

OXU Drive

A Post-Quantum, Zero-Knowledge Drive System
with Encrypted Folder Structure

Cryptography White Paper

Authors

OXU Drive Security Team

September 2026

Abstract

This white paper presents the cryptographic architecture of OXU Drive, a zero-knowledge, cloud-synchronized drive system designed to provide strong client-side data protection with post-quantum confidentiality for file content. All encryption, decryption, and key derivation run on the client; the service back end synchronizes only ciphertext, encrypted manifests, and ML-KEM-encapsulated keys. We describe the hierarchical key-management system built on Argon2id and HKDF, ML-KEM-768 post-quantum key encapsulation of per-file data encryption keys, AES-256-GCM authenticated encryption, BLAKE3-based Merkle-tree file integrity, Ed25519 request signing (with ML-DSA implemented in the cryptographic core, pending activation), end-to-end encrypted vault and file sharing, key rotation, and multi-device transfer. We further present the metadata-privacy properties of vault synchronization — encrypted folder relationships, position-independent item keys, and context-bound chunk encryption — and we state explicitly which guarantees this design does and does not provide, including that it does not hide which space a record belongs to and offers no deniability of hidden content against a coercing adversary. The paper analyzes the system against a threat model that includes physical device access, user coercion, an honest-but-curious service operator, and future quantum adversaries.

Contents

1	Introduction	4
1.1	Deployment Status	4
1.2	Two Storage Products: Drive and Vault	4
1.3	Contributions	4
1.4	Paper Organization	5
2	Threat Model and Security Goals	5
2.1	Adversary Model	5
2.2	Security Objectives	6
2.3	Trust Assumptions	6
3	Cryptographic Primitives	7
3.1	Symmetric Encryption: AES-256-GCM	7
3.2	Hash Functions	8
3.3	Key Derivation Functions	8
3.3.1	Argon2id	8
3.3.2	HKDF-SHA512	8
3.4	Post-Quantum Key Encapsulation: ML-KEM	8
3.5	Digital Signatures	9
3.5.1	Ed25519	9
3.5.2	ML-DSA (FIPS 204)	9
3.6	Secure Random Generation	9
3.7	The libsilver Cryptographic Core	9
4	System Architecture	9
4.1	Design Principle: Client-Side Cryptography Only	9
4.2	Client Architecture	9
4.3	Service Architecture	10
4.4	The Zero-Knowledge Boundary	10
4.5	Authentication Layers	10
4.6	FileProvider Key Sharing	11
5	Key Hierarchy and Management	12
5.1	Key Objects	12
5.2	Key Derivation Flow	12
5.3	Per-Space Keypairs	13
5.4	Post-Quantum Encapsulation of DEKs	14
5.5	Secure Storage Integration	15
6	Vault Key Storage and Local Databases	16
6.1	The Vault Key Database	16
6.2	Slot Records	16
6.3	Integrity Verification	16
6.4	Local Database Layout	17
6.5	SQLCipher Configuration	17
6.5.1	Cipher Mode Note	17

7	File Encryption, Integrity, and Synchronization	17
7.1	Per-File DEKs and Chunking	17
7.2	File Integrity: BLAKE3 Merkle Trees	18
7.2.1	AEAD Context Binding (AAD)	19
7.3	Upload and Download Pipeline	19
7.4	Synchronization Model	19
7.5	Pull and Parent Resolution	20
7.6	Push	20
7.7	Vault Data Directory	21
8	Metadata Privacy	21
8.1	Model and Terminology	21
8.2	The Claim — and What Is Not Claimed	21
8.3	Analysis	23
8.4	Mechanisms	23
8.4.1	Sentinel Parent Field	23
8.4.2	Parent Inside the Encrypted Metadata	23
8.4.3	On-Device Resolution with Ancestor Climb	24
8.4.4	Flat Root Wrapping	24
8.4.5	Rotation and Deletion	24
8.4.6	Compatibility	24
8.5	Cost	24
8.6	Observation Channels	25
8.7	Explicit Residual: Structure Inference Beyond Parent Edges	25
8.8	Known Limits	25
8.9	Explicitly Not Offered	26
9	Digital Signatures and Request Authentication	27
9.1	Ed25519 Signing with SHA-256 Prehash	27
9.2	Signing Keys and Signing-Object Types	27
9.3	Signing-Object Structure and Verification	27
9.4	Relationship to Session Authentication	28
10	Sharing Cryptography	29
10.1	Two-Layer Key Wrapping	29
10.2	Share Records and Authorization	29
10.3	Recipient Flow	30
10.4	Revocation and Expiry	30
11	Key Rotation	30
11.1	What Rotation Changes	31
11.2	Protocol	31
11.3	State Machine and Safety	32
11.4	Cryptographic Effect	32
12	Multi-Device Support and Recovery	33
12.1	Device Transfer Protocol	33
12.2	Vault List Synchronization	33
12.3	Recovery Model	33
12.4	Multi-Device Restore	34
12.5	Deletion Options and Panic	34

13 Security Analysis	35
13.1 Analysis Against the Threat Model	35
13.2 Key Compromise Resilience	35
13.2.1 Isolation Boundaries	35
13.2.2 Forward Secrecy Limitation	36
13.3 Known Weaknesses and Open Items	36
14 Implementation Notes	36
14.1 Performance Considerations	36
14.2 Memory Security	36
14.3 Nonce Management	37
14.4 Error Handling	38
15 Related Work	38
15.1 Encrypted Drive Solutions	38
15.2 Post-Quantum Cryptography Adoption	39
15.2.1 Hybrid Classical/Post-Quantum Considerations	39
16 Conclusion and Future Work	39
16.1 Summary	39
16.2 Future Work	39
16.3 Security Audit Status	40
A Algorithm Parameter Specifications	41
B HKDF Info-Label Registry	41
C Vault Key Database Schema	42
D Key Derivation Pseudocode	43

1 Introduction

The increasing sophistication of cyber threats and the anticipated arrival of cryptographically relevant quantum computers present fundamental challenges to traditional data-protection mechanisms. Devices and cloud accounts that store sensitive personal and professional data are vulnerable to physical-access attacks, service-side compelled disclosure, and “harvest now, decrypt later” strategies employed by adversaries anticipating future quantum capabilities.

OXU Drive addresses these challenges with a cryptographic architecture that combines client-side (end-to-end) encryption, post-quantum key encapsulation, and metadata-private synchronization of its vault product, while using a zero-knowledge cloud back end for synchronization, backup, sharing, and multi-device access.

1.1 Deployment Status

OXU Drive ships as native clients for macOS and iOS, integrated with the platform file manager through the NSFileProvider framework and backed by the OXU cloud services. All clients share a single cross-platform cryptographic core (Section 3.7): a portable Rust library with production Swift bindings, and additional language bindings that provide the path to further platforms. The properties described in this paper are properties of the Apple-platform clients and the cloud services they connect to, unless stated otherwise. The folder-structure protection of Section 8 applies to vault items written by the current macOS and iOS clients; items written by earlier client versions are treated as legacy data there. The behaviours described are those of one fixed revision of the client and service source, reviewed for this document; whether a particular installed build carries every described mechanism is a release-management matter outside its scope, and where a build predates a mechanism the legacy treatment stated in the relevant section applies.

1.2 Two Storage Products: Drive and Vault

OXU Drive exposes two distinct storage concepts built on the same cryptographic foundation:

Drive (“My Drive”) Everyday encrypted file storage with familiar file/folder UX and file-level sharing. One Drive per user, protected by a device PIN for local access. The Drive synchronizes through the original manifest protocol, whose records carry plaintext identifier graph keys (Section 4.4).

Vault The advanced security layer. A user can create multiple vaults; each vault contains one or more password-protected *spaces* (also called *slots*). Each space synchronizes its own records, whose folder relationships are encrypted on the server (Section 8). Vaults support vault-level sharing, key rotation, and panic actions.

Both products encrypt all content on the client before upload and integrate with the platform file manager through the NSFileProvider framework.

1.3 Contributions

1. Post-quantum secure key encapsulation of per-file data encryption keys using ML-KEM-768
2. Hierarchical key derivation with Argon2id password stretching and domain-separated HKDF
3. A zero-knowledge synchronization back end: the server stores only ciphertext, encrypted manifests, and encapsulated keys

4. File integrity via BLAKE3 chunk hashes and a signed BLAKE3 Merkle root per file
5. End-to-end encrypted vault and file sharing with two-layer key wrapping (share password + ML-KEM-768 to the recipient)
6. Client-driven key rotation that re-encrypts a space's content under fresh keys, with a batched server-side promotion of the new manifests
7. Encrypted folder structure for vault items: explicit parent relationships are encrypted, item keys are wrapped independently of position, and chunks are bound to their identity — with an explicitly scoped (and explicitly limited) statement of what this hides and what remains visible

1.4 Paper Organization

- **Section 2** defines the threat model and security objectives
- **Section 3** describes the cryptographic primitives and the libsilver core
- **Section 4** presents the client and service architecture and the zero-knowledge boundary
- **Section 5** details the key hierarchy and key management
- **Section 6** specifies vault key storage and the local database layout
- **Section 7** describes file encryption, chunking, integrity, and synchronization
- **Section 8** presents the metadata-privacy properties of vault synchronization and their limits
- **Section 9** describes digital signatures and request authentication
- **Section 10** covers vault and file sharing
- **Section 11** covers key rotation
- **Section 12** covers multi-device transfer and recovery
- **Section 13** analyzes the design against the threat model
- **Section 14** discusses implementation considerations
- **Section 15** compares with existing solutions
- **Section 16** concludes with future work

2 Threat Model and Security Goals

2.1 Adversary Model

We consider the following classes of adversaries:

Definition 2.1: Physical Access Adversary

An adversary who gains temporary or permanent physical access to the device, potentially with the ability to extract storage contents, open local databases with forensic tooling, and analyze file structures.

Definition 2.2: Coercion Adversary

An adversary who can compel the user to reveal a password and unlock a space (“rubber-hose” attack), typically combined with physical seizure and forensic analysis. The design offers this adversary no deniability of hidden content: Section 8.2 states what such an adversary learns, and Section 8.3 shows why content-bearing spaces cannot be denied under this design.

Definition 2.3: Honest-but-Curious Service Operator

The OXU Drive back end (or any party with access to its databases and object storage, including via subpoena) is assumed to follow the protocol but to inspect everything it stores and observes: ciphertext blobs, encrypted manifests, request metadata, timing, and volumes.

Definition 2.4: Network Adversary

An adversary who observes or tampers with network traffic between clients and the service. Transport security (TLS) is assumed; the design must not depend on the network for confidentiality of content or keys.

Definition 2.5: Quantum Adversary

A future adversary with access to cryptographically relevant quantum computers capable of breaking classical public-key cryptography, including one who records ciphertext today for later decryption.

2.2 Security Objectives

1. **Confidentiality:** file content, file/folder names, and keys must remain unreadable to the server and to any party without the proper credentials
2. **Integrity:** tampering with stored content must be detectable by the client
3. **Authenticity:** state-changing operations must be verifiably authorized by a legitimate key holder
4. **Metadata Privacy (scoped):** the service operator must not learn, from the synchronization records of current vault items, the folder relationships of stored items (see Section 8.2 for the precise, deliberately limited claim)
5. **Post-Quantum Resistance:** confidentiality must hold against quantum adversaries, including harvest-now-decrypt-later

2.3 Trust Assumptions

OXU Drive operates under the following trust assumptions:

1. **Secure Hardware Foundation:** The device’s hardware security mechanisms and OS secure-storage services (Apple Keychain on the shipped platforms) are not compromised and correctly enforce access controls.
2. **Operating System Integrity:** A compromised operating system can read keys and plaintext while a space is unlocked; this is outside the threat model.

3. **Cryptographic Primitive Security:** The underlying implementations (ML-KEM, AES-GCM, Ed25519, Argon2id, HKDF, BLAKE3) are assumed correct and secure against their respective threat models.
4. **Random Number Generation:** The platform CSPRNG produces unpredictable output suitable for key generation and nonce creation.
5. **User Password Entropy:** Security against brute-force attacks on a space password depends on password strength; Argon2id stretching raises the per-guess cost but cannot rescue very weak passwords. We recommend high-entropy passwords for high-risk use.
6. **Client Binary is Public:** Adversaries are assumed to possess and reverse-engineer the client binary. Metadata privacy therefore cannot rely on secret client behavior (Section 8.8).
7. **Limits Against Coercion:** A coercing adversary who obtains the device, its seed material, and one or more space passwords learns the content of those spaces. The design does not hide the existence of the other spaces: the device’s registry and the server’s records identify every space and its records (Section 8.3).

3 Cryptographic Primitives

OXU Drive employs a deliberately small suite of well-analyzed primitives, implemented in a dedicated cross-platform cryptographic core (Section 3.7).

Table 1: Primitive Suite Overview

Role	Algorithm	Status	Notes
Symmetric AEAD	AES-256-GCM	Implemented	default for all data encryption
Symmetric AEAD (alt.)	ChaCha20-Poly1305	Implemented	available in the core library
Password KDF	Argon2id	Implemented	$m = 64$ MiB, $t = 3$, $p = 1$
Hierarchical KDF	HKDF-SHA512	Implemented	domain-separated info labels
General hash	SHA-256	Implemented	key versioning, Ed25519 prehash
Content hash	BLAKE3	Implemented	chunk hashes, Merkle-tree file root
KEM (post-quantum)	ML-KEM-768	Implemented	FIPS 203, NIST Level 3
Signatures	Ed25519	Implemented	request signing, integrity
Signatures (post-quantum)	ML-DSA-44/65/87	Implemented, not active	FIPS 204; pending activation
Database encryption	SQLCipher (AES-256-CBC)	Implemented	raw pre-derived 32-byte key
CSPRNG	OS-native	Implemented	key and nonce generation

3.1 Symmetric Encryption: AES-256-GCM

All data encryption uses AES-256-GCM (Galois/Counter Mode) [5], providing:

- 256-bit key strength
- Authenticated encryption with associated data (AEAD)
- 96-bit nonce with 128-bit authentication tag

The default backend is AWS-LC-RS; validated configurations of its underlying AWS-LC library exist, but the shipped build does not itself claim FIPS validation. A pure-Rust RustCrypto backend is available as an alternative, and ChaCha20-Poly1305 is available for platforms without AES hardware acceleration.

3.2 Hash Functions

Table 2: Hash Functions

Algorithm	Output Size	Use Case
SHA-256	256 bits	Ed25519 prehashing, key versioning, slot hashes
SHA-512	512 bits	HKDF and HMAC operations
BLAKE3	256 bits	Per-chunk content hashes; Merkle-tree file root

BLAKE3 [8] is used on the content-integrity path because of its high throughput on large files and its native tree-hashing construction; see Section 7.2.

3.3 Key Derivation Functions

3.3.1 Argon2id

Password-based key derivation uses Argon2id (RFC 9106) [4], the winner of the Password Hashing Competition, providing resistance against GPU and ASIC attacks. OXU Drive uses memory cost 64 MiB, time cost 3 iterations, and parallelism 1 (Appendix A). Every password-derived key in the current derivation passes through Argon2id before any fast KDF is applied; spaces created before that derivation was introduced are opened through the earlier, unstretched derivation until re-keyed (Section 5.2).

3.3.2 HKDF-SHA512

Hierarchical key derivation uses HKDF (RFC 5869) [6] with SHA-512. No two derived keys share both input keying material and `info` label, so under the pseudorandomness of HKDF-SHA512 no two keys are drawn from the same output stream; the label registry, including the salt used on each path, is given in Appendix B.

3.4 Post-Quantum Key Encapsulation: ML-KEM

OXU Drive implements the NIST-standardized ML-KEM (FIPS 203) [2], formerly CRYSTALS-Kyber, at the 768 parameter set (NIST security category 3).

Table 3: ML-KEM Parameter Sets

Variant	Security Level	Public Key	Ciphertext	Shared Secret
ML-KEM-512	NIST Level 1	800 bytes	768 bytes	32 bytes
ML-KEM-768	NIST Level 3	1,184 bytes	1,088 bytes	32 bytes
ML-KEM-1024	NIST Level 5	1,568 bytes	1,568 bytes	32 bytes

3.5 Digital Signatures

3.5.1 Ed25519

Classical elliptic-curve signatures over Curve25519 (RFC 8032) [7], providing 128-bit classical security with compact 64-byte signatures. All request signing and integrity signatures currently use Ed25519 (Section 9).

3.5.2 ML-DSA (FIPS 204)

The cryptographic core implements ML-DSA-44, ML-DSA-65, and ML-DSA-87 (module-lattice signatures, FIPS 204) [3]. These are *implemented but not yet activated* in the product protocol: no shipped flow produces or verifies ML-DSA signatures. Activation — including key-versioning and migration — is tracked as future work (Section 16.2). Statements about signature security in this paper therefore refer to Ed25519 unless noted.

3.6 Secure Random Generation

Cryptographically secure random bytes are generated using the platform CSPRNG (e.g. `SecRandomCopyBytes` on Apple platforms, `/dev/urandom` or equivalent elsewhere). Every seed is at least 64 random bytes obtained from the platform CSPRNG.

3.7 The libsilver Cryptographic Core

All primitives are provided by *libsilver*, a Rust library shared by all clients:

- **Backends:** AWS-LC-RS (default) and RustCrypto (portable alternative); no FIPS validation is claimed for the shipped build.
- **Bindings:** Swift (production, macOS/iOS), Node.js and Kotlin (paths to further platforms).
- **Memory hygiene:** key material inside the core is held in self-zeroizing buffers (zeroize-on-drop); the client layers hold key material in ordinary language objects and release the references on lock, without a demonstrated overwrite of the bytes (Section 14.2).
- **Testing:** the primitive layer ships with an extensive unit and known-answer test suite exercised in CI.

4 System Architecture

4.1 Design Principle: Client-Side Cryptography Only

Content encryption and decryption, secret-key derivation, and signing run on the client device. The cloud back end synchronizes opaque material, verifies signatures, terminates TLS, and enforces authorization, quotas, and consistency; it is never a decryption authority. “Zero-knowledge” is used throughout in this content-confidentiality sense, not in the sense of a zero-knowledge proof system. This boundary is the core security property of the architecture and each subsequent section states precisely what crosses it.

4.2 Client Architecture

Figure 1 shows the layered topology on both sides of the zero-knowledge boundary.

The client consists of:

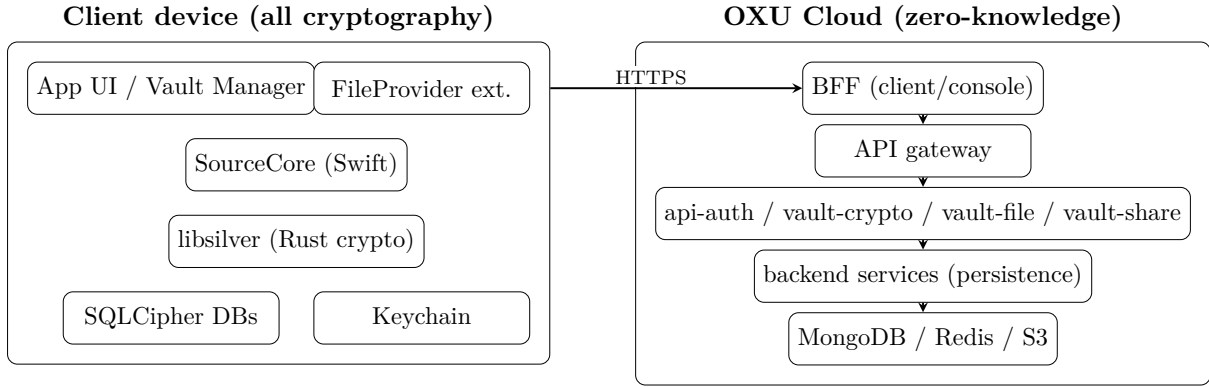


Figure 1: System architecture. All cryptographic operations run on the client; the service layers store and route only opaque material.

- **Application** (macOS/iOS, Swift): UI, vault/drive management, sync orchestration, panic actions.
- **FileProvider extension**: integrates vault content with Finder/Files. It runs in a separate process and receives keys through a controlled channel (Section 4.6).
- **SourceCore**: the shared Swift package implementing key management, vault format, sync engine, and storage.
- **libsilver**: the Rust cryptographic core (Section 3.7).
- **Local storage**: SQLCipher-encrypted databases and encrypted chunk files (Section 6); platform Keychain for seeds.

4.3 Service Architecture

The back end is a layered Go microservice system: clients reach a backend-for-frontend (BFF) over HTTPS; an API gateway routes to API-logic services (**api-auth**, **api-vault-crypto**, **api-vault-file**, **api-vault-share**) which call persistence services (**backend-auth**, **backend-vault-crypto**, **backend-vault-file**, **backend-vault-share**) over internal RPC. Infrastructure comprises MongoDB (primary document store), Redis (caching, sessions, rotation locks), and S3-compatible object storage for encrypted chunks, accessed by clients through short-lived presigned URLs.

4.4 The Zero-Knowledge Boundary

The “does observe” row is deliberate honesty: zero-knowledge refers to *content and keys*, not to all metadata. Section 8 describes what the vault synchronization protocol withholds from this metadata — folder relationships — and Section 8.7 states what remains visible.

4.5 Authentication Layers

OXU Drive separates *user authentication* from *cryptographic request authorization*:

1. **OIDC + PKCE session layer**: users authenticate against the OXU identity provider using OpenID Connect with PKCE; the back end issues session tokens (JWT) and enforces workspace/organization permissions through an IAM integration. This layer controls *who may call the API*.

Table 4: What the server stores, verifies, and never sees

Category	Detail
Stores (opaque)	Encrypted chunks (S3); encrypted manifest metadata (<code>metadata_encrypted</code>); ML-KEM-encapsulated DEKs (<code>encap_dek_by_vault</code> , <code>encap_cipher_text_by_vault</code>); encrypted slot records and slot hashes; encrypted share records; encrypted recovery blobs (Section 12.3).
Verifies	Ed25519 signatures on state-changing requests against registered public keys and key versions (Section 9); OIDC session tokens and IAM permissions; rotation state transitions; ownership of a shared space and the per-recipient share limit (Section 10).
Never sees	Plaintext file content or names; passwords or PINs; <code>user_seed</code> , <code>vault_seed</code> , <code>slot_key</code> , DEKs, or any decryption key.
Does observe (metadata)	Space and record identifiers, so which records form each space; per-space activity — object sizes, counts, timestamps, version chains, and access patterns; registered public keys per space; presence of recovery blobs. For <i>current vault items</i> the parent identifier is encrypted (Section 8). For the Drive, and for items written before the current protocol, the plaintext manifest graph keys (<code>file_id</code> , <code>parent_id</code>) remain visible and hence the folder-graph <i>shape</i> . See Sections 8.7 and 13.3.

2. **Ed25519 request-signing layer:** state-changing vault operations additionally carry an Ed25519 signature produced by a key derived from the user’s or vault’s seed (Section 9). This layer proves *possession of the client-side key material* and is what the zero-knowledge design relies on — a compromised session token alone cannot forge vault operations. The signature binds a request to its group or space and to a time window, not to the specific operation or request body (Section 9.3).

4.6 FileProvider Key Sharing

The FileProvider extension needs vault keys to serve file operations, but runs out-of-process. Key handoff is platform-specific:

- **iOS:** the app and extension share an app-group Keychain; the vault key bundle for the FileProvider is stored with accessibility `afterFirstUnlockThisDeviceOnly` (not synchronized, available to background operation after first unlock). Seeds themselves use `whenUnlockedThisDeviceOnly`.
- **macOS:** the app writes a one-shot FileProvider key bundle to the shared app-group Keychain before domain registration; the extension loads that bundle at startup, applies the keys to its runtime store, and deletes the Keychain item. After bootstrap, keys are served from the runtime store and cached in memory with a 5-minute TTL; if the runtime key is absent or expired, the extension must be re-provisioned by the app.

Keychain items are namespaced per user, workspace, and vault: each seed entry is stored under the SHA-256 hash of an app-internal identifier derived from the user, workspace, and vault name, preventing cross-vault key confusion.

5 Key Hierarchy and Management

5.1 Key Objects

Table 5 names the key objects used throughout this paper, with their derivation sections and storage locations.

Table 5: Named Key Objects

Key	Description	Storage
<code>user_seed</code>	64-byte master seed, unique per user across devices; derives the user ML-KEM keypair and user signing key	Keychain
<code>user_recovery_key</code>	Hex export of <code>user_seed</code> ; re-importable on a new device	User-held
<code>vault_seed</code>	64-byte seed, unique per vault (all spaces of a vault share it); derives group keypairs	Keychain
<code>vault_recovery_key</code>	Encrypted recovery object for a vault (Section 12.3)	User-held / server
<code>pwd_stretched</code>	<i>Argon2id</i> (password, <code>vault_salt</code>); never stored	RAM only
<code>slot_key</code>	Per-space AES-256 key (Section 5.2); not persisted by the client, but contained in the recovery object when escrow is enabled (Section 12.3)	RAM; recovery object if escrowed
<code>encap_seed</code>	Per-space ML-KEM-768 seed; not persisted by the client except in the FileProvider key bundle while a space is unlocked (Section 4.6) and in the recovery object when escrow is enabled. The ML-KEM private key derived from it is stored in the space's key database	RAM; FP bundle while unlocked; recovery object if escrowed; derived private key in <code>slot-info.db</code>
space signing key	Ed25519 keypair derived from <code>encap_seed</code> (Section 9.2); the private key is stored with the ML-KEM private key in the space's SQLCipher-encrypted key database	<code>slot-info.db</code>
<code>slot_db_key</code>	Random 32-byte SQLCipher key inside the encrypted slot record	Slot record
group signing key	Ed25519 keypair derived from <code>vault_seed</code> (Section 9.2)	Derived on demand
root DEK	Per-space random 256-bit key under which item DEKs are wrapped (Section 5.4); held in the FileProvider key bundle for shared spaces while unlocked	Slot row (encapsulated); FP bundle for shared spaces
DEK	Per-file random 256-bit key, wrapped under the root DEK (or, for legacy and fallback items, the parent DEK) and ML-KEM-encapsulated; stored in the clear inside the space's data database for every item the device has opened	Encrypted in manifest; <code>slot-data.db</code> for opened items

5.2 Key Derivation Flow

Key derivation combines Argon2id password stretching with domain-separated HKDF. The password is always stretched before entering the fast KDF, and every derived key uses a distinct HKDF info label with `vault_salt` supplied through the HKDF salt parameter, so sibling keys are cryptographically independent output streams (Figure 2):

$$\begin{aligned}
\text{pwd_stretched} &= \text{Argon2id}(\text{password}, \text{vault_salt}, 64 \text{ B}) & (m = 64 \text{ MiB}, t = 3, p = 1) & \quad (1) \\
\text{ikm} &= \text{vault_seed} \parallel \text{pwd_stretched} & & \quad (2) \\
\text{slot_key} &= \text{HKDF-SHA512}(\text{ikm}, \text{salt} = \text{vault_salt}, \text{info} = \text{"v2"}, 32 \text{ B}) & & \quad (3) \\
\text{encap_seed} &= \text{HKDF-SHA512}(\text{ikm}, \text{salt} = \text{vault_salt}, \text{info} = \text{"encap-seed-v2"}, 64 \text{ B}) & & \quad (4)
\end{aligned}$$

Spaces created before this derivation was introduced are unlocked through the previous, unstretched derivation until they are re-keyed; the client tries the stretched derivation first and falls back only for such legacy spaces.

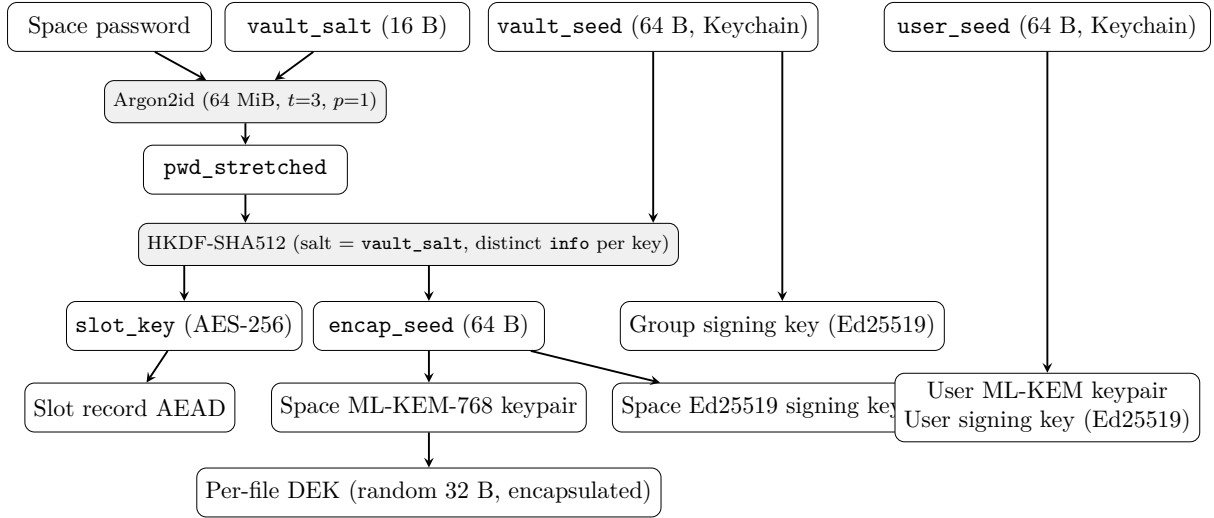


Figure 2: Key hierarchy. The space password is stretched with Argon2id before entering HKDF; **slot_key** and **encap_seed** are independent HKDF output streams separated by distinct **info** labels. **vault_seed** both feeds the HKDF input material and independently derives the vault-wide group signing key; **user_seed** (right) derives the user’s own ML-KEM and signing keypairs.

Security rationale. The Argon2id step imposes a 64 MiB, 3-iteration cost on every password guess, raising the cost of high-rate GPU/ASIC guessing against a seized device. The **vault_seed**, held in the platform Keychain and never derivable from the encrypted vault file, adds a second factor: exfiltrating the encrypted vault file alone does not enable an offline password attack, although the seed itself travels to the user’s other devices through transfer or escrow (Section 12). Distinct **info** labels guarantee that no derived key is a prefix or function of a sibling key (RFC 5869 §3.2 domain separation); the full label registry is in Appendix B.

5.3 Per-Space Keypairs

From **encap_seed**, each space deterministically derives:

- its **ML-KEM-768 keypair** (encapsulation of file DEKs, Section 5.4), and
- its **Ed25519 signing keypair** (Section 9) via

$$\text{HKDF}(\text{encap_seed}, \text{salt} = \text{vault_salt}, \text{info} = \text{"ed25519-signing-key"}, 32 \text{ B}).$$

Because both are derived from credentials, neither private key needs to be stored: unlocking a space regenerates them.

5.4 Post-Quantum Encapsulation of DEKs

Each space has one *root DEK*, a random 32-byte key encapsulated to the space’s ML-KEM-768 public key:

1. Generate a random 32-byte root DEK.
2. $(ct, ss) \leftarrow \text{ML-KEM-768.Encapsulate}(pk_{\text{space}})$.
3. Wrap the root DEK under the shared secret:
 $\text{encap_dek_by_vault} = \text{AES-256-GCM.Encrypt}(\text{root DEK}, \text{key} = ss)$.
4. Store $(ct, \text{encap_dek_by_vault})$ with the space’s slot row (Section 6.1); discard ss and the plaintext key from memory after use.

Unlocking regenerates the space keypair from credentials, decapsulates ct to recover ss , and unwraps the root DEK. Note that the shared secret is used directly as the AES-256 wrapping key; it is a uniformly random 32-byte KEM output used for exactly one wrap operation.

Vault items: flat wrapping under the root. Every file and folder in a vault carries its own random DEK. Under the current protocol that DEK is wrapped *directly* under the space’s root DEK, at every depth of the folder tree:

$$\text{dek_encrypted} = \text{AES-256-GCM.Encrypt}(\text{DEK}_{\text{item}}, \text{key} = \text{root DEK}).$$

Opening any item therefore costs one unwrap under the root, independent of its position, and no stored key relationship links an item to its parent. This flatness is what allows the vault synchronization protocol to omit parent identifiers from manifests without the key material reproducing the tree (Section 8); it also means that a vault share is a grant of the whole space (Section 10) and that moving an item already wrapped under the root rewrites no key material. Two kinds of item are not root-wrapped: items written before root wrapping, which keep their parent wrap until they are rewrapped, and items created by a current client — owner’s or recipient’s — that could not reach the root at creation time and fell back to a parent wrap. Section 8.8 states the consequence.

Two stored forms of each item key. A vault manifest carries its item DEK in two forms: a symmetric wrap (dek_encrypted) — under the space root for items created or rewrapped with the root available, under the parent’s DEK for the legacy and fallback items just described — and separately an encapsulation to the space’s ML-KEM-768 public key, exactly as the root itself is ($\text{encap_dek_by_vault}$, $\text{encap_cipher_text_by_vault}$). The root wrap is the primary open path — one symmetric unwrap per item. The encapsulation is the fallback path: it opens items written before root wrapping was introduced, and any item whose root wrap cannot be used (Section 7.5). A rotation issues a fresh item DEK in both forms (Section 11); the encapsulation keypair changes only when the space’s password changes.

Drive items: hierarchical wrapping. In the Drive, child DEKs are wrapped under their parent folder’s DEK ($\text{encrypted_dek_by_parent}$), so a folder DEK doubles as a *key-encryption key* (KEK) for its subtree and a folder share transmits one wrapped key covering all descendants; Figure 3 shows this structure. It applies to Drive file and folder sharing only.

Traversal cost. In a vault, opening n items costs n AES-256-GCM unwraps under the cached root DEK after the one ML-KEM decapsulation performed at unlock; there is no depth term, and the work is independent of tree shape. Along the Drive’s parent-wrapped path, key material

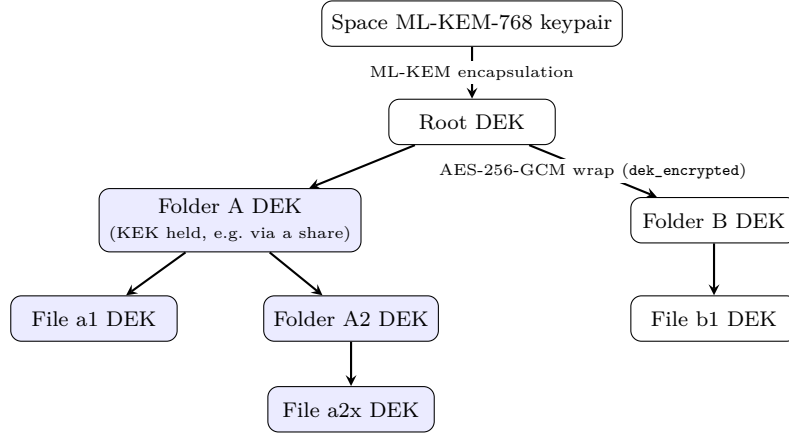


Figure 3: Hierarchical DEK wrapping in the Drive. Every manifest node carries its own DEK; each child DEK is stored wrapped under its parent’s DEK, and the root DEK is encapsulated to the owner’s ML-KEM-768 public key. A party holding Folder A’s DEK can unwrap exactly the shaded subtree — never siblings or ancestors. Vault items are *not* chained this way: each is wrapped directly under the space root.

resolves top-down, one unwrap of a wrapped 32-byte key per edge: a party holding a folder DEK obtained through a share opens a descendant at relative depth d in

$$T_{\text{open}}(d) = d \cdot T_{\text{unwrap}} + T_{\text{content}}, \quad (5)$$

where T_{unwrap} is on the order of microseconds (Table 10), so depth is effectively free there too, and enumerating a subtree of n nodes with cached parent DEKs is again n unwraps. The one-wrap-per-node structure is what makes a Drive subtree grant constant-size — sharing a folder transmits a single wrapped DEK no matter how many descendants it covers — while access is strictly one-directional: holding Folder A’s DEK yields no path to siblings or ancestors (Section 13.2).

5.5 Secure Storage Integration

Seeds are stored in the platform Keychain, namespaced by user, workspace, and vault: the account identifier for each seed entry is the SHA-256 hash of an app-internal string derived from the user, workspace, and vault name, so distinct vaults (or workspaces) never resolve to the same Keychain item, and the stored item name itself does not expose the vault name.

- Seeds (`user_seed`, `vault_seed`) use `whenUnlockedThisDeviceOnly`: never synchronized to iCloud, not restored onto a different device from a backup, available only while the device is unlocked.
- The FileProvider vault key bundle (iOS) uses `afterFirstUnlockThisDeviceOnly` to permit background file operations after first unlock (Section 4.6).
- Derived keys (`slot_key`, `encap_seed`, DEKs) are derived at unlock and held in memory. The copies that *are* persisted — the FileProvider key bundle while a space is unlocked, the per-space private keys inside the space’s encrypted key database, and the recovery object when escrow is enabled — are listed in Table 5 and Section 14.2.

6 Vault Key Storage and Local Databases

6.1 The Vault Key Database

Each vault has a dedicated SQLCipher-encrypted *vault key database* holding one row per space in a slot-configuration table. The database itself is opened with a per-user database key derived from `user_seed` via $HKDF(\text{user_seed}, \text{salt} = \text{app constant}, \text{info} = \text{"dbContext"})$; because `user_seed` lives only in the Keychain of the user's devices, the database opens only where that seed is present. This outer layer is *not* what protects space secrets: each row's slot record is independently encrypted under its space's password-derived `slot_key`, so opening the database reveals only uniform, undecryptable rows.

Each row carries (Appendix C):

- the **encrypted slot record**: a fixed 512-byte AES-256-GCM record — 12-byte nonce, ciphertext of the 484-byte padded plaintext, 16-byte tag (Section 6.2) — so every row's record has identical length;
- the 16-byte `vault_salt`, replicated identically on every row;
- the space's **encapsulated root-DEK material**: the AES-wrapped DEK and the ML-KEM-768 ciphertext (Section 5.4); and
- optionally, the encrypted recovery object (Section 12.3).

Row hygiene. The table is written defensively against forensic row analysis: row identifiers are random UUIDs; creation and modification timestamps are written as a fixed constant date, so they carry no activity information; and whenever the set of spaces changes (a row is added), the complete row set is deleted and re-inserted, shuffled, with fresh identifiers, while in-place updates to a row's contents preserve its random identifier — row order and identifiers therefore reveal nothing about which rows are old or new. Unchanged slot ciphertext, by contrast, is the same across snapshots, so a reader with two images can match rows whose contents did not change.

6.2 Slot Records

Each slot record is the AES-256-GCM encryption, under that space's `slot_key`, of a fixed 484-byte plaintext — stored as a fixed 512-byte record once the 12-byte nonce and 16-byte tag are added — containing: a record version; the random 32-byte `slot_db_key` (SQLCipher key for the space's databases); a locator blob (`PtrBlob`) pointing to the space's on-disk directory; a remote-slot key reference; and zero padding to the fixed record length. Fixed-size records ensure that a record's byte length reveals nothing about its contents.

Try-all-slots decryption. Unlocking derives `slot_key` from the entered password and attempts to decrypt the rows in turn; the first successful AEAD decryption yields that space's `slot_db_key` and path. Failed decryptions are indistinguishable from one another (AEAD authentication failure) and do not abort the scan, so a wrong password produces the same generic failure regardless of how many rows exist or which of them are real.

6.3 Integrity Verification

Integrity of the vault key database uses two authenticated layers:

- **Database pages:** SQLCipher authenticates every page with a per-page HMAC under the database key (Section 6.5); page-level tampering is detected on read.

- **Record authentication:** each slot record carries a 128-bit GCM authentication tag under `slot_key`; any tampering causes decryption failure, independently of the database layer.

6.4 Local Database Layout

Local metadata lives in SQLCipher-encrypted SQLite databases. For a vault, the layout is:

Vault key database Vault-level: the slot-configuration rows of Section 6.1, including each space’s encapsulated root DEK.

slot-info.db Per-space: slot configuration, key references, and the space’s key history.

slot-data.db Per-space: the decrypted manifests of the space — file and folder metadata with real parent identifiers, item DEKs, chunk references — and the space’s synchronization cursor; populated while the space is unlocked and opened with the space’s random `slot_db_key` (Section 6.2).

Encrypted chunk files are stored in the space’s data directory (Section 7.7). Further databases exist for account, Drive, and share bookkeeping; the list above describes the vault layout, not an inventory of every database on the device.

6.5 SQLCipher Configuration

Databases are opened with SQLCipher [10] using *library-default* page encryption (AES-256-CBC with per-page HMAC-SHA512 in SQLCipher 4). Two points deserve precision:

- **Raw pre-derived keys.** The application supplies each database key as a raw 32-byte value (`PRAGMA key = "x'...'"`), pre-derived from the OXU Drive key hierarchy. SQLCipher’s internal PBKDF2 password-stretching step therefore does not apply to the main key — by design, since the supplied key is already a full-entropy 256-bit value: the random `slot_db_key` for per-space databases, or the `user_seed`-derived database key for the vault key database (Section 6.1). Password stretching happens earlier, in the Argon2id stage of the key hierarchy.
- **Defaults, not tuning.** Page size, HMAC algorithm, and KDF parameters are SQLCipher-4 defaults; OXU Drive does not tune them. The security-relevant property is per-page authenticated encryption under a full-entropy key.

6.5.1 Cipher Mode Note

SQLCipher uses AES-256-CBC with per-page HMAC (encrypt-then-MAC) rather than AES-GCM. Both provide authenticated encryption; page-level encryption with a fresh random IV per page write and a per-page HMAC that covers the page number enables random access and detects a page moved to another location. It does not by itself detect replacement of a page or of the whole database by an older valid copy.

7 File Encryption, Integrity, and Synchronization

7.1 Per-File DEKs and Chunking

Each file is encrypted with its own random 256-bit DEK, wrapped under the space root DEK (vault) or its parent folder’s DEK (Drive) as in Section 5.4. Large files are split into chunks to enable streaming, random access, and incremental sync:

- **Chunk sizes:** 2 MB default; 50 MB in the FileProvider bulk path; 1 MB for background transfers.
- **Chunk encryption:** each chunk is encrypted directly with the file DEK using AES-256-GCM with a fresh random 96-bit nonce; the stored chunk is $[\text{nonce} \parallel \text{ciphertext} \parallel \text{tag}]$. There is no per-chunk key derivation; random nonces keep the collision probability under one DEK at about $q(q-1)/2^{97}$ for q encryptions (Section 14.3), and cross-file isolation comes from the per-file DEK.
- **Chunk naming:** on the device, encrypted chunks are stored under the item identifier and chunk index (Section 7.7), which is also the chunk’s authenticated context (Section 7.2.1); on the server they are stored under service-assigned object keys, and each chunk’s content hash is registered with its upload so the client can verify what the server recorded before finalizing a manifest.

7.2 File Integrity: BLAKE3 Merkle Trees

Integrity of file content is provided by a two-level BLAKE3 construction:

1. Per-chunk hash with index binding:

$$h_i = \text{BLAKE3}(\text{LE32}(i) \parallel \text{nonce}_i \parallel \text{ciphertext}_i \parallel \text{tag}_i),$$

where $\text{LE32}(i)$ is the chunk index as a 4-byte little-endian integer and the remainder is the stored chunk exactly as written (Section 7.1). Binding the index prevents reordering or substituting chunks within a file.

2. **Merkle root:** the chunk hashes, in index order, are combined level by level, each parent being $\text{BLAKE3}(\text{left} \parallel \text{right})$ of its two children; an odd trailing node is paired with a copy of itself, and a single-chunk file’s root is its chunk hash. The 32-byte root is stored in the file manifest and, when the writer’s user signing key is available, signed with it (Section 9.2). The signed message contains the root only: the file identifier, version, and chunk count are associated with the root by the manifest record, not by the signature, and a manifest without a signature is still accepted — the signature is an integrity attestation by the writer, not a condition of acceptance.

When the synchronization engine downloads a file’s chunks it recomputes each chunk hash and records it. On the signed download path — a manifest that carries the writer’s signing object — it then verifies the Merkle root of the received chunks against the signed root, and a mismatch marks the file as failed. A download without a signing object — the signature being optional, as noted above, or the download resuming a waiting file from the local database — skips that final comparison, including the comparison against the root stored on the manifest, and in every case the chunk files and their records are written before the check runs. When a stored file is later materialized for the file manager from the local chunk store, the chunk hashes are recomputed again but the Merkle root is not, and a chunk that fails to read, authenticate, or decrypt at that stage is skipped rather than propagated as a failure of the whole file. The implementation therefore does not guarantee rejection of a corrupted file across all paths: rejection is enforced for signed downloads at download time, and nowhere else. Substituting a complete older manifest together with its own consistent chunks is not detected by the hash tree; that case belongs to the version chain and to the authenticated context of Section 7.2.1.

7.2.1 AEAD Context Binding (AAD)

Each content chunk is bound to its identity through the AEAD’s associated data: the UTF-8 encoding of the chunk’s stored name,

$$\text{AAD} = \text{“}\langle \text{item id} \rangle_ \text{chunk}_ \langle i \rangle\text{”},$$

where the item identifier is the client-generated identifier of the manifest the chunk belongs to and i is the chunk index. Owner and recipient store the chunk under that name, so both derive byte-identical associated data without reconstructing the index from row order. A chunk moved to another item or another position therefore fails authentication on read, complementing the Merkle-tree layer of Section 7.2. Chunks written before this binding was introduced carry no associated data; on read the client tries the bound form first and falls back to the unbound form, which does not weaken a bound chunk (a relocated bound chunk fails both), and a key rotation re-encrypts every chunk in the bound form. The binding covers content chunks only: encrypted metadata and wrapped keys carry no associated data, and replacing a complete older manifest together with its own consistently named chunks is not detected by this layer — that case belongs to the version chain. The associated data carries no space label beyond what the item identifier already implies.

7.3 Upload and Download Pipeline

1. **Draft:** the client registers an encrypted manifest (`POST /v1/manifest/draft`) carrying the encrypted metadata, the encapsulated DEK material, key versions, and an Ed25519 signature.
2. **Chunk upload:** the client requests short-lived S3 presigned URLs (`generate-url`) and uploads encrypted chunks directly to object storage; the service never proxies plaintext or holds keys.
3. **Submit:** the manifest is finalized (`PUT /v1/manifest/submit`), making the version active.

Downloads reverse the pipeline: pull manifests (delta sync by change timestamp), fetch chunks for changed manifests via presigned URLs, verify hashes, then decrypt locally. Unchanged, fully pulled manifests are skipped.

7.4 Synchronization Model

Two synchronization relationships exist. Both are mediated exclusively by the zero-knowledge back end — devices never communicate directly, and the server orders and authorizes updates without being able to read them.

Same user, multiple devices. Every device of a user holds the same seed material (`user_seed` imported at setup; `vault_seed` obtained through device transfer or recovery, Section 12) and therefore derives identical keys. Each unlocked space synchronizes its own manifests: a device pushes that space’s changes with the draft/upload/submit pipeline of Section 7.3, signed with the space’s signing key (Section 9.2), and the others pull the space’s deltas — manifests changed since their last synchronization cursor — and open them locally. Client-generated version identifiers, checked against the prior version when a draft is created, provide optimistic-concurrency ordering. The server verifies signatures and refuses ordinary drafts and delta pulls for a space while a rotation is registered (Section 11). Vault-level state (the space registry, recovery blobs, share records) converges through the vault-list synchronization of Section 12. Every synchronization request therefore names the space it concerns; what the request does not carry, for

vault items, is the item’s position in the folder tree (Section 8). For the Drive, records carry plaintext identifier graph keys as well.

Across users (shared content). Sharing never re-uploads content: a recipient synchronizes against the *owner’s* records through share-scoped endpoints, authorized by the share record (Section 10) and the recipient’s own signing keys — never by the owner’s vault credentials. For a vault share the recipient pulls the shared space’s manifests by version change, with a persisted cursor, and opens them under the shared root exactly as the owner does; for a Drive share the recipient pulls the shared subtree and unwraps down the DEK tree (Figure 3) from the shared node. Revocation and expiry propagate through the same channel: the share record flips to **REVOKED** on the server, recipient devices honor it on their next sync, and the owner’s rotation re-encrypts the space’s content (Sections 10.4 and 11.4).

7.5 Pull and Parent Resolution

Vault manifests are pulled per space (Section 7.4). The pull runs while the space is unlocked in the session and proceeds as follows:

1. **Cursor.** The client requests the space’s manifests — active and deleted alike — changed since a timestamp cursor, in pages, and advances the cursor as pages complete. A device that does not yet hold the space walks all of it.
2. **Open.** For each row the client unwraps `dek_encrypted` under the space’s root DEK (falling back to decapsulating the row’s encapsulated key for records written before root wrapping was introduced) and decrypts the metadata under the item DEK.
3. **Parent resolution.** The item’s real parent identifier is read from the decrypted metadata. The wire value of the parent field is used only for records written by the original protocol, whose parent field is real; for current records it is the sentinel and carries nothing. If the parent is not yet held locally, the client fetches it from the server *by identifier*, decrypts it, and continues upward until it reaches the root of the space or an ancestor already present, then persists the chain top-down. The local database always stores the real parent, never the sentinel.
4. **Chunks.** Content chunks are fetched lazily per file, through presigned URLs, into the space’s data directory (Section 7.7), verified against their recorded hashes, and decrypted with the item DEK under the chunk’s authenticated context (Section 7.2.1).

Deletions are new manifest documents with a deleted status and no chunks; the space that opens them removes the projected item.

Two consequences are stated here because they are visible to the operator. Every pull names the space and shows when that space is active on a device; and resolving a hidden ancestor fetches it by identifier immediately after its child arrived, so on a device that does not yet hold the ancestor the parent–child edge is revealed by the request sequence at that moment, even though no stored field carries it (Section 8.7). A device that already holds the ancestor performs no such fetch.

7.6 Push

Pending vault manifests are pushed per space through the draft/upload/submit pipeline of Section 7.3: each item is drafted with the space identifier, the sentinel parent field, the encrypted metadata (which carries the real parent), the item DEK in its root-wrapped and encapsulated forms, and the space’s key version; its chunks are uploaded; and the manifest is submitted. Items are pushed in the order they were queued; no shuffling or timing jitter is applied.

7.7 Vault Data Directory

Encrypted chunks are stored per space, named by item identifier and chunk index:

```
<group-container>/<workspace>/<vault-hash>/<space>/Data/<item-id>_chunk_<i>.enc
```

The chunk file name is also the context bound into the chunk’s authenticated encryption (Section 7.2.1), so owner and recipient derive the same context from the stored name. Each space keeps its two database files (`slot-info.db`, `slot-data.db`) and its data directory in its own subdirectory; these exist and grow only for spaces that have been unlocked on the device.

8 Metadata Privacy

Content and names are encrypted end to end (Section 7). This section is about what the *synchronization records* of a vault reveal beyond that. For records written in the current format, the stored manifest omits the explicit parent identifier, so the service operator — and any party holding its databases — cannot read from stored fields which item sits in which folder. That omission does not prevent relationship inference from request history, timing, or legacy records, and the section says where it does not. The protocol is *not* designed to hide which space a record belongs to, how many spaces a vault has, or how active they are, and it provides no deniability of hidden content to a user under coercion. The section states the model (Section 8.1), the claim and its exclusions, why the exclusions are necessary, the mechanisms, their cost, the observation channels (Section 8.6), and what remains observable.

8.1 Model and Terminology

Vault A vault identity with exactly one `vault_seed` (Keychain-resident) and a registered group identifier derived from it (Section 9.2), used for vault-level operations.

Space (slot) An access path into a vault: its own password, its own derived keys (Section 5), its own root DEK (Section 5.4), and its own registered identifier and public keys on the server. Every manifest, chunk record, upload record, and share record of the space carries that identifier. The user creates spaces explicitly; each creation is a signed, server-visible registry event.

Manifest The synchronization record of one file or folder version: the item’s metadata — name, *parent identifier*, attributes — encrypted under the item DEK (`metadata_encrypted`); the item DEK wrapped under the space root (`dek_encrypted`) and encapsulated to the space public key; integrity fields; and plaintext bookkeeping — the space identifier, record and file identifiers, the sentinel parent field, timestamps, status, key version, and the version chain.

Observers The honest-but-curious operator of Section 2.1 sees every record of every space, request metadata, and logs. The physical-access adversary holds the device image including the Keychain seeds. The coercion adversary additionally obtains one or more space passwords.

8.2 The Claim — and What Is Not Claimed

For vault items written by the current protocol (Section 1.1), the design provides the following.

1. **Encrypted folder relationships.** A vault manifest’s parent identifier travels only inside `metadata_encrypted`. The top-level parent field carries a fixed sentinel value on the wire

and in storage, for every item at every depth. The operator cannot read the parent–child edges of current vault items from stored fields.

2. **Key material that encodes no structure.** Every item created or rewrapped with the root available has its DEK wrapped directly under the space root (Section 5.4). No stored wrap links such an item to its parent, and moving it does not rewrite any key material, so the wrapping layer does not reproduce the tree that the parent field hides. An item written before root wrapping, or created by a client that fell back to a parent wrap because the root was unavailable, keeps its parent wrap until it is rewrapped, and the claim does not cover it (Section 8.8).

Table 6 states the paths on which the claim holds.

Table 6: Scope of the metadata claim by synchronization path

Path	Parent en- cryptd	Space identi- fied	Notes
macOS and iOS clients, vault items written by the current protocol	yes	yes	the claim
Recipient of a shared space, current protocol	yes	yes	writes are user-signed; the share record names the space
Vault items written before the current protocol	no	yes	real parent identifiers retained server-side; historical mappings persist
Drive	no	n/a	plaintext identifier graph keys by design
Older clients	no	yes	the server accepts and serves any parent value

Explicitly **not** claimed:

- **Which space a record belongs to is not hidden.** Every manifest, chunk and upload record, share record, and request names its space by its registered identifier and carries the space’s key version; manifests are signed with the space’s key. The operator knows exactly which records form each space.
- **The number and activity of spaces are not hidden.** Creating a space is a signed registry event; every space has its own registry entry, public keys, and settings; each space synchronizes separately, so the operator sees how many spaces a vault has, when each is added, and when each is active.
- **No deniability of hidden content.** A coercer holding the device and one password sees the other spaces listed in the registry and their records on the server; nothing about a withheld space is deniable (Section 8.3).
- **Structural inference beyond parent edges is not prevented.** Stable file identifiers and version chains, ciphertext lengths, timing, and — on a device that does not yet hold an ancestor — the by-identifier fetch that resolves a hidden parent still support inference about structure (Section 8.7).
- **Legacy items and the Drive are out of scope.** Items written before the current protocol carry plaintext parent identifiers, and an operator’s historical mappings are not erased by later rewriting; the Drive synchronizes with plaintext identifier graph keys (Section 4.4).

- **Not a schema guarantee.** The sentinel is client behavior. The server stores and serves whatever parent value a client sends; an older client populates the field.
- **No volume, timing, or padding protection.** Space sizes, object counts, and request timing are visible; nothing is padded (Section 8.9).

8.3 Analysis

The exclusions above are not caution; each follows from the design.

Why nothing about a space is deniable. Every record of a space is labelled with the space’s identifier, on the server and in the device’s registry. A coercer who holds the device and the password of one space sees, without any cryptography, that the vault has N registered spaces and which records belong to each of them; the adversary of Section 2.1 needs no trial decryption to attribute a withheld space’s records, only to read its content. The design therefore makes no claim about what a user can deny; it limits the coercer to the content of the passwords actually revealed. Making attribution itself hard would require records that carry no space label and are attributed only on the device — a different synchronization protocol, listed as a research direction in Section 16.2.

Creation-event leakage. Adding a space is a signed registry event; the count of registry entries per vault is the number of spaces. The design claims nothing about hiding it.

What encrypting the parent hides and does not hide. With the parent field encrypted, the operator’s stored view of a space is a flat set of records: it cannot tell from stored fields which items are folders’ children, how deep the tree is, or how items are grouped. It can still count items, see their sizes and how often each changes, and observe timing. One dynamic channel remains: a device that receives a child whose parent it does not yet hold fetches that parent by identifier in the same synchronization pass, so an operator watching 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. Renames and moves produce new manifest versions with unchanged key material, which reveals that an item changed, not where it moved.

8.4 Mechanisms

Figure 4 summarizes how the mechanisms fit together.

8.4.1 Sentinel Parent Field

On every vault push (Section 7.6) — owner, recipient, and rotation — the top-level parent field of a manifest is set to a fixed sentinel value, for every item at every depth. The same record layout is used for every vault manifest, so the sentinel itself carries no information.

8.4.2 Parent Inside the Encrypted Metadata

The real parent identifier is a field of the metadata object that is encrypted under the item DEK, next to the item’s name and attributes. It is decrypted only on a device holding the space’s keys, and the local database stores the real value.

8.4.3 On-Device Resolution with Ancestor Climb

When a pulled item's parent is not held locally, the client fetches the parent by identifier, decrypts its metadata to find the next parent, and repeats until it reaches the space root or a known ancestor, persisting the chain top-down (Section 7.5). Records written by the original protocol, whose parent field is real, are resolved from the wire value; no separate migration rewrites them.

8.4.4 Flat Root Wrapping

Every item DEK is wrapped directly under the space root (Section 5.4). A parent-chained wrap would require the parent to be known when an item is written and would change on every move; the flat wrap makes item keys independent of position, so neither the wrap nor a move reveals structure. Records written before root wrapping are opened through their encapsulated key, and a recipient falls back to the parent DEK for such records.

8.4.5 Rotation and Deletion

Key rotation re-encrypts every item of a space under fresh DEKs wrapped under the new root and pushes the rewritten manifests with the sentinel parent field (Section 11). Deleting a vault removes every space registered under its group identifier (Section 12.5).

8.4.6 Compatibility

The server requires only that the parent field be present; it neither validates nor blanks its value. Older clients, and the Drive, keep sending real values; current clients read those values when they meet them.

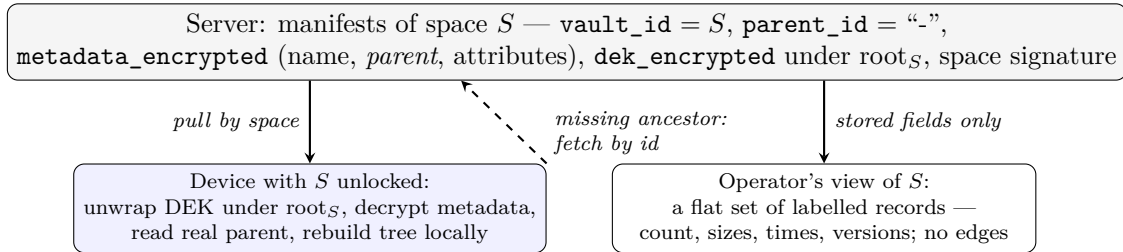


Figure 4: Folder-structure privacy for a vault space. The server holds labelled, per-space records whose parent field is a sentinel; the stored fields do not reproduce the tree, which is rebuilt after decryption on a device that holds the space's keys. The dashed path is the one dynamic leak: an ancestor fetched by identifier when it is not yet held locally.

8.5 Cost

Encrypting the parent adds little to storage and bandwidth: the sentinel replaces a value of the same field, the parent identifier inside the encrypted metadata adds a few dozen bytes per manifest, and the ancestor fetches of Section 7.5 add requests on a device's first walk of a space. Opening an item costs one AES-256-GCM unwrap under the cached root DEK instead of a walk down parent wraps, and a move of a root-wrapped item rewrites no key material. The remaining cost is on-device: an ancestor that is missing locally when its descendant arrives is fetched and decrypted, and a device repeats such fetches when its local store is rebuilt or a fetch is retried. These costs were not measured for this document; the figures of Table 10 indicate the per-operation constants.

8.6 Observation Channels

Table 7 lists the channels through which an operator or a forensic reader could learn structure or attribution, and what the design does about each. Only the first two rows are claims; the rest are residuals that the design accepts.

Table 7: Observation channels and their status

Channel (observer)	Design response	Status
Parent-child edges of current vault items, from stored fields (operator)	Parent identifier encrypted; top-level field is a sentinel	Claimed (current items)
Tree structure reproduced by key wrapping (operator)	Flat wrap under the space root; moves rewrite no key material	Claimed
Which space a record belongs to (operator)	None: the space identifier and key version are on every record and request	Visible by design
Number of spaces, their creation, their activity (operator)	None: per-space registry entries and per-space synchronization	Visible by design
Parent edge revealed by an ancestor fetched by identifier (operator)	None: fetch happens only when the ancestor is not yet held locally	Residual (Section 8.7)
Item counts, sizes, version chains, timing (operator)	None: no padding or cover traffic	Residual
Per-space databases and data directory on disk (forensic reader)	Exist and grow only for spaces unlocked on the device	Residual
Legacy vault rows and Drive rows (operator)	None: plaintext parent identifiers	Residual

8.7 Explicit Residual: Structure Inference Beyond Parent Edges

Encrypting the parent identifier removes the operator’s stored view of the folder graph; it does not remove every source of structural information. A file identifier is stable across versions, so an operator links a file’s manifests over time and sees how often and how much it changes; superseded versions are archived before deletion, and version chains are visible. Ciphertext lengths reflect metadata and content sizes. The ancestor fetch of Section 7.5 reveals an edge whenever a device resolves a parent it does not yet hold — on a device’s first walk of a space, potentially many edges in sequence. Every record carries the writing user’s key version and, for files, the writer’s integrity signature over the content root, so the operator sees which account wrote each record. And the operator’s records of legacy vault items — rows written with plaintext parent identifiers before the current protocol — are not erased when those items are rewritten; whatever mapping was learned then persists. Hiding these channels would require padding, oblivious access, or prefetching ancestors without revealing which are missing, none of which is offered (Section 16.2).

8.8 Known Limits

The following limits are part of the specification and are accepted explicitly.

1. **Space identity is visible.** Every record and request names its space. There is no protection against an operator or coercer learning which records belong to which space,

or how many spaces exist.

2. **No coercion deniability.** A user who reveals M real passwords cannot deny the existence or the extent of the remaining spaces; the registry lists them and the server holds their labelled records (Section 8.3).
3. **Ancestor-fetch channel.** Resolving a hidden parent that is not yet held locally fetches it by identifier, revealing the edge to an operator who observes request order.
4. **Legacy items rewritten without rewrapping.** An item written before root wrapping keeps its parent wrap. When the current client rewrites such an item — a move or a rename — it hides the parent on the wire without rewrapping the item key under the root, and both the owner’s and a recipient’s client fall back to a parent wrap when the root is unavailable at creation and push the sentinel all the same. The owner still opens such an item through its encapsulated key; a recipient who does not already hold the item key cannot, because the recipient’s fallback needs the plaintext parent that the sentinel now hides. Rewrapping under the root — which a key rotation does for every item — restores access.
5. **vault_seed compromise.** Whoever extracts `vault_seed` (Keychain extraction, memory dump at unlock) can derive the group signing key and perform vault-level operations — rename, password reset, deletion — but decrypts no content, which needs a space password as well.
6. **Stable-identifier and lifecycle channels.** File identifiers, version chains, and signer identity link records to each other (Section 8.7).
7. **On-disk state of unlocked spaces.** A space’s databases and data directory are created when the space is first unlocked on a device and grow with use; a forensic reader learns which spaces have been used on that device and roughly how much.
8. **Memory disclosure at unlock.** While a space is unlocked, its keys are in RAM; a same-time memory dump reveals them. This is the standard limit of every encrypted-storage application.
9. **Backup and cloud-container exposure.** Backups of the group container hand the forensic problem to whoever holds the backup, with the same properties and the same limits.
10. **Client paths outside the verified set.** The sentinel is client behavior. Older clients send real parent identifiers, the server accepts and serves them, and the Drive is unchanged.

Interaction with recovery escrow

Recovery-key escrow (Section 12.3) is a per-space decision. If some spaces of a vault have server-side recovery blobs and others do not, the presence pattern of those blobs is per-space metadata visible to the operator alongside the registry entries themselves. This interaction is a known property of the current escrow design (Section 12.3).

8.9 Explicitly Not Offered

The following are **not offered** by the current design; they are listed so that readers do not assume them:

- **Hidden volumes / duress passwords** (“the data does not exist” deniability) — not offered.

- **Hiding which space a record belongs to** — not offered; group-scoped synchronization with on-device attribution is a research direction (Section 16.2).
- **Automatically generated decoy spaces or decoy manifests** — not offered; a research direction, and not by itself sufficient for deniability.
- **Fixed-size or padded containers, chunk padding, cover traffic** — not offered; an observer learns each space’s volume envelope and activity.
- **Hiding the number of spaces** — not offered; creation is a visible registry event.

9 Digital Signatures and Request Authentication

9.1 Ed25519 Signing with SHA-256 Prehash

All request and integrity signatures use Ed25519 over the SHA-256 hash of the message:

Algorithm 1 Signing

Require: *data*, *sk* (Ed25519 private key)

Ensure: *sig* (64 bytes)

- 1: *hash* $\leftarrow$ SHA256(*data*)
 - 2: *sig* $\leftarrow$ Ed25519.Sign(*hash*, *sk*)
 - 3: **return** *sig*
-

Prehashing allows signing large payloads without buffering them. The construction is plain Ed25519 over the SHA-256 digest of the message; it is *not* the Ed25519ph variant of RFC 8032, which uses SHA-512 and domain separation, and it is not interoperable with it. The client signs the exact UTF-8 message string carried in the signing object (typically compact JSON containing `time` plus a request-specific field such as `vault_id` or `checksum`); the server hashes and verifies that same message string.

9.2 Signing Keys and Signing-Object Types

The system uses three Ed25519 keys, each deterministically derived from a seed. The user and group keys are derived on demand; the space key is derived when the space is created and stored, with the space’s ML-KEM private key, in the space’s encrypted key database (Section 14.2):

A manifest operation is signed by the key of the space it concerns; the signed message names the space, and the server verifies the signature against that space’s registered public key at the key version the request carries. The per-space key version (`vault_key_version`, the SHA-256 of the space’s encapsulation public key) changes when the space’s encapsulation keypair changes — on password change or reset. Superseded versions remain registered, and the verifier accepts any registered version (Section 9.3). Alongside the space keys, the vault registers a group ML-KEM public key derived from `vault_seed`; the SHA-256 of that key is the vault’s stable group identifier (`group_key_version`), fixed at creation and unaffected by password changes, root rotations, and space additions.

9.3 Signing-Object Structure and Verification

Each signed API request carries a signing object. The key-version field is object-type specific: `vault_key_version`, `group_key_version`, or `user_key_version`.

The server recomputes the message hash, verifies the Ed25519 signature against the registered public key, and checks that `vault_key_version`, `group_key_version`, or `user_key_version` names a registered key of that space, vault, or user. For a space the registry keeps every key

Table 8: Signing keys and signing-object types

Object type	Key derivation	Used for
User	from <code>user_seed</code> (Appendix B)	user-scoped API requests; a recipient’s share-scoped reads and writes (Section 10); signing the BLAKE3 Merkle root of each file (Section 7.2)
Space	from the space’s <code>encap_seed</code> via HKDF (<code>info</code> = “ed25519-signing-key”); public key registered per space	every manifest operation of that space — draft, submit, abort, delta pull, chunk upload, rotation manifests — and space-scoped registry operations
Group	the 32-byte SHA-256 hash of <code>vault_seed</code> , used directly as the Ed25519 seed; public key registered once at vault creation	vault-level operations: vault registration, rename, password change and reset, vault deletion

```

1 {
2   "signing": {
3     "message":           "{ \"vault_id\": \"<space id>\", \"time
4     \"signature\":       \"<base64 Ed25519 signature over SHA256(message
5     \"vault_key_version\": \"<SHA-256 of the space encapsulation public
6   }
7 }
```

Listing 1: Signing object

version it has issued, and the verifier does not require the current one: a request signed under a keypair that a password change has superseded still verifies while that version remains registered. A password change therefore does not by itself cut off a holder of the previous space signing key at this layer, and the rotation that follows it (Section 11) re-encrypts content without removing the old registry row. For the vault the key version is the stable group identity (Section 9.2). The server also checks that the message’s `vault_id` matches the request’s space and that the timestamp is not older than a fixed window; there is no bound on timestamps in the future, no server-side record of consumed signatures, and no binding of the operation or request body. Within the window a captured signature therefore remains valid for requests to the same space, and Ed25519 being deterministic, identical messages yield identical signatures. Freshness here is an implementation parameter, not a cryptographic guarantee relied on elsewhere in this paper.

9.4 Relationship to Session Authentication

Signatures complement — and do not replace — the OIDC/PKCE session layer (Section 4.5). A valid session token authorizes API access; a valid signature proves possession of client-side key material. State-changing vault operations require both.

10 Sharing Cryptography

OXU Drive supports vault-level sharing — a whole space shared with a bounded set of recipients (five by default, a workspace-configurable client limit), with roles and expiry — and file/folder-level sharing in the Drive (which carries a server-enforced cap of five active recipients per shared item). Both rest on the same principle: key material is re-wrapped end-to-end to the recipient, and the server stores only wrapped material it cannot open. Vault sharing additionally wraps under a user-chosen share password (Section 10.1) and records the grant server-side (Section 10.2); Drive file sharing relies on the ML-KEM recipient wrap alone. Because vault wrapping is flat (Section 5.4), a vault share is a grant of the space’s root DEK and therefore of the whole space; Drive shares are subtree grants (Figure 3).

10.1 Two-Layer Key Wrapping

To share a space, the owner wraps its root DEK under *two* independent factors — a share password (communicated out of band) and the recipient’s post-quantum public key. A fresh random share salt is generated for every share:

$$\begin{aligned} \text{pw_share_32b} &= HKDF\text{-SHA512}(\text{Argon2id}(\text{pw_share}, \text{share_salt}, 64\text{ B}), \\ &\quad \text{salt} = \text{share_salt}, \text{info} = \text{"share-v2"}, 32\text{ B}) \end{aligned} \quad (6)$$

$$\text{share_dek_enc} = \text{AES-256-GCM.Encrypt}(\text{root_dek}, \text{pw_share_32b}) \quad (7)$$

$$(\text{ct}_{\text{share}}, \text{ss}) = \text{ML-KEM-768.Encapsulate}(pk_{\text{recipient}}) \quad (8)$$

$$\text{encap_dek_by_receiver} = \text{AES-256-GCM.Encrypt}(\text{share_dek_enc}, \text{ss}) \quad (9)$$

Recovering `root_dek` requires *both* the recipient’s private key (derived from their `user_seed`) and the share password. The server, holding `(ct_share, encap_dek_by_receiver, share_salt)`, can recover neither layer. The share password is stretched with Argon2id exactly as space passwords are (Section 3.3); the same derivation runs on both ends of a share, and there is no unstretched fallback.

10.2 Share Records and Authorization

The share request carries the wrapped material above, the identifier of the shared space, the recipient identifier and role, the expiry, an encrypted clone of the slot record (`slot`) whose `slot_db_key` is *re-randomized* — the recipient never learns the owner’s database key — and a client-computed share identifier

$$\text{unique_hash} = \text{SHA-256}(\text{receiver} \parallel \text{pw_share_32b} \parallel \text{share_salt} \parallel \text{group id}),$$

together with a signing object. Recipient lookup resolves the recipient’s registered public key through a workspace-membership-validated directory search, so shares are encapsulated only to keys registered within the workspace; the directory answer is authenticated only by the session layer and TLS, which are classical (Section 13.1). Before recording a share the server checks that the calling account owns the named space’s registry entry, validates the recipient’s workspace membership, rejects a resubmission of the same share, and enforces the per-recipient share limit. One record per shared space and recipient is a client-side rule; the share record names the shared space, so the operator sees which spaces of a vault a recipient holds.

Two properties of `unique_hash` are stated for completeness. First, every input other than `pw_share_32b` is visible to the operator, so the stored hash is an *offline verifier* for the share password: an operator can test a candidate password by running the share derivation with the public salt and comparing hashes, at the Argon2id cost per guess and without the recipient’s

private key. Learning the share password this way does not by itself yield `root_dek`, which remains under the recipient’s ML-KEM layer; but the share password should be chosen as a real secret, not as a convenience. Second, because the salt is fresh per share, equal passwords produce different hashes across shares; any policy against reusing a share password is therefore enforced by the client, not by the server.

10.3 Recipient Flow

Recipients delta-sync their share list. On unlock, the recipient decapsulates with their user private key, decrypts the password layer with the entered share password, and recovers `root_dek`. From there the recipient pulls the shared space’s manifests by version change (Section 7.5), unwraps each item DEK under the shared root — falling back to the parent DEK for items written before root wrapping whose parent field is still in the clear — decrypts the metadata, resolves parents exactly as the owner does, and stores the result in its own `slot-data.db` under a recipient-side database key. A legacy item that the owner has since rewritten under the current protocol without rewrapping it cannot be opened by a recipient who does not already hold its key (Section 8.8); a rotation of the space clears the condition. The recipient holds the ciphertext of the shared space only, and it remains on the recipient’s device until the share is removed locally.

A recipient with the `EDITOR` role writes through the same push path, signed with the recipient’s `user` key; the server authorizes each request by the active share record for that space and recipient, so a recipient’s authority is confined to the shared space. Recipient writes reuse the stored item DEK (Section 11.4). The owner’s `vault_seed` is never transmitted; a recipient gains the shared space’s root and nothing that signs as the owner.

10.4 Revocation and Expiry

Revoking a share (or letting it expire) flips the share record to `REVOKED`. From that point the recipient’s API access to the shared space ends: the server no longer serves the space’s records to that account. Recipient devices honor `REVOKED` on their next sync by deleting local share artifacts; a non-cooperating client is not bound by that deletion and keeps whatever it had synchronized. Presigned object URLs already issued remain usable until they expire; they are short-lived by configuration.

Because the recipient already holds `root_dek`, the sharing protocol prescribes a key rotation of the space after revocation (Section 11). In the current implementation the rotation is initiated by the owner rather than triggered by the revocation; the coupling that is wired runs in the other direction — after a rotation completes, the server requests revocation of all outstanding shares of the rotated space, processed asynchronously; a refused request is logged and does not fail the rotation (Section 11.2). A completed rotation is therefore not evidence that revocation was requested, let alone that every share record has been marked; an owner who rotates to cut off a former recipient should confirm the share list. What rotation does and does not achieve against a former recipient is stated in Section 11.4: it re-encrypts the space’s current content under keys the recipient never receives, and it does not retroactively protect content the recipient synchronized before.

11 Key Rotation

Key rotation re-encrypts an entire space under a fresh root DEK and fresh item DEKs. It is initiated explicitly by the user, or as the confirmation step offered after a password change or reset; the sharing protocol additionally prescribes it after share revocation (Section 10.4), where it is likewise owner-initiated in the current implementation. Section 11.1 states what a rotation

changes; Sections 11.2 and 11.3 give the protocol and its state machine; Section 11.4 states the cryptographic effect and its limits.

11.1 What Rotation Changes

A rotation of a space replaces:

- **the root DEK:** a fresh random root is generated and encapsulated to the space’s ML-KEM public key;
- **every item DEK:** each file and folder receives a fresh DEK, wrapped under the new root (Section 5.4) and encapsulated to the space public key;
- **every chunk:** each chunk is decrypted under the old DEK and re-encrypted under the new one with fresh nonces and the chunk’s authenticated context (Section 7.2.1), then uploaded again; and
- **every manifest:** the encrypted metadata is re-encrypted under the new item DEK and the manifest is pushed with the sentinel parent field (Section 8).

Unchanged are the space’s encapsulation keypair (unless the rotation follows a password change or reset), its signing key, its registered identifier, and the vault’s group identifier. A rotation uploads the space’s entire content volume once.

11.2 Protocol

1. **Prepare:** pull-sync the space to the latest state; cancel any existing rotation for it.
2. **New root DEK:** generate a random root and encapsulate it to the space public key with ML-KEM-768; register the rotation server-side (status **WAITING**). The server takes a rotation lock so only one device can rotate the space at a time.
3. **Queue phase:** walk the local tree from its roots, generating for every item a fresh DEK wrapped under the new root and encapsulated to the space public key, and queue the item.
4. **Sync phase:** in batches, create draft manifests marked as rotation manifests, re-encrypt each item’s chunks under its new DEK, upload them through presigned URLs, and submit. The marker keeps rotation manifests separate from the live set until commit.
5. **Commit:** when the client reports completion (status **COMPLETED**), the server queues a commit job, records the completed status, registers the new encapsulated root as the space’s current key record, and requests revocation of every outstanding share of the rotated space (Section 10.4). These are four separate steps, not one transaction: a completed status shows that the commit job was queued, not that the root was registered or revocation requested, and a completion request repeated after a failure at a later step is answered as already complete without redoing that step. The commit job promotes each rotation manifest in batches: it archives and removes the superseded manifest and its chunk records, inserts the replacement manifest, and moves the replacement chunk records into the live set — each a separate write without a surrounding transaction. The job does not check the outcome of those writes and reports success regardless. An interruption or a failed write can therefore leave a file with neither generation live, or with a live manifest whose chunk records are missing, and because the job may already have deleted the rotation record it would need, a repeat run is not guaranteed to repair the file. The chunk queries read at most 1,000 records per manifest without paging, so a file with

more chunk records than that is not fully promoted. The rotation lock is released when the commit job completes.

6. **Finalize:** the client records the new root in its slot configuration, clears its local data directory and manifest tables for the space, and pull-syncs the committed state.

11.3 State Machine and Safety

Rotation states are `WAITING` → `COMPLETED` (then the server-side commit job), with `PAUSED`, `FAILED`, and `CANCELLED` branches. A paused rotation resumes from its persisted queue; a failed or cancelled rotation is rolled back server-side — its rotation manifests are deleted — and recovery is a fresh rotation, not a resume.

While a rotation is registered, the space is read-only: ordinary manifest drafts for the space are refused by the server with an explicit rotation-in-progress error and by the FileProvider on the rotating device, version-based pull synchronization for the space is withheld, and browsing reads are served from the rows that were live before the rotation. Promotion is atomic neither across the space nor within a file (Section 11.2): a device that pulls during it can receive rows of both generations, or a file with no live row. An owner device opens new-generation rows through the encapsulated form of the item DEK — a pure rotation leaves the encapsulation keypair unchanged — until it learns the new root; a recipient device, which holds only the old root, cannot open them. Vault mutations (rename, password change, delete) are refused; share and revoke actions are removed from the client UI during rotation but are not additionally blocked by the share service itself.

11.4 Cryptographic Effect

After a rotation whose promotion has completed for every intended record — a condition that neither the completed status nor the commit job’s success return establishes by itself (Section 11.2) — every current manifest and chunk of the space is encrypted under keys that did not exist before it: fresh item DEKs wrapped under a fresh root. A party that held the old root, the old item DEKs, or the old wraps — including a former recipient — cannot open any current record. The space’s key version does not change on a pure rotation, since the encapsulation keypair is unchanged; it changes only when the rotation follows a password change or reset.

What rotation does *not* achieve is stated plainly:

- It does not retroactively protect content that an authorized party downloaded or synchronized before the rotation; a former recipient who kept the ciphertext and its keys, or the plaintext, keeps it.
- It does not rotate keys that a content update leaves in place: between rotations an item keeps its DEK across content updates, so a party who learned one item’s DEK reads that item’s later revisions until the next rotation.
- Superseded manifests and chunks are archived by the commit job and deleted by a follow-up job; they exist on the server until that job runs.
- Revocation of the rotated space’s outstanding shares is requested after commit and processed asynchronously; a refused request does not fail the rotation (Section 10.4).

See also the forward-secrecy discussion in Section 13.2.

12 Multi-Device Support and Recovery

12.1 Device Transfer Protocol

Vault metadata can be moved to a new device without ever exposing keys to the server, using a four-phase, human-verified key exchange between the sender device D1 and receiver D2 (the server relays opaque records):

1. **Key-exchange initiation:** D2 publishes its `user_public_key`; both devices independently compute a hash of the key and a time window.
2. **Short-number verification:** D2 displays a short number (range 1–999) derived deterministically from the hash; D1 displays the same derived number among random decoys, and the user must select on D1 the number shown on D2. A wrong selection aborts the transfer. This is a *human-verified* check of D2’s public key against man-in-the-middle substitution, with roughly $\log_2 999 \approx 10$ bits of comparison entropy per window; it is not a cryptographic key-agreement binding (no PAKE is used). The check assumes the attacker cannot choose the substitute key after seeing the code: an attacker free to generate candidate keys needs on the order of a thousand attempts to find one whose derived number matches, so the protection rests on the registration of D2’s key with the service and on retry and expiry limits, not on the comparison alone.
3. **DEK exchange:** D1 generates a random transfer DEK and wraps it under an ML-KEM shared secret encapsulated to D2’s verified public key; D2 decapsulates and unwraps.
4. **Data exchange:** D1 encrypts the transfer payload — the per-vault `vault_seed` values and vault names — under the transfer DEK; D2 downloads and decrypts, stores the seeds in its Keychain, and rebuilds local slot records through the vault-list synchronization below. File content is not transferred — it re-syncs from the cloud as ciphertext. Note that the transfer channel carries the raw vault seeds (not merely encrypted slot records); the server relays only the DEK-encrypted blob and overwrites it once the transfer completes.

12.2 Vault List Synchronization

A device holding `user_seed` can enumerate the user’s vaults directly (`/v1/my-vault/by-user-key`, signed with the user key) and rebuild the local vault key databases (Section 6.1), matching slot records by their server-registered hashes.

12.3 Recovery Model

Two recovery objects exist:

user_recovery_key A hex export of `user_seed`, displayed once at onboarding for the user to store offline. Re-importing it on a new device restores the user identity layer; the import is validated against the user’s registered public key when the server can supply one (the check is best-effort, not enforced on every path).

vault_recovery_key An encrypted recovery object per vault/space containing the material needed to restore access (`vault_seed`, `encap_seed`, salt, slot identifier, slot key), wrapped under a key derived from `user_seed` via HKDF. It supports vault-seed restoration on new devices and the forgot-password flow (decrypt the recovery object, rebuild the slot under the new password, and re-wrap the encapsulation seed; a full key rotation, Section 11, is offered as a follow-up step rather than performed inside the flow).

The vault recovery object can be escrowed server-side (`vault_recovery_key_encrypted`): for Drives this escrow always occurs; for Vaults it is controlled by a per-vault setting that defaults to the workspace policy.

Trust implications of the escrow — stated plainly

OXU Drive does **not** claim “no key recovery by design.” Readers should understand the trust consequences of the recovery design as it stands:

- The escrowed recovery object contains, once unwrapped, the complete key material of the space — its wrapping is the *only* barrier.
- The wrapping key is derived from `user_seed` alone — the same secret the user exports as `user_recovery_key`. No password or memory-hard KDF participates in the wrap.
- Consequently, any party holding *both* the user’s exported recovery key file *and* the server-side escrow blob can reconstruct the vault’s keys off-device, without any space password. The device-binding property of `vault_seed` (Section 5.2) does not apply to this path.
- The presence/absence pattern of escrow blobs is itself server-side metadata (see the metadata interaction in Section 8.8).

This trade-off is deliberate: the escrow is load-bearing for multi-device restore (Section 12.4), and its cost is that recovery security reduces to the secrecy of the user-exported recovery key plus access control on the server blob. Strengthening the escrow wrapping — for example with an Argon2id-stretched recovery passphrase distinct from `user_seed`, or KMS/threshold escrow — is planned work (Section 16.2).

12.4 Multi-Device Restore

After restoring `user_seed` and `vault_seed` on a new device (through transfer, or automatically from an escrowed recovery object, Section 12.3), the client rebuilds each vault’s directory, configuration, and vault key database through the vault-list synchronization above. A space’s manifests and chunks are *not* acquired at that point: they are pulled when the space is first unlocked on the device (Section 7.5), and on that first walk the ancestor-fetch channel of Section 8.7 applies. A restored device therefore holds slot rows for every space of every vault but content only for the spaces it has opened.

12.5 Deletion Options and Panic

Vaults support three deletion levels — hide (local disable flag), delete local (disable and remove local artifacts), and delete on server (an authenticated, group-signed deletion of the vault and every space registered under its group identifier) — plus a panic action that operates on local state first so it works offline: it locks every vault, removes the file-manager integration, disables the vault configurations, and stops synchronization. Server-side deletion removes the spaces’ registry entries, manifests, chunk records, and stored objects, and revokes their shares.

Two limits are stated plainly. Local deletion and panic remove what the client can reach; whether every per-space artifact is removed under every failure condition has not been verified for this document, so a forensic reader may recover ciphertext after a panic. And no local deletion affects copies that the server retains until its deletion jobs run or that a recipient has synchronized (Section 10.3). Deletion is a housekeeping operation, not a coercion remedy.

13 Security Analysis

13.1 Analysis Against the Threat Model

Table 9: Security Properties vs. Threat Model

Adversary	Primary protections	Residual exposure
Physical access	Argon2id-stretched passwords; Keychain-bound seeds (device-only accessibility); AEAD everywhere; SQLCipher local databases; fixed-size slot records	weak passwords remain brute-forceable in principle; unlocked-state memory (Section 14.2)
Coercion (rubber-hose + forensics)	Argon2id-stretched passwords; per-space keys derived only from the revealed password, seed, and salt	no deniability of hidden content: the registry and the server’s labelled records identify every space (Section 8.3); limits in Section 8.8
Honest-but-curious server	client-only keys; encrypted content, names, manifests; ML-KEM-encapsulated DEKs; signature-gated writes; encrypted folder relationships on current vault items (Section 8)	per-space metadata: which records belong to which space, sizes, counts, timing, version chains, ancestor fetches (Section 8.7); folder-graph shape for the Drive and legacy items; recovery-escrow blob if enabled (Section 12.3)
Network	TLS; end-to-end encryption beneath it; signed state changes; presigned, short-lived upload URLs	traffic analysis (volumes/timing), as with any sync system
Quantum	ML-KEM-768 KEM (Level 3); AES-256 (~128-bit post-quantum); harvest-now-decrypt-later resistance for content	Ed25519 signatures are classical — forgeable by a future quantum adversary; ML-DSA implemented, pending activation

The quantum row deserves emphasis. What is post-quantum is *passive* confidentiality: ciphertext captured today and encrypted to authentic ML-KEM public keys cannot be decrypted by a later quantum adversary. *Authentication* is not: Ed25519 request signatures, the session layer, TLS, and the directory that hands out recipients’ public keys are all classical. A future quantum adversary able to defeat those could not decrypt stored content, but could forge requests and, in an active attack on a future sharing or enrollment step, influence which public key a client encrypts to. Activating ML-DSA addresses the request-signature part of that gap (Section 16.2); the transport, identity, and directory layers need their own post-quantum assessment.

13.2 Key Compromise Resilience

13.2.1 Isolation Boundaries

- **Space isolation:** each space’s keys derive from its own password through Argon2id + domain-separated HKDF; compromising one space password reveals nothing about sibling spaces, provided the passwords differ — under the shared `vault_seed` and `vault_salt`, equal passwords derive equal keys.
- **Per-file DEKs:** compromise of one file’s DEK exposes that file only. In a vault, every item’s DEK is wrapped directly under the space root DEK (Section 5.4), so a compromised root exposes that space and nothing else. In the Drive, where child DEKs are wrapped

under their parent’s DEK (Figure 3), compromise of a *folder’s* DEK additionally exposes its descendant subtree — though never ancestors or siblings.

- **Vault isolation:** each vault has its own `vault_seed`; keys are namespaced per (user, workspace, vault).
- **Seed compartmentalization:** `user_seed` by itself does not decrypt vault content — *unless* the recovery escrow path applies, in which case `user_seed` + the escrow blob suffices (Section 12.3). This is the single most important caveat in the compromise analysis.

13.2.2 Forward Secrecy Limitation

The design does *not* provide forward secrecy: if a space’s ML-KEM private key (equivalently `encap_seed`, equivalently the password + `vault_seed` + salt) is compromised, all previously stored KEM ciphertexts for that space can be decapsulated and all historical DEKs recovered. Key rotation (Section 11) re-encrypts the space’s current content under fresh keys, but does not retroactively protect data an adversary already captured under old keys, and old private keys are not subject to a secure-deletion protocol with verifiable erasure.

13.3 Known Weaknesses and Open Items

For readers making deployment decisions, the honest list:

- **Recovery escrow trust** (Section 12.3): server + user-recovery-file jointly recover vault keys.
- **Per-space metadata** visible to the server — which records belong to which space, counts, sizes, timing — and folder-graph shape for the Drive and legacy items (Section 8.7).
- **No deniability of hidden content** — see Sections 8.2 and 8.8.
- **Rotation re-encrypts current content but cannot recall copies** a former recipient already synchronized; between rotations an item keeps its DEK across content updates (Section 11.4).
- **Classical signatures** until ML-DSA activation.
- **No forward secrecy** for content keys (above).

14 Implementation Notes

14.1 Performance Considerations

The Argon2id cost is incurred once per unlock attempt by design — it is the anti-brute-force budget. ML-KEM operations are sub-millisecond and add negligible overhead per file. Encrypting the parent relationship (Section 8.5) adds one AES-256-GCM unwrap per item under the cached root DEK and one extra fetch and decryption per ancestor that is missing locally when its descendant arrives; a key rotation re-uploads the space’s content once (Section 11).

14.2 Memory Security

Key material is released when no longer needed. The Rust core erases its buffers; the client layers, which hold keys in ordinary language objects, drop their references:

- password buffers are cleared immediately after key derivation;

Table 10: Cryptographic Operation Performance (Apple M1; re-benchmark scheduled before publication)

Operation	Time	Throughput	Memory
AES-256-GCM Encrypt (1MB)	~2ms	500 MB/s	Streaming
ML-KEM-768 KeyGen	~0.1ms	–	<1 MB
ML-KEM-768 Encapsulate	~0.1ms	–	<1 MB
ML-KEM-768 Decapsulate	~0.1ms	–	<1 MB
Ed25519 Sign	~0.05ms	–	<1 KB
Ed25519 Verify	~0.1ms	–	<1 KB
Argon2id (64 MiB, $t=3$, $p=1$)	~500ms	–	64 MB
HKDF-SHA512 (32-byte output)	<0.01ms	–	<1 KB
SQLCipher page encrypt	~0.05ms	80 MB/s	Per-page

- derived keys (`slot_key`, `encap_seed`, DEKs) are removed from the runtime store on vault lock or application termination — a release of references, not a demonstrated overwrite of the bytes;
- ML-KEM shared secrets are cleared after the single wrap/unwrap they serve;
- the Rust core zeroizes its buffers on drop.

Not every copy of key material is a transient object, and the list above describes those only. Three kinds of retained copy exist. The FileProvider key bundle (Section 4.6) sits in the Keychain — encrypted under a Secure Enclave key where one is available, otherwise as a plain serialized object — from the moment the app writes it until the extension consumes and deletes it at startup, and carries its own lock deadline. The space’s ML-KEM private key and signing private key are stored in the space’s key database, and the item DEK of every item the device has opened is stored in the clear in the space’s data database; both databases are SQLCipher-encrypted under `slot_db_key` and persist until the space is deleted locally. And in-process caches hold material for a fixed time from the moment they are populated: five minutes for the extension’s space-key and encapsulation-key caches, eight minutes for the synchronization engine’s root DEK, while the engine holds the encapsulation seed for the life of the unlocked session. Expiry is checked on the next lookup, which then reloads the value from the runtime store or resolves it again; it is not verified erasure of every copy, so the five-minute figure bounds that one cache and not the client’s total key exposure, and no maximum lifetime across all copies is asserted. On lock, the extension removes the space’s entries from its runtime store and clears its space-key cache; the encapsulation-key cache is left to expire.

Complete memory security cannot be guaranteed on commodity platforms: copies made by language runtimes, swap, hypervisors, cold-boot attacks, or crash dumps remain possible while material is live.

14.3 Nonce Management

AES-GCM security depends critically on nonce uniqueness per key:

- **File chunks:** fresh random 96-bit nonces per encryption. For q encryptions under one DEK the probability of a repeated nonce is about $q(q-1)/2^{97}$, where q counts every encryption invocation under that exact DEK for any purpose — content chunks and the encrypted metadata, by any writer or device, including a retry that re-encrypts rather than resends — from the DEK’s creation until rotation replaces it (Section 11.4); a rotation’s re-encryption counts under the new DEK. Neither file size nor a rotation schedule alone

bounds q ; only a bound on the work done under a DEK between rotations does. As an illustration only, one revision of a 2^{40} -byte file is 2^{20} chunks at the smallest chunk size of Table 14 (taking its sizes as binary megabytes) and about $2^{14.4}$ at the largest, giving per-revision probabilities near 2^{-57} and 2^{-68} ; neither figure bounds the lifetime of a DEK. The client enforces no invocation budget per key and does not count q ; the limit of 2^{32} random-nonce invocations per key across all users of the key in SP 800-38D [5] is therefore a budget the owner must meet by bounding the work between rotations, not a mechanism of the client.

- **No in-place re-encryption:** updates produce new ciphertexts with new nonces; nonces are stored with the ciphertext, never regenerated.
- **Database pages:** SQLCipher generates a fresh random IV for every page write.
- **Derived keys:** HKDF domain separation (Appendix B) guarantees distinct keys for distinct purposes, so a nonce collision across purposes never occurs under the same key.

14.4 Error Handling

Failure messages are sanitized to prevent oracles: authentication failures return a generic decryption error without distinguishing wrong password, corrupted ciphertext, or invalid tag; the try-all-slots unlock (Section 6.2) does not reveal which record failed; database unlock failures do not reveal whether the key was wrong or the file corrupted.

15 Related Work

15.1 Encrypted Drive Solutions

Table 11: Comparison with Existing Solutions

Feature	OXU Drive	VeraCrypt	Cryptomator	Standard Notes
Post-Quantum KEM	✓	–	–	–
Cloud sync (zero-knowledge)	✓	–	✓	✓
Folder structure hidden from storage	✓ [†]	✓	✓	–
Hidden volume / duress password	–	✓	–	–
Native file-manager integration	✓	mount	✓	–
Open-source crypto primitives	✓	✓	✓	✓

[†] for current vault data only: explicit parent relationships are encrypted (Section 8.2); which space a record belongs to, sizes, counts, timing, and identifier reuse remain visible, and the Drive sends plaintext graph keys. VeraCrypt conceals all structure inside a fixed-size container; Cryptomator obfuscates the directory hierarchy on storage while exposing object counts, sizes, and timestamps. Cells reflect the vendors’ public documentation as of this writing; a dash for the post-quantum KEM row means no post-quantum public-key encapsulation, not the absence of quantum-resistant symmetric protection. OXU Drive offers no hidden-volume deniability.

VeraCrypt’s hidden volumes [11] hide data in the free space of an outer volume of fixed size — strong “does not exist” deniability for a local container, but incompatible with per-file cloud synchronization, which necessarily exposes object counts and sizes to the server. OXU Drive takes a different trade: it accepts that a synced vault’s existence, its group-level activity, and its volume envelope are visible, and instead encrypts the folder structure within each space (Section 8). It offers no hidden-volume deniability. Deniable encryption in the sense of Canetti et al. [13] and robust hidden-volume constructions such as HIVE [12] address the coercion setting directly, at the cost of oblivious or fixed-size storage; OXU Drive does not address it (Section 8.2).

15.2 Post-Quantum Cryptography Adoption

NIST concluded the initial PQC standardization [1] in 2024 with FIPS 203 (ML-KEM), FIPS 204 (ML-DSA), and FIPS 205 (SLH-DSA). Industry adoption began with hybrid TLS deployments and Signal’s PQXDH. OXU Drive adopts ML-KEM-768 (Level 3) for key encapsulation — balancing security margin, key/ciphertext sizes on mobile, and standard compliance — and implements ML-DSA-44/65/87 in its core library, with protocol activation pending (Section 16.2). The signing-object key-versioning mechanism (Section 9.3) provides the algorithm-agility hook for that migration.

15.2.1 Hybrid Classical/Post-Quantum Considerations

OXU Drive currently uses pure ML-KEM rather than an ML-KEM + X25519 hybrid. Rationale: a single, NIST-standardized KEM minimizes implementation surface, and the primary threat motivating the choice (harvest-now-decrypt-later) is addressed by the PQ component. The natural hybrid successor is X-Wing [9], the CFRG hybrid KEM that combines ML-KEM-768 and X25519 under a fixed, security-analyzed combiner — a drop-in replacement at the encapsulation interface, since it exposes the same KEM API and a 32-byte shared secret. The key-indirection layer sketched in Section 16.2 would make this migration a per-space re-encapsulation of a single wrapped key rather than a re-wrap of every stored manifest.

16 Conclusion and Future Work

16.1 Summary

This paper presented the cryptographic architecture of OXU Drive: a zero-knowledge, cloud-synchronized drive system with client-only cryptography; a hierarchical key system built on Argon2id stretching and domain-separated HKDF; ML-KEM-768 encapsulation of per-file DEKs; BLAKE3 Merkle-tree integrity with signed roots; end-to-end encrypted sharing with two-layer wrapping; staged, re-encrypting key rotation; human-verified device transfer; and the metadata-privacy properties of vault synchronization with an explicitly scoped claim and an explicit list of what they do not provide.

16.2 Future Work

- **ML-DSA activation:** the FIPS 204 implementation exists in the core; activating it in the protocol (dual-signing migration, key versioning, size impact on signing objects) makes request authentication post-quantum; transport, identity, and directory trust remain separate items.
- **Recovery redesign:** resolving the escrow trust issue (Section 12.3) — password-protected recovery wrapping, KMS/TEE-backed escrow (a TEE-based KMS integration is designed but not implemented), or threshold/Shamir schemes.
- **Metadata protection beyond parent edges:** the vault protocol encrypts explicit folder relationships, but sizes, counts, stable identifiers and their reuse, version chains, activity timing, and the by-identifier fetch of a missing ancestor still support structural inference (Section 8.7); padding, oblivious access, or prefetching ancestors without revealing which are missing would be needed to close those channels, and legacy plaintext identifier graphs need to be retired.
- **Hiding which space a record belongs to:** today every record names its space. Synchronizing all spaces of a vault as one group, with records carrying no space label and

attributed on the device by trial decryption, would remove that label at the cost of whole-vault storage on every device and per-row trial work; it is a research direction, not a commitment.

- **Deniability, currently not offered** (Section 8.9): automatically generated decoy spaces, cover traffic or chaff generation, and hidden volumes or duress passwords are research directions. None is claimed by this design; Section 8.3 explains why decoys or cover traffic that appear only alongside real activity do not by themselves restore a deniability bound.
- **Fresh keys per revision**: between rotations an item keeps its DEK across content updates (Section 11.4); minting a fresh DEK for every content revision would narrow what a single leaked item key exposes.
- **Hybrid KEM via X-Wing**: replacing pure ML-KEM-768 with X-Wing [9] (ML-KEM-768 + X25519 under a fixed combiner) for defense-in-depth during the post-quantum transition (Section 15.2).
- **Retiring the per-item encapsulations**: the root DEK already provides key indirection for the primary open path (Section 5.4), but every vault manifest still carries its own ML-KEM encapsulation to the password-derived space keypair. A password change or a KEM migration such as the move to X-Wing therefore still touches every manifest. Making the root encapsulation the only KEM object per space — and attributing rows across key generations through retained root material instead — would reduce such a migration to one re-encapsulation per space.
- **Forward secrecy**: space private keys are password-derived and regenerable, so deleting them achieves nothing by itself; forward secrecy would need non-derivable per-epoch key material with verifiable deletion of superseded epochs, together with content re-encryption.
- **Formal verification** of the protocol layer (ProVerif / Tamarin) and independent third-party audit (below).

16.3 Security Audit Status

Disclosure: OXU Drive’s cryptographic implementation has undergone internal security review (whose findings shaped this document, including the scoping of the metadata-privacy claims and the recovery-escrow disclosure), but has *not yet* undergone an independent third-party audit. The primitives used are individually well analyzed; the integration requires independent verification. We are committed to engaging a reputable auditing firm, publishing findings and remediations, and maintaining a vulnerability disclosure program.

References

- [1] NIST. Post-Quantum Cryptography Standardization. <https://csrc.nist.gov/projects/post-quantum-cryptography>, 2024.
- [2] NIST FIPS 203. Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM), 2024.
- [3] NIST FIPS 204. Module-Lattice-Based Digital Signature Standard (ML-DSA), 2024.
- [4] Biryukov, A., Dinu, D., Khovratovich, D. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. Euro S&P 2016. (See also RFC 9106.)

- [5] NIST SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM), 2007.
- [6] Krawczyk, H., Eronen, P. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, 2010.
- [7] Josefsson, S., Liusvaara, I. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, 2017.
- [8] O'Connor, J., Aumasson, J.-P., Neves, S., Wilcox-O'Hearn, Z. BLAKE3: One Function, Fast Everywhere. <https://github.com/BLAKE3-team/BLAKE3-specs>, 2020.
- [9] Connolly, D., Schwabe, P., Westerbaan, B. X-Wing: The Hybrid KEM You've Been Looking For. IACR Communications in Cryptology, 1(1), 2024. (See also IETF CFRG draft-connolly-cfrg-xwing-kem.)
- [10] Zetetic LLC. SQLCipher: Full Database Encryption for SQLite. <https://www.zetetic.net/sqlcipher/>, 2024.
- [11] IDRIX. VeraCrypt Documentation (Hidden Volume). <https://veracrypt.io/>, 2024.
- [12] Blass, E.-O., Mayberry, T., Noubir, G., Onarlioglu, K. Toward Robust Hidden Volumes Using Write-Only Oblivious RAM. ACM CCS 2014.
- [13] Canetti, R., Dwork, C., Naor, M., Ostrovsky, R. Deniable Encryption. CRYPTO 1997.

A Algorithm Parameter Specifications

Table 12: Argon2id Parameters

Parameter	Value
Variant	Argon2id, version 1.3
Memory Cost	64 MiB
Time Cost	3 iterations
Parallelism	1 lane
Output Length	64 bytes (space and share passwords)

Table 13: HKDF-SHA512 Parameters

Parameter	Value
Hash Function	SHA-512
Salt	<code>vault_salt</code> (16 bytes) where applicable
Info	distinct per derived key (Appendix B)
Output Length	32 or 64 bytes per key

B HKDF Info-Label Registry

No two derived keys share both input keying material and HKDF `info` label (Section 3.3). Salts: `vault_salt` (16 random bytes per vault) for the password-derived keys and the space signing seed; the per-share random salt for the share key; a fixed application constant for the

Table 14: Chunking Parameters

Path	Chunk size
Default	2 MB
FileProvider bulk path	50 MB
Background transfers	1 MB
Chunk format	nonce (12 B) ciphertext tag (16 B)
Chunk hash	BLAKE3(LE32(index) nonce ciphertext tag)

vault key database key; empty for the user signing seed. All labels are UTF-8 strings without terminator; output lengths are as listed. The recovery-object wrapping key is derived from `user_seed` with a separate fixed label (Section 12.3). The registry below is normative; adding a derived key requires adding a row here.

Table 15: HKDF domain-separation labels

Derived key	IKM	Info label
slot_key	vault_seed pwd_stretched	v2 (salt = vault_salt)
encap_seed	vault_seed pwd_stretched	encap-seed-v2 (salt = vault_salt)
Space signing seed	encap_seed	ed25519-signing-key
pw_share_32b	Argon2id(pw_share, share salt)	share-v2 (salt = share salt)
Vault key database key	user_seed	dbContext (fixed app salt)
Group signing seed*	SHA-256(vault_seed)	(none — hash used directly)
User signing seed	user_seed	ed25519-group-signing-key

* The 32-byte SHA-256 of `vault_seed` is used directly as the Ed25519 seed of the group signing key (Section 9.2); no HKDF expansion is applied on that path. Per-space databases are opened with the random `slot_db_key` carried in the slot record (Section 6.2), not with a derived key.

C Vault Key Database Schema

The vault key database (Section 6.1) is a SQLCipher database whose slot-configuration table holds one row per space:

Table 16: Slot-configuration row

Column	Content
id	random UUID; re-randomized only when a space-set change rewrites the table
created / updated	fixed constant date (deliberately carries no activity information)
slot	fixed 512-byte AES-256-GCM slot record (Section 6.2): 12-byte nonce ciphertext of the 484-byte padded plaintext 16-byte tag
salt	16-byte vault_salt, identical on every row
encap_algorithm	AEAD identifier for the DEK wrap
encrypted_dek_by_vault	root DEK wrapped under the ML-KEM shared secret (Section 5.4)
encap_cipher_text_by_vault	ML-KEM-768 ciphertext (1,088 bytes)
vault_recovery_key_encrypted	optional encrypted recovery object (Section 12.3)

All rows carry a slot record of the same fixed length and the same salt; the recovery column is populated only for spaces with escrow enabled. The whole table is re-inserted shuffled with

fresh identifiers whenever the set of spaces changes (Section 6.1).

D Key Derivation Pseudocode

Algorithm 2 Complete Key Derivation Chain

```

1: function DERIVESPACEKEYS(pwd, vaultSalt, userId, wsId, vaultName)
2:   vaultSeed  $\leftarrow$  Keychain.get(userId, wsId, vaultName)
3:   if vaultSeed =  $\perp$  then
4:     return RECOVERYREQUIRED  $\triangleright$  an existing vault's seed must come from transfer or
       recovery; only vault creation generates one
5:   end if
6:   stretched  $\leftarrow$  Argon2id(pwd, vaultSalt, 64)  $\triangleright m=64$  MiB,  $t=3$ ,  $p=1$ 
7:   ikm  $\leftarrow$  vaultSeed  $\parallel$  stretched
8:   slotKey  $\leftarrow$  HKDF(ikm, vaultSalt, "v2", 32)
9:   encapSeed  $\leftarrow$  HKDF(ikm, vaultSalt, "encap-seed-v2", 64)
10:  (pkK, skK)  $\leftarrow$  ML-KEM-768.KeyGen(encapSeed)
11:  signSeed  $\leftarrow$  HKDF(encapSeed, vaultSalt, "ed25519-signing-key", 32)
12:  (pkS, skS)  $\leftarrow$  Ed25519.KeyGen(signSeed)
13:  return (slotKey, pkK, skK, pkS, skS)
14: end function

```

Algorithm 3 File Encryption with Post-Quantum KEM

```

1: function ENCRYPTFILE(id, plaintext, rootDEK, pkK)
2:   DEK  $\leftarrow$  SecureRandom(32)  $\triangleright$  new item or rotation only; a content update reuses the
       item's DEK
3:   dekEnc  $\leftarrow$  AES-256-GCM.Encrypt(DEK, key = rootDEK)  $\triangleright$  primary form,
       Section 5.4
4:   (ct, ss)  $\leftarrow$  ML-KEM-768.Encapsulate(pkK)
5:   encDEK  $\leftarrow$  AES-256-GCM.Encrypt(DEK, key = ss)  $\triangleright$  encapsulated form
6:   chunk0, ..., chunkm-1  $\leftarrow$  split(plaintext)  $\triangleright$  Table 14; indices are zero-based
7:   for i  $\leftarrow$  0 to m - 1 do
8:     ni  $\leftarrow$  SecureRandom(12)
9:     aadi  $\leftarrow$  UTF8(id  $\parallel$  "__chunk__"  $\parallel$  decimal(i))  $\triangleright$  Section 7.2.1
10:    ci  $\leftarrow$  ni  $\parallel$  AES-256-GCM.Encrypt(chunki, DEK, nonce = ni, aad = aadi)  $\triangleright$ 
       ciphertext  $\parallel$  tag
11:    hi  $\leftarrow$  BLAKE3(LE32(i)  $\parallel$  ci)
12:  end for
13:  root  $\leftarrow$  MerkleRootBLAKE3(h0, ..., hm-1)  $\triangleright$  pairwise, odd node duplicated; signed by
       the user key when available
14:  return (dekEnc, ct, encDEK, c0, ..., cm-1, root)
15: end function

```
