

OXU

OXU Note

Cryptographic Architecture for End-to-End Encrypted Notes

Client-side encryption, post-quantum key encapsulation,
secure synchronization, recovery, and concealed spaces

Cryptographic Architecture Specification

Version 1.1

OXU Note Security and Engineering

September 2026

Abstract

This document specifies the cryptographic architecture of OXU Note for iOS and macOS. Clients shall encrypt note content and selected metadata before synchronization, generating an independent 256-bit data-encryption key per note and attachment object, protecting each such key through ML-KEM-768 key encapsulation followed by AES-256-GCM wrapping, storing local structured data in SQLCipher databases, and maintaining a per-space encrypted FTS5 index for local search. Encrypted note bodies shall be divided into authenticated chunks, with BLAKE3 hashes binding each ciphertext chunk to its index and forming a Merkle root. The architecture further specifies user-seed recovery, post-quantum protected QR device transfer, and PIN-selected concealed spaces.

The architecture is end-to-end encrypted with respect to note plaintext, but it is deliberately not metadata-free. OXU Cloud receives account and device context, space and object identifiers, versions, timing, ciphertext lengths, key encapsulations, the vault salt, and recovery ciphertexts. For objects written by the current protocol, the parent identifier travels only inside the encrypted metadata and each object key is wrapped directly under the space root key, so stored service records do not reproduce the notebook, note, and attachment tree; request order, legacy records, and other channels still permit structural inference. Section 9 enumerates this exposure in full, because a design that hides its metadata surface cannot be reasoned about. Stealth Mode conceals space selection in the normal locked-app interface; it does not conceal the existence or number of spaces from the service, from device forensics, from live memory capture, or from a compromised endpoint.

Contents

1	Introduction	1
1.1	Specified architectural properties	1
1.2	What this specification does not claim	1
1.3	Scope, conformance, and normative language	2
1.4	Document organization	3
2	Threat Model, Goals, and Non-Goals	3
2.1	Adversaries	3
2.2	Security goals	4
2.3	Non-goals and environmental assumptions	4
3	Cryptographic Primitives	5
3.1	Authenticated encryption	6
3.2	Post-quantum key encapsulation	6
3.3	Hashing, Merkle construction, and signatures	7
3.4	Key derivation	7
4	System Architecture and Trust Boundaries	8
4.1	Components	8
4.2	Client boundary	8
4.3	Service boundary	9
4.4	Asset placement	10
5	Identities, PINs, and Key Hierarchy	10
5.1	Identity authentication is not content encryption	10
5.2	Root seed and user keys	11
5.3	Default PIN	11
5.4	Vault seed, salt, and per-space outputs	11
5.5	Object key hierarchy	12

5.6	Key storage and lifetime	13
6	Local Storage and the Note Data Model	14
6.1	Storage domains	14
6.2	Logical object model	15
6.3	Search index and convenience state	16
7	Note Lifecycle	16
7.1	Creation and first persistence	16
7.2	Open, edit, and autosave	17
7.3	Backgrounding and lock	17
7.4	Deletion	18
8	Content Encryption and Integrity	18
8.1	Chunk encryption	18
8.2	Indexed leaf hashes and Merkle root	18
8.3	Signing objects	19
8.4	What authentication covers	19
9	Cloud Synchronization and Metadata	19
9.1	Synchronization transaction	19
9.2	Parent privacy	20
9.3	Service-visible information	22
9.4	Conflict, rollback, and availability	24
10	Authentication and Signatures	25
10.1	Three distinct identity layers	25
10.2	Canonicalization and domain separation	25
10.3	Authorization boundary	25
11	Recovery and Device Transfer	25
11.1	User Recovery Key	25
11.2	Per-space recovery object	26
11.3	Post-quantum protected QR transfer	26
12	Stealth Mode and Concealed Spaces	28
12.1	PIN-directed selection	28
12.2	What the service can infer	29
12.3	What a device examiner can infer	29
12.4	Explicitly not offered	30
13	Security Analysis	30
13.1	Scenario analysis	30
13.2	Key compromise matrix	32
13.3	Forward secrecy and post-compromise security	32
13.4	Cryptographic agility	33
14	Implementation and Operational Boundaries	33
14.1	Logging	33
14.2	Memory and temporary data	33
14.3	Platform parity	34
14.4	Dependency provenance	34
14.5	Backend and build boundaries	35

15 Related Design Families	35
16 Conclusions and Priorities	36
A Algorithm and Parameter Registry	38
B KDF and Label Registry	38
C Protocol Field Inventory	39
D Normative Pseudocode	40
D.1 Encrypt a note object	40
D.2 Recover a note object	40
D.3 Resolve the parent of a pulled object	41

1 Introduction

Notes routinely contain information whose sensitivity exceeds its apparent size: credentials, personal reflections, medical or legal details, business plans, and fragments copied from other protected systems. Encrypting the disk or the connection to a cloud service addresses only part of the risk. A cloud-synchronized note system must decide where plaintext is created, which keys exist on each device, which metadata remains visible to the service, how new devices recover access, and what happens when a device or recovery secret is compromised.

OXU Note is a native Apple-platform note application built around client-side cryptography. Its editor produces structured note documents. Before cloud upload, the client encrypts note bytes and user-facing metadata; the service coordinates manifests and opaque chunks. The cryptographic construction combines conventional authenticated encryption and signatures with NIST-standardized post-quantum key encapsulation. Local databases and the local search index are encrypted with SQLCipher.

This specification uses *end-to-end encrypted* in a precise, scoped sense: ordinary server-side storage does not receive the plaintext note body, title, notebook name, or attachment content, and the content keys needed to decrypt those fields are produced or recovered on clients. The term does not mean the server observes no metadata, cannot deny service, cannot replay old records, or cannot attack a compromised client.

1.1 Specified architectural properties

Conforming clients shall use shared-core cryptographic interfaces that provide the following architectural properties. A conforming client shall implement all of them:

1. a separate random 32-byte DEK for each new note and attachment object;
2. AES-256-GCM encryption of note chunks and selected metadata;
3. ML-KEM-768 encapsulation of a shared secret that wraps each object DEK;
4. a flat object-key layout in which every notebook, note, and attachment DEK is wrapped directly under the space root DEK (in addition to its ML-KEM-768 encapsulation), so that no stored key wrap encodes the object tree;
5. a parent identifier carried only inside the encrypted object metadata, with the top-level parent field of every synchronized manifest set to a fixed sentinel value;
6. BLAKE3 hashing of encrypted chunks with explicit little-endian chunk index binding and a deterministic Merkle-tree construction;
7. SQLCipher-protected per-space metadata, key-material, and FTS5 databases;
8. a user-seed recovery model and an ephemeral ML-KEM-768 QR transfer channel for installing the same seed on a new device; and
9. a shared unlock surface that can select either a default space or a PIN-derived concealed space.

1.2 What this specification does not claim

This specification does not claim:

- that OXU Cloud cannot observe which space an object belongs to, object counts, sizes, versions, timing, devices, or the existence and number of multiple spaces;
- that the object tree cannot be inferred from request order, stable identifiers, or records written before the current protocol — encrypting the parent identifier removes it from stored fields only;

- decoy spaces, decoy objects, padding, or cover traffic;
- protection after endpoint compromise or while an attacker can observe an unlocked editor or live process memory;
- post-quantum authentication—Ed25519 and OIDC remain conventional;
- hidden-volume, steganographic, or service-operator deniability;
- cryptographic erasure of server backups or old versions;
- forward secrecy for content encrypted under a recoverable seed; or
- that conformance to this specification has been demonstrated for any particular build.

1.3 Scope, conformance, and normative language

This specification applies to OXU Note for iOS version 4.0.0 and OXU Note for macOS version 1.0.0, and to the shared cryptographic core and cryptographic library those clients depend on.

The key words *shall*, *shall not*, *should*, *should not*, and *may* are to be interpreted as requirement levels. *Shall* denotes an absolute requirement of this specification; *should* denotes a recommendation that may be departed from only with a documented rationale; *may* denotes an optional feature. A client that fails any *shall* requirement does not conform, regardless of whether the failure is observable in ordinary use.

Several requirements in this document — in particular key-derivation domain separation, associated-data binding, integrity enforcement on download, crash-consistent publication, and server-side session expiry — can be satisfied or violated without any change to the user-visible behavior of the application. They are stated explicitly for that reason. Each is a property of the construction rather than of the interface, and an implementation that omits one is outwardly indistinguishable from one that does not.

Where this specification and any product or technical document disagree, this specification governs.

Changes in version 1.1. Version 1.1 aligns OXU Note with the metadata-privacy design of the OXU Drive vault. The parent-child relationship of notebooks, notes, and attachments moves from a plaintext manifest field into the encrypted object metadata, and object DEKs are wrapped directly under the space root DEK instead of along a notebook/note/attachment key chain (Sections 5.5 and 9.2). Version 1.1 makes no decoy or deniability claim; Section 12.4 lists what is explicitly not offered. Throughout this document, an object *written by the current protocol* is one that was created, or has since been rewrapped, under these rules: its DEK is wrapped under the space root DEK, and its latest manifest carries the sentinel parent field with the real parent inside the encrypted metadata. Any other object is a *legacy object*.

The mechanisms introduced in version 1.1 apply to objects written by clients that implement them. Objects written by a covered release that predates a mechanism are legacy objects, and the legacy treatment of Sections 5.5 and 9.2 applies to them. Version 1.1 supersedes version 1.0 in full.

Table 1: Scope of this specification.

Component	Covered version	Treatment
iOS client	4.0.0	Normative for every requirement in this document
macOS client	1.0.0	Normative for every requirement in this document. Platform parity is itself a requirement; see table 14
Shared cryptographic core	As pinned by the covered clients	Normative for the primitive constructions, key derivations, and integrity checks specified here
Cryptographic library	As pinned by the shared core	Normative for primitive parameters and serialization
Backend services	Not versioned by this document	This specification states the requirements the client places on the service. Service-side conformance, including retention, session expiry, and deletion, is out of scope here and shall be specified separately

1.4 Document organization

Section 2 defines the adversaries and non-goals. Section 3 describes primitives. Sections 4 and 5 cover trust boundaries and the key hierarchy. Sections 6 to 8 trace local storage and note encryption. Sections 9 and 10 explain synchronization, metadata, and signatures. Sections 11 and 12 analyze recovery and concealed spaces. Sections 13 and 14 set out residual risk and operational requirements. The appendices carry the normative algorithm and derivation registries, the protocol field inventory, the normative pseudocode, and the conformance requirements.

2 Threat Model, Goals, and Non-Goals

A security claim is meaningful only relative to an adversary. OXU Note spans an editor, local encrypted storage, platform key storage, network authentication, service APIs, databases, and object storage. A control in one layer does not automatically protect another.

2.1 Adversaries

Network observer.

Can record, delay, replay, or block traffic but cannot break correctly validated TLS. The observer sees endpoints, timing, and volume.

Honest-but-curious service.

Executes the documented protocol and stores records correctly while trying to learn from plaintext fields, identifiers, timing, sizes, and correlations.

Service breach or malicious storage.

Can read and modify metadata databases, recovery ciphertexts, manifests, and encrypted chunk objects. It may omit, reorder, replace, or replay records.

Locked-device examiner.

Obtains an application container or filesystem image while the device and app are locked. Keychain accessibility and SQLCipher key availability determine what can be recovered.

Live-memory attacker.

Can inspect a running or recently backgrounded process. This adversary may find plaintext editor state, DEKs, database keys, seeds, or a Stealth PIN.

Compromised endpoint.

Controls the OS, app binary, keyboard, screen, or accessibility layer while the user works. Client-side cryptography cannot hide plaintext from the client that must display it.

Recovery-key holder.

Possesses the exported User Recovery Key, which is an encoding of the `user_seed`. With authenticated service access and synchronized recovery objects, this actor may reconstruct broad account cryptographic state.

Coercive UI observer.

Can ask the user to unlock the normal application interface but does not have server records, a forensic image, or live-memory tooling. This is the narrow adversary relevant to Stealth Mode.

Future quantum adversary.

Records current ciphertext and later gains a cryptographically relevant quantum computer. ML-KEM-768 addresses KEM confidentiality in this scenario; Ed25519 and OIDC authentication do not become post-quantum merely because the DEK wrapper uses ML-KEM.

2.2 Security goals

The architecture targets:

- confidentiality of note bodies, note titles, notebook names, and attachment bytes against routine service storage;
- authenticated encryption of each content chunk under an object DEK;
- isolation between object DEKs so compromise of one notebook, note, or attachment DEK does not decrypt any other object written by the current protocol;
- encryption of the parent–child relationships of objects written by the current protocol, so that stored service records do not reproduce the object tree;
- client-controlled recovery of object DEKs through a per-space encapsulation key;
- encrypted local persistent storage for note metadata and full-text content;
- corruption and reordering detection through indexed chunk hashes and a Merkle root, when verification is actually enforced;
- authenticated service requests through OIDC session authentication plus Ed25519 signing objects; and
- narrow UI concealment of space selection.

2.3 Non-goals and environmental assumptions

The architecture does not aim to hide:

- account, organization, workspace, device, address, or session data;
- connection time, request frequency, total ciphertext volume, object count, or ciphertext size;
- which space an object belongs to, or the number of spaces;
- stable object identifiers, version chains, or structure inferable from request order and legacy objects;
- the presence of recovery ciphertexts or space-creation operations; or

- content after a user exports, copies, screenshots, or opens it outside the app-controlled storage boundary.

The design assumes a trustworthy client distribution channel, a correct OS cryptographic random source, correct primitive implementations, non-compromised Apple platform security, validated TLS, and server authorization that rejects cross-account requests. Those assumptions are independent of content encryption and must be tested separately.

Table 2: Threat-to-control summary.

Threat	Primary control	Residual
Network interception	TLS plus ciphertext payloads	Endpoint, timing, and volume remain visible
Object-storage disclosure	Per-object AES-256-GCM and wrapped DEKs	Size, count, hashes, and access patterns remain
Metadata database disclosure	Encrypted user-facing metadata, including the parent identifier; sentinel parent field; flat root wrapping	Space and object IDs, versions, status, timing, key records, and legacy parent edges remain
Locked-device image	Keychain and SQLCipher	UserDefaults metadata, platform backups, and key accessibility require platform validation
Chunk modification	GCM tags; indexed BLAKE3 Merkle construction	Download-time verification shall fail closed; enforcement is a conformance requirement
Quantum ciphertext archive	ML-KEM-768 DEK encapsulation	Ed25519/OIDC are not post-quantum; endpoint capture still wins
Coercive normal-UI unlock	PIN-selected concealed space	No server, forensic, memory, or traffic deniability

3 Cryptographic Primitives

The application-facing crypto API is provided through SourceCore and Swift bindings to libsilver. The cryptographic library uses Rust libraries for AES-GCM, hashing, HKDF, Argon2id, Ed25519, ML-KEM, and OS randomness. Availability in the crypto library does not prove that a primitive is called by a particular product flow.

Table 3: Cryptographic mechanisms and required use.

Primitive	Parameters	Role	Status
AES-256-GCM	32-byte key, 12-byte random nonce, 16-byte tag	Note chunks, metadata, slot records, DEK wraps, recovery and transfer payloads	Required. Note-content calls presently pass empty AAD; see section 8
ML-KEM-768	1184-byte public key, 2400-byte private key, 1088-byte ciphertext, 32-byte shared secret	Wrap object and root DEKs; QR transfer shared secret	Active in app call sites
Ed25519	32-byte seed/public key, 64-byte signature	Request signing and Merkle-root signing objects	Active generation; content verification enforcement limited
HKDF-SHA512	variable IKM, optional salt/info	Seed expansion, DB keys, signing keys, PIN-derived slot outputs	Required
Argon2id	64 MiB, three passes, parallelism one in the cryptographic library	Password KDF capability	Available; not called by the slot path
SHA-256	32-byte digest	Identifiers, PIN hashes, key versions, prehash before Ed25519	Active
SHA-512	64-byte digest	HKDF HMAC backend	Active
BLAKE3	32-byte digest	Ciphertext chunk hashes and Merkle parents	Active
SQLCipher	Raw 32-byte app key passed with PRAGMA key; WAL enabled	Encrypted local relational and FTS storage	Active; exact linked build parameters shall be pinned and recorded
OS CSPRNG	platform OsRng in the cryptographic library	Seeds, DEKs, nonces, salts, KEM randomness	Required for all key, nonce, and salt generation

3.1 Authenticated encryption

The cryptographic library shall serialize an AES-GCM ciphertext as:

$$C = N_{96} \parallel \text{GCM.Enc}_K(N, P, A),$$

where the returned GCM body contains ciphertext followed by the 128-bit tag. Thus a plaintext of length n produces $n + 28$ bytes. Nonces are generated from the OS random source for each call.

GCM authenticates both ciphertext and optional associated data [2]. Note-content calls in the current implementation use `Crypto.encrypt(chunkData, key: dek)` without an AAD argument. The effective A is empty. Object identity, version, manifest ID, and chunk index are therefore not cryptographically bound through GCM AAD. The index is instead incorporated into a separate BLAKE3 hash. This is a meaningful limitation: a future revision should use a canonical, versioned AAD structure and reject any context mismatch. Remediation does not require a new primitive. The cryptographic library already exports AAD-accepting encryption and decryption entry points alongside the two-argument functions the note path calls; only the call sites and a canonical context encoding are missing.

3.2 Post-quantum key encapsulation

ML-KEM is a KEM standardized by NIST in FIPS 203 [3]. It establishes a 32-byte shared secret and does not directly encrypt an arbitrary note or DEK. OXU Note uses the composition:

$$(ct_{\text{kem}}, ss) \leftarrow \text{MLKEM.Encaps}(pk_{\text{space}})$$

$$E_{\text{DEK}} \leftarrow \text{AESGCM.Enc}_{ss}(\text{DEK}, A = \epsilon).$$

The manifest carries ct_{kem} , E_{DEK} , the public key, and a public-key version. On decryption, the client regenerates or loads the matching private key, decapsulates ss , and decrypts the DEK.

The 32-byte shared secret is used directly as the AES-256-GCM key; no additional key-derivation step is applied between decapsulation and wrapping. ML-KEM produces its shared secret through an internal hash-based derivation, so the value is already uniform and of exactly the required length, and direct use is a defensible construction. It nevertheless means that no protocol version, space identifier, or object identifier is bound into the wrapping key, and the same absence of context binding recurs in the empty GCM associated data discussed above. A future revision that introduces a canonical AAD structure should bind the same context into a derivation over the shared secret so that the two mechanisms agree.

This protects the encapsulated DEK against a future quantum adversary under the assumptions of ML-KEM and the composition. It does not make Ed25519 signatures, OIDC sessions, server authorization, or the endpoint post-quantum. NIST guidance for KEM use emphasizes treating encapsulation as key establishment rather than direct bulk encryption [4].

3.3 Hashing, Merkle construction, and signatures

For encrypted chunk C_i , the leaf hash is:

$$h_i = \text{BLAKE3}(\text{LE32}(i) \parallel C_i).$$

At each tree level, adjacent nodes are concatenated and hashed:

$$p_j = \text{BLAKE3}(h_{2j} \parallel h_{2j+1}).$$

If the level has an odd number of nodes, the final node is duplicated. A single leaf is itself the root. This construction binds ordering to the 32-bit little-endian index and detects ciphertext changes when the computed root is compared with an independently authenticated expected root.

Signing objects prehash their JSON message with SHA-256 and then apply Ed25519:

$$\sigma = \text{Ed25519}.\text{Sign}_{sk}(\text{SHA256}(\text{UTF8}(m))).$$

This is an application-specific prehash construction, not the Ed25519ph mode defined in RFC 8032 [6]. Verification must reproduce the exact JSON byte string; semantically equivalent reserialization is not necessarily the same message.

3.4 Key derivation

HKDF is an extract-and-expand KDF [5]. Every derived output in this architecture shall carry a distinct, versioned info label. Two outputs shall not be produced from the same input keying material, salt, and info and distinguished only by requested length: HKDF expansion is deterministic, so a shorter output derived that way is a prefix of the longer one and the two are not independent keys. Appendix B is the normative registry of labels; a derivation that does not appear there does not conform.

This requirement is directed at a specific failure. The `slot_key` and `encap_seed` outputs described in Section 5 are derived from identical input, salt, and info, differing only in length, which makes the 32-byte slot key the leading 32 bytes of the 64-byte encapsulation seed. The group signing seed is likewise required to pass through the KDF; deriving an Ed25519 seed as a bare digest of the vault seed does not conform, because a digest provides no domain separation and no label. Both derivations shall be migrated to the labels given in Appendix B, and migration shall preserve access to slots created under the earlier construction.

Argon2id support exists in the cryptographic library with 64 MiB, three passes, and one lane, consistent with a memory-hard password KDF interface [7]. The slot path calls HKDF rather

than Argon2id. This specification therefore makes no claim of Argon2id hardening for the six-digit Stealth PIN, and the PIN shall be treated throughout as a low-entropy selector rather than a content-encryption secret. Routing the PIN through a reviewed memory-hard construction is specified as future work in Section 16.

4 System Architecture and Trust Boundaries

4.1 Components

Figure 1 shows the principal trust boundaries. Plaintext editor state and raw keys exist on the client side. The service receives authenticated requests, ciphertext, manifests, public keys, encapsulations, and plaintext routing metadata.

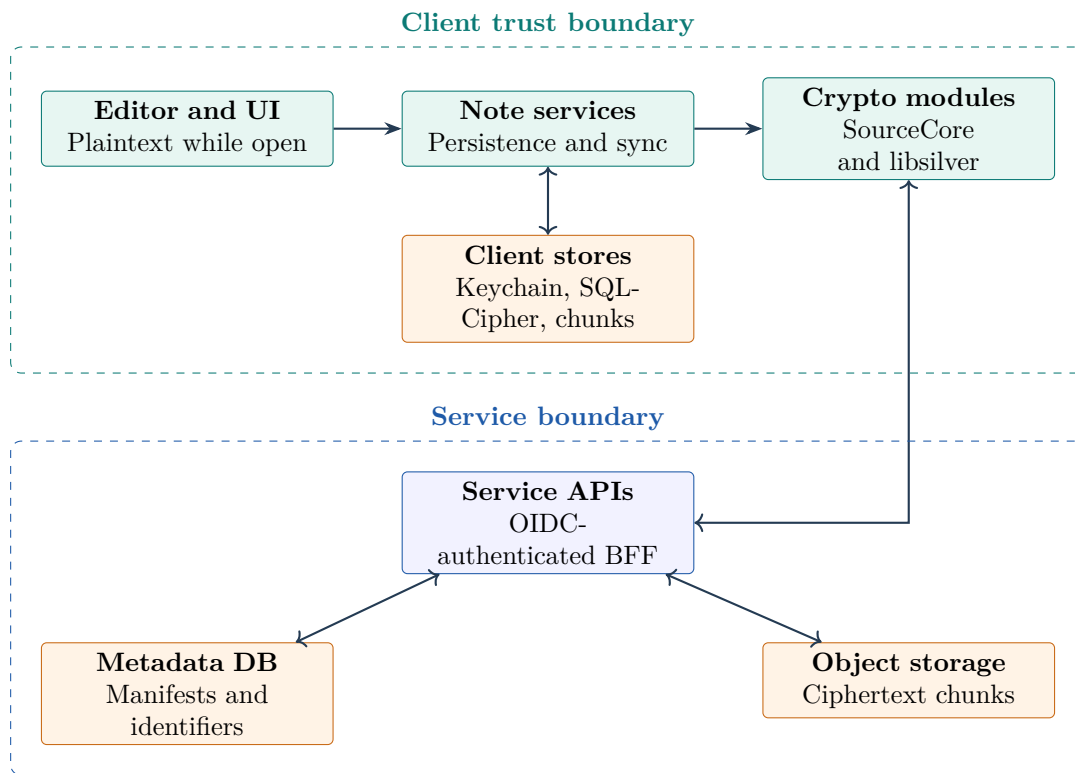


Figure 1: End-to-end trust boundary. Cryptographic operations and plaintext processing occur on the client; the service routes and stores ciphertext plus observable metadata.

4.2 Client boundary

The Apple clients contain:

- authentication and PIN-entry flows;
- editor serialization and title extraction;
- notebook, note, attachment, search, and sync services;
- SourceCore storage, security, vault, network, and sync modules;
- libsilver Swift bindings to lower-level cryptographic primitives;
- Apple Keychain storage for long-lived seeds;
- SQLCipher databases for user, slot, note, key-history, and search records; and

- filesystem directories containing only encrypted content chunks in the ordinary note-data path.

Apple describes Keychain Services as encrypted storage for small secrets [12]. The shared core requests the `whenUnlockedThisDeviceOnly` accessibility class for generated seeds by default. On iOS, that class is accessible only while the device is unlocked and is not migrated to another device [13]. The actual entitlements, access groups, and macOS behavior must be validated for each release; a Keychain class is not a substitute for testing backup and process access.

4.3 Service boundary

The service architecture separates identity/session authentication from vault and file operations. OIDC provides the account identity layer [14]; request-signing objects add application-level proof tied to user or vault keys. Manifest APIs coordinate drafts, versions, encrypted metadata, and DEK wraps; for objects written by the current protocol the top-level parent field is a sentinel and the real parent travels inside the encrypted metadata. Chunk APIs issue upload or download URLs and receive content length and content hash. Object storage holds the encrypted chunk bytes.

The service does not need note plaintext to synchronize ordinary content, but it necessarily handles plaintext routing metadata. Calling the entire backend “zero knowledge” would obscure that distinction. This specification uses the more precise phrase *server-unreadable note content* and enumerates the remaining metadata in Section 9.

4.4 Asset placement

Table 4: Asset placement and routine service visibility.

Asset	Client plaintext	Client persistent form	Service form
Note body	While edited/open	AES-GCM chunks; extracted copy inside SQLCipher FTS	Ciphertext chunks
Note title	While displayed/indexed	Plaintext field inside SQLCipher; encrypted in sync metadata	Ciphertext in metadata_encrypted
Notebook name	While displayed	Plaintext field inside SQLCipher	Ciphertext in encrypted metadata
Object DEK	During crypto operations	Usually wrapped; notebook DEK also stored raw inside SQLCipher	AES-GCM-wrapped under KEM secret and under the space root DEK
Space private keys	During derivation/use	Base64 inside SQLCipher key history in the key-history store	Not intentionally uploaded as plaintext
User seed	During recovery and derivation	Keychain	Never ordinary plaintext; encrypted transfer/recovery consequences apply
Parent relationship	In local manifest	Real value inside SQLCipher, never the sentinel	Ciphertext inside metadata_encrypted; top-level parent_id is a fixed sentinel (legacy objects: plaintext)
Search keywords	While searching	Recent terms in UserDefaults	Not part of ordinary sync protocol

Scoped confidentiality claim. Given an uncompromised client, correct cryptographic implementations, and absence of the relevant client-held or recovered key material, routine OXU Cloud storage does not receive the plaintext note body, title, notebook name, or attachment content. The service can still learn and correlate the metadata listed in Table 8.

5 Identities, PINs, and Key Hierarchy

5.1 Identity authentication is not content encryption

The OXU identity session authenticates a user to service APIs and selects account and workspace context. It does not itself decrypt a note. Conversely, possession of a content key does not automatically create an authorized service session. The system relies on both:

1. OIDC/session credentials for network identity and authorization;
2. an Ed25519 signing hierarchy for application signing objects; and
3. symmetric and KEM-derived material for content confidentiality.

5.2 Root seed and user keys

In the shared core, `user_seed` is 64 random bytes generated from the OS CSPRNG and stored in Keychain. It is exported as hexadecimal for the User Recovery Key. The seed deterministically regenerates:

- the account ML-KEM-768 key pair;
- an Ed25519 user-signing key pair derived with HKDF-SHA512 and the `ed25519-group-signing-key` label; and
- a common user-database key derived with a fixed 16-byte salt and `dbContext` information label.

Because the same `user_seed` is portable across devices, compromise has a large blast radius. It is a recovery root, not a disposable per-device token. The exported PDF or recovery string must be protected like a master key.

5.3 Default PIN

The standard unlock PIN is six decimal digits. The app stores `SHA256(UTF8(PIN))` inside the encrypted user SQLCipher database and compares the hash on unlock. After three failed attempts, the UI stores a ten-minute lockout timestamp and attempt count in `UserDefaults`.

The default PIN is therefore local access control, not a high-entropy content encryption secret. The default Note vault password shall be derived as:

$$P_{\text{note}} = \text{HKDF-SHA512}(\text{IKM} = \text{UTF8}(\text{userId}), \text{salt} = \epsilon, \text{info} = \text{my-note-password}, L = 32),$$

and then hex-encoded. The input is the user identifier, not `user_seed`. This specification therefore does not treat the default PIN as a default-space cryptographic root, and does not treat this password as secret from an actor who knows the user identifier and the construction. Its purpose is deterministic cross-device agreement, not confidentiality.

Derivation shall fail closed. If a key-derivation or hashing call fails, an implementation shall propagate the error; it shall not substitute the input material for the derived output, and shall not write a persistent record containing such a substitution.

5.4 Vault seed, salt, and per-space outputs

A 64-byte `vault_seed` shall be generated for each user, workspace, and vault tuple and stored in Keychain. The same vault seed is reused when deriving multiple spaces under one Note vault; spaces under a common vault therefore do not have independent root entropy, and this specification does not claim that they do. A 16-byte vault salt is obtained from the vault structure and, as Table 8 records, the same value is transmitted to the service in the space-creation request. It is a domain-separation input, not a secret, and shall not be relied upon to raise the cost of offline PIN search. For a Stealth PIN P :

$$I = \text{vault_seed} \parallel \text{vault_salt} \parallel \text{UTF8}(P).$$

The two outputs shall be derived under distinct, versioned labels:

$$\text{slot_key} = \text{HKDF-SHA512}(I, \text{vault_salt}, \text{OXU.note.slot-key.v2}, 32),$$

$$\text{encap_seed} = \text{HKDF-SHA512}(I, \text{vault_salt}, \text{OXU.note.encap-seed.v2}, 64).$$

The slot key encrypts serialized slot data. The encapsulation seed deterministically regenerates the space ML-KEM-768 key pair. The vault Ed25519 signing seed shall be derived separately from `encap_seed` with the vault salt and the `ed25519-signing-key` label.

5.5 Object key hierarchy

Within an opened space:

- the space has one root DEK, a random 32-byte key encapsulated to the space ML-KEM-768 public key and recovered when the space is unlocked;
- each new notebook, note, and attachment receives its own random 32-byte DEK; an attachment has its own manifest and is a logical child of the note that references it, not of the containing notebook;
- every object DEK shall be wrapped directly under the space root DEK and shall, in addition, be encapsulated to the space ML-KEM-768 public key for space-level recovery; and
- no object DEK written by the current protocol shall be wrapped under another object's DEK. The logical tree — root, notebook, note, attachment — is carried by the parent identifier inside each object's encrypted metadata (Section 9.2), not by key wrapping.

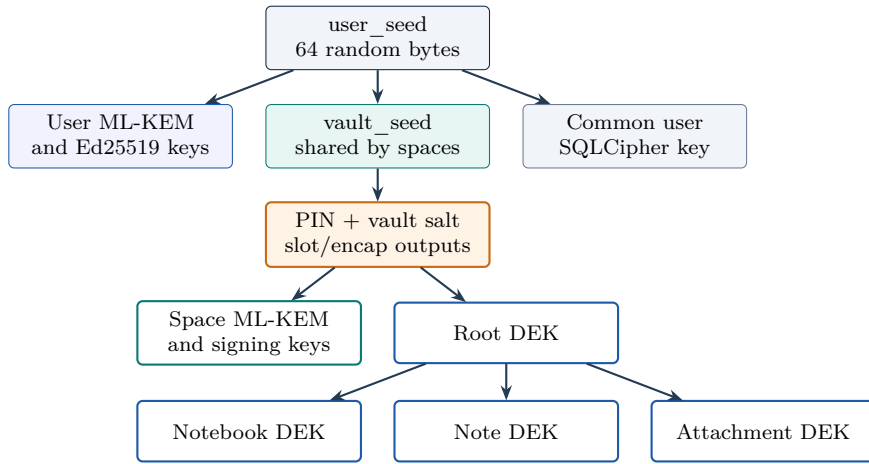


Figure 2: Logical key hierarchy. Edges show derivation above the slot boundary and wrapping below it, not independent entropy at every node. Object DEKs are siblings under the space root DEK: a notebook DEK does not wrap its notes' DEKs, and a note DEK does not wrap its attachments' DEKs. The notebook, note, and attachment tree exists only in encrypted metadata. vault_seed is shared across spaces in this design.

For an object DEK K_f in a space with root DEK K_r , the space-level encapsulation is:

$$(ct_f, ss_f) \leftarrow \text{MLKEM.Encaps}(pk_{\text{space}}), \quad W_f = \text{AESGCM.Enc}_{ss_f}(K_f),$$

and the root wrap is:

$$W_{f,r} = \text{AESGCM.Enc}_{K_r}(K_f).$$

The root wrap is the ordinary opening path: an unlocked space holds K_r and opens any of its objects with one AES-256-GCM unwrap. The encapsulation is the recovery path and opens legacy objects.

The flat layout has three consequences that this specification relies on.

1. **Key material encodes no structure.** No stored wrap links an object to its parent, so the wrapping layer does not reproduce the tree that the encrypted parent identifier hides.
2. **Moves rewrite no keys.** Moving or renaming an object produces a new manifest version whose encrypted metadata carries the new parent. The client shall not generate a new object DEK, or rewrap the object DEK or any descendant DEK, because the object moved.

3. **Compromise does not propagate down the tree.** For objects written by the current protocol, a notebook DEK decrypts that notebook’s metadata only, and a note DEK decrypts that note and not its attachments. The space root DEK reaches every object of the space, as it does in any design in which the root can open all objects.

A legacy object keeps the parent wrap it was written with, and the client shall open it through its encapsulated key. A move or rename of a legacy object writes the sentinel parent field and places the parent inside the encrypted metadata, but it does not rewrap the object DEK: the object remains a legacy object, the DEK of its former parent still opens it, and it is outside claim 2 of Section 9.2 until its DEK is wrapped under the space root. This version defines no operation that rewraps legacy objects, so an object that is never rewritten remains legacy.

5.6 Key storage and lifetime

Table 5: Key object storage and compromise impact.

Object	Origin	Storage/lifetime	Compromise impact
user_seed	64 random bytes	Keychain; exported as recovery key; present during recovery/derivation	Regenerates account KEM and signing keys; unlocks synchronized recovery capabilities
vault_seed	64 random bytes	Keychain; also inside encrypted recovery object	Enables derivation across spaces when combined with salt and PIN
slot_key	HKDF from vault seed, salt, PIN	Memory; recovery object in the recovery flow	Decrypts serialized slot record
encap_seed	64-byte HKDF output	Memory; recovery object; historical rewrap data	Regenerates space ML-KEM private key and contributes to signing key derivation
slotDBKey	Random field in slot data	Decrypted into memory; retained through background suspension	Opens note, info, and search SQLCipher DBs for one space
Root DEK	Random 32 bytes	Wrapped in slot config; memory while open	Unwraps every current object DEK of the space; reaches legacy objects through their parent-wrap chain
Notebook DEK	Random 32 bytes	Raw base64 inside SQLCipher in notebook creation; also wrapped under the space root DEK	Decrypts that notebook’s metadata; unwraps the DEKs of legacy notes still wrapped under it
Note DEK	Random 32 bytes	Wrapped in manifest under the space root DEK; transient cache and memory	Decrypts one note; unwraps the DEKs of legacy attachments still wrapped under it

Object	Origin	Storage/lifetime	Compromise impact
Attachment DEK	Random 32 bytes	Wrapped in manifest under the space root DEK (legacy: under the owning note DEK); transient cache and memory	Decrypts one attachment object
Space ML-KEM private key	Deterministic from <code>encap_seed</code>	Recomputed; also base64 in SQLCipher key history	Decapsulates current or historical space-level object DEKs
Space Ed25519 private key	HKDF-derived from <code>encap_seed</code>	Base64 in SQLCipher key history	Produces vault signing objects

6 Local Storage and the Note Data Model

6.1 Storage domains

The client shall separate account-level state from space-level state. The space storage consists of three SQLCipher databases and a ciphertext chunk tree:

Table 6: Required local storage domains, relative to the selected space directory. Filenames and layout describe the current implementation and are not a cryptographic protocol guarantee.

Path	Principal contents	Protecting key	Cleanup behavior
<code>key-stealth-vault.db</code>	Encrypted slot and space configuration, including wrapped root material	SQLCipher; key derived from the PIN or recovered for the selected space	Persists for the life of the space; slot record rewritten on PIN change or recovery reset
<code>slot-info.db</code>	Space key history, including base64-encoded ML-KEM and Ed25519 private material in the current implementation	SQLCipher under the space database key	Retains historical key versions to support rewrap; no specified expiry; retention is bounded by rewrap policy
<code>slot-data.db</code>	Notebook, note, manifest, chunk, version, and PIN-related records	SQLCipher under the space database key	Soft-deletion state precedes chunk cleanup; rows persist until convergence
<code>slot-data-index.db</code>	FTS5 rows containing note title and body text	SQLCipher under the space database key	Rewritten on note save; not separately purged on lock
<code>Data/</code>	Committed authenticated ciphertext chunks, named by object identifier	Per-object AES-256-GCM under that object's own note or attachment DEK	Superseded chunks are removed only after the corresponding database transaction commits; a crash in that window leaves stale ciphertext
<code>Tmp/⟨hashed object id⟩/</code>	Staged ciphertext chunks awaiting publication	Same per-object AES-256-GCM	Removed on the success path and on error; the whole Tmp tree is deleted when the space is next opened, so partial state is discarded rather than resumed

A separate account database uses a key derived as

$$K_{\text{userdb}} = \text{HKDF-SHA512}(\text{user_seed}, \text{defaultSaltBytes}, \text{"dbContext"}, 32),$$

where defaultSaltBytes is a fixed 16-byte application constant. This key separation keeps the account database distinct from a selected space, although compromise of user_seed still enables its recomputation.

Databases shall be opened with a raw hexadecimal key passed through PRAGMA key, with write-ahead logging enabled, synchronous mode set to NORMAL, a configured busy timeout, and WAL checkpointing as part of database management. Application code shall reach SQLCipher only through the shared-core interfaces; it shall not open databases directly. SQLCipher provides encrypted database pages and authenticated page storage when correctly configured [8, 9]. The linked SQLCipher release, cipher compatibility level, page size, and KDF iteration parameters shall be pinned and recorded as part of the dependency manifest required in section 14; this specification does not restate them, because a parameter that is not pinned is not a property of the design.

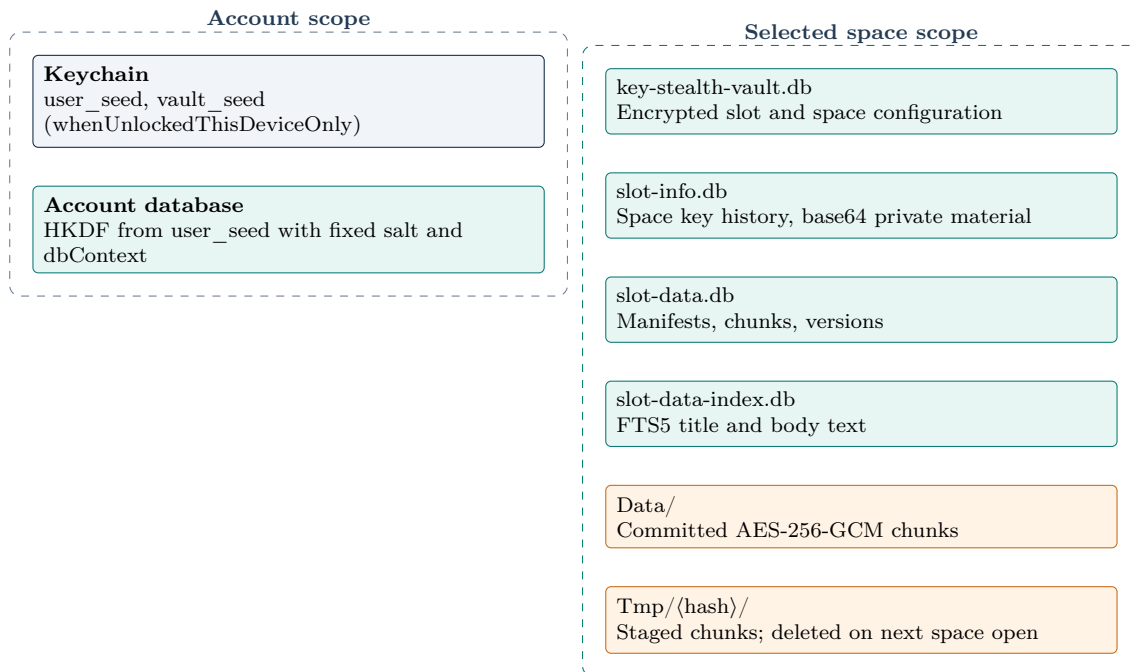


Figure 3: Local storage layout. Account-scoped material is separated from space-scoped material; the four space databases share the space database key, while chunk files carry independent per-object encryption. Only the selected space is opened at a time.

6.2 Logical object model

A space has a root object. Notebooks are children of the root; notes are children of notebooks; attachments are children of the note that references them. Each encrypted object has identifiers, a manifest, version data, encrypted metadata, one or more ciphertext chunks, and DEK wrapping material. The parent relationship is part of each object's encrypted metadata; the service stores a sentinel in the top-level parent field and cannot read the tree from stored fields (Section 9.2). Clients rebuild the tree after decryption and store the real parent locally.

Note content is a structured document rather than an unstructured text blob. The client serializes the document, derives a display title from its content, and limits the derived title to 100 characters. Both title and body are placed in the local search index after decryption.

6.3 Search index and convenience state

The FTS5 schema stores content-bearing rows with columns `file_id`, `title`, `body`, and `parent_id`; it uses the unicode61 tokenizer. FTS5 supplies the local full-text query mechanism [10]. Because the table contains plaintext title and body data, the confidentiality of search content depends on the SQLCipher database and on correct closure of its handles. It is not a contentless or hash-only index.

The indexer admits up to 1,000,000 combined title and body characters for an indexed item. This is an operational bound, not a cryptographic limit. Separately, the clients keep up to ten recent search terms and up to twenty recently accessed note or notebook identifiers with timestamps in `UserDefaults`. These convenience records are scoped by user and workspace, and are not encrypted per space. They can therefore reveal search intent or access patterns to a local examiner even when the note databases remain locked.

Local encryption does not remove all endpoint residue. SQLCipher protects database contents at rest, and object encryption protects chunk files. Operating-system preferences, filenames and directory structure, logs, temporary files, memory, backups, and system snapshots remain separate evidence surfaces. The endpoint threat model must treat them individually.

7 Note Lifecycle

7.1 Creation and first persistence

Note creation shall be performed entirely on the client:

1. The editor produces a structured note document and derived metadata.
2. The client generates a fresh 32-byte note DEK using the platform cryptographic random-number generator.
3. The serialized note is split into chunks of at most 50 MiB.
4. Each chunk is encrypted independently with AES-256-GCM using a fresh random nonce.
5. Ciphertext chunks are written to a temporary object directory.
6. The client computes indexed BLAKE3 leaf hashes and a Merkle root.
7. Metadata, including the parent notebook identifier, is encrypted under the note DEK. The DEK is encapsulated to the space key material and wrapped under the space root DEK; it is not wrapped under the notebook DEK.
8. Encrypted chunks are moved from the temporary directory into the committed data directory one file at a time.
9. Database rows — chunk records and the manifest, including the new Merkle root — are then committed in a single transaction.
10. Superseded chunk files are deleted only after that transaction commits.

Two properties of this sequence matter and are easy to state incorrectly. First, the file move precedes the database transaction; it does not follow it. Second, there is no atomic directory rename: publication is a per-file move or replace loop, so the transition from old to new content is not a single filesystem operation.

The two write paths therefore have different crash behavior. On the update path, each chunk is written under a fresh name qualified by a random identifier, so newly published chunks

never overwrite the chunks the committed manifest still references. A crash before commit leaves orphaned files but a readable note, and the old content remains authoritative until the transaction succeeds. This is a sound copy-on-write publication order. On the creation path, chunk names are derived deterministically from the object identifier, and an existing file of the same name is replaced in place. If a creation path runs for an identifier that already has chunks, a crash between the move and the commit can leave committed rows whose stored Merkle root no longer matches the ciphertext on disk, and the note will fail verification when reopened.

A crash after the commit but before cleanup leaves superseded ciphertext on disk. Staged state is not resumable: the temporary tree is deleted when the space is next opened, so an interrupted save is discarded rather than completed.

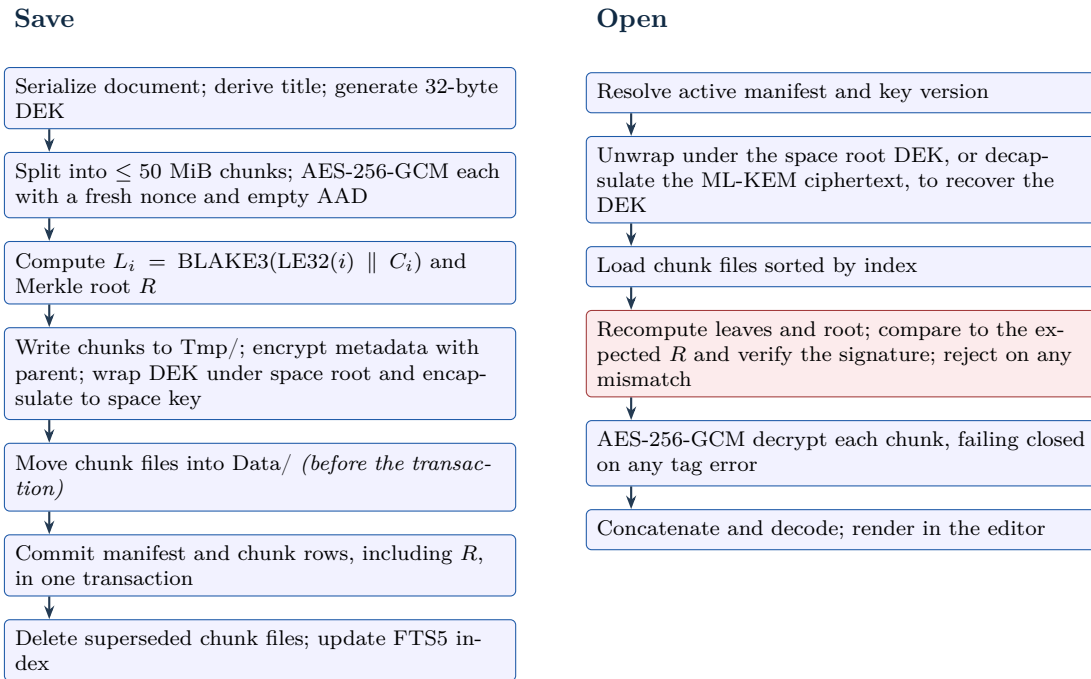


Figure 4: Note save and open. The save path moves ciphertext into its committed location before the database transaction and performs no atomic directory rename. On the open path, integrity verification precedes decryption and rejects the object on any mismatch.

7.2 Open, edit, and autosave

Opening a note selects its current local version, recovers the object DEK, sorts chunks by index, decrypts every chunk, concatenates the plaintext, and decodes the structured document. The editor works on plaintext in process memory. Saving creates encrypted content and version state before the sync engine exposes it to the service.

Autosave improves durability but expands the number of versions and timestamps that can reveal editing cadence. The service can observe upload time, ciphertext size, chunk count, and version relationships even though it cannot read the note body under the stated assumptions.

Moving a note to another notebook writes a new manifest version whose encrypted metadata carries the new parent. The object DEK and its wraps are unchanged, and the service observes that the object changed, not where it moved (Section 5.5).

7.3 Backgrounding and lock

On background suspension clients shall close space database handles. They intentionally retain several values, including the space database key, encapsulation seed, and vault DEK, so that the active session can resume. A full reset path applies `memset_s` to owned mutable byte buffers

before releasing them. However, the active Stealth PIN is also held as a Swift String; immutable language-managed strings cannot be reliably zeroized. The distinction is important:

- closing a database handle prevents ordinary queries;
- retaining its key preserves a live-session capability;
- zeroizing selected buffers reduces, but does not eliminate, memory recovery risk; and
- a compromised process can observe plaintext while the user edits.

7.4 Deletion

The local synchronization model uses soft-deletion state before later chunk cleanup. This supports convergence across devices but means that “delete” is not a cryptographic erasure guarantee. Locally, superseded ciphertext also survives an interrupted save, because chunk files are removed only after the database transaction commits. The covered client source does not establish service retention, backup expiration, object-store lifecycle, or physical media behavior. A claim of irreversible deletion would require a separately specified server and backup policy, and tests against it.

8 Content Encryption and Integrity

8.1 Chunk encryption

For a plaintext object P , let P_i denote the i -th chunk, with $|P_i| \leq 50$ MiB. The application performs:

$$K_f \xleftarrow{\$} \{0, 1\}^{256}, \quad C_i = \text{AESGCM.Enc}_{K_f}(N_i, P_i, \epsilon),$$

where N_i is a fresh 96-bit random nonce and ϵ is empty additional authenticated data. The stored form is:

$$C_i = N_i \parallel \text{ciphertext}_i \parallel T_i,$$

with a 128-bit authentication tag. Thus storage adds 28 bytes per encrypted chunk before container or database overhead.

Independent per-chunk nonces support large objects without holding the entire ciphertext in memory. Security nevertheless depends on never repeating a nonce with the same DEK. Nonces are generated randomly for every call; there is no persistent counter, collision detector, invocation budget, or rekey threshold in the current implementation.

The relevant invocation count is larger than one object. A note DEK is fresh per object but is reused for every subsequent version: updating a note re-encrypts all of its chunks under the same key. Invocations under a single DEK therefore accumulate as the product of saves and chunks, not as the chunk count of one save. NIST SP 800-38D caps the number of authenticated-encryption invocations at 2^{32} for a given key under the random-IV construction that this design uses [2]. At that bound the collision probability for 96-bit random nonces remains below 2^{-33} , so realistic note-editing volumes stay far inside the safe region and no practical nonce-reuse risk is identified here. The limitation is that nothing in the implementation measures or enforces the bound. A future revision should track invocations per DEK and rotate the key on threshold, so that the property is enforced rather than inferred from expected usage.

8.2 Indexed leaf hashes and Merkle root

The leaf for chunk i is:

$$L_i = \text{BLAKE3}(\text{LE32}(i) \parallel C_i).$$

Binding the little-endian index prevents a valid encrypted chunk from being silently moved to a different position without changing the leaf. Parent nodes are

$$H_j^{(r+1)} = \text{BLAKE3}(H_{2j}^{(r)} \parallel H_{2j+1}^{(r)}).$$

If a level has an odd number of nodes, the final node is duplicated. A one-leaf tree uses that leaf as its root. BLAKE3 is specified as a cryptographic hash with a tree construction suited to parallel processing [11].

8.3 Signing objects

Signing objects shall sign

$$\text{SHA256}(\text{UTF8}(m))$$

with Ed25519. This is an application-defined prehash construction. It must not be described as the standardized Ed25519ph mode in RFC 8032 [6], because Ed25519ph uses different domain separation and a SHA-512 prehash.

For content authorization, m is commonly a canonical JSON object carrying a checksum field whose value is the Merkle root. For selected request authorization, m is a small JSON object carrying `vault_id` and `time`. Security depends on deterministic serialization and on verifiers reconstructing exactly the same bytes.

8.4 What authentication covers

AES-256-GCM detects modification of a chunk under the correct DEK. The indexed leaf hash detects substitution or reordering relative to an expected Merkle root. A verified signature can bind that root to a signing identity. These layers do not automatically bind every server-visible manifest field: the top-level parent field, status, version pointers, timestamps, algorithm labels, and other request fields require explicit signature coverage or a trusted transport/service decision. The real parent, carried in the encrypted metadata, is protected against modification but not against replay of an earlier metadata version of the same object (Section 9.2).

Empty AES-GCM AAD also means object identity and version are not authenticated inside each chunk. A ciphertext moved between contexts will usually fail because the DEK differs, but this is an incidental consequence of key selection, not an explicit cryptographic binding. Future protocol revisions should place a canonical protocol version, space ID, object ID, version ID, chunk index, and content type into AAD.

9 Cloud Synchronization and Metadata

9.1 Synchronization transaction

The client performs a manifest-oriented upload:

1. Create a draft manifest containing encrypted metadata (which carries the real parent), wrapping records, the sentinel parent field, algorithms, hashes, and signing objects.
2. Request one or more upload URLs with ciphertext length and content hash.
3. Upload encrypted chunks directly to the supplied object-storage endpoint.
4. Submit or finalize the draft so that other devices can discover the version.
5. Pull manifest changes and ciphertext chunks on another device, then update the local encrypted databases and chunk tree.

TLS protects the account session and network path. OpenID Connect authenticates the user to the application service [14]; it does not replace content encryption and is not a content-signing identity.

9.2 Parent privacy

Titles, bodies, notebook names, and attachment bytes are encrypted (Section 8). This subsection specifies what the synchronization records of a space reveal about the *structure* of its objects: which note sits in which notebook and which attachment belongs to which note.

Claim. For objects written by the current protocol:

1. **Encrypted parent relationships.** The parent identifier of a notebook, note, or attachment travels only inside `metadata_encrypted`, next to its name and attributes. The top-level `parent_id` field of the manifest shall carry a fixed sentinel value on the wire and in storage, for every object at every depth, on every push path. The sentinel shall be a fixed value that is not a valid object identifier and differs from the value that denotes the space root. The service cannot read the parent-child edges of such objects from stored fields.
2. **Key material that encodes no structure.** Every object DEK is wrapped directly under the space root DEK (Section 5.5). No stored wrap links an object to its parent, and a move rewrites no key material.

Mechanisms. A conforming client shall:

1. on every push — including creations, content updates, moves, renames, and deletions, a deletion being a new manifest version with a deleted status and no chunks — draft the manifest with the space identifier, the sentinel parent field, the encrypted metadata carrying the real parent, the object DEK in its root-wrapped and encapsulated forms, and the space key version. The sentinel is the same for every object, so it carries no information;
2. on every pull, unwrap each object DEK under the space root DEK (falling back to decapsulation for legacy objects), decrypt the metadata, and read the real parent from it. The wire value of the parent field is used only when it is not the sentinel, as for legacy objects written under version 1.0 rules;
3. when the parent of a pulled object is not held locally, fetch that parent from the service by identifier, decrypt it, and continue upward until it reaches the space root or an ancestor already present, then persist the chain top-down. Because the tree has fixed depth, a resolution needs at most two fetches; a longer chain, a revisited identifier, a failed fetch, or a current-protocol object whose metadata carries no parent shall leave the object unresolved, retained, and retried on a later pull, never discarded or hidden (Appendix D); and
4. store the real parent in the local database, never the sentinel.

Objects are pushed in the order they were queued; no shuffling or timing jitter is specified.

Migration and interoperability. A client shall not write the sentinel for an object unless the same manifest carries the real parent inside its encrypted metadata. A client that does not implement this version reads the parent field as real and cannot place a sentinel-parented object. A client therefore shall not write current-protocol objects to a space while any device of the account may synchronize that space with such a client, and this condition shall be enforced rather than assumed. The two parts of the mechanism — the sentinel with the parent in encrypted metadata, and flat root wrapping — are both required; implementing one does not satisfy the other.

Not claimed.

- **Which space an object belongs to.** Every manifest, chunk record, and request names its space by identifier and carries the space key version; the service knows exactly which records form each space.
- **The number and activity of spaces.** Each space is registered separately and synchronizes separately (Section 12).
- **Structure inferable from other channels.** A device that receives a child before its parent fetches the parent by identifier in the same synchronization pass, so a service that observes request order learns that edge at that moment. This happens on a device's first walk of a space and whenever an ancestor arrives after its descendant; a device that already holds the ancestor makes no such request, but a device repeats such fetches whenever it rebuilds its local store or retries a fetch. Stable object identifiers, version chains, and signer identity link an object's manifests over time and to the account that wrote them; ciphertext lengths, counts, and timing remain visible.
- **Object kind.** A notebook carries no content chunks and no content signature, so the service can distinguish notebooks from notes and attachments and count each. The fixed depth of the tree is part of this specification and is not hidden.
- **Replay of an earlier parent.** The encrypted parent is protected against modification but not against the service substituting an earlier `metadata_encrypted` of the same object, which would silently revert a move.
- **Legacy objects.** Objects written under version 1.0 rules carry a plaintext parent identifier and a parent wrap. Rewriting them does not erase what the service recorded earlier; any mapping learned then persists.
- **Legacy objects rewritten without rewrapping.** A move or rename of a legacy object sets the sentinel but keeps its parent wrap; such an object is outside claim 2 until rewrapped, and the DEK of its former parent still opens it (Section 5.5).
- **A service-enforced property.** The sentinel is client behavior. The service stores and serves whatever parent value a client sends; a client that does not conform to this version populates the field.

Table 7: Structure-observation channels and their status.

Channel (observer)	Design response	Status
Parent-child edges from stored fields (service)	Parent encrypted; top-level field is a sentinel	Claimed, current objects
Tree reproduced by key wrapping (service)	Flat wrap under the space root; moves rewrite no keys	Claimed, current objects
Which space an object belongs to (service)	None	Visible by design
Number, creation, and activity of spaces (service)	None	Visible by design
Edge revealed by an ancestor fetched by identifier (service)	None; fetch occurs only when the ancestor is not held locally	Residual
Counts, sizes, version chains, timing (service)	None; no padding or cover traffic	Residual
Legacy objects (service)	None; plaintext parent retained	Residual
Object kind: notebook versus content object (service)	None; notebooks carry no chunks or content signature	Residual
Real parent in local databases (device examiner)	SQLCipher	Protected while the space database key is unavailable
Per-space databases and data directory on disk (device examiner)	Exist and grow only for spaces opened on the device	Residual

9.3 Service-visible information

Table 8: Service visibility and confidentiality boundary.

Data class	Service visibility	Security interpretation
Account, device, session, and network context	Plaintext to service or infrastructure	Needed for authentication and transport; exposes stable linkage and IP-derived context
Space, object, manifest, and version IDs	Plaintext	Reveal space membership, cardinality, cross-version linkage, and activity patterns
Top-level parent field	Plaintext sentinel	Carries no information for objects written by the current protocol; legacy objects carry the real parent. The real parent is inside the encrypted metadata (Section 9.2)
Status, reason, and action type	Plaintext	Can reveal operations; creation of a Note space includes a <code>MANUAL_CREATE_NOTE</code> action

Data class	Service visibility	Security interpretation
Vault salt	Plaintext	A 16-byte input to the Stealth derivation $I = \text{vault_seed} \parallel \text{vault_salt} \parallel \text{UTF8(PIN)}$, transmitted in the space-creation request. It must not be treated as a secret that raises the cost of offline PIN search: an adversary holding service records and <code>vault_seed</code> faces only the six-digit PIN space
Vault name	Plaintext	The name string embeds the user identifier, workspace identifier, and original vault creation timestamp, and is identical for every space under one vault
Encapsulation algorithm labels and key versions	Plaintext	The manifest and encapsulation labels correctly identify ML-KEM-768; they fingerprint client generation and migration state
Vault-settings labels	Plaintext, and inaccurate	Transmitted as hardcoded literals that do not describe the implemented construction; see Section 13 and treat as legacy declarative metadata rather than a negotiated suite
Per-space ML-KEM and Ed25519 public keys	Plaintext	Required for encapsulation and verification; stable keys can link activity
Vault-wide group signing key and group key version	Plaintext	Derived without the space PIN and therefore byte-identical across every space under one vault; binds concealed spaces to a single vault and account directly
Encrypted metadata	Ciphertext	Protects serialized names, attributes, and the parent identifier if the DEK remains secret
Note and attachment content	Ciphertext chunks	Protects content under per-object DEKs
Wrapped DEKs and KEM ciphertexts	Ciphertext/wrapping records	Enable authorized recovery of object keys without sending the private space key; every current wrap is under the space root, so wraps do not encode the tree
Request order during pull	Observable	A by-identifier fetch of a missing ancestor reveals that parent-child edge at that moment
Ciphertext length and chunk count	Plaintext	Approximate content size and editing behavior remain visible
Per-chunk hashes and Merkle root	Plaintext	Integrity commitments can correlate identical ciphertext records and expose chunk structure

Data class	Service visibility	Security interpretation
Signature objects	Plaintext	Bind selected canonical values when verified; do not cover all request fields
Recovery-object ciphertext presence	Plaintext existence, encrypted value	Reveals that recovery material exists for a space
Timing and synchronization cadence	Observable	Reveals access and editing patterns even without plaintext

Readable by the service

Account, device, session, and network context; organization and workspace association

`vault_id`, `file_id`, manifest and version IDs, `prev_version_id`, status, reason; sentinel `parent_id` (legacy objects: real parent)

Action type; vault name embedding user and workspace identifiers; vault salt

Per-space and vault-wide public keys; key versions; algorithm labels, including inaccurate settings labels

Ciphertext length, chunk count, per-chunk hashes, Merkle root, signature objects, timestamps

Opaque to the service

Note body and attachment bytes, as AES-256-GCM chunks under a per-object DEK

Note title, notebook name, parent identifier, and other serialized attributes inside `metadata_encrypted`

Root-wrapped DEKs and ML-KEM ciphertexts

Encrypted slot record; per-space recovery ciphertext, whose existence is visible while its content is not

Figure 5: Server visibility. The left column is plaintext to ordinary service storage and supports correlation across objects, versions, spaces, and time. The right column is protected under client-held keys. Confidentiality applies to the right column only; it does not extend to space membership, sizes, or timing on the left, nor to edges revealed by request order.

The encrypted metadata routine serializes object metadata, including the parent identifier, as JSON and encrypts it under the object DEK. Titles, note bodies, notebook names, parent relationships, and attachment bytes are therefore intended to remain outside service plaintext. This confidentiality claim assumes correct client encryption, protected keys, and no plaintext logging or alternate telemetry path.

9.4 Conflict, rollback, and availability

Version pointers and previous-version identifiers allow the service and clients to represent a history. This specification does not require a globally append-only log, transparency mechanism, monotonic client counter, or signed causal chain over every version field. A malicious or faulty service could withhold updates, return an older but otherwise valid encrypted version, fork views between devices, or deny uploads. End-to-end encryption alone does not solve these availability and rollback problems.

The service is also on the availability path for synchronization, recovery session coordination, and new-space registration. Users with only one device remain dependent on local backups or the recovery key when the service is unavailable.

10 Authentication and Signatures

10.1 Three distinct identity layers

Table 9: Authentication layers. Passing one layer does not imply possession of the others.

Layer	Key or credential	Purpose
Account authentication	OIDC tokens and service session	Authorizes API access and associates a device with a service account
User cryptographic identity	ML-KEM and Ed25519 keys derived from user_seed	Protects account recovery transfer and signs selected user-level statements
Space cryptographic identity	ML-KEM and Ed25519 keys derived from encap_seed	Wraps object DEKs and signs selected space or manifest statements

The separation is useful: compromise of a service token need not expose note plaintext, and a stolen encrypted manifest does not contain the private space key. Conversely, a live unlocked endpoint can exercise both account and cryptographic capabilities.

10.2 Canonicalization and domain separation

The use of small JSON messages makes signature inputs auditable, but generic JSON admits multiple valid encodings. Field order, slash escaping, Unicode normalization, number formatting, and whitespace must be deterministic. The current formulas also reuse the same helper for different message classes without an explicit protocol prefix in every signed string.

A safer signature envelope would include:

$$m = \text{CanonicalEncode}(\text{protocolVersion}, \text{purpose}, \text{userID}, \text{spaceID}, \text{objectID}, \text{versionID}, \text{merkleRoot}, \text{timestamp}).$$

The purpose value should be unique for manifest content, space registration, recovery transfer, and other operations. Verifiers must reject unknown versions and purposes.

10.3 Authorization boundary

A valid signature establishes possession of a private key for the exact signed bytes. It does not independently establish that the signer is still authorized for a space, that the version is the newest, or that a server field not included in the message is trustworthy. Authorization therefore combines account policy, current space membership or ownership, key selection, signature verification, and version policy.

11 Recovery and Device Transfer

11.1 User Recovery Key

The user-facing Recovery Key is the hexadecimal encoding of the 64-byte user_seed. It is a high-value bearer secret. From it the client can reproduce the account ML-KEM keypair, the account Ed25519 signing seed, and the common user database key. The client stores the seed in the platform Keychain using the accessibility class kSecAttrAccessibleWhenUnlockedThisDeviceOnly, which makes the item available only while the device is unlocked and prevents migration to a different device through ordinary Keychain backup [12, 13].

Export converts the seed to hexadecimal. Import normalizes the entered string by retaining only alphanumeric characters and lowercasing them, so separators and formatting are discarded, and then requires at least 128 normalized characters; the hexadecimal parser itself does not enforce a 64-byte result. Length parsing is therefore not the mechanism that rejects a malformed key.

Correctness is enforced cryptographically instead. Before installation the client verifies that the candidate seed reproduces both the account ML-KEM-768 public key and the account Ed25519 signing public key recorded for the user. The second check is necessary rather than redundant: an ML-KEM-768 encapsulation key depends only on the first 32 bytes of the 64-byte seed, so verifying the KEM key alone would leave the remaining 32 bytes unauthenticated, and a seed altered in that range would pass. Deriving the Ed25519 signing key over the full seed closes that gap. Import fails closed on either mismatch.

Users must store the Recovery Key outside OXU. Loss of every device and the key means the service does not have the secret needed to reconstruct the protected account identity.

11.2 Per-space recovery object

This design serializes a recovery object containing:

- the vault salt;
- the shared vault_seed;
- the space encap_seed;
- the slot identifier; and
- the slot_key.

It derives

$$K_{\text{recovery}} = \text{HKDF-SHA512}(\text{user_seed} \parallel \text{UTF8("V2")}, \emptyset, \emptyset, 32)$$

and encrypts the serialized object with AES-256-GCM. The resulting vault_recovery_key_encrypted value can be synchronized. The space PIN is not an input to this recovery-wrap key. Possession of user_seed and the recovery ciphertext is therefore sufficient to recover the included space material in this design.

Recovery changes the PIN security boundary. The Stealth PIN gates ordinary local selection and participates in local space derivation. It is not an independent factor against an attacker who has both the User Recovery Key and the encrypted per-space recovery object, because that object already contains the slot and encapsulation material.

11.3 Post-quantum protected QR transfer

The transfer protocol uses an ephemeral ML-KEM-768 keypair:

1. The receiving device generates an ephemeral keypair, computes a desired expiry 30 minutes ahead, and submits that value as a field of the session-creation request.
2. Its QR payload contains the session identifier, the first 16 bytes of $\text{SHA256}(\text{userID})$, and $\text{SHA256}(pk_{\text{ephemeral}})$.
3. The sending device obtains the session, validates both hashes, and encapsulates to the receiver public key.
4. It encrypts user_seed with AES-256-GCM under the resulting 32-byte shared secret and uploads the KEM ciphertext plus encrypted seed.

5. The receiver polls, decapsulates, decrypts the seed, and verifies that its derived ML-KEM and Ed25519 public keys match the account records before installation.
6. The client completes or cancels the session and clears local recovery state.

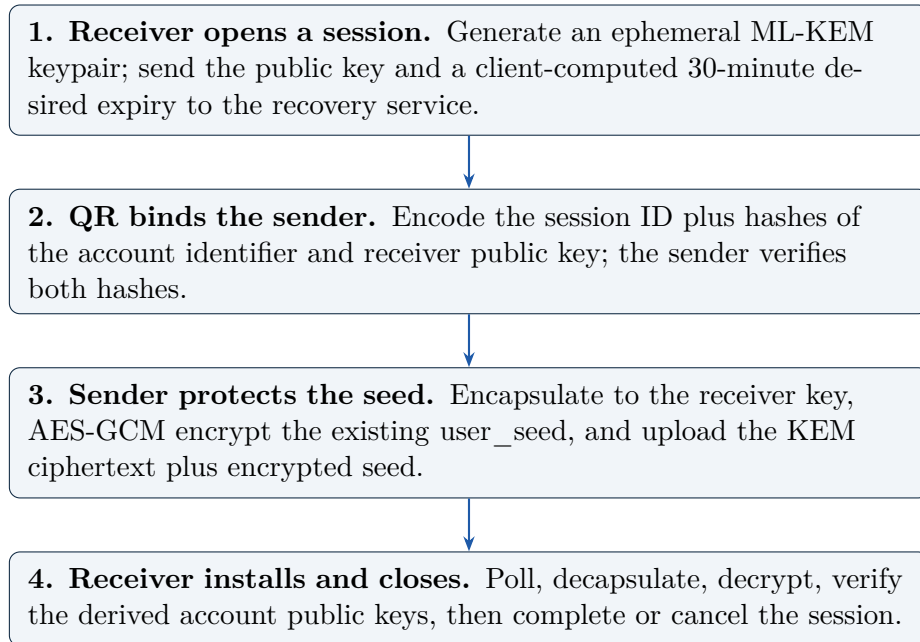


Figure 6: Recovery transfer. The service coordinates delivery but does not receive the plaintext user seed under the stated client and endpoint assumptions.

The 30-minute value is client-proposed and client-checked. The receiving device computes it locally, submits it as a request field, and later compares the expiry returned by the service against its own clock. The covered client source does not establish that the service enforces or clamps that value, rejects client-supplied expiries, deletes expired sessions, enforces single use, maintains anti-replay state, or bounds retention after completion. Until backend behavior is verified, the 30-minute figure should be read as a client-side hint, not as a bound on the exposure of an abandoned session carrying the encrypted user seed. These are backend requirements, not properties that can be inferred from client calls.

The account-identifier hash in the QR payload is also computed inconsistently by the two devices. The receiver hashes the whitespace-trimmed identifier with case preserved, while the sender lowercases it before hashing and comparing. The binding therefore succeeds only while account identifiers are already lowercase. That invariant is assumed rather than enforced, and a mixed-case identifier would cause the transfer to fail closed. The two paths should be normalized identically and the normalization should be specified.

Table 10: Recovery capability matrix. “Space material” means the slot key and encapsulation seed needed to open a space; PIN entry is a local selection step, not an independent factor once recovery material is available.

Material obtained	Default space	Concealed space	Other workspaces	Server cooperation needed?
Unlocked device, active session	Yes	Active space only	Only if opened	No
Stealth PIN alone, on the device	No	Yes, that space	No	No
user_seed (Recovery Key) alone	No, without synchronized recovery objects	No	No	Yes, to fetch recovery ciphertexts
user_seed plus synchronized per-space recovery ciphertext	Yes	Yes, without knowing the PIN	Yes, where such objects exist	Yes
vault_seed plus vault salt plus PIN	Yes	Yes, that PIN’s space	No	No, for derivation
encap_seed for one space	No	Yes, that space	No	Manifests required to obtain wrapped DEKs
Service records alone	No	No	No	Not applicable

12 Stealth Mode and Concealed Spaces

Scoped Stealth Mode claim. Stealth Mode conceals space selection in the normal locked-app user interface. It does not claim to hide the existence or number of spaces from OXU’s service infrastructure, a forensic examination of the device, live memory capture, or a compromised endpoint.

12.1 PIN-directed selection

One lock screen accepts a PIN. The default user PIN selects the ordinary space after its stored hash is verified. Other PIN candidates are used to derive and probe locally known or restored concealed-space slots. The client handles conflicts if more than one route would select the same input. An observer limited to the ordinary locked UI is not shown a list of concealed spaces or a separate “hidden vault” affordance.

For a Stealth space, the derivation input is:

$$I = \text{vault_seed} \parallel \text{vault_salt} \parallel \text{UTF8(PIN)}.$$

The resulting slot_key decrypts the slot record; encap_seed reproduces space key material. Creation obtains the vault salt, generates a new server slot identifier, creates the space keypair and root DEK, encrypts the local slot record, and registers a separate Note-space record with the service.

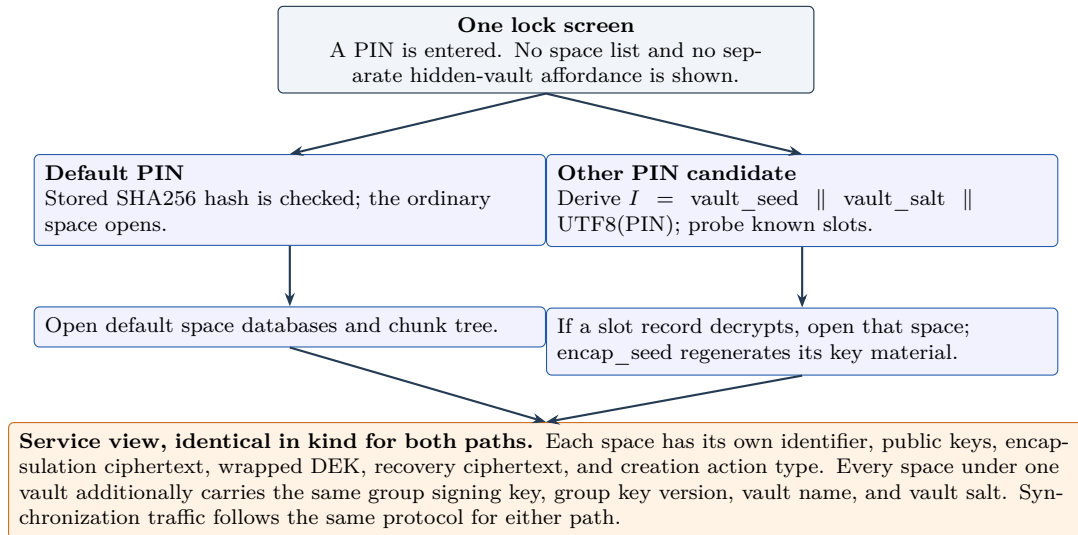


Figure 7: Multi-Space unlock and synchronization. The concealment is the absence of a space list at the lock screen. Once a space is open, its synchronization behavior and its service-side record are of the same form regardless of which PIN selected it.

12.2 What the service can infer

Registration transmits a distinct vault or slot identifier, per-space public keys, encapsulation ciphertext, wrapped DEK, recovery ciphertext, and an `MANUAL_CREATE_NOTE` action type. Synchronization thereafter names the space identifier on every record and request; the parent relationships of its objects are encrypted as specified in Section 9.2. The service can therefore count and link spaces associated with an account according to its own records even if the client UI does not display them.

Registration additionally transmits values that do not depend on the space PIN and are consequently identical for every space under the same vault: the group signing public key, the group key version, and the vault name string, which embeds the user identifier, workspace identifier, and original vault creation timestamp in plaintext. The vault salt, an input to the PIN derivation, is sent in the same request. The service therefore receives cryptographic and textual constants that bind concealed spaces to one vault and account directly, rather than requiring inference from timing, session, or transport context. This does not weaken the claim stated above, which never asserted concealment from the service; it does mean that the linkage is explicit in the protocol rather than merely inferable.

12.3 What a device examiner can infer

The local filesystem contains space databases and chunk trees. SQLCipher and slot encryption protect contents, but filenames, counts, sizes, timestamps, preferences, PIN fingerprints, and backups can reveal that additional state exists. The workspace-selection store indexes choices by a SHA256(PIN) fingerprint together with mode and user context. SHA-256 is not a password hash; the six-digit PIN domain is small enough for exhaustive testing if an attacker can recognize a correct record.

During an active Stealth session, the client explicitly tracks Stealth mode, holds the selected PIN as a Swift string, and retains derived keys for resume. A process compromise or live memory capture can therefore identify and read the active space.

Table 11: Stealth Mode by observer class.

Observer	Concealment	Residual visibility
Casual observer at locked UI	Strong for space list and selection	May observe unlock behavior, timing, or coercion response
OXU service	None claimed for existence or count	Separate identifiers, creation action, public keys, traffic, request order, and timing
Offline device examiner	Limited	Encrypted contents, but database and directory artifacts, preferences, counts, sizes, and timestamps
Live-memory examiner	None for active space	PIN strings, derived keys, plaintext document, current mode, and identifiers
Compromised client or OS	None	Can capture input and plaintext before encryption or after decryption

12.4 Explicitly not offered

The following are not offered by this specification. They are listed so that readers do not assume them from the presence of concealed spaces or of the parent privacy of Section 9.2:

- **Hidden volumes or duress PINs** in the sense of “the data does not exist” deniability.
- **Coercion deniability.** A user who reveals the PIN of one space cannot deny the existence or extent of other spaces; each is registered with the service under its own identifier and leaves local artifacts.
- **Hiding which space a record belongs to.** Every record names its space. Space-agnostic synchronization with on-device attribution would be a different protocol.
- **Decoy spaces or decoy objects.** Not offered; decoys would not by themselves establish deniability.
- **Padding, fixed-size containers, or cover traffic.** Each space’s volume and activity are visible.
- **Hiding the number of spaces.** Creation is a visible registration event.

13 Security Analysis

13.1 Scenario analysis

Table 12: Security controls and residual risk by scenario.

Scenario	Assets exposed	Principal controls	Residual risk
Service database theft	Manifests, IDs, public keys, wrapped keys, encrypted metadata, hashes, timing	Client-side object encryption; private keys absent from ordinary service records; encrypted parent identifiers	Space membership, counts, version chains, legacy parent edges, and traffic remain visible; recovery ciphertext can be targeted

Scenario	Assets exposed	Principal controls	Residual risk
Object-store theft	Ciphertext chunks, sizes, chunk grouping	Fresh per-object DEKs; random AES-GCM nonces; authenticated tags	Deletion and retention depend on backend policy; ciphertext can be preserved indefinitely
Network interception	Account requests and ciphertext in transit	TLS plus end-to-end content encryption	Traffic analysis; compromised trust store or endpoint; service metadata is intentionally plaintext after TLS termination
Lost locked device	Local encrypted DBs, chunks, preferences	Platform device lock, Keychain accessibility, SQLCipher, object encryption	Weak device passcode, preference leakage, backup artifacts, PIN fingerprint brute force
Live unlocked device	All active note plaintext and keys	App lock, operating-system isolation, session reset	Malware, debugging, screen capture, memory extraction, clipboard, input method
Stolen OIDC token	Service API capability	Token lifetime and server authorization; content keys remain client-side	Metadata access, traffic manipulation, denial of service, recovery-session abuse if server policy is weak
Stolen User Recovery Key	Account seed and derivable identity	512-bit random seed; user custody	With synchronized recovery objects, can recover space material without the Stealth PIN
Guessed six-digit PIN	Local space selection	Three-attempt UI lockout for ten minutes	Lockout is client state; offline artifacts or modified clients can bypass UI rate limiting
Malicious service	Metadata, ordering, availability	Ciphertext confidentiality and selected signatures	Withholding, rollback, equivocation, traffic analysis, and unsigned-field substitution remain possible

Scenario	Assets exposed	Principal controls	Residual risk
Compromised build pipeline	Client executable and dependencies	Release review, code signing, dependency pinning	Malicious client can exfiltrate plaintext or keys before encryption

13.2 Key compromise matrix

Table 13: Compromise scope. Historical manifests and recovery records can increase the practical scope beyond the immediate value.

Compromised value	Immediate capability	Not automatically exposed
One note or attachment DEK	Decrypt that object's content and metadata; a note DEK also unwraps legacy attachments still wrapped under it	Other notes, the note's current-protocol attachments, notebooks, or account identity
Notebook DEK	Decrypt that notebook's metadata, and the DEKs of legacy notes still wrapped under it	The notebook's current-protocol notes and their attachments
Space root DEK	Unwrap every root-wrapped object DEK of that space, reach legacy objects through their parent-wrap chain, and through them all content and metadata of the space	Other spaces
Space ML-KEM private key	Decapsulate space-level wrapped object DEKs in available manifests	Other spaces with independent encapsulation keys
Space database key	Read local SQLCipher databases for that space	Chunk plaintext without corresponding object DEKs, unless DB rows expose or wrap them
encap_seed	Reproduce space ML-KEM and signing key material	Account user_seed by one-way derivation
vault_seed alone	Reproduce the vault-wide group signing key and sign vault-level requests	Space content, which also needs a space PIN
vault_seed plus PIN and salt	Reproduce Stealth derivation outputs	Account identity without user_seed
user_seed	Reproduce account identity and user DB key; decrypt available recovery objects	Note plaintext if no recovery ciphertext is available
Active process	Read plaintext, keys, PIN input, and session tokens	No meaningful cryptographic boundary remains within that process

13.3 Forward secrecy and post-compromise security

The object design provides key granularity, not full forward secrecy. Historical manifests preserve KEM ciphertexts and wrapped DEKs. Later compromise of the corresponding long-term space private key can decapsulate those stored records unless keys and ciphertexts were rotated and destroyed. The QR transfer uses an ephemeral ML-KEM receiver key, but the transferred secret is the long-term user_seed; endpoint compromise after installation exposes it.

Similarly, the design has no demonstrated post-compromise healing protocol. Generating a new note DEK limits reuse for new objects, but an attacker retaining a space private key can continue to unwrap new manifests until the space key version changes and authorization rejects the old key.

13.4 Cryptographic agility

Every transmitted algorithm label shall name the suite actually in use, and shall name it precisely enough to distinguish a mode. Manifest encapsulation labels and key versions meet this today; they identify ML-KEM-768 correctly. The vault-settings object transmitted at space creation does not: the current implementation sends hardcoded literals declaring ECDH key exchange, PEM public-key encoding, and RSA-PSS signatures, while the same request carries genuine ML-KEM-768 and Ed25519 material and names ML-KEM-768 in its own encapsulation fields, and its encryption label states AES-256 without a mode. Those literals shall be corrected or removed. These strings do not describe the implemented construction and must be read as legacy declarative metadata rather than a negotiated suite.

Consequently the settings block provides no agility and no migration hook today, and a backend or future client that honored it would be wrong about the construction in use. Correcting or removing these fields is a prerequisite for any real migration mechanism. A secure migration must then define allowed suites, downgrade resistance, how old data is rewrapped, how retired private keys are protected or deleted, and which clients may write each version. Merely accepting a string algorithm label is not downgrade protection, and accepting an incorrect one is worse.

14 Implementation and Operational Boundaries

14.1 Logging

Application and shared-core sources contain diagnostic statements that can emit note titles, search queries, metadata, user-seed material, vault seeds, encapsulation values, slot values, recovery fields, or private-key material on some paths. Signing paths additionally log public keys, key versions, the exact signed message string, and the resulting signature; space creation logs the vault name, which embeds the user identifier. The shared core also enables SQLCipher's own diagnostic channel by directing cipher_log to standard error, which is a separate sink from the application logger and must be disabled or redirected independently. Whether a given statement executes in a production build depends on compiler flags, log wrappers, and runtime sinks not fully established here.

Before release, production logging should enforce a deny-by-default policy: no plaintext content, search terms, recovery secrets, PINs, DEKs, private keys, seeds, decrypted metadata, or full identifiers. Redaction must occur before a message reaches unified logging or a third-party crash reporter. Automated tests should scan both source and captured release logs using synthetic canary secrets.

14.2 Memory and temporary data

Swift and Apple platform frameworks copy data as part of serialization, string handling, UI rendering, and cryptographic bridging. Selective memset_s calls cannot prove that every copy was erased. Temporary ciphertext directories reduce plaintext filesystem exposure because the pipeline encrypts before writing chunks, but decoded documents, editor state, thumbnails, crash dumps, swap, and operating-system snapshots remain relevant.

14.3 Platform parity

Both platforms are in scope for every requirement in this document. Parity is a requirement, not a consequence of sharing a dependency: a platform that obtains a specified behavior from neither its application layer nor the shared core does not conform, and the two clients shall be shown to satisfy Sections 6 to 8 independently rather than by assumption.

Even where both applications resolve the same shared-core revision, their version, application lifecycle, preferences, Keychain behavior, file protection, and release flags are not identical merely because shared source is used. Security claims should be backed by the platform-by-platform test matrix in Table 14.

Table 14: Required platform parity. “App-level” marks a capability implemented in the application layer today; **Partial** marks one that is required but not present there; **Not established** marks one whose conformance requires runtime observation.

Capability	iOS 4.0.0	macOS 1.0.0	Basis and residual
Account seed, Keychain storage, account key derivation	App-level	App-level	Required on both; Keychain accessibility class is specified in section 4
Multi-Space slot creation and PIN-directed unlock	App-level	App-level	Required on both platforms
User Recovery Key import and account-key verification	App-level	App-level	Required on both; dual ML-KEM and Ed25519 checks
QR device transfer	Required	Required	Server-side expiry and single use shall be enforced
Note chunk encryption, indexed BLAKE3 leaves, Merkle root	App-level	Partial	Required on both; the macOS application layer has no equivalent
Local FTS5 search index	App-level	Partial	Required on both
Note create/update publication ordering	App-level	Partial	Required on both; see section 7
Sentinel parent field, parent inside encrypted metadata, flat root wrapping, ancestor resolution on pull	Required	Required	Required on both; see sections 5.5 and 9.2
Background retention and zeroization behavior	Not established	Not established	Conformance requires runtime observation on both platforms
Production logging and crash-reporter redaction	Not established	Not established	Conformance requires release-binary log capture on both platforms

14.4 Dependency provenance

Both applications shall pin the same revision of the shared cryptographic core and, through it, the same revision of the cryptographic library. Pinning shall be exact; a branch reference is not a pin, and two clients that name the same revision while tracking different branches shall not be assumed to resolve to the same code.

Each release shall be accompanied by a dependency manifest recording the resolved revision of every cryptographic dependency, the linked SQLCipher build parameters, and the compiler flags under which the release was produced. That manifest, and the exact revision identifiers, are release artifacts held alongside the build rather than restated in this document. A requirement

in this specification that cannot be traced to a pinned revision is not demonstrable, which is why the manifest is itself a release requirement.

14.5 Backend and build boundaries

Service-side behavior is outside the scope of this document and shall be specified separately. Because the client cannot enforce these properties, this specification states no requirement it could not verify from the client side, and makes no claim about:

- server-side ciphertext retention after deletion;
- backup expiration or operator-access controls;
- recovery-session single use and deletion;
- deployed rate limits or abuse detection;
- exact linked versions and compile flags of crypto dependencies; or
- absence of extra telemetry in distributed binaries.

Each of these is a requirement on the service or the build pipeline, and each shall be covered by its own specification and its own tests. A client-side specification cannot substitute for either.

15 Related Design Families

OXU Note combines several established design families:

- **Envelope encryption:** random object DEKs encrypt content, while a higher-level key mechanism wraps those DEKs. This limits routine bulk encryption to a symmetric primitive.
- **Authenticated chunking:** large objects are split into independently authenticated records and committed through a Merkle tree.
- **Encrypted local databases:** SQLCipher protects structured local state, while application-layer encryption separately protects chunk objects.
- **Key-derived spaces:** a user-entered selector participates in deriving the local capability that opens a concealed space.
- **Client-held recovery secrets:** a high-entropy seed recovers account cryptographic identity without asking the service to store the plaintext seed.
- **Hybrid KEM plus symmetric wrapping:** ML-KEM-768 establishes a shared secret, and AES-256-GCM uses that secret to wrap a compact DEK, as recommended for KEM-based application designs [4].

These categories describe architecture, not a claim of equivalence to any other product. Security comparisons require identical threat models, versions, defaults, recovery policies, metadata definitions, and deployment evidence. This edition deliberately makes no comparative security claim about any named competitor, because doing so responsibly would require primary documentation and version-matched evidence for each system, which was outside the covered scope.

One distinction must nevertheless be stated explicitly, because the surface similarity is misleading. OXU Note Multi-Space is *not* equivalent to a hidden-volume system such as VeraCrypt, nor to steganographic storage, nor to cover-traffic systems. Hidden-volume designs aim to make the existence of the concealed data indistinguishable from unallocated or random storage, so that an examiner holding the medium cannot establish that a second volume exists. OXU Note makes no such attempt and cannot: each concealed space is registered with the service under its own identifier, with a creation action type, public keys, a recovery ciphertext, and vault-wide constants

that link it to the same account, and locally it leaves distinct database and directory artifacts. The concealment offered is a user-interface property at the lock screen, as stated in Section 12. A design targeting genuine deniability would need a separate protocol, a separate threat model, and an independent assessment; it is listed as future work, not as a property of this system.

16 Conclusions and Priorities

OXU Note has a coherent client-side encryption architecture: high-entropy seeds and per-object DEKs, standardized encryption and post-quantum encapsulation primitives, a flat root-wrapping model with encrypted parent relationships, encrypted local databases, authenticated ciphertext chunks, and a recovery transfer that does not intentionally send the plaintext user seed to the service. Its Stealth interface offers a meaningful UI property when stated narrowly: the locked application does not enumerate concealed spaces.

The design also has deliberate limits, and they are part of the specification rather than caveats to it. The service observes substantial structural metadata, including the vault salt that participates in the Stealth derivation and vault-wide constants that link concealed spaces to one account. The Stealth PIN is a six-digit selector with client-side rate limiting, not a high-entropy encryption secret against offline examination; because the salt is service-visible, it shall not be assumed to add search cost. Recovery bypasses the PIN by design when the User Recovery Key and a per-space recovery ciphertext are both available. Active endpoints see plaintext. Spaces under one vault do not have independent root entropy.

Several requirements carried over from version 1.0 are not met by the implementation as it stands: the slot and encapsulation derivations are not domain separated, the group signing seed is a bare digest, note-content encryption passes empty associated data, download-time verification does not fail closed, the creation path publishes chunks under names that can overwrite referenced ciphertext, and the vault-settings algorithm labels do not describe the implemented suite. Each is stated as a requirement above. A release shall not be described as conforming to this specification until each has been remediated.

The highest-priority engineering work is therefore:

1. inspect and test the exact pinned SourceCore and crypto dependency, and map every security-relevant path;
2. make Merkle and signature verification mandatory and fail closed;
3. add independent, versioned KDF labels for the slot, encapsulation, and group-signing derivations, replacing the bare digest used for the last;
4. define canonical signature envelopes and bind context with AES-GCM AAD, using the associated-data entry points the crypto library already provides;
5. correct or remove the vault-settings algorithm labels so that no transmitted field misdescribes the implemented suite;
6. adopt fresh-name chunk publication on the note creation path and establish crash consistency by fault injection rather than by ordering;
7. normalize the QR account-identifier hash identically on both devices;
8. remove sensitive logs, including the SQLCipher diagnostic sink, and validate the distributed binaries;
9. document and test backend retention, recovery-session expiry and single use, rollback, and deletion behavior; and
10. run platform-specific adversarial, migration, and recovery tests, and complete the parity matrix in Table 14.

This specification intentionally favors verifiable requirements over broad assurances. As the protocol evolves, it should be versioned alongside code, test vectors, dependency provenance, and server policy so that every security statement can be reproduced.

A Algorithm and Parameter Registry

Table 15: Specified algorithms and parameters.

Purpose	Primitive	Parameters	Required encoding or use
Content and metadata encryption	AES-GCM	256-bit key, 96-bit random nonce, 128-bit tag	nonce ciphertext tag; empty AAD in note path
DEK wrapping	AES-GCM	Same	Under ML-KEM shared secret and under the space root DEK; legacy objects may carry a parent-DEK wrap
Key encapsulation	ML-KEM-768	Public key 1184 bytes; private key 2400 bytes; ciphertext 1088 bytes; shared secret 32 bytes	Deterministic keypair generation from application seed; randomized encapsulation
KDF	HKDF-SHA512	Output length varies by purpose	Salt and info listed in table 16
User and space signatures	Ed25519 over app prehash	Signs the SHA-256 digest of the UTF-8 message	Not standardized Ed25519ph
Chunk and tree hashing	BLAKE3	32-byte digest	Indexed leaves and binary Merkle parents
PIN storage	SHA-256	Six decimal digits in normal UI	Digest in encrypted DB; UI rate limiting is not a password KDF
Password KDF available	Argon2id	64 MiB memory, 3 iterations, parallelism 1	Not called by the slot derivation; see section 16
Database encryption	SQLCipher	Linked defaults shall be pinned	Raw hex application key; WAL and NORMAL synchronous mode

AES is standardized in FIPS 197 [1]; GCM is specified in NIST SP 800-38D [2]; ML-KEM is standardized in FIPS 203 [3]; HKDF is specified in RFC 5869 [5]; Ed25519 in RFC 8032 [6]; and Argon2 in RFC 9106 [7].

B KDF and Label Registry

Table 16: Specified derivation inputs. Empty fields are written as $\emptyset$.

Output	Input key material	Salt	Info / length
User signing seed	user_seed	Implementation default	ed25519-group-signing-key; 32 bytes
Common user DB key	user_seed	Fixed 16-byte application salt	dbContext; 32 bytes
Default Note vault password	UTF8(userID)	$\emptyset$	my-note-password; 32 bytes, hex encoded

Output	Input key material	Salt	Info / length
Stealth slot_key	vault_seed vault_salt UTF8(PIN)	∅	∅; 32 bytes
Stealth encap_seed	Same as slot key	∅	∅; 64 bytes
Space signing seed	encap_seed	vault_salt	ed25519-signing-key; 32 bytes
Group signing seed	SHA256(vault_seed), used directly as the Ed25519 seed	— (no KDF on the taken path)	No label. An HKDF fallback carrying ed25519-group-signing-key exists but is unreachable, because SHA-256 always returns the 32 bytes the branch tests for
Recovery-wrap key	user_seed UTF8("V2")	∅	∅; 32 bytes

C Protocol Field Inventory

Table 17: Draft and upload fields the protocol defines.

Field family	Representative names	Role and boundary
Hierarchy	parent_id, vault_id, file_id	Space association and object identity in plaintext; parent_id is a fixed sentinel and the real parent is inside metadata_encrypted (legacy objects: real parent)
Versioning	manifest ID, version ID, prev_version_id, status, reason	Synchronization state; not all fields are proven signature-covered
Content protection	encrypt_algorithm, metadata_encrypted	Declares suite and carries encrypted metadata
Space key	vault public key and version	Selects the space KEM key
DEK encapsulation	encapsulation algorithm, KEM ciphertext, encrypted DEK	Enables space private key to recover object DEK
Root wrap	dek_encrypted	Object DEK wrapped under the space root DEK; the ordinary opening path. Legacy objects carry a parent-DEK wrap here
Integrity	content hash, per-chunk hashes, Merkle root	Commits to uploaded ciphertext; verification shall fail closed
Authorization	vault signature, user signing	Selected statements; exact canonical message matters
Upload	content length, content hash, manifest ID	Lets object storage and service allocate and validate transfer

D Normative Pseudocode

D.1 Encrypt a note object

1. $K_f \leftarrow \text{RandomBytes}(32)$.
2. $P \leftarrow \text{SerializeStructuredNote}(\text{note})$.
3. For each indexed 50 MiB chunk P_i :
 - (a) $N_i \leftarrow \text{RandomBytes}(12)$.
 - (b) $C_i \leftarrow \text{AESGCM}(K_f, N_i, P_i, \epsilon)$.
 - (c) $L_i \leftarrow \text{BLAKE3}(\text{LE32}(i) \parallel C_i)$.
 - (d) Write C_i to the temporary object directory.
4. $R \leftarrow \text{MerkleRoot}(L_0, \dots, L_n)$, duplicating an odd final node.
5. Encrypt metadata, including the parent identifier, under K_f ; encapsulate K_f to the space key and wrap it under the space root DEK K_r . Set the top-level parent field to the sentinel.
6. Move each C_i from the temporary directory into the committed data directory, one file at a time. On the update path the chunk names are qualified by a fresh random identifier, so no referenced ciphertext is overwritten; on the creation path names are deterministic and an existing file is replaced in place.
7. Persist the manifest and chunk rows, including R , in a single database transaction. This follows the file move; there is no atomic directory rename.
8. After the transaction commits, delete superseded chunk files.
9. Publication shall be crash-consistent across the database and the filesystem on both write paths. Ordering alone does not achieve this; chunk names shall be freshly qualified on both paths so that no referenced ciphertext is ever overwritten.

D.2 Recover a note object

1. Resolve the active manifest and selected key version.
2. Unwrap K_f under the space root DEK K_r ; if that fails, as it does for a legacy object, decapsulate the ML-KEM ciphertext instead.
3. Load ciphertext chunks sorted by index.
4. Recompute indexed leaves and the Merkle root; compare to the expected root and verify the applicable signature, rejecting the object on any mismatch.
5. For every chunk, parse nonce, ciphertext, and tag; AES-GCM decrypt under K_f , failing closed on any authentication error.
6. Concatenate plaintext and decode the structured document.

Step 4 is not optional and not advisory. A verification routine that returns success before reaching its Merkle and signature checks satisfies no part of this specification, and neither does one whose checks are unreachable. A modified chunk, a reordered index, a substituted root, and a forged signature shall each cause the object to be rejected. See also section 8.

D.3 Resolve the parent of a pulled object

1. Recover K_f as above and decrypt the object's metadata M_f .
2. If the wire parent field is not the sentinel, $p \leftarrow$ the wire value. Otherwise $p \leftarrow M_f.\text{parent}$; if M_f carries no parent, the object is malformed: retain it unresolved and stop. The sentinel is never used as a parent.
3. While p is neither the space root nor held locally:
 - (a) if two fetches have already been made, or p was already visited, treat the chain as malformed: retain the object and the chain unresolved and stop;
 - (b) fetch the manifest of p by identifier; if the fetch fails or returns a deleted object, retain the object and the chain unresolved and retry on a later pull;
 - (c) otherwise recover its DEK, decrypt its metadata, apply step 2 to it, push it onto a stack, and set p to its parent.
4. Persist the stacked ancestors top-down, then the object, each with its real parent. An unresolved object shall remain retrievable and shall not be discarded or hidden.

Step 3 is the only structural request the protocol makes; it reveals each fetched edge to a service that observes request order (Section 9.2).

References

- [1] National Institute of Standards and Technology, *Advanced Encryption Standard (AES)*, FIPS 197, updated 2023. <https://csrc.nist.gov/pubs/fips/197/final>
- [2] National Institute of Standards and Technology, *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*, NIST SP 800-38D, 2007. Current at the date of this document; NIST announced in March 2024 that the publication will be revised. <https://csrc.nist.gov/pubs/sp/800/38/d/final>
- [3] National Institute of Standards and Technology, *Module-Lattice-Based Key-Encapsulation Mechanism Standard*, FIPS 203, August 2024. NIST published a planning note in November 2025 recording an identified issue to be corrected in a future update, with an accompanying errata list; implementations should be re-checked against that list. <https://csrc.nist.gov/pubs/fips/203/final>
- [4] National Institute of Standards and Technology, *Recommendations for Key-Encapsulation Mechanisms*, NIST SP 800-227, 2025. <https://csrc.nist.gov/pubs/sp/800/227/final>
- [5] H. Krawczyk and P. Eronen, *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*, RFC 5869, 2010. <https://www.rfc-editor.org/info/rfc5869>
- [6] S. Josefsson and I. Liusvaara, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, RFC 8032, 2017. <https://www.rfc-editor.org/info/rfc8032>
- [7] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, *Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications*, RFC 9106, 2021. <https://www.rfc-editor.org/info/rfc9106>
- [8] Zetetic, *SQLCipher Design*. <https://www.zetetic.net/sqlcipher/design/>
- [9] Zetetic, *SQLCipher API*. <https://www.zetetic.net/sqlcipher/sqlcipher-api/>
- [10] SQLite, *SQLite FTS5 Extension*. <https://www.sqlite.org/fts5.html>

- [11] BLAKE3 Team, *The BLAKE3 Hashing Framework*. <https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf>
- [12] Apple, *Keychain Services*. <https://developer.apple.com/documentation/security/keychain-services/>
- [13] Apple, *kSecAttrAccessibleWhenUnlockedThisDeviceOnly*. <https://developer.apple.com/documentation/security/ksecattraccessiblewhenunlockedthisdeviceonly>
- [14] OpenID Foundation, *OpenID Connect Core 1.0 incorporating errata set 2*, 2023. https://openid.net/specs/openid-connect-core-1_0.html