

SprinTelex End-to-End Encryption: Technical Whitepaper

Version 0.3 · 13 October 2026

Status: Living document. This paper describes the SprinTelex Messenger encryption architecture as implemented in the shipped product. It is updated as the product changes. Where a feature is mid-migration, the paper says so.

1. Introduction

SprinTelex is a messaging application (mobile client, application server) in which person-to-person and group conversations are end-to-end encrypted (E2EE). This paper documents the cryptographic protocols, key management, and trust assumptions behind that encryption, and states plainly what the system does and does not protect against.

Scope: 1:1 messaging, group messaging, the two implementations of the protocol (a TypeScript engine on every platform and Sige, our native Swift engine on iOS), message decryption inside the iOS notification extension, key verification, encrypted backups, registration lock, contact discovery, and view-once media. Out of scope: payments, bots, embedded mini-apps (bot chats are explicitly non-E2EE and are labeled as such in the client), stories, calls signaling beyond what is noted.

Nothing in this paper is a marketing claim. Every property stated here is grounded in the shipped product, and Section 7 lists the honest limitations. Version 0.3 corrects several claims in earlier versions that had stopped matching the product, most importantly about group messaging; Section 10 lists every correction.

2. Cryptographic notation

The formulas in Section 4 use the following notation, following the conventions of the Signal protocol specifications:

- $X \parallel Y$ is the concatenation of byte sequences X and Y .

- $DH(PK1, PK2)$ is the shared-secret byte sequence from an X25519 elliptic-curve Diffie-Hellman exchange between the key pairs identified by the public keys $PK1$ and $PK2$ (RFC 7748).
- $KDF(KM)$ is 32 bytes of output from HKDF (RFC 5869) with input key material $F || KM$, where F is 32 bytes of $0xFF$ (domain separation, as in X3DH), a zero-filled salt, and an application info string naming the derivation (`WhisperText`, `WhisperMessageKeys`, `WhisperRatchet`).
- $HMAC(K, m)$ is HMAC-SHA256 of message m under key K .
- $XSig(IK, m)$ is an XEdDSA signature on m made with a device's Curve25519 identity key IK , as specified by Signal. XEdDSA produces Ed25519-format signatures from an X25519 key pair, so one identity key serves both key agreement and signing. Signed prekeys, post-quantum key bindings, and the notification extension's requests (Section 4.8) are signed this way.
- $Sig(PK, m)$ is a standard Ed25519 signature (RFC 8032) under a dedicated key PK that is not a device identity key: the key-transparency directory and witness keys (Section 4.5) and the update-signing key (Section 3.5).
- $(CT, SS) = KEM-ENC(PK)$ is ML-KEM-1024 encapsulation (FIPS 203) under public key PK , producing ciphertext CT and a 32-byte shared secret SS ; $KEM-DEC(sk, CT)$ decapsulates with the corresponding private key.

3. Architecture overview

3.1 Components

- **Client:** Mobile application (iOS and Android) and a web companion. All encryption and decryption happen on the client. Key material never leaves the device in cleartext; the two exceptions are public keys published to the server (identity keys, signed prekeys, one-time prekeys, PQ public keys) and a PIN-encrypted copy of the identity keypair uploaded at signup to a dedicated server-side record for account recovery (Section 5.2).
 - Every client runs the protocol in a TypeScript engine. On iOS a second, native implementation, **Signe** (Section 4.7), performs the per-message work. Both engines read and write the same session records.

- On iOS, the **notification extension** (a separate process that iOS runs when a push notification arrives) is also a client component: it can decrypt an incoming message with Sige before the app opens (Section 4.8), so it holds access to the same encrypted key stores.
- **Server:** A store-and-forward relay and a public-key directory. It is not trusted with plaintext.

3.2 What the server sees

The server stores and can observe:

- Ciphertext messages (opaque blobs it cannot decrypt).
- Public key bundles: identity public keys, signed prekeys, one-time prekeys, and post-quantum public keys.
- Routing metadata: which user IDs and device IDs exchange messages, when, and approximate sizes. **The server always knows who talks to whom and when.** E2EE hides content, not metadata. Sealed sender (hiding the sender from the server) is not deployed.
- Profile directory data (usernames, display names, avatars) and, during contact discovery, truncated-SHA-256 bucket keys (see Section 6): the server no longer receives full phone numbers on this path.
- Push notification tokens, delivery bookkeeping, and the visible part of each push: the sender's display name or the group's name, looked up by the server, and a message-kind label from a closed list (Section 3.6). Pushes may also carry an end-to-end encrypted preview that the server relays without being able to read it.

3.3 What the server never sees

- Message plaintext or media plaintext.
- Notification preview text. A preview is either encrypted by the sender for the recipient or decrypted from the message itself on the recipient's device (Section 3.6).
- Private keys in cleartext. Exception: a PIN-encrypted copy of the identity private key is held in a dedicated server-side record for account recovery (Section 5.2): the server holds only the ciphertext, and recovering the key requires the user's recovery PIN (offline brute force against a weak PIN is the residual risk, which is why new

backups require an 8+ alphanumeric PIN). Signed and one-time prekey privates and sender-key chains never leave the device.

- Backup plaintext or backup encryption keys (the server holds only ciphertext; Section 5).
- Registration-lock secrets (the server holds a verifier, not the secret; Section 5.3).

3.4 Transport security

All client-to-server traffic (API, key directory, storage, realtime channel) uses HTTPS endpoints. On iOS, the OS refuses cleartext HTTP for these connections. Certificate pinning is deployed on both platforms for our API endpoints through the platform's network security configuration: the pin-set carries the live leaf SPKI as primary with the issuing intermediate and root as backups, and a mismatch fails closed. A network attacker with a fraudulently issued certificate from a trusted CA can no longer intercept client-server traffic without also defeating the pin-set; E2EE still protects message content in any case (the server never holds plaintext or private keys). Pin rotation follows a documented runbook (leaf expiry 2026-11-24; pin-set expiry 2027-10-01).

3.5 Update authenticity (OTA signing)

The Android client can receive over-the-air JavaScript bundle updates. This channel previously verified only the SHA-256 of the advertised bundle: integrity without authenticity. A compromised server or deploy machine could have pushed arbitrary JS to every install, which in an E2EE messenger is silent remote code execution.

Every bundle is now signed offline with an Ed25519 key whose private half never leaves the founder's secure machine. The update service serves the signature alongside the update check; the client verifies it against the baked-in public key before applying the bundle, and the platform's existing SHA-256 check still runs after download. Both gates must pass.

The posture is fail-closed: an unconfigured key, a missing signature, or a bad signature refuses the update with a loud log, never silently. Key rotation ships as an app update (new baked-in public key).

Trust assumption, stated plainly: update authenticity reduces to the security of the single offline signing machine (see limitation 10). iOS installs do not use this channel

(App Store distribution).

3.6 Push notifications and lock-screen previews

Push notifications (APNs/FCM) never carry message content in a form the server, Apple, or Google can read.

The visible banner is server-rendered from a closed set. The push service looks up the sender's display name (and, for groups, the group's name) itself and renders the banner text from a closed list of kinds ("New message", "Photo", "Missed voice call", and similar). Free-text title and body fields from a user caller are ignored by design, so no caller can put arbitrary words on another user's lock screen or exfiltrate message text through the push channel. This generic banner is what the platform push service sees, whatever the user's preview setting.

Readable previews are end-to-end encrypted. The real text can reach the lock screen in two ways, both decrypted only on the recipient's device:

1. **Sender-encrypted preview.** Each account holds a random 32-byte push preview key, which it shares, inside the E2EE channel, with everyone it sends messages to (its 1:1 contacts and the members of groups it posts in). A sender who holds the recipient's key encrypts a small preview (text or caption, sender name, conversation) with AES-256-GCM and attaches it to the push; the server relays it without the key. The recipient's notification extension decrypts it with the key the app stored in a keychain group shared only with the app's own extensions.
2. **Decryption of the message itself (iOS).** The notification extension can decrypt the device's own copy of the message with Sige (Section 4.8). Its text is used when no sender-encrypted preview arrived.

The user decides what the lock screen shows. The "Show previews" setting (name and message, name only, or nothing) is applied on the device by the extension. "Nothing" also replaces the sender's name with the app's name. Every step fails open to the generic banner: a missing key, a failed decryption, or the extension running out of time never suppresses a notification and never reveals more than the generic banner.

3.7 Web companion

The web client is a second client of the same protocols, not a separate product. It runs the TypeScript engine only. It enforces the phone's device-security contracts rather than bypassing them: on link it adopts the phone's auth-boundary lifecycle for the Signal store, and it presents the registration-lock PIN gate before publishing its key bundle (Section 5.3). In the browser, key material and message history are stored encrypted with AES-256-GCM under a non-extractable WebCrypto key, so the key's raw bits are never visible to page scripts. Server-side access controls remain the backstop: a web session that somehow skipped the gate still cannot read another device's keys. Backup restore on web verifies per-file MACs and content hashes exactly like the phone (Section 5.2).

4. Cryptographic protocols

4.1 Session establishment (X3DH)

New 1:1 sessions are established with X3DH, the Signal triple Diffie-Hellman handshake, extended to PQXDH (Section 4.4). Session establishment is performed by the TypeScript engine, a Signal-protocol library maintained as a fork of the upstream open-source code it derives from. A manual differential audit of the fork against the Signal specification confirmed it is faithful: DH ordering, the $0xFF \times 32$ prefix, and the HKDF info strings (`WhisperText`, `WhisperMessageKeys`, `WhisperRatchet`) match the spec. On iOS, Sige processes incoming session-establishing (PreKey) messages (Section 4.7).

Each device publishes:

- A long-term **identity key pair**: a Curve25519 key pair used for X25519 agreement and, through XEdDSA, for signatures (the Signal design).
- A **signed prekey**, rotated on a 30-day cadence, signed by the identity key (XSig). Superseded signed prekeys are pruned after 60 days, but never the one the directory currently advertises.
- A pool of **one-time prekeys** (pool size 100, replenished when fewer than 50 remain; low-pool telemetry is emitted for monitoring).

On launch the client checks the bundle the directory advertises for this device against the keys it actually holds (identity key, signed prekey, PQ key, and the whole one-time pool). If they disagree it republishes the bundle under the same identity key, so safety numbers do not change.

Our fork of the TypeScript library carries four modifications from the upstream code:

1. A normalization of the public signature-verification API, which in the upstream code returned inverted booleans (false for valid signatures). The modification makes `true` mean valid. Internal session establishment used the throwing verifier and was never affected.
2. An async fix in the session-processing path, where a missing await on the identity-trust check could leave session state half-applied.
3. **Clone-then-commit decryption.** A decryption attempt works on a copy of the session record and is written back only if it succeeds, so a failed or malicious message can never advance or corrupt a healthy session. The build refuses to compile without this modification.
4. Threading of the post-quantum shared secret into the handshake (Section 4.4).

4.2 The Double Ratchet

Once a session is established, messages are encrypted under the Double Ratchet: a symmetric-key ratchet (per-message forward secrecy: each message key is derived and then deleted, so a later key compromise does not expose earlier messages) combined with a Diffie-Hellman ratchet (post-compromise security: each DH step heals the session after a compromise). Out-of-order messages are tolerated within a 2000-message window, matching the Signal specification. Both engines enforce the same window.

Message cipher. Each message key is derived as `HKDF(chainKey, zeroSalt, 'WhisperMessageKeys')`, which yields three 32-byte values: cipher key, MAC key, and IV material. The body is encrypted with AES-256-CBC under the 32-byte cipher key and a 16-byte IV (the first 16 bytes of the third value). Authenticity is HMAC-SHA256 under the 32-byte MAC key, computed over both parties' 33-byte identity keys, a version byte, and the encoded message, then truncated to 8 bytes and appended to the envelope. The MAC is verified in constant time before any decryption is attempted.

Decryption failures and recovery. Failures follow a 3-strike budget: the first two leave the ciphertext queued and retry silently. The third archives the session (it is kept, not deleted, so messages already in flight under it can still decrypt) and sends the sender's device a **resend request** for that message. To answer such requests, each device keeps the plaintext of the messages it sent, inside its encrypted local store, for 7 days (at most the 300 most recent). It re-encrypts a message only for a device that was one of its original recipients, and it rate-limits requests per requester (excess requests queue rather than drop). A notice that a message could not be delivered appears only if no resent copy arrives within an hour. Control messages (receipts, sync, repair) are marked by the sender so their loss produces no notice. A client change rolling out with the next release stops requests for messages older than the 7-day window, which no sender could answer.

4.3 Group messaging

Today every group message is encrypted pairwise. The sender encrypts one Signal-protocol copy for each device of each member (and for each of the sender's own other devices), and the server relays each copy to its device. Group messages therefore get the full guarantees of the pairwise Double Ratchet (Section 4.2), including per-message forward secrecy and post-compromise security. The cost is one encryption per recipient device; group membership itself is managed by the server.

Sender keys: implemented, not active. The clients contain a sender-key scheme of our own design (it is not Signal's SenderKey message format), and devices already generate and distribute sender keys inside pairwise sessions, but no message is encrypted with them: a sender switches a recipient device to sender keys only after that device acknowledges the key, and devices do not send that acknowledgement. Activating the scheme is a separate, planned rollout. Its design, for completeness:

- Each sender device holds a per-(group, sender-device) **sender key**: a 32-byte chain key plus an Ed25519 signing key pair. Per send, a message key is derived as $\text{HMAC}(\text{chainKey}, 0x01)$ and the chain advances as $\text{HMAC}(\text{chainKey}, 0x02)$. The message key is split into an AES-256-CBC encryption key and an HMAC-SHA256 MAC key. The receiver verifies the MAC (constant time) before decrypting, and verifies the Ed25519 signature over the group ID, sender identity, iteration, and

ciphertext before deriving anything. Binding the group and sender into the signature defeats cross-group replay of a captured envelope.

- Sender keys are distributed inside pairwise-encrypted Signal sessions (never in the clear); only the public signing key is shared. A joiner receives keys at the current iteration and cannot read pre-join history.
- **Rotation:** when a member leaves, every participant discards all sender keys for the group and generates a fresh key on next send; keys are also replaced after seven days. A per-epoch rotation counter defeats replayed stale distributions.
- **Fallback:** devices that have not acknowledged a sender key keep receiving pairwise-encrypted messages. Sender-key envelopes carry a dedicated type tag; old clients never see them.
- Out-of-order tolerance is a 100-key window per sender stream.

Security note, applying once the scheme is active: the sender-key chain is a one-way hash ratchet with no per-message DH step, so post-compromise security is weaker than the pairwise double ratchet. A leaked chain key exposes future messages in that sender's stream until the next rotation. This is the accepted trade-off of every sender-key design, shared with the reference implementations.

4.4 Post-quantum session establishment (PQXDH)

Classical X3DH relies on elliptic-curve Diffie-Hellman, which a sufficiently powerful quantum computer would break retroactively: an adversary recording today's ciphertext could decrypt it years from now ("harvest now, decrypt later"). Signal shipped the fix as PQXDH in September 2023; to our knowledge WhatsApp has not deployed one.

SprinTelex implements the same hybrid construction:

- Each device generates an **ML-KEM-1024** keypair (NIST FIPS 203, the standardized successor to Kyber-1024) per signed-prekey generation. The public key is published alongside the prekey bundle; the private key lives in encrypted storage next to the signed-prekey private key. Keys rotate on the 30-day cadence and are pruned on the 60-day schedule. Devices that run Sige store the private key as its 64-byte ML-KEM seed.
- During session establishment, the initiator encapsulates against the responder's PQ public key. The 32-byte KEM shared secret is appended to the X3DH handshake

buffer **before** the HKDF that derives the root key ($F \parallel DH1 \parallel DH2 \parallel DH3 \parallel DH4 \parallel PQ$, where F is 32 bytes of $0xFF$ and PQ is the KEM shared secret), exactly mirroring Signal's PQXDH. The mixed root key therefore exists before any chain or message key is derived, on both sides, by construction.

- The 1568-byte KEM ciphertext travels in an app-level envelope around the PreKey message. Every PreKey message of a session carries it until the peer replies, so the responder can complete the handshake from whichever of those messages reaches it first. It never appears in the per-message path once the session is established.
- **Hybrid security:** recovering the session requires breaking both X25519 and ML-KEM-1024. If ML-KEM ever proves flawed, the classical half still stands alone.
- **Fallback:** the sender uses the PQ handshake if and only if the peer's bundle carries a PQ public key **with a valid identity signature**. Only new builds publish signed keys, so old clients never receive PQ envelopes; without a usable PQ key the handshake is byte-identical classical X3DH.
- **Identity binding:** each PQ public key is signed with the device identity key ($XSig$) over a versioned, domain-separated message binding the key. The initiator verifies the binding before encapsulating. A missing signature (pre-signing peer, the transition state) or an invalid one falls back to classical X3DH without aborting the session: the attacker gains nothing beyond pre-PQ security either way, and stripping the key only downgrades to classical X3DH, which the attacker already gets.

4.5 Key transparency (phases 1 and 2)

A signed public-key directory defends against a server that hands different keys to different people (equivocation / man-in-the-middle via the key directory).

Phase 1: an hourly server job builds a Merkle tree over every device's published bundle (identity key + signed prekey) and publishes a signed tree head (Sig , Ed25519, under the directory key). When the client fetches a contact's prekey bundle, it also fetches the Merkle inclusion proof for that device (at most one check per epoch per contact) and verifies: the head signature, the recomputed leaf, and the path to the signed root. Signatures are checked as strict Ed25519 (RFC 8032, without the more lenient ZIP-215 rules). Epoch monotonicity is enforced in encrypted storage. Hard failures (bad signature, bad proof, leaf mismatch, epoch rollback) surface through the safety-number

warning path; transient issues (network failure, missing snapshot, device published after the last snapshot) defer quietly, because staleness is not an attack.

Phase 2 closes the phase-1 trust hole (the signing-key holder) in three steps. First, chained heads: every epoch head commits to the previous epoch's head hash, forming a hash chain, and carries a second directory signature over the chained message; a history rewrite breaks the chain. Second, consistency proofs: any client or auditor can fetch the signed head chain between two epochs and verify the history only ever appended. Third, witness countersignatures: independent witnesses countersign each chained head, and clients require a quorum (currently 1) before trusting a new epoch. Publishing is atomic: the server refuses to write a head that breaks the chain or lacks witness signatures, failing loud instead of forking silently. Once a client has verified any phase-2 epoch, a v1-only head is rejected as a downgrade.

Trust assumption, stated plainly: phase 2 detects equivocation by anyone who does not hold **both** the directory signing key and a witness key (two separately stored keys; a single-key compromise no longer suffices for an undetected equivocation), and any fork is publicly visible through the head-chain endpoint. It does **not** stop the operator itself going rogue while the only enrolled witness is first-party (a second keypair held separately from the directory signing key, not decentralization). Real third-party witnesses are the operational next step; the directory's public endpoints and the client quorum logic are built for them.

4.6 Safety-number verification

Out-of-band identity verification uses 60-digit safety numbers (12 groups of 5 digits, the Signal shape), derived from both parties' identity keys, sorted so both sides compute the same digits. The client also supports QR verification: the QR payload is derived from the exact same identity-key bytes as the digit display, so the two can never diverge. Scanning verifies against the locally stored identity key for the current contact and records verification through the same persistence path as manual digit confirmation. Malformed, wrong-version, wrong-contact, and mismatched payloads fail closed.

Safety numbers are trust-on-first-use unless the user verifies them. Key transparency (Section 4.5) raises the cost of a first-use MITM but does not replace verification.

4.7 Sige: the native protocol implementation (iOS)

What it is. Sige is SprinTelex's own implementation of the Signal protocol in Swift, written clean-room from the published specifications and the wire format. It contains no libsignal code. We wrote it so that the iOS notification extension can decrypt messages (Section 4.8), which requires the ratchet in native code, and so that per-message cryptography runs natively in the app.

Why not Signal's own library. Signal's current native library cannot talk to SprinTelex's existing clients: recent releases no longer accept classical X3DH sessions and use a different message version. Adopting it would have cut off every older build and the web companion. Sige speaks exactly the wire format our clients already use.

What it does, and what stays in TypeScript.

Operation	Engine on iOS
Encrypting on an established session (including the follow-up PreKey messages of a pending session)	Sige
Decrypting regular messages and incoming PreKey messages (the responder side of X3DH/PQXDH)	Sige
Decrypting sender-key group envelopes (when the scheme is active, Section 4.3)	Sige
Starting a new session as initiator, generating and publishing keys, archiving sessions	TypeScript
Android and the web companion	TypeScript only

One store, two engines. Both engines read and write the same serialized session records, so a session can pass from one engine to the other between any two messages. If Sige meets anything it does not handle, or any error, it writes nothing and the TypeScript engine processes that message, so the fallback is always the long-standing implementation.

Primitives. Sige uses Apple's CryptoKit for X25519, SHA-512, HMAC-SHA256, HKDF, ML-KEM-1024, and Ed25519 signature verification, and the system's CommonCrypto for AES-256-CBC. XEdDSA signing is implemented in Sige from

Signal's specification; it is the only place Sige implements curve arithmetic itself, and its signatures are verified through CryptoKit's Ed25519 after the specification's key conversion.

Moving a device to Sige. When a device switches to Sige it generates a fresh, seed-based ML-KEM-1024 key and a new signed prekey and republishes its bundle. Its identity key does not change, so safety numbers stay the same.

Verification. Sige was built against the TypeScript engine as an oracle: the engine generates test vectors (complete session transcripts, with the stored record before and after every step), and Sige must reproduce every byte. A loopback harness runs sessions in both directions between the two engines, and each rollout stage passed a gate on two physical phones before the next began.

Status. Sige has been enabled for all iOS users since 2026-10-11, on app builds 1.1 (124) and later; a device switches over on its next launches after receiving the setting. The decision can be reversed per feature from the server, which returns devices to the TypeScript engine.

4.8 Decryption in the notification extension (iOS)

When a message arrives for a device whose app is closed or suspended, iOS runs the app's notification extension, a separate process with a short time budget. With this feature on, the extension decrypts the device's own copy of the message before the app opens, so the message is already on the device and the notification can show its text (subject to the preview setting, Section 3.6).

- **Scope.** The extension handles established 1:1 messages and sender-key group envelopes only. PreKey messages, media, and anything it cannot finish are left untouched for the app to process later, exactly as before.
- **Fetching without a session token.** The extension holds no login session. It fetches and acknowledges message rows with requests signed by the device's own identity key (XSig) over a domain-separated message that binds the action, the user and device IDs, a timestamp, and the row IDs. The server accepts a timestamp within five minutes and only ever returns or acknowledges undelivered rows addressed to that exact device. A replay inside the window repeats a read of ciphertext the device already holds, or an idempotent acknowledgement.

- **Shared stores, one writer at a time.** The app and the extension use the same encrypted stores (Section 5.1). Every store operation in either process runs under a cross-process file lock, which is never held across a network wait; the extension gives up if it cannot take the lock within a few seconds.
- **Crash-safe order.** Inside the lock the extension decrypts in memory, writes the message to an encrypted spool, marks the row as consumed, writes the advanced session record, and only then releases the lock and acknowledges the row. A crash at any point leaves the message deliverable. The app checks the consumed mark under the same lock before it decrypts any row, so one ciphertext is never decrypted twice and the ratchet stays consistent.
- **Locked phones.** The stores' keys are readable after the phone's first unlock since boot (Section 5.1). Before that first unlock, the extension cannot decrypt and the generic banner shows; the app processes the message after unlock.

5. Data at rest

5.1 On-device key storage

Identity keys, session state, sender keys, PQ private keys, key transparency epochs, and the 7-day send log (Section 4.2) live in encrypted on-device storage, whose AES key is held in the OS-provided keystore (Keychain on iOS, Keystore on Android). Key material is never logged.

On iOS, the protocol stores live in a container shared between the app and its notification extension (an App Group), still encrypted, and their keys sit in a keychain access group shared only with the app's own extensions. These keychain items are readable only after the device's first unlock since boot and are bound to the device: they are not synced to other devices and do not move to a new phone through device backups.

Two integrity safeguards protect the stores. On every open, an encrypted store is checked against what its file holds: if the storage library reports fewer entries than the file contains, the app sets the file aside and recovers from it, and if nothing is readable it refuses to open the store rather than overwrite it. (This guards against a defect we observed in the storage library on 2026-10-13; it rolls out with the next release.)

Separately, numeric key-store state (registration ID, key-ID counters) is repaired on open if an earlier storage migration damaged it.

5.2 Encrypted backups

Chat history backup encrypts everything on-device before upload:

- Each backup generates fresh random data keys; blobs are encrypted with AES-256-CBC + HMAC-SHA256.
- The data keys are wrapped under a KEK derived from the user's recovery PIN via **scrypt** (N=32768, r=8, p=1; 64 bytes derived, split into AES and HMAC keys).
- The server stores only ciphertext in an owner-only bucket.
- Manifest v2 additionally includes media: attachments are encrypted under stable per-backup media keys (also wrapped by the PIN-derived KEK), uploaded under opaque HMAC-SHA256 blob names keyed by the media HMAC key (naming v2; pre-fix backups used plaintext-SHA-256 names and still restore), deduplicated across chats and backup generations, capped at 100 MB per file, with per-file fail-open so one bad file cannot abort a backup. The media blob-name list in the backup meta is PIN-sealed, never plaintext. Restore verifies MAC and content hash before repointing attachments. Manifest v1 restores remain supported.

Backup security reduces to the recovery PIN: a weak PIN permits offline brute force of the backup by anyone who obtains the ciphertext. New backups require a recovery PIN of 8+ ASCII alphanumeric characters; legacy backups keep their original (often 6-digit) PINs, which the restore path continues to accept unchanged.

The identity keypair itself is backed up separately at signup: a PIN-encrypted copy (scrypt N=32768, r=8, p=1, same envelope as above) is uploaded to a dedicated server-side record, so a new device can recover the long-term identity and peers' safety numbers stay valid. The envelope's MAC is verified before decryption, so a wrong PIN and a tampered record fail the same way. Current envelopes (v3) also bind the key-derivation recipe into the MAC, so an attacker cannot relabel a record to be read with weaker parameters; older envelopes remain readable. Signed/one-time prekey privates are never backed up; they are re-minted on restore. The backup record is readable only by the account owner through server-side access controls, but a server operator or

database reader can read the ciphertext: its security likewise reduces to the recovery PIN, which is the honest reason Section 3.3 lists it as the one private-key exception.

5.3 Registration lock

Account re-registration is protected by a user-chosen secret: 6 to 20 ASCII alphanumeric characters. The server stores a verifier derived with PBKDF2-SHA256 at 210,000 iterations (per-user salt); the secret itself never leaves the device. Rate limiting (five free guesses, doubling lockouts) and a seven-day escape hatch apply unchanged. Existing 6-digit verifiers continue to work with no migration.

6. Contact discovery

Current state (as of this writing): private contact discovery via a bucket scheme. The client uploads only 20-bit truncated SHA-256 bucket keys of normalized numbers, no raw phone numbers and no full hashes, to the contact-discovery service, which returns candidate profiles *without* phone numbers (a match confirmation the client checks locally lets it confirm exact matches). The server can no longer reconstruct the address book or the social graph from this path; a per-user daily quota (100k bucket keys) blunts directory enumeration. Honest caveat: as with every contact-discovery scheme including Signal's, targeted probing ("is this one specific number registered?") remains possible; what the scheme kills is mass collection.

The legacy raw-upload endpoint is retained for old clients and marked deprecated; current builds never call it.

7. Threat model and honest limitations

Defended against:

- A passive network eavesdropper: all message content is encrypted with forward secrecy (Section 4.2), group messages included (Section 4.3).
- A future quantum adversary recording traffic today: PQXDH (Section 4.4).
- A compromised or malicious database serving wrong keys: key transparency (Section 4.5); phase 2 additionally requires both the directory key and a separately stored witness key for an undetected equivocation.

- A server, or a push provider, trying to read notification previews: previews are end-to-end encrypted or decrypted on the device (Section 3.6).
- A stolen backup blob: encryption under a PIN-derived script KEK (Section 5.2); security scales with PIN strength.
- Casual account takeover on re-registration: registration lock (Section 5.3).

Not defended against (limitations):

1. **Metadata.** The server always knows who talks to whom, when, and roughly how much. E2EE hides content, not the social graph of communication. Sealed sender is not deployed. Each push also shows the platform push service the sender's or group's name and a message-kind label.
2. **Device compromise.** If the endpoint is compromised (malware, unlocked device, forensic extraction past the OS keychain), encryption cannot help. This includes the last 7 days of sent messages kept for resends (Section 4.2), which sit in the same encrypted store as the chat history. On iOS the notification extension can read the same stores as the app, so it is part of the same trust boundary.
3. **Key-transparency trust.** Phase 2 needs both the directory signing key and a separately stored witness key for an undetected equivocation, and forks are publicly visible. It still does not stop the operator itself while the only witness is first-party; third-party witnesses are the next step.
4. **Contact discovery (legacy path).** Old clients still upload raw phone numbers via the deprecated endpoint; current builds use the bucket scheme (Section 6).
5. **Targeted number probing.** Even after the rebuild, checking whether one specific number is registered remains possible, as in every deployed contact-discovery system.
6. **Unverified safety numbers.** Without out-of-band verification, identity is trust-on-first-use.
7. **View-once media.** Forwarding, copying, saving, and OS sharing are blocked in-app, and screenshot/screen-recording detection notifies the sender, but no software can prevent a recipient from photographing a screen with another device. View-once is a speed bump, not a guarantee.
8. **Lock-screen previews.** With previews set to "name and message", decrypted text appears on the lock screen, where anyone holding the phone can read it; "name

only" and "nothing" hide it. A user's push preview key is held by everyone that user has sent messages to (including members of groups they post in), and it is not rotated today: an operator who recorded push payloads and colluded with any one of those people could read the previews of messages sent to that user. Previews produced by the extension's own decryption (Section 4.8) do not depend on this key.

9. **Group cost.** Group messages are pairwise today (Section 4.3), which keeps the full Double Ratchet guarantees but costs one encryption per recipient device. Once sender keys are activated, their weaker post-compromise security (Section 4.3) will apply to groups.
10. **OTA signing-key custody.** Update authenticity reduces to the security of the single offline signing machine (Section 3.5). Compromise of that machine is compromise of the update channel for every Android install. Key rotation ships as an app update.
11. **Two implementations.** Running two protocol engines on iOS adds the risk that they disagree on some input. It is mitigated by Sige's byte-exact test vectors from the TypeScript engine, the cross-engine loopback tests, and the rule that Sige writes nothing on any error and hands the message to the TypeScript engine (Section 4.7).

8. Comparison with WhatsApp and Signal

Stated only where we have public evidence; where evidence is thin, we say so.

Property	SprinTelex	Signal	WhatsApp
1:1 E2EE (X3DH + Double Ratchet)	Yes	Yes	Yes
Group messages	Pairwise per recipient device; sender keys implemented, not yet active	Sender keys	Sender keys (documented)
Post-quantum handshake	Yes (ML-KEM-1024 hybrid)	Yes (PQXDH, 2023)	Not publicly deployed (to our knowledge)

Property	SprinTelex	Signal	WhatsApp
Key transparency	Phase 2: chained heads, consistency proofs, witness quorum (one first-party witness so far)	Yes (long-standing)	Yes (deployed AKD; still more mature: their auditors are third-party, ours are not yet)
Private contact discovery	Yes (bucket scheme; legacy raw-upload path retained for old clients)	Yes (SGX-based)	No (address-book upload)
Safety-number QR verification	Yes	Yes	Yes
Decrypting messages in the iOS notification extension	Yes, for established sessions (Sige)	Yes	Not documented
Native protocol implementation	Sige (Swift, iOS); TypeScript elsewhere	libsignal (Rust, open source)	Not public

We do not claim superiority where we have not earned it: WhatsApp's key transparency deployment is more mature than ours, and Signal's contact discovery remains the reference design.

9. References

- RFC 7748, "Elliptic Curves for Security" (X25519, X448). <https://www.rfc-editor.org/rfc/rfc7748.html>
- RFC 8032, "Edwards-Curve Digital Signature Algorithm (EdDSA)". <https://www.rfc-editor.org/rfc/rfc8032.html>
- RFC 5869, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)". <https://www.rfc-editor.org/rfc/rfc5869.html>
- FIPS 203, "Module-Lattice-Based Key-Encapsulation Mechanism Standard" (ML-KEM). <https://csrc.nist.gov/pubs/fips/203/final>
- Signal, "The X3DH Key Agreement Protocol", Revision 1, 2016-11-04. <https://signal.org/docs/specifications/x3dh/>

- Signal, "The XEdDSA and VEdDSA Signature Schemes", Revision 1, 2016-10-20.
<https://signal.org/docs/specifications/xeddsa/>
- Signal, "The Double Ratchet Algorithm", Revision 4, 2025-11-04.
<https://signal.org/docs/specifications/doubleratchet/>
- Signal, "The PQXDH Key Agreement Protocol", Revision 3, 2023-05-24.
<https://signal.org/docs/specifications/pqxdh/>
- Signal, "The Sesame Algorithm: Session Management for Asynchronous Message Encryption". <https://signal.org/docs/specifications/sesame/>
- WhatsApp, "WhatsApp Encryption Overview" (technical whitepaper).
<https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf>

10. Changelog

- **2026-10-13, v0.3:** New sections for Sige, our native Swift implementation of the protocol on iOS (Section 4.7), and for message decryption in the iOS notification extension, including its signed, token-free row access, cross-process locking, and crash-safe write order (Section 4.8). Section 3.6 rewritten: previews are end-to-end encrypted (sender-encrypted, or decrypted from the message itself) and governed by the user's preview setting; new limitation 8 covers lock-screen previews and the unrotated push preview key. Section 4.2 documents resend requests and the 7-day sender-side send log (limitation 2 updated). Section 5.1 covers the shared iOS stores, their keychain accessibility, and the store-integrity safeguards. Section 5.2 documents the v3 identity-backup envelope. New limitation 11 (two implementations). Corrections to earlier versions:
 - **Group messaging (Section 4.3, comparison table, limitation 9).** Earlier versions said group messages use sender keys. They never have in the field: a defect in the acknowledgement step kept the scheme dormant from its first release, and it now stays off deliberately until a planned rollout. Every group message has been encrypted pairwise per recipient device. This is a correction of a feature claim, not a weakening: pairwise encryption carries stronger post-compromise security.
 - **Signature schemes (Sections 2, 4.1, 4.4).** Signatures made with device identity keys (signed prekeys, PQ key bindings) are XEdDSA, not plain Ed25519

as the notation previously said. Plain Ed25519 is used only by the dedicated directory, witness, and update-signing keys.

- **Key-transparency verification (Section 4.5).** From 2026-10-02 to 2026-10-10 the client verified tree-head signatures with the XEdDSA check instead of Ed25519, so every check failed and raised false safety-number warnings. It failed closed (no false acceptance). Fixed in builds from 1.1 (123).
- **Signed-prekey rotation (Section 4.1).** The 30-day rotation did not take effect on builds before 1.1 (123), and a separate migration defect could leave the directory advertising keys a device no longer held. Builds from 123 rotate correctly, never prune the advertised key, and check and repair the published bundle on launch.
- **PQ ciphertext (Section 4.4).** Earlier versions said the KEM ciphertext rides only on the initial PreKey message. Before builds from 1.1 (123), a follow-up PreKey message of the same pending session did not carry it, so a responder that saw one first could not decrypt it. Every pending PreKey message now carries it. The "one-time backfill" for older installations was removed.
- **Library modifications (Section 4.1).** Four, not three: clone-then-commit decryption was missing from the list.
- **2026-10-05, v0.2 (rev 2):** Removed internal implementation details (file paths, table names, function names, library names) to match whitepaper conventions; no cryptographic claims changed.
- **2026-10-05, v0.2:** Update authenticity (Section 3.5): Android OTA bundles are now Ed25519-signed offline, with fail-closed verification against a baked-in public key; the single-machine custody assumption is stated as limitation 9. Push notifications carry no message content (Section 3.6): banner text is server-rendered from a closed kind enum. Web companion security model documented (Section 3.7): the web client enforces the phone's auth-boundary and registration-lock contracts, with server-side access controls as backstop.
- **2026-10-03:** Corrected three stale claims: certificate pinning is deployed on both platforms (Section 3.4); the bucket-scheme contact discovery has shipped (Section 6, limitation 4, comparison table); the "private keys never leave the device" claim now honestly discloses the PIN-encrypted identity-key copy held server-side

(Sections 3.1, 3.3, 5.2), and notes the new 8+ alphanumeric recovery-PIN floor for new backups plus the v2 HMAC media-blob naming.

- **2026-10-02, v0.1:** Initial draft. Covers X3DH/Double Ratchet, group sender keys, PQXDH, key transparency phase 1, safety numbers + QR, encrypted backups (manifest v2), registration lock, contact discovery current state and in-progress rebuild, threat model.
- **2026-10-02:** Added cryptographic notation (Section 2), transport security (Section 3.4), explicit pairwise message cipher (Section 4.2), and references (Section 9).
- **2026-10-02:** Key transparency phase 2 (Section 4.5): chained heads, consistency proofs, witness quorum; trust assumption and limitations updated (one first-party witness so far).
- **2026-10-02:** PQ public keys are now identity-signed (Section 4.4); the "unsigned PQ key" limitation is closed.
- Future entries go here, newest first. Update this paper alongside the product change it describes.