 (BE)
- Data, in the above example it would be UTF-8 encoded text

A message can contain many components, or even blank for certain message types. Applications would pre-define which image formats to accept, or videos. Text is always UTF-8 encoded. The server will always set the download_allowance count to 1 for normal message, as it is never read by multiple entries.

Message_identifier allows modifications of existing sent messages by sending an edit, or requesting 'unsend', for the client not server. A matching message Identifier to an already stored message, will replace it with the new sent message, a empty message is an unsend/deletion request.

Special MIME identifiers, which can have a zero Message_identifier:

- application/x-special-sent_to_device_ids Data is a list of device ids, each one 16 bytes long. This inclusion allows a recipient to indicate to the user that the remote client might not have all their devices, and offer a creation. Also included

  The allows detection and correction of the edge-case where two contacts which have just linked add additional devices at the exact same time (the new devices would not know about each other). See section 10 for use.

- application/x-special-binary a linked binary, only applicable to message_type=0 or 7, see section 4.7

1 notify_id

Very first communication on a new channel setup through bootstrap, each direction:

Device_ID[16] | | len16(User_Name) | | User_Name utf-8 string | | len16(Device_Name) | | Device_Name utf-8 string

Device_ID	16 bytes of random data, remains constant for a application
User_Name	friendly name of the person, is sent to the other party

Device_Name identifier of the device, such as Phone, Tablet, Laptop, PC

2 multi device comms channel creation

Very first message sent to a new linked device by the creator, comprises of this entry repeated (see section 10), then also sent to each existing known device creating new channels:

Device_Type byte | | IsEntity1 byte | | Device_Group_ID uint32 BE | | Entropy[32] | | Device_ID[16] | | len16(User_Name) | | User_Name utf-8 string | | len16(Device_Name) | | Device_Name utf-8 string

Device_Type 0= Own Linked Device 1=Remote Contact.

Informs if the device belongs to the person, or is a new connection to a remote:

Own Linked Device should only be accepted from an existing own device linked channel. A remote connection (another person) cannot link as one of yours.

Remote Contact informs a remote contact of a new linked device, this new device should be added to the same contact list as the entity which sent it.

IsEntity1 0=no 1=yes

Device_Group_ID a grouping ID, if Alice is linking a new device, and Bob has two devices, the new device needs to link those 2 devices together, a matching Device_Group_ID in the list ensures that those two devices are stored under one contact. This value is only used by the new linked device. Bob for example can assign Alice's new device to the existing channel. The test would be If Entity1=1 then use device_group_ids

Entropy	used to create the communication initial keys
Device_ID	device ID assigned to the new contact, if all zeros then is not known to by creator
User_Name	of new contact, it can be zero length for unknown
Device_Name	for new contact, it can be zero length for unknown

3 security notification (such as new linked device added),

4 remove linked device

Removal_Type byte | | Device_ID[16]

Removal_Type	0=close this comms channel (other end wants to close) 1=remove device matching device ID
Device_ID	ID to remove, when Removal_Type=1. Note only remove devices assigned to same contact as sender

5 new group member add,
6 remove group member,
7 group message,

8 ratchet forward

9 L4_init_payload key transfer (L4),

.

11 Lightweight Recovery

Performs a recovery for broken communication channels. No message payload is required, AAD authenticates the ratchet_secret. AAD direction is Entity1-to-Entity2. See section 3.5

4.5 Random Padding

Padding, of random-bytes, is appended to each L2_pre_encryption_bundle. Whilst a bundle can be less than 128 bytes if encrypting a short message, Level 4 adds approximately 3224 bytes overhead per message. The padding is designed to hide L2 or L4 based on size to effectively hide L4 encryption from observers:

min_rnd_bytes = len(L2_pre_encryption_bundle)

min_rnd_bytes is then scaled up to the next bucket size:

4096, 8192, 16384, 32768, 65536, 131072, 262144

If above 262144 then is scaled to the next multiple of 262144.

Padding is added:

L2_pre_encryption_bundle_padded = L2_pre_encryption_bundle || padding

The recipient strips padding after decryption using a length prefix embedded in the bundle, at the start of the message. A minimum message size of 4096 bytes.

4.6 Encryption

Message encryption comprises:

For normal L2 messages:
aad = len16(protocol_id) || protocol_id || version || direction || mailbox_address

For Linked Binary messages:

aad = len16(protocol_id) || protocol_id || version || direction || mailbox_address || binary_special

protocol_id = fixed ASCII string "NoPrying"
version = uint8 protocol version: 0 v1.00,
direction = uint8 identifying Entity1-to-Entity2 (0) or Entity2-to-Entity1 (1), or Linked Binary (2)
mailbox_address = the 32-byte mailbox address into which this message is delivered
binary_special = only used if message_type is 10 (Linked Binary Object):

For non-group messages it is the sender device_id [16 bytes].
For group messages it is the group ID [16 bytes]

The Additional Authenticated Data, or AAD, is not encrypted, but it is authenticated by AES-GCM-SIV. If any AAD field is changed, or ciphertext is moved to a different mailbox address, decryption fails.

ciphertext || tag = AEAD-AES-256_GCM-SIV-Encrypt([KEY] key, [NONCE] nonce, [AAD] aad, [PLAINTEXT] L2_pre_encryption_bundle_padded)

Normal 2 message encryption:

key = L2_message_key
nonce = gcmsiv_nonce
aad = normal_l2_aad
plaintext = L2_pre_encryption_bundle_padded

Linked binary object encryption:

key = blob_key
nonce = 0^96
aad = aad with binary_special
plaintext = L2_pre_encryption_bundle_padded (note ratchet_secret is all zeros and unused)

The final message envelope stored in the mailbox is:

stored_envelope = version || ciphertext || tag

With field sizes:

version = 1 byte, uint8
ciphertext = variable length, same length as L2_pre_encryption_bundle_padded
tag = 16 bytes, AES-GCM-SIV authentication tag

The nonce does not itself detect tampering. Tampering is detected by the AES-GCM-SIV authentication tag, which authenticates both the ciphertext and the AAD.

On receipt, version is parsed from the envelope as untrusted bytes. The receiver constructs AAD using those exact bytes, the expected direction, and the expected mailbox address, nonce.

If AEAD-AES-256-GCM-SIV authentication fails, the message is discarded and the ratchet state MUST NOT be advanced as if a valid message had been received. If authentication fails, the client MUST NOT advance the ratchet. The message should be quarantined and the mailbox rechecked. If repeated failure persists, the application should report possible tampering, corruption, or desynchronisation, and offer re-key.

For retry handling, a sender MUST NOT re-create the L2_pre_encryption_bundle for the same pending send. The sender encrypts once, stores the exact pending envelope and pending next ratchet state locally, and retransmits the identical envelope until the server acknowledges storage. Only after server acknowledgement is received does the sender commit the new chain key and next mailbox address.

After decryption, the recipient should reject malformed L2_pre_encryption_bundle for:

- minimum length: fixed items before user message;
- length_prefix <= remaining_bundle_after_fixed_fields;
- maximum message size;
- valid UTF-8 only when content type says text;
- binary payload framing;
- unknown message types;
- unknown protocol versions;
- timestamp sanity bounds;
- padding length limits.

4.7 Binary Linked Message

All components, except for text are stored as a linked binary. The reasons for this are two-fold. The server is able to purge binaries sooner than normal messages, without risk of breaking a chain. Also for a group message, as there could be 100 members, sending a 4MB image to all 100 members individually would be a huge upload, instead a linked binary is used.

The normal message contains a MIME type of application/x-special-binary, the data part of the component contains a link:

```
blob_mailbox_address[32] || blob_key[32]

blob_mailbox_address = a random 32-byte mailbox address, generated by the sender
blob_key = random 32-byte encryption key, generated by the sender
```

Notes about a binary message:

- Encrypted using AEAD-AES-256-GCM-SIV with the blob_key and an all-zero gcmsiv_nonce (safe since the key is single-use).
- The sender can include the download_allowance for a binary, for normal communication channels it is (remote device count) + (own device count – 1). For groups it is number of group members + (own device count – 1).

The contents of the linked binary are constructed as any existing message; including L2_pre_encryption_bundle and can contain any standard components. Therefore, the process is:

```
Normal message / group message:
        contains one or more mime types: application/x-special-binary which points to the binary object.

Binary message:
        message_type = 10 Linked Binary Object
        stored at blob_mailbox_address, encrypted with blob_key
        not a normal L2 ratchet message
```

usual message construction, from multiple components
aad has a special normal message address component, to ensure a message and binary are linked
ratchet_secret all zeros, ignored

For linked binary objects, AAD construction is selected by the retrieval context. A linked binary object is fetched only after an authenticated normal L2 message has supplied an application/x-special-binary descriptor. That descriptor tells the receiver to treat the fetched object as message_type = 10 and supplies the required binary_special context into AAD.

Linked binary upload:

write-once object store
identical upload retry MAY be accepted
different bytes at same blob_mailbox_address MUST be rejected

Linked binary download:

multi-download up to download_allowance
message purged on download_allowance exhaustion
no ratchet advancement
no mailbox-chain advancement
purge when download_allowance is exhausted or expiry occurs

5. Message Delivery

Messages are delivered over TCP/HTTPS, an extra layer of security.

Entity1 delivers into the current Entity1 derived mailbox_address; the server records each message as delivered before accepting the next. The sender knows not to advance the ratchet until the server responds to record receiving the message. The possibilities of delivering a message:

1. Alice sends a message, the server checks mailbox is empty, stores and acknowledges; Alice applies the ratchet (chain key and mailbox).

2. Alice sends a message, the server checks mailbox is empty, stores and acknowledges. Alice misses this acknowledgment and retries the send. On the retry the exact same payload should be transmitted, including same padding, there is no need for Alice to re-encrypt the message. The mailbox is no longer empty; however, the server sees the contents are identical, so acknowledges receipt; Alice applies the ratchet (chain key and mailbox).

3. If the current send mailbox is occupied by a non-identical envelope, the channel is in an unsafe or ambiguous state.

   Possible causes include:

   - local rollback / backup restore,
   - another local device using stale state,
   - server replay or corruption,
   - an attacker with channel state.

   Recovery advancement is only possible across envelopes that are still available and can be authenticated/decrypted by the recovering client. A collected or purged mailbox cannot supply the missing ratchet_secret; in that case, OOB re-key is required.

Sender Exact Order

1. Derive message key, mailbox address, next chains.
2. Generate L2_pre_encryption_bundle, with padding, nonce, ciphertext, tag once.
3. Persist but not commit next_xxxx states.
4. Open communication channel; check the mailbox has never been used, upload payload if all clear on same connection. Indicate to server download_allowance.
5. If server says “stored” or “already stored identical envelope,” commit next_xxxxx states to secure storage.
6. If step (4) returned “occupied with different envelope,” stop and require re-key.

A random collision in a 256-bit mailbox space is cryptographically negligible.

Once the receiver reads a message, the old mailbox on the client device is forgotten. The server purges mailboxes after collection, or an extended time, such as 6 months.

Receiver Exact Order:

1. Derive the next expected mailbox address from the current mailbox chain without committing state.
2. Query the mailbox
3. Persist: UniqueCollectionID, If mailbox occupied, receive contents. If mailbox purged, or already read, inform a re-key required.
4. Derive the expected message key, nonce, next message chain, and next mailbox chain without committing state.
5. Parse version as untrusted envelope bytes
6. Reconstruct AAD using those exact envelope bytes, expected direction, and expected mailbox address
7. Attempt AEAD-AES-256-GCM-SIV authentication/decryption
8. If authentication fails, do not advance state and do not ACK collection
9. If authentication succeeds, validate `L2_pre_encryption_bundle` framing (for L4 implementations, it would be passed to L4 for decoding).
10. Persist decrypted message/display record and next chain state atomically.
11. Only after durable local commit, send ACK_COLLECT to the server, the server can purge the message.
12. Advance new chain state and commit to secure storage.

Without this, a crash after server collection but before local state commit loses the ratchet_secret and breaks the chain.

After a mailbox has been used, it will always be flagged as used, even if it has been purged, for a period of up to 1 year (purge window for messages is 6 months). If the receiving Entity is offline and the message is purged due to non-pickup, then on eventual read the server would indicate 'Server Purged due to extended time expiry'. The client would not be able to ratchet the keys forward, only a OOB rekey will resync the connection.

If any read mailbox cannot decrypt/authenticate, it should be indicated to the user and a re-key offered.

Messages in a single accepted mailbox chain are read in chain order.

Note: used-but-purged indications are not cryptographic evidence of a past valid message. They are trusted server metadata used only for recovery from expiry. A malicious server can cause skipped messages or desynchronization prompts.

6. Secret Sender & Secret Receiver

The sender does not need to attach any identity metadata to the message.

The server code will not include IP address logging (beyond temporary in memory lists to deter server abuse); the code for the server will be Open Source, allowing for self-compile and self-hosted servers. A deployed server has a level of trust with the end user; a company self-hosting, or a trusted server implementation ensures best practice in metadata collection.

The sender delivers to a mailbox derived from their own current Entity mailbox identifier (acting as an outbox). The server does not know who owns either Entity mailbox, as each ratchet creates a new unique box.

The protocol avoids account identifiers and static addresses. Mailbox addresses are not linkable without chain state; however, a maliciously deployed server can still observe network IP, timing, polling, and delivery metadata. This is not a traffic-analysis-resistant protocol.

7. Key Storage

All key material is stored exclusively in the operating system's secure storage facility. Implementations MUST document the exact platform storage classes used, backup exclusion policy, biometric/passcode requirements, and device migration procedures.

Key material is never written to general application storage, logs, or backups.

Temporary encryption keys and other temporary keys are disposed of securely (by overwriting with zeros) before being destroyed: best-effort zeroization of mutable buffers; avoid immutable strings for secrets; disable screenshots/backups/logging; use OS non-exportable storage where possible.

8. Out-of-Band Re-Key

At any time, either party may initiate a full re-key OOB. Both parties restart from Section 1 — generating new word lists, exchanging them by phone, completing the verification fingerprint, and re-deriving all chain keys, mailboxes, and device IDs from scratch.

This is the correct response to any suspected compromise, or application anomaly indication. It takes approximately two minutes, and fully resets the entire key hierarchy. It should be made easy and prominent in the application UI.

A quicker lightweight recovery can also be offered, see section 3.5.

9. One-time Collection & Canary

One-time Collection: When a physical package is sent through the mail, it waits for collection after-which it is no longer available for collection.

Canary: historically canaries were used in mines, if there was an issue with the air, the canary would indicate this to miners, informing something was wrong.

9.1 Design

These two simple systems can be combined to increase security, through defence-in-depth and anomaly detection of a symmetric based encryption system.

A message sent from Alice to Bob has one channel, when the real Bob reads the message, it does not need to stay on the server. Leaked keys from a stale compromised-backup could enable a message to be read later on if it were still available. A message box has payload states:

1. MailBoxEmpty=0

2. AvailableForCollection=1: server stores download_allowance on upload. Each new UniqueCollectionID collection attempt is stored with mailbox

   When client reads all contents, and acknowledges receipt, the UniqueCollectionID is moved to a a separate collected list, where the server will not give out the contents with the same ID again.

3. FullyCollectedPurged=2 – when download_allowance is exhausted (when the collected list *size* equals the allowed count), the contents are purged.

4. ExpiryPurged=3: due to non-collection in extended time expiry

5. CanaryTriggered=4 for Mailbox, too many collections with different UniqueCollectionIDs. A CanaryTriggered state is never overwritten, even if later the ExpiryPurged takes place.

When a client wishes to collect a message it creates a new collection identifier UniqueCollectionID, a 16-byte CSPRNG(16), which is stored by the server. The server builds a list of unique collection IDs until they equal the allowed collection count, then the message is purged. If an app is restarted during collection, the same collection identifier is used, the purge would not be triggered on this recollection where the receiver had part of the download. The server records UniqueCollectionID immediately when it returns the payload, not only after ACK_COLLECT.

A legitimate re-collection / triggered use case could be when Bob tries to re-collect previously collected messages; his device might have been restored from a previous state; in such instance the anomaly notification would include this possibility. Bob and Alice can communicate off NoPrying to clarify the detected anomaly and offer a lightweight connection repair.

9.2 Benefits

Reduced ciphertext-retention window

One-time collection reduces the time during which ciphertext remains available to a server attacker, later forensic process, or attacker who obtains stale chain state after the recipient has already collected the message.

Partial mitigation for stale backup compromise

This protects only against delayed access to server-retained ciphertext. It does not protect if the attacker polls the mailbox before the legitimate recipient, continuously monitors the mailbox, or has already captured the ciphertext in transit or from the server.

Detection of rollback or cloned-state problems

The canary is valuable beyond attackers. It can reveal:

- restored devices using stale state,
- duplicated app state from backups,
- cloned devices,
- buggy clients retrying old mailboxes,
- multi-device state confusion,
- server replay/corruption.

Better replay resistance at the delivery layer

The canary is not required for message authentication, but it provides a delivery-layer signal that a one-time mailbox has been reused after collection.

Cleaner data minimisation story

The server does not keep messages indefinitely, and even encrypted payloads are removed as soon as they are no longer needed.

Useful UX warning

A canary event gives users a clear prompt to re-key or investigates. This is especially important because the protocol otherwise has no global identity system, no server accounts, and no conventional device-management centre.

9.3 Limitations

First-collector-wins

The canary cannot authenticate the first collector. If an attacker with valid mailbox state collects the message

before the intended recipient, the server cannot distinguish that attacker from the recipient. The legitimate recipient's later attempt may trigger a canary, but the message may already have been disclosed.

Malicious server can lie

Canary alerts are advisory server metadata. They are useful against honest-server compromise, stale clients, and accidental duplicate access, but they are not cryptographic proof. A malicious server can suppress, fabricate, delay, or misreport canary events.

A server crash or bug could cause an inconsistent mailbox state.

Passive ciphertext capture is not addressed

If an attacker can passively record ciphertext while it is being delivered or collected, one-time server deletion does not erase the attacker's copy. If that attacker later obtains the relevant chain state, the recorded ciphertext may still be decryptable.

Canary creates metadata

One-time collection reduces retained payload data but may increase retained mailbox-state metadata. Implementations SHOULD minimise canary metadata, avoid long-term retention, and make read-receipt-like behaviours configurable.

10. Multiple Device Support

It stands to reason that Alice might have multiple devices she messages using, such as a phone and a PC. The following text assumes that Alice is Entity1 and is using No Prying on a phone and wants to also add a new PC, Alice has 300 contacts (including sub-devices of existing contacts) who would need automatically informing of the new device.

Note: Unlike other messaging systems, which allow access through a central server, with a simple password, which retrieves years of past messages, protected by a short password — a security hole. When a new device is added, it only has access to new messages going forward, not past.

When linking a new linked device, it must not already be in use with any have any contacts. Therefore, it should be a fresh install of No Prying. The existing device doing the linking, has contacts already.

10.1 New Linked Device - Initializing Communication Channel

Before starting a linking process, the existing device should read all pending messages from the server and send any pending.

Alice has two existing devices A1 (phone), and A2 (laptop), and wants to add a new device A3 (PC). Either A1 or A2 can setup the link. The new device (A3) installs No Prying, there are two options after an install: link to one of my already in use devices, or link to a remote contact.

On A1, the option ‘link to my own new device’; the application asks a device name is chosen/created (PC in this example).

A1 generates a random 256-bit. The creator (A1) takes Entity2 designation and passes the 256-bit entropy value to A3, which takes the designation Entity1, by either QR Code or single 24-word list.

A QR code is preferred to pass this entropy quickly through screen and onboard camera, with built-in CRC checks. An alternative is a 24-word lists with relevant security risk warnings about the transfer (such as snapshotting codes and emailing). An adapted bootstrap verification from Section 1 is used on A3 which will only receive a correctly encoded 24 word list, to ensure no errors on transfer and the link is brought into operation by creating the required keys on A1 and A3. The HKDF-SHA256-Extract per Section 1.2 to form the master however using salt:

"NoPrying/L2/device-link/master".

From which all chain keys, mailboxes, and device IDs are derived. This communication channel is used as the permanent channel between A1 and A3 going forward. The same device-link salt is used when creating the keys between all linked devices.

A single word list is chosen, as a different way of setting up the channel, this is a linked device, rather than a new contact, which uses two 12-word lists. The use of different salt, protects the setting up incorrectly as a remote contact instead of a linked device.

10.2 Creation of Communication Channels

At this point there is an active channel between A1 ←→ A3, A1 holds: User_Name the name associated with this A1/A2/A3 entity, Device_Name for A3 (PC in this example) and a created Device_ID.

Informing Everyone

A single multi-id list is created and sent to A3 as a multi device comms channel creation message, where multiple entries are appended:

> The very first entry contains A1's own User_Name, Device_Name and Device_ID. Entropy is 0^32 (as the comms channel is already created).
>
> The 2nd Entry relates to the device doing the setup, A3 is informed of A1's details, again Entropy 0^32
>
> The remainder of Linked devices and known contacts are added, each adding new fresh entropy which will be used by A3 and the other device to create a new comms channel.
>
> This bundle is sent to A3 as one multi device comms channel creation message.

Next 301 individual multi device comms channel creation messages (one own linked device and 300 contacts) are sent to all the other entities informing them of A3, comprising of just one entry using the same entropy which was sent to A3.

A3 MAY begin sending on A3 ↔ contact chains immediately after the multi device comms channel creation message is received. Until the contact has processed the introduction message on the existing A1 ↔ contact chain, A3's messages will sit in mailboxes the contact does not yet poll.

A Notify_ID should be sent after linking two devices, from each side, this ensures the other party has a Device_ID of the other party, which might not have been available to the device creating the linking (A1 and B1 create a new channel, but B1 has yet to get online, A1 pairs A2, but does not have B1s device ID), it also introduces fresh entropy from the actual devices.

Edge Case – missing Device Link

If Alice and Bob both start linking a new device at exactly the same time, after pairing, they retire their old devices. Bob does not know about Alice's new device and Alice does not know about Bob's new device; the creation of channels would miss the interlinking of the new devices. Applications can monitor the list of device_ids which are included in application/x-special-sent_to_device_ids. If the sender is missing any current IDs can ask the user if the missing device should be added. Care needs to be taken around recent pairings and older sent messages (which might have been queued).

10.3 Removal of Devices

For a device to be removed, the source can only be from a linked device, for example Alice gets a new phone, sets it up (A3), she has now 3 devices A1 (phone), A2 (laptop), A3 (PC). On setup every contact would have been informed of the new phone. Any of her devices can remove A1, including itself, and all linked devices

should be informed of the removal (including remote devices), through a notification message 4 remove linked device.

The other choice of removing, is if Alice no longer wishes to communicate with Bob, she can for each of her device IDs send the removal request to Bob device(s), and close those channels.

A remote contact would only remove a device it is associated with that channel, to stop a bad actor spamming removals to all their contacts for a device they are pretending to be. The device_id is used to identify the channel to remove.

Once a device is removed, it should clear all its keys. All devices notified of the removal should remove that message channel completely.

If a removed device is restored from a backup, it could have incorrect and stale contacts, there should be a provision for resyncing lost contacts.

10.4 New Bootstrap Remote Contact, Where Local Has Multiple Devices

Alice has 2 devices (A1 + A2) and adds Charlie (which also has 2 devices C1 + C2) as a new contact, this can be done on any of her devices. After adding the communication channel with Charlie, Alice introduces Charlie to each of her linked devices and informs Charlie of the linked devices. Section 10.2 (**Informing Everyone**) onwards but only targeting Charlie rather than every contact.

Note after the boot strap: these channels exist: A1 ←→ C1, A2 ←→ C1, C2 ←→A1, however A2 ←→ C2 would not be linked as they were not known until after the multi-device messages have been send and received.

10.5 Additional Functionality

Messages can be sent from device to device owned by the same person, this has a use case for transferring files between devices, easily and securely.

10.6 Multi-device Security

If a sender is sending messages to multiple devices, to defeat an attack where Alice might have lost control of her phone and the attacker silently added one of their devices as an additional device to Alice, Bob might not know, just thinks Alice has two devices. To defeat this, any message sent to multiple devices should automatically include details of the devices sent to, so it can be shown to the user. Section 4.4 details a custom MIME type of: application/x-special-sent_to_device_ids

11. Notifications (Metadata)

Various notifications are expected in a messaging app: last time a user was seen, and if a message has been received, as a general indication it might have been read.

On a system which does not track users through accounts, the system would have to use mailboxes as the go-between for passing notifications. Each Entity knows the next address they will receive from; each check can log a ‘last seen’ time. This can be checked by the other Entity.

Each Entity also keeps the last mailbox address previously left a message in, this can indicate when the other Entity has received the message as picking up a message can store the time stamp also.

The date of message composure is also included encrypted inside the message envelope.

Times would be stored in UTC.

Notifications should be user configurable allowing them to be disabled (except composition date which is encrypted), limiting metadata exposure.

12. Group Chat

Groups pose an interesting problem for secure messaging designs, the design should allow communications with all group members, yet be secure, there will always be trade-offs when it comes to group chat. There are two designs possible:

- Fan out design: where every person in the group writes an individual encrypted message to each member of the group when composing a new message, this is preferable, but a group of 10,000 members would be 10,000 separate messages.
- The server implements group cleverness, in that there are less encryptions needed, but relies on grouping such as tree designs. This places much trust in the server.

NoPrying has opted for the most secure option in group design, the first option, no trust in the server. However this places a limit on practical group sizes, which is 100 participants.

Creating the Group

Group chats are initiated by a single person, who creates a new group and assigns it a name, the group would have a unique 16-byte identifier. This name is stored locally by the initiator; it can be changed later on by sending a control message to existing members.

Adding Members

The requirement for adding members, they must already be contacts of the group initiator. There are three people Alice, Bob and Charlie. A new group is created; a similar process is used as with adding multi-devices:

- Each member of the group receives one channel (each way) per other person in the group, so a group with Alice, Bob, and Charlie would create the following channels:

  Alice >> Bob, Bob >> Alice, Alice >> Charlie, Charlie >> Alice, Bob >> Charlie, Charlie >> Bob.

  The group creator would choose which party is Entity1 send the key-set over to both parties over the existing messaging channel, therefore to add a person to a group they must be known by the group creator. This communication channel is exclusive to the group, upon invite the name of the group is sent with an ID. Any messages on said channel are exclusively for group chat only.

- It should be noted that Bob and Charlie might not know each other, the group creator creates a list of all contacts in the group, and an identifier for each contact. This identifier is sent with group id, and friendly name to the contacts and they create their own lists.

As the group creator has seen both sets of keys, a mandatory automatic ratchet message (no payload) sent between both parties for each new channel created, this ensures that the group creator no longer has active keys, the canary system would ensure the creator cannot collect the new entropy without discovery. A ratchet-only message uses message_type = 8, length_prefix = 0, no user-visible body, a fresh ratchet_secret, normal random padding, and normal AEAD authentication. It advances both message and mailbox chains exactly like a normal message. No messages should be sent to a group member until the initial ratchet has been received.

Sending Messages

A group message ID is used when sending, the receiver knows which group it is for, that channel is associated with a group and specific person in said group.

Removing a member

Officially only the group initiator can add and remove different members of a group, this is to stop a situation where Bob sends to Alice the group creator that he wants to self-remove, but does not send to the other members, effectively he is still in the group except Alice >> Bob messages, Alice the group creator would think he is gone.

Any group removal message needs to be checked it originates from the group creator.

It should be allowed for a group member to simply ignore a group, remove it themselves locally, the messages would still be left on the server, Alice would still see them as a member.

If a member who was removed, carried on sending messages to the group, as each connection is individual, they would be never picked up, if the order of checking for new messages is: Group creator always first, act on any removals, then group members. This ensures a member who has been offline for a period of time would not receive a message from an expelled member.

Message Binaries

There needs to be a trade-off when including non-text in groups, take a group with 100 members, if a 5MB image was uploaded, then the client would have to send 500MB, as each member has their own communication channel with said member. Instead, a different approach is needed, binaries are to be uploaded with a message_type of 10, see section 4.7

13. Security Design Philosophy

13.1 The Case for Pure Symmetric Design

This protocol excludes an asymmetric key exchange, which would establish chain keys dynamically, for the following reasons:

The chain key source was already available. Both parties already possess both halves of the word list from the bootstrap. Deriving chain keys from this material requires no transmission, no synchronisation, and no asymmetric operations. The security of the chain keys is exactly equivalent to the security of the bootstrap itself.

Asymmetric operations are the mathematical risk surface. The original motivation for this protocol was scepticism about mathematical assumptions in public key cryptography. Removing asymmetric operations entirely is consistent with and reinforces that motivation. The protocol now has no elliptic curve operations, no lattice operations, and no mathematical structures that could harbour undiscovered weaknesses.

Symmetric primitives are the most battle-hardened. AES-256 and HKDF (built on HMAC-SHA-256) have been subjected to decades of intense public scrutiny by the global cryptographic community. No practical attacks are known against AES-256 or HMAC-SHA-256 when used correctly with appropriate keys, nonces, and implementation discipline. They are also the primitives considered most resistant to quantum computing attacks — see Section 13.2.

Complexity removed is complexity that cannot fail. By excluding an asymmetric layer, the protocol avoids the complexity of key generation, public key transmission, encapsulation, decapsulation, hybrid combination, and direction-change signalling — each a potential source of implementation error.

This protocol provides post-compromise security against snapshot adversaries via a fresh per-message symmetric injection, refer to Section 3 for caveats.

13.2 Quantum Resistance

Counterintuitively, removing the asymmetric layer improves the quantum resistance of this protocol rather than diminishing it.

Quantum computers pose a serious threat to asymmetric cryptography — algorithms such as Shor's algorithm can efficiently break elliptic curve and RSA schemes. Grover's algorithm provides a quadratic speedup against symmetric primitives, effectively halving key length — meaning AES-256 provides approximately AES-128 equivalent security against a quantum adversary, which remains far beyond any known attack.

This protocol has no asymmetric operations whatsoever. The bootstrap is a human-mediated out-of-band exchange — it has no mathematical structure for a quantum algorithm to exploit. The encryption is AEAD-AES-256-GCM-SIV. The key derivation is HKDF over HMAC-SHA-256. None of these are meaningfully threatened by any known quantum algorithm.

The protocol is therefore natively quantum-resistant without requiring any post-quantum cryptographic primitives.

13.3 Forward Exposure Window

A snapshot adversary who exfiltrates chain state at a single point in time loses the chain in the direction of the sender, as a message contains a fresh ratchet_secret (Section 3). This provides post-compromise security against snapshot exfiltration. It does not protect against persistent network observers or persistent endpoint compromise; for these, only OOB re-key remediates.

14. Real-World Attack Scenarios

Scenario 1: Active malware or spyware

The most common meaningful attack on messaging applications.

Spyware reads plaintext before encryption or after decryption. No single device cryptographic mechanism in any protocol defends against this. Dynamic chain keys, double ratchets, and post-quantum algorithms all provide zero benefit. The only remedy is removing the malware from the device.

This protocol: No defence. No cryptographic protocol can protect plaintext (on the device) while an endpoint is actively compromised. Users should be educated that device security is the primary security boundary, not the messaging protocol.

Scenario 2: Physical device seizure

An adversary with physical access to an unlocked device can access key material in secure storage if they can bypass OS protections. Future messages are exposed until re-key on a clean device. Past messages are protected by forward secrecy — advanced keys are gone.

Dynamic chain keys would not help here: the attacker also obtains every new key as it is generated, since they have the device.

This protocol: Forward secrecy protects past messages (in respect of messages not accessible on said seized device). OOB re-key on a clean device closes the window for future messages. This is the only realistic scenario where asymmetric self-healing would help, however if the adversary has network level access (a state actor) the re-key can be intercepted with a MITM attack.

Scenario 3: Server compromise

An attacker gains access to the message server.

This protocol: Strong. The server holds only ciphertext. It has no key material and cannot decrypt any content. AEAD-AES-256-GCM-SIV authentication means tampered messages are detected and discarded on receipt.

Scenario 4: Snapshot key exfiltration

An attacker silently copies key material at a point in time, then loses all device access.

This protocol: Snapshot-only attackers (no continued network or device access) lose the chain after the next message in the send direction, due to the ratchet_secret injection in Section 3. If the attacker also has continued network access (such as HTTPS bypass, capturing each future ciphertext), the injection does not help and OOB re-key is required.

A normal attacker who can only read messages as any other user from the server, if they fetch the ratchet_secret from the server, they will likely be discovered through the canary system and one-time-collection system.

Scenario 5: Man-in-the-middle at session establishment

This protocol: Strong. The bootstrap is Out of Band. A MITM would need to intercept the communication. The verification fingerprint detects active substitution or transcription errors. It does not detect passive recording, successful voice impersonation, compromised devices, or an attacker who obtains the word lists. This is substantially stronger than Trust On First Use (TOFU).

For the highest-security bootstrap, an in-person exchange is preferred. If a voice call must be used, consider that targeted phone surveillance could record the word exchange.

Scenario 6: Quantum adversary

Future / theoretical. This protocol: Strong. There are no asymmetric operations. HKDF and AES-256 are considered quantum-resistant with wide safety margins. See Section 13.2.

15. Local Message Storage Considerations

Care should be given to the process of storing decrypted messages on the device. Ideally the messages should be protected by local database encryption. With thoughts to:

- whether keys are OS-bound or user-passphrase-derived
- backup exclusion
- migration rules
- deletion semantics
- screenshot/clipboard/indexing policy
- attachment storage
- notification message preview policy

16. Known Limitations

No long-term identity.

There are no persistent identity keys. Session continuity is provided by the out-of-band relationship, not the protocol. This is more of a practical limitation, which is balanced against the extra security it provides.

Consider that many “end-to-end encrypted” messaging systems allow a simple short password to be entered into a new device, and this loads and decrypts all previously sent messages. Security is an illusion in such a system, where the server holds the encryption keys.

Ongoing device compromise.

No protocol can recover from real-time, persistent attacker access to the device.

Server DoS

Because there are no accounts, the server needs an anti-abuse design:

- All binary message components, should use linked binary, which allows a shorter purge time on the content (such as a large video). A purged binary would not affect the chain, as no ratchet_secret is used by binaries.
- There should be a maximum message length set, lower than 2GB.

This is not cryptographic, but it is required for a deployable public server. Whilst a private company server could be protected other methods, such as password access the server, or a VPN to access server. A public general server implementation would need:

- 2nd server which verifies a user has correct credentials to access the messaging server, such as a subscription to service. The verification can give out a token which can be used once and is based on action, such as read mailbox, or upload to mailbox, is verified by the messaging server and is not stored. The 2nd server will count total stored over a period, a token storage request should indicate the size to store and the token should encode this size into the token. Users who store too much (a DoS) can be blocked.
- 2nd server counts the number of actions the user takes and places limits if they exceed fair use limits.

Server crash

This could involve a re-key for many participants, if sent a message during crash / storage internally.

A restore from backup would involve many more re-key requests. With this in mind, a server design should implement two distinctly separate storage locations, and everything duplicated. Large binary storage is less critical to message chain integrity, as such storage could be less critical for such (less requirement to duplicate)

Groups

The security of a group is very much tied to the group creator; they select people to join or remove. Users of the group should run outside the group checks to ensure individuals who are added to the group are who they are indicated to be. The group creator could spoof new users.

When groups use shared binary messages, there is a single collection point which a malicious server could tie together all users of a group.

17. Security Properties Summary

Property	Status	Notes
Bootstrap authentication	✅ OOB	Stronger than TOFU
Bootstrap verification	✅ short digit verification	Detects MITM substitution and errors
Asymmetric operations	✅ None	No mathematical key exchange assumptions
Quantum resistance	✅ Native	AES-256 + HKDF only; no asymmetric operations
Forward secrecy	✅ Per-message	Past messages safe on key exposure
Chain key synchronisation	✅ Zero — derived locally	Both known from bootstrap
Message integrity	✅ AEAD-AES-256-GCM-SIV	Tampering causes decryption failure
Nonce uniqueness	✅ unique nonce per key rotation	AEAD-AES-256-GCM-SIV usage mitigates reuse
Length obfuscation	✅ Padding	Obscures message size
Key storage isolation	✅ OS secure storage only	No key material in app storage
Implementation complexity	✅ Minimal	Fewest possible moving parts
Server metadata exposure	✅ No Accounts (Secret Sender and Receiver)	Server holds no accounts. A malicious server could observe mailboxes, message sizes and shared binary collections, linking to IP addresses.
Group chats	✅ Fan-out design	Strongest Encryption, per connection individual.
Multi-device	✅ Introduction authenticity	Inherited from existing chain & channel forward secrecy
One-time ciphertext collection	✅ Defence-in-depth	Payload deleted after download_allowance exhausted, reducing server-side ciphertext retention
Self-Healing	✅ / ⚠️ Split	Snapshot-only attacker: closed after next ratchet_secret injection. Network level attacker: until OOB re-key.

Property	Status	Notes
Lightweight recovery	✅ Quick resync for broken message chains	Avoids full re-key, authenticated by prior shared state + fresh two-party entropy