Gossip: decentralized pseudonymous private messaging protocol with quantum-resistant double ratchet and sealed metadata

Damir Vodenicarevic
dv@massa.net

2025

Contents

1 Overview

Design goals

Gossip is a new type of mobile-first instant messaging system built around the following pillars:

- **Privacy by default:** all your messages are encrypted before they leave your device, and only yourself and the person you are talking to can see them in the clear. Perfect forward secrecy and post-compromise security ensure that even in the case of key leaks, previous and future messages remain secret. Metadata is also encrypted, ensuring that no outsider knows who is talking to whom. Post-quantum encryption and authentication ensures that even attackers with powerful quantum computers can't read your messages or impersonate Gossip users.

- **No personal linkability:** users authenticate based on a passphrase they choose and are identified by a unique pseudonymous ID. No phone number or personally identifying information is required, new identities are easy to create and manage. The system also provides plausible deniability, ensuring that even if the user you are talking to leaks your conversation, you can still deny ever having sent those messages, since there is no cryptographic proof that they did not just forge them because both participants are aware of all message encryption secrets and messages are not signed.
- **Censorship resistance:** Gossip does not depend on central servers, it uses a decentralized network of nodes, thus ensuring network resilience, availability and strong resistance to censorship. Gossip also provides an alternative Progressive Web App hosted on the Massa Decentralized Web (DeWeb) that remains available in case the app is censored on mobile app stores. Even if Gossip's authors disappear or try to shut down the project, Gossip will still function.
- **Accessibility:** Gossip aims to be as feature-complete and usable as possible, hiding the underlying cryptographic complexity to ensure a smooth user experience. Gossip aims to be politically and socially neutral, welcoming anyone to use it.
- **Trustlessness:** you don't need to trust Gossip's creators to feel secure. Gossip's source code has been publicly available since the project's inception. It aims to be independently audited, and formally verified as much as possible. Only cryptographic primitives with strong public scrutiny are used.

2 Global architecture

Gossip relies on a decentralized network of nodes that propagate encrypted messages to each other and temporarily store them locally in their cache for 2 weeks. Anyone can retrieve data from any node. Anyone with a computer can easily become a node of the network.

The illustration below explains how the network functions, assuming Alice wants to send an encrypted message to Bob:

- The Gossip app on Alice's phone encrypts the message in a way that only herself or Bob can decrypt it, and sends it to one or more nodes in the Gossip network.
- The nodes of the Gossip network propagate the encrypted message and temporarily store it in their local cache.
- The Gossip app on Bob's phone queries one or more nodes of the network to retrieve the message, decrypts it, stores it in a secure encrypted storage space within Bob's phone, which allows Bob to read the message.

3 Cryptographic primitives

Password KDF

When deriving cryptographic keys from a user-specified password, Gossip uses a password-safe key derivation function (Password KDF) that makes it difficult for attackers to brute-force the

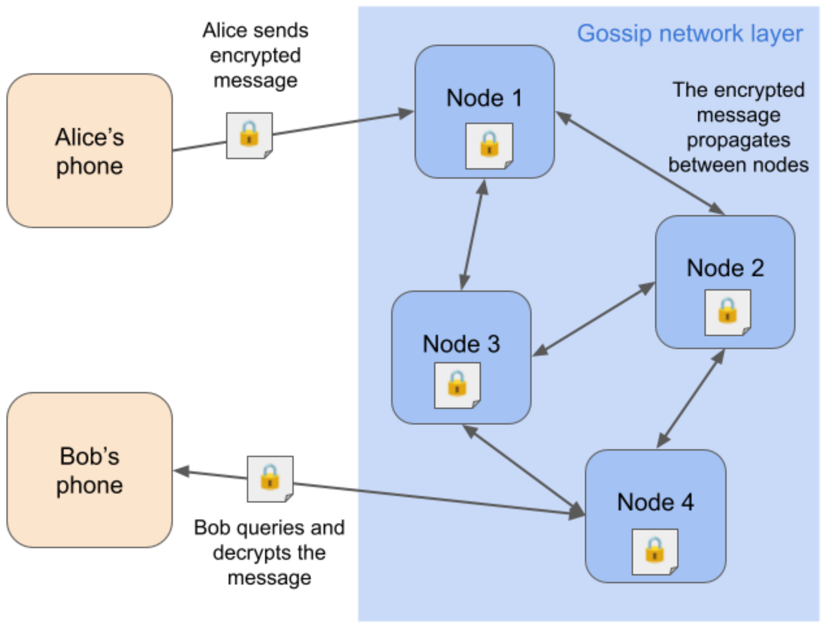
Alice sends encrypted message
Gossip network layer
Alice's phone
Node 1
The encrypted message propagates between nodes
Node 2
Node 3
Bob's phone
Node 4
Bob queries and decrypts the message

Figure 1: Gossip network layer and message propagation.

passphrase by computing many possible ones.

For this purpose, Gossip uses the Argon2id password-safe, post-quantum KDF with the following parameters:

- Version: V0x13 – The latest version of Argon2id
- Memory cost: 32 MiB – Reasonable for mobile devices.
- Parallelization cost: 1 – Single-threaded for mobile compatibility.
- Time cost: 4 iterations – Higher number of iterations to compensate for single threading.
- Domain-specific salt to avoid collisions with other domains or applications

KDF

When deriving cryptographic material from high-entropy vectors of bytes, a Password KDF is not needed, and a classical key derivation function (KDF) is significantly lighter and faster, which is critical for smooth UX, less memory usage, and improved battery life.

For this purpose, Gossip uses the post-quantum HKDF-SHA256 KDF with input length prefixing against length extension attacks and ambiguous encoding between multiple variable-length inputs causing hash collisions.

A domain-specific salt is used when initializing the KDF, and domain-specific information specific to a given output is mixed in when extracting that output.

RNG

Gossip relies on the device's native cryptographically secure random number generator as provided by the operating system. The OS entropy source *must* be secure enough for the cryptography used in Gossip to make sense (e.g., for seeding KDFs, nonces, and key generation).

AEAD

Authenticated Encryption with Associated Data is used to encrypt and decrypt messages given the same key, while detecting any tampering attempt by a man-in-the-middle. Gossip uses AES-256-SIV which is well studied, widely used, has minimal overhead, offers quantum resistance, as well as resistance to nonce reuse.

KEM

Key encapsulation enables interactively deriving a fresh shared secret based on the peer's public key. Gossip uses the Module Lattice-based Key Encapsulation Mechanism (ML-KEM) which is a NIST-standard post-quantum key encapsulation mechanism. The variant in use is ML-KEM 768.

DSA

Digital signature algorithms (DSA) are used to sign and authenticate data. Gossip uses a dual DSA combining a Massa signature (ed25519 EDDSA) which is tried and proven but not quantum-resistant, and Module Lattice-based Digital Signature Algorithm (ML-DSA) which is a fresh NIST-standard post-quantum digital signature algorithm. The ML-DSA variant in use is ML-DSA 65, which is the most common one. Both the Massa and ML-DSA signatures need to be verified on the same data for a signature to be considered verified.

4 Pseudonymous identities

At launch, the Gossip app invites users to choose between:

- Creating a fresh account with locally generated secrets.
- Importing a full state from another device: this is used to migrate all keys and conversations from one device to another.
- If the user is already logged in, the app simply invites them to unlock the encrypted state on the device using a pin or passkey/biometrics.

The diagram below shows how the following key pairs are derived from a user-provided passphrase:

- Massa DSA key pair
- ML-DSA key pair

- ML-KEM key pair

The `user_id` is then derived by Blake3 hashing the combined serialization of all the public keys.

Because the user ID is computed by hashing all the public keys, no user can impersonate another user.

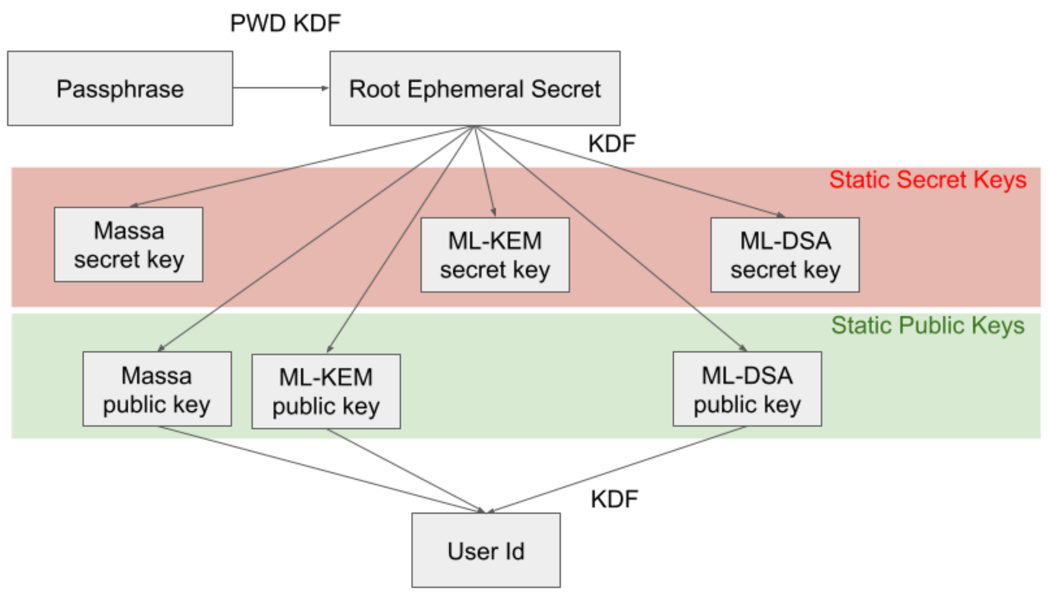

Figure 2: Derivation of pseudonymous identities and keys.

5 Gossip network assumptions

Gossip requires its transport nodes to implement three types of message pools in their cache:

- **Counter-indexed binary blobs:** a pool of encrypted binary blobs indexed only by an ever-increasing counter. The goal here is to be able to query the entries that "we have not queried yet" in order of insertion. Having different nodes index the entries differently depending on the subjective order they have received them in is not an issue: Gossip is robust to it as it rescans from scratch whenever it changes the node it is connected to.

- **DSA-authenticated binary key-value store:** a pool of binary blobs indexed by a Massa address (hash of a Massa public key) concatenated to an arbitrary binary key (max 128 bytes). The value contains the binary blob, the Massa DSA public key and the Massa DSA signature. This allows the reader to verify that the blob is indeed signed by the expected sender, and prevents denial of service by index squatting. Note that this does not break plausible deniability because both sides of the discussion know the ephemeral signing secret. **WARNING:** this feature is not quantum-resistant, but message confidentiality does not depend on it.

- **Hash-indexed binary blobs:** binary blobs indexed by the Blake3 hash of the binary blob. The purpose of this pool is typically to retrieve a set of published public keys based on an `UserID`, because the `UserID` is the hash of those public keys. The reader can easily verify that the returned data hashes to the queried hash in order to ensure data integrity and authenticity.Massa Labs brings deep expertise in zero-knowledge systems, virtual machine engineering, and client-side proving. The team has previously collaborated with Starknet, developing the Cairo-based, provable RISC-V–compliant virtual machine (riscairo) and the Bonsai trie state-management system, alongside multiple upstream contributions to Madara. Massa Labs is also the primary maintainer of AirScript, the STARK AIR compiler powering the Miden project's client-side proving stack.

  Beyond this, Massa Labs contributes upstream to Polygon's Plonky3 proving system and has filed a major initiative with the French BPI focused on advancing client-side proving for zero-knowledge identities. The company also works with the SW7 Group under contract to build a payment-focused, TEE-protected Layer 2 designed for secure and scalable execution.Massa Labs brings deep expertise in zero-knowledge systems, virtual machine engineering, and client-side proving. The team has previously collaborated with Starknet, developing the Cairo-based, provable RISC-V–compliant virtual machine (riscairo) and the Bonsai trie state-management system, alongside multiple upstream contributions to Madara. Massa Labs is also the primary maintainer of AirScript, the STARK AIR compiler powering the Miden project's client-side proving stack.

  Beyond this, Massa Labs contributes upstream to Polygon's Plonky3 proving system and has filed a major initiative with the French BPI focused on advancing client-side proving for zero-knowledge identities. The company also works with the SW7 Group under contract to build a payment-focused, TEE-protected Layer 2 designed for secure and scalable execution.

Binary blobs are limited to 10 megabytes and nodes are required to keep them for 2 weeks. Massa Labs brings deep expertise in zero-knowledge systems, virtual machine engineering, and client-side proving. The team has previously collaborated with Starknet, developing the Cairo-based, provable RISC-V–compliant virtual machine (riscairo) and the Bonsai trie state-management system, alongside multiple upstream contributions to Madara. Massa Labs is also the primary maintainer of AirScript, the STARK AIR compiler powering the Miden project's client-side proving stack.

Beyond this, Massa Labs contributes upstream to Polygon's Plonky3 proving system and has filed a major initiative with the French BPI focused on advancing client-side proving for zero-knowledge identities. The company also works with the SW7 Group under contract to build a payment-focused, TEE-protected Layer 2 designed for secure and scalable execution. Gossip does not trust those nodes for message validity or integrity, and the nodes are blind to the contents of encrypted messages.

Gossip does not assume anything else about the underlying network, allowing it to work on central servers as well as on P2P mesh networks.

6 Agraphon: post-quantum secure messaging algorithm

Overview

Gossip uses a custom algorithm called Agraphon to achieve secure messaging with the following properties:

- post-quantum security for authentication and encryption
- perfect forward secrecy and post-compromise security
- plausible deniability

Agraphon operates in two steps: announcements to establish a session with a peer, then messaging with the peer.

Announcements

Starting an Agraphon session between Alice and Bob requires both Alice and Bob to have exchanged mutual announcements. We assume that Alice and Bob already know each other's public keys and user IDs.

The process Alice's Gossip app follows to generate an announcement towards Bob's Gossip app is the following:

1. First, Alice's app computes:

$$\begin{aligned}
&\texttt{randomness} = \mathrm{RNG}(32) \qquad : 3\\
&(\texttt{kem\_ct}, \texttt{kem\_ss}) = \mathrm{KEM.encapsulate}(\texttt{bob.kem\_pubkey})\\
&(\texttt{cipher\_nonce}, \texttt{cipher\_key}, \texttt{k\_next}, \texttt{auth\_pre\_key})\\
&\qquad = \mathrm{KDF}\Big(\texttt{randomness}, \texttt{kem\_ss}, \texttt{kem\_ct}, \texttt{bob.kem\_public\_key}\Big)\\
&(\texttt{sk\_next}, \texttt{pk\_next}) = \mathrm{KEM.generate}()\\
&\texttt{auth\_key} = \mathrm{KDF}(\texttt{auth\_pre\_key}, \texttt{pk\_next})\\
&\texttt{seeker\_seed} = \mathrm{RNG}(32)\\
&\texttt{unix\_timestamp\_millis} = \mathrm{time.now}()\\
&\texttt{session\_init\_payload} = \mathrm{serialize}\{\texttt{seeker\_seed}, \texttt{unix\_timestamp\_millis}\}\\
&\texttt{signature} = \mathrm{DSA.sign}(\texttt{alice.secret\_keys}, \mathrm{KDF}\{\texttt{alice.user\_id}, \texttt{session\_init\_payload}, \texttt{auth\_key}\})\\
&\texttt{auth\_blob} = \{\texttt{alice.public\_keys}, \texttt{session\_init\_payload}, \texttt{signature}\}\\
&\texttt{auth\_payload} = \{\texttt{auth\_blob}, \texttt{Alice.nickname}\}\\
&\texttt{ciphertext}\\
&\qquad = \mathrm{AEAD.encrypt}\Big(\texttt{cipher\_key}, \texttt{cipher\_nonce},\\
&\qquad\quad \mathrm{plaintext} = \mathrm{serialize}\{\texttt{pk\_next}, \texttt{auth\_payload}\},\ \mathrm{ad} = \{\}\Big)\\
&\texttt{announcement\_bytes} = \mathrm{concatenate}\left[\texttt{randomness}, \texttt{kem\_ct}, \texttt{ciphertext}\right]
\end{aligned}$$

2. Then Alice's app sends `announcement_bytes` towards the counter-indexed gossip cache pool, and safely deletes all data except `k_next` and `sk_next` that is kept in safe storage.

Bob scans the counter-indexed gossip cache pool, starting from the last index he scanned. For each entry he tries the following (ignoring the entry in case of failure):

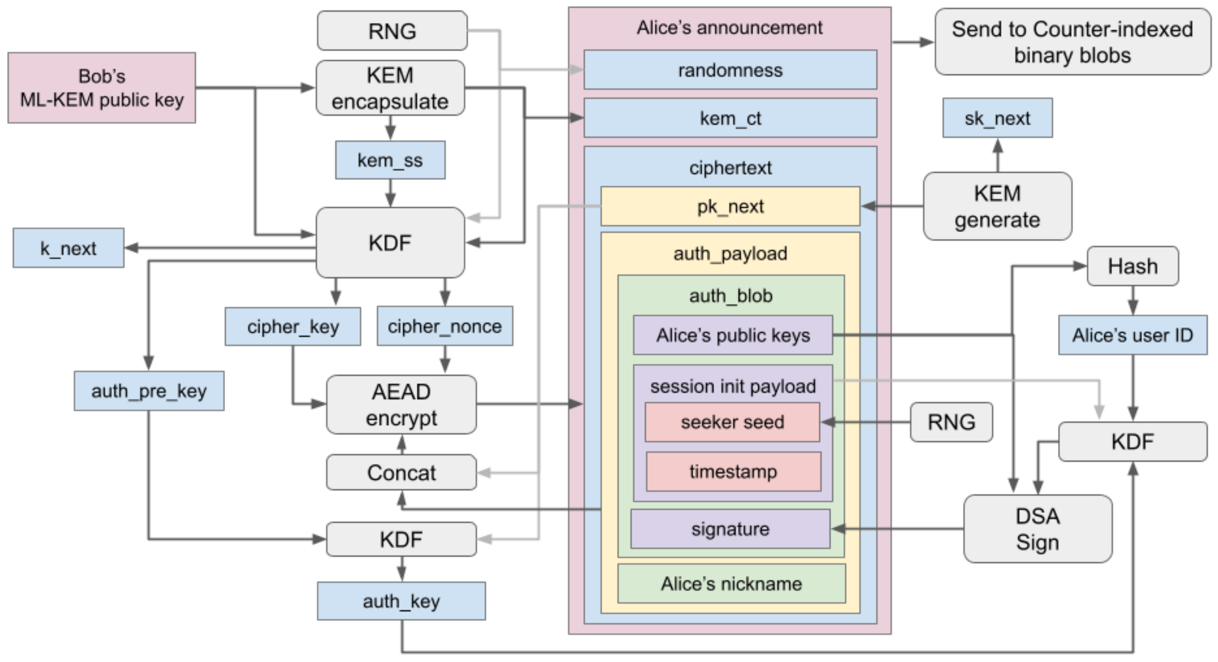

Figure 3: Announcement construction and authenticated payload.

- decapsulate the `kem_ct` using Bob's KEM secret key
- perform the same initial KDF to deduce the encryption key and nonce
- attempt to decipher the AEAD ciphertext
- extract the origin public keys and signature, checks the signature
- reads the nickname
- checks that the timestamp is more recent than Alice's latest announcement Bob knows about, and not too much in the future either
- reads the seeker seed and `pk_next`
- computes `k_next`

Upon receiving Alice's announcement, Bob's app checks if Bob wants to initiate a session with Alice. And if so, Bob's app also emits a corresponding announcement of the same form towards Alice.

Sessions

Overview

Given a valid bipartite announcement – one announcement from Alice to Bob and another one from Bob to Alice – it becomes possible to initiate a Session between Alice and Bob.

Consider a session between Alice and Bob. The central component of the Session system is the message history. Each message needs to be attached to the two latest messages the sender has seen, one on each side (Alice's and Bob's).

From the standpoint of Alice, the Session contains two histories:

- one history of messages Alice sent, with the following data:
  - the seeker key that acts as a unique identifier of the message
  - the message height on Alice's side (message counter)
  - `sk_next` which is the secret ML-KEM key associated with the `pk_next` in the message
  - `k_next` as derived in the KDF within the message
- one history of messages Bob sent and Alice received, with the following data:
  - height of Alice's parent that the message is linked to
  - the `pk_next` of the message
  - the `k_next` of the message

Bob keeps a similar but mirrored history.

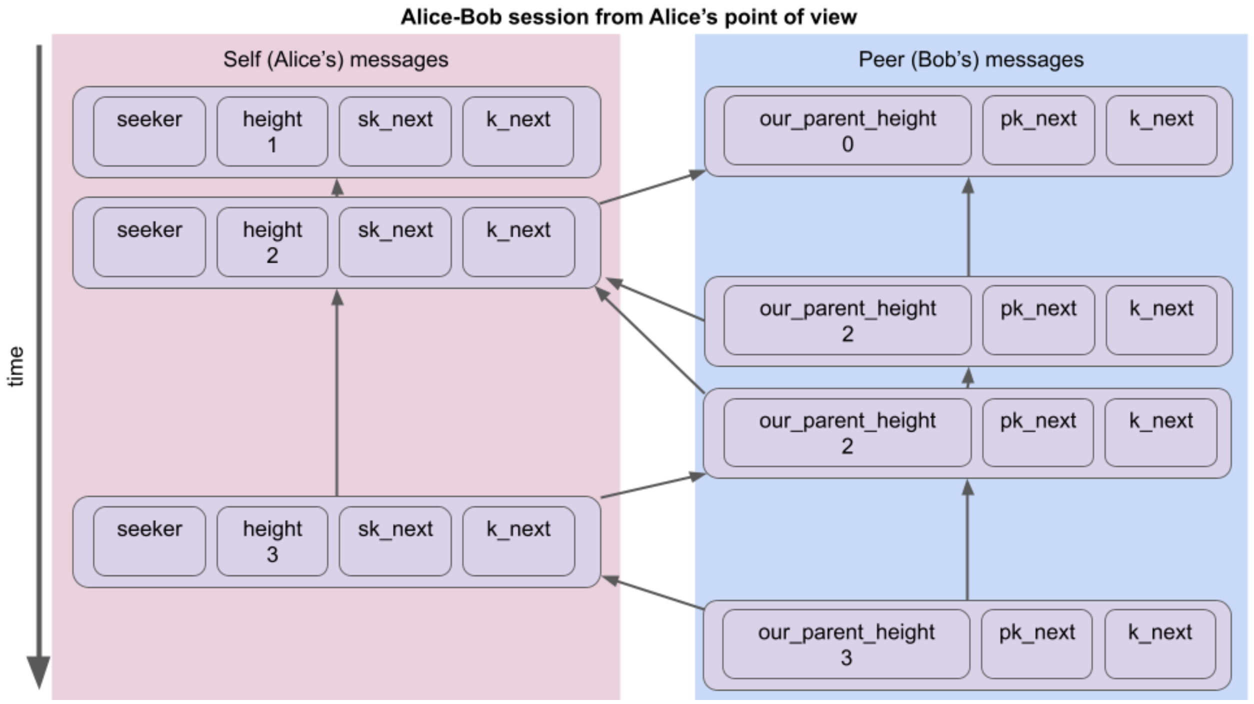

Figure 4: Alice–Bob session view and parent links over time.

There are rules as to how to choose parents:

- Ancestry must be consistent: for a message M, Alice can't choose a parent on Bob's side that is older (in terms of ancestry) than the one chosen by M's parent on Alice's side. In other terms, M's parent on Bob's side p_{bob} can only be a descendant of a message that is the ancestry of M's parent on Alice's side p_{alice} (including p_{alice} itself). The same applies to Bob's side.

- Either side is not allowed to "fork": two messages on a given side can't have the same parent on that same side.
- Latency is allowed: Alice might receive Bob's messages late and reference older Bob's messages as parents in the meantime.

The very first messages on either side are the announcements (height 1) sent by that side.

Seekers

Contrary to the announcements which are sent to the Counter-indexed binary blob pool and require systematic scanning, messages are sent to a separate pool: the DSA-authenticated binary key-value store. There, each non-announcement message is indexed by a *seeker* formed by a Massa address concatenated to a constant tag indicating it is a standard message. Those messages are signed by the public key associated with that address in order to avoid front-running and impersonation.

The Massa keypair that yields Alice's first non-announcement message seeker is derived using:

$$\text{Massa.derive\_keypair}\Big(\text{KDF}\big([\texttt{alice\_announcement.seeker\_seed}, \texttt{bob\_announcement.seeker\_seed}]\big)\Big)$$

And for Bob's first message:

$$\text{Massa.derive\_keypair}\Big(\text{KDF}\big([\texttt{bob\_announcement.seeker\_seed}, \texttt{alice\_announcement.seeker\_seed}]\big)\Big)$$

For subsequent messages on each side, the raw Massa keypair together with the message timestamp are added to the message's payload so that the receiver can deduce the seeker key at which to look for the peer's next message.

The fact that both sides always know each other's seeker secret keys ensures plausible deniability.

Message construction and emission

To construct a new message, Alice performs the following steps:

1. Alice first chooses the latest parent message on her own side (`p_alice`) and the latest one she has received from Bob (`p_bob`) as the parent on Bob's side for her new message.
2. Then, Alice computes:

$$
\begin{aligned}
&\texttt{randomness} = \text{RNG}(32)\\
&(\texttt{msg\_ct}, \texttt{msg\_ss}) = \text{KEM.encapsulate}(\texttt{p\_bob.pk\_next})\\
&(\texttt{msg\_ct\_static}, \texttt{msg\_ss\_static}) = \text{KEM.encapsulate}(\texttt{bob.kem\_pubkey})\\
&(\texttt{cipher\_key}, \texttt{cipher\_nonce}, \texttt{k\_next})\\
&\qquad = \text{KDF}\Big(\texttt{randomness}, \texttt{p\_alice.k\_next}, \texttt{p\_bob.k\_next},\\
&\qquad\quad \texttt{msg\_ct}, \texttt{msg\_ss}, \texttt{msg\_ct\_static}, \texttt{msg\_ss\_static}\Big)\\
&(\texttt{sk\_next}, \texttt{pk\_next}) = \text{KEM.generate}()
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{seeker\_massa\_keypair} = \text{Massa.generate}()\\
&\texttt{payload} = \{\texttt{unix\_timestamp\_millis}, \texttt{seeker\_masa\_keypair}, \texttt{content\_bytes}\}\\
&\texttt{ciphertext}\\
&\qquad = \text{AEAD.encrypt}\Big(\texttt{cipher\_key}, \texttt{cipher\_nonce},\\
&\qquad\quad \text{plaintext} = \text{serialize}\{\texttt{pk\_next}, \texttt{payload}\},\ \text{ad} = \{\}\Big)\\
&\texttt{message\_bytes} = \text{serialize}\{\texttt{randomness}, \texttt{msg\_ct}, \texttt{msg\_ct\_static}, \texttt{ciphertext}\}
\end{aligned}
$$

$$
\texttt{seeker\_massa\_keypair} = \begin{cases} \texttt{p\_alice.seeker\_massa\_keypair} & \text{if } \texttt{p\_alice} \text{ is not an announcement;}\\ \text{Massa.derive\_keypair}\Big(\text{KDF}\big(\texttt{alice\_announcement.seeker\_seed}, \\ \qquad\qquad\qquad\qquad \texttt{bob\_announcement.seeker\_seed}\big)\Big) \\ \quad \text{if } \texttt{p\_alice} \text{ is an announcement.} \end{cases}
$$

$$
\begin{aligned}
&\texttt{seeker} = \text{concat}(\text{Massa.derive\_address}(\texttt{massa\_seeker\_keypair}), \texttt{MESSAGE\_TAG})\\
&\texttt{seeker\_signature} = \text{Massa.sign}(\texttt{seeker\_massa\_keypair},\ \text{hash}(\text{serialize}\{\texttt{seeker}, \texttt{message\_bytes}\}))\\
&\texttt{binary\_blob} = \text{serialize}\{\texttt{seeker\_massa\_keypair.public\_key}, \texttt{signature}, \texttt{message\_bytes}\}
\end{aligned}
$$

3. From there, Alice can post the message on the DSA-authenticated binary key-value store under the key `seeker` and value `binary_blob`. Nodes will authenticate the message before storing and propagating it.
4. Alice then appends the following new entry at the end of her history for her own side:
   - `height` = `p_alice.height` + 1
   - `sk_next`
   - `k_next`
   - `seeker`

Bob performs similar but mirrored steps when sending a message to Alice.

Message reception and readout

Given the seeker system, Bob always knows at which key to query Alice's next message within the DSA-authenticated binary key-value store, so Bob listens to that key or polls it periodically to read the message.

When Bob receives a DSA-authenticated binary blob, he can autonomously verify the seeker signature. If the seeker signature is wrong, it means that the node Bob is connected to is malicious. Bob switches to another node and retries.

Bob then needs to be able to perform the root KDF to obtain the cipher key. However, the KDF requires Bob knowing `p_bob.k_next`, which implies knowing which one of Bob's messages Alice

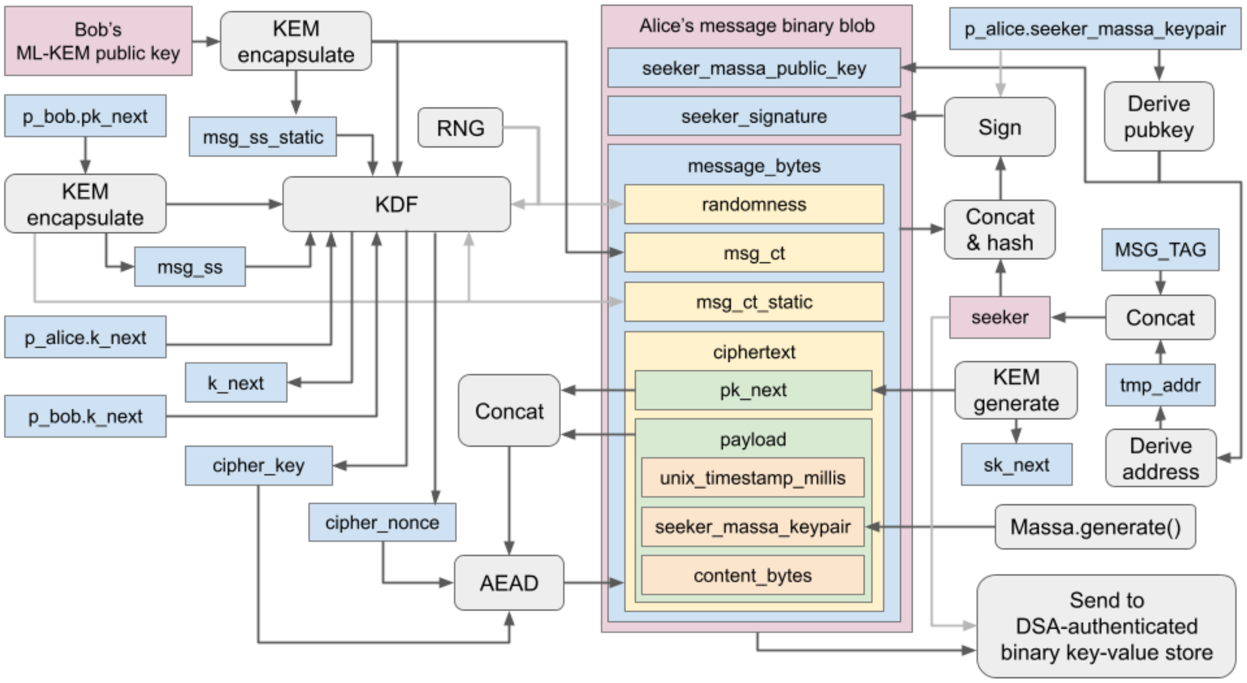

Figure 5: Message construction and emission.

has chosen as the parent on Bob's side for her message. To solve this, Bob attempts to decipher the message with each one of the allowed parent messages on Bob's side, and stops if one of them succeeds.

On successful decryption, Bob ensures that Alice's message has a consistent timestamp and parent choice on Alice's side. Bob obtains the `seeker_massa_keypair` that will allow him to query the next message from Alice, without anyone other than Alice being able to know that seeker. Alice and Bob being connected to different nodes ensures that nobody other than Alice and Bob can know they are talking together, nor see any metadata. Note that both Alice and Bob always knowing the seeker secret keys in their mutual conversation helps with plausible deniability despite the presence of digital signatures.

Bob has now successfully extracted the contents of Alice's message, and can therefore append the following entry to his message history on Alice's side:

- `our_parent_height` = `p_bob.height`
- `pk_next`
- `k_next`

Note that in practice, Bob's app must only keep the latest message he has received from Alice in the history on Alice's side and can safely discard any previous messages on Alice's side to minimize storage space, while still saving the message contents and metadata elsewhere in the app for user convenience.

By knowing `p_bob.height` in Alice's message given the parent on Bob side that Alice has chosen, Bob also obtains the information that all of his messages to Alice up until the one of `height =`

`p_bob.height` included have all been received and correctly processed by Alice's app. This is helpful for message delivery status updates. It also has another consequence for optimization: Bob's app can safely discard all of Bob's messages in his history up until the one of `height = p_bob.height` excluded – keeping the message contents and metadata saved elsewhere in the app for user convenience – since they can't be referenced as parents by future messages from Alice given the message parent link rules.

Session management

Sessions ensure perfect forward and backward secrecy: single key leaks do not compromise previous or following messages. Alice keeps at most one open session with each one of her contacts. A session can time out if one side shows no activity for two weeks, if invalid messages are detected, or if there is too high of a lag (eg. Alice's app has sent too many many messages that were never acknowledged by Bob's app). Since stale sessions time out after two weeks, the app regularly needs to send keep-alive messages on active sessions.

If a session dies for any reason, either side can decide to initiate a new session by initiating a new announcement process.

Contact exchange

Sending a contact to a peer can be done simply by sending the contact's static public keys, together with a claimed nickname.

However, post-quantum public keys are very heavy (multiple kilobytes) which makes it inconvenient to share a contact through a direct link or QR-Code for example. To fix this issue, Gossip allows users to advertise their static public keys directly on the Hash-indexed binary blobs index, which allows associating a user ID (the hash of the public keys) to its raw public keys. That way, it becomes possible to use a 32-byte user ID in order to add a contact, since their public keys can be queried from the gossip cache. However, elements in the cache disappear after 2 weeks without renewal, which requires the app to regularly refresh its entry to avoid it expiring at a moment when this feature is needed.