أكاديمية كاوست
KAUST ACADEMY

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

QUAROM: Quantum Secure Data Rooms for Storage and Sharing

Turki Ahmed Alsuhaibani, Mohammed Mansour Albulayhi, Abdullah Saleh Alfallaj,
Saleh Abdullah Alkhafaji, Hatim Mohammad Alhosaini
Supervisor: Dr. Muhammad Shahbaz Khan, Aston University, Birmingham, UK

Abstract

QUAROM is a secure data-room platform for storing and sharing sensitive, long-lived documents. It applies client-side encryption and post-quantum cryptography, ensuring that plaintext files, raw file keys, and unencrypted private keys are not exposed to the backend. Access is granted on a need-to-know basis through recipient-specific key envelopes, with additional support for quorum-sealed files that require multiple custodians to approve access and hash-chained audit records for sensitive actions. The main beneficiaries are government, defence, healthcare, legal, financial, and other organizations that manage highly confidential or regulated information requiring long-term protection. QUAROM can be offered as a B2B cybersecurity software product or managed secure data-room service. Its main advantage is that sensitive documents remain protected from the storage provider and unauthorized users while also reducing exposure to harvest-now, decrypt-later threats associated with future quantum computers.

Code: `https://github.com/TurkiOS/Kaust_Project` (Private Repo, contact us for access)

Date: August 21, 2026

Contents

1 Introduction **3**

1.1 Background and Motivation . . . 3
1.2 Problem Statement . . . 3
1.3 Project Objectives . . . 4
1.4 Scope and Contributions . . . 4

2 Related Work **5**

2.1 Encrypted Storage on Untrusted Infrastructure . . . 5
2.2 Virtual Data Rooms . . . 6
2.3 Post-Quantum Cryptography in Deployed Systems . . . 6
2.4 Standardized Post-Quantum Primitives . . . 6
2.5 Tamper-Evident Logging and Key Transparency . . . 7
2.6 Gap Addressed by Quarom . . . 7

3 Requirements and Threat Model **8**

3.1 Functional and Security Requirements . . . 8
3.2 Assets, Adversaries and Trust Assumptions . . . 8
3.3 Limitations and Out-of-Scope Threats . . . 11

4 System Design and Security Architecture **11**

4.1 Architecture . . . 12
4.2 Cryptographic Protocol . . . 12
4.2.1 What each layer authenticates . . . 13
4.2.2 The Protection Profile . . . 14
4.3 Data Rooms and Access Control . . . 15
4.4 Key Management . . . 15
4.4.1 User key lifecycle . . . 16
4.4.2 Recipient key authentication . . . 16
4.5 Quorum Access . . . 16
4.6 Auditing . . . 17

5 Implementation **17**

5.1 Technology Stack . . . 17
5.2 Implemented Architecture . . . 18
5.3 Main Features . . . 18
5.3.1 The protection profile as built . . . 18
5.3.2 Need-to-know sharing . . . 19
5.3.3 Quorum-sealed files . . . 19
5.3.4 Auditing as built . . . 19
5.4 Implementation Status and Traceability . . . 20
5.5 The Delivered Application . . . 20

6 Evaluation and Results **23**

6.1 Test Environment . . . 23

6.2 Functional Tests . . . 23
6.2.1 Automated verification . . . 23
6.2.2 Functional coverage by requirement . . . 24
6.3 Security and Negative Tests . . . 24
6.4 Performance Benchmarks . . . 24
6.4.1 Post-quantum primitives . . . 24
6.4.2 File encryption and decryption . . . 26
6.4.3 Sharing scalability . . . 26
6.4.4 Quorum sealing and unsealing . . . 27
6.4.5 Audit chain verification . . . 27
6.5 Storage Overhead . . . 28
6.6 Selecting the Protection Profile . . . 28
6.7 Results and Discussion . . . 29
6.8 Threats to Validity . . . 30

7 Security Analysis and Limitations **30**
7.1 Analysis by Adversary . . . 30
7.2 The Audit Gap . . . 31
7.3 The Key-Substitution Gap . . . 32
7.4 No Forward Secrecy . . . 32
7.5 Single-Family Cryptographic Risk . . . 33
7.6 Residual Risk Summary . . . 33

8 Conclusion and Future Work **33**
8.1 Future Work . . . 34
8.2 Closing Remarks . . . 35

References **35**

A Appendix **37**
A.1 Reproducing the Measurements . . . 37
A.1.1 Harnesses . . . 37
A.1.2 Running the cryptographic harness . . . 38
A.1.3 Running the test suites . . . 38
A.2 Security Test Case Catalogue . . . 38
A.3 Cryptographic Parameters . . . 38
A.4 Authenticated Context Formats . . . 38
A.5 Measurement Output . . . 40

1 Introduction

1.1 Background and Motivation

Confidential documents are increasingly exchanged through *virtual data rooms* (VDRs) rather than the general file sync-and-share services (GFSS) that dominate ordinary collaboration, such as Google Drive, Dropbox, and OneDrive. The difference is governance: a data room is organized around a transaction, whether due diligence in a merger, a clinical submission, or a regulatory audit, and treats access as a governed relationship of per-room roles, sensitivity-driven permissions, revocation, and an audit trail meant to be relied on afterwards.

What the model does not change is the cryptography underneath. The established providers, among them Intralinks, Datasite, DFIN Venue, iDeals, Ansarada, and Firmex, alongside the lighter-weight Digify, DocSend, and Papermark, publish broadly similar descriptions: AES-256 at rest, TLS in transit, and key management resting on RSA or elliptic-curve primitives. None documents post-quantum key establishment for stored documents.

That matters because those primitives have a known expiry date. Shor's algorithm solves integer factorization and discrete logarithms in polynomial time on a sufficiently large quantum computer [1], which breaks RSA and elliptic-curve key establishment outright; Grover's search algorithm weakens symmetric primitives only quadratically [2], which is why AES-256 survives and key *wrapping* does not. NIST has been preparing the migration since 2016 [3], concluding the selection process in 2022 [4] and publishing the first standards in 2024 [5–7].

The timing argument is what makes this urgent rather than theoretical. Mosca frames it as a comparison of three durations: how long data must stay confidential, how long a migration takes, and how long until a cryptographically relevant quantum computer exists [8]. If the first two together exceed the third, the data is already exposed, because an adversary can record ciphertext now and decrypt it later. This *harvest-now, decrypt-later* exposure falls hardest on exactly the long-lived records data rooms exist to hold: healthcare, defence, and regulated collaboration keep documents valuable for decades.

A second exposure is structural rather than cryptographic. A conventional data room places substantial trust in its provider: when a backend handles plaintext files, raw content keys, or unencrypted private keys, a breach or a malicious administrator can expose the documents whatever the governance layer says. The protection rests on the provider's conduct, which is precisely the assumption a long-lived confidential record should not have to make.

1.2 Problem Statement

The two exposures above are usually addressed separately, and each existing answer gives up what the other provides.

End-to-end encrypted storage removes the provider from the trust boundary but is built for personal file sync, not for governed multi-party disclosure, and it wraps file keys with the classical primitives that harvest-now, decrypt-later defeats. Virtual data rooms provide the governance but keep the provider inside the trust boundary. Deployed post-quantum protocols solve key establishment convincingly, but for *sessions*, where a compromised secret expires when the session does, rather than for objects that must stay sealed for decades.

No existing system covers the intersection. The problem this project addresses is therefore:

> Can a data room provide the governance an organization needs, namely per-room roles, sensitivity-driven policies, revocation, and an auditable history, while keeping the hosting infrastructure outside the trust boundary and protecting stored documents

against an adversary who records ciphertext today and obtains a quantum computer later, and at what measurable cost in latency and storage?

1.3 Project Objectives

The project set out to meet seven objectives. They are numbered here because Section 6 reports the evidence for each in turn, and Section 5.4 states which are met in full, which in part, and which are not met.

O1. Remove the provider from the trust boundary. No plaintext document, no unwrapped file key, and no unencrypted private key may reach the backend or the storage provider, under any operation the system supports.

O2. Make key management post-quantum for data at rest. Every mechanism that protects a stored file key must rest on standardized post-quantum primitives, so that a ciphertext recorded today is not opened by a future quantum adversary.

O3. Encrypt once, share many times. A document must be encrypted and stored exactly once regardless of how many people can read it. Granting or revoking one recipient must cost one small object and must never require rewriting or re-uploading the ciphertext.

O4. Make access a cryptographic claim rather than an identity claim. Membership of a room must not by itself confer the ability to read a file in it. A member should obtain nothing without a key envelope wrapped to their own key, and the server should not disclose the existence of files they cannot open.

O5. Remove the single-holder assumption where classification demands it. For documents governed by a two-person rule, no individual, and no single compromised account, may hold a complete capability to open the file.

O6. Make the audit record verifiable without trusting its keeper. The history of who did what must be checkable by a party who does not trust the backend that stores it, and any modification of that history must be detectable.

O7. Deliver and honestly evaluate a working prototype. The result must be a running desktop system, evaluated for functional correctness, security behaviour under attack, performance, and storage overhead, with the evaluation stating plainly which claims the evidence supports and which it does not.

1.4 Scope and Contributions

Quarom addresses these objectives through client-side encryption under post-quantum key management. For each upload the client's crypto process draws a random 256-bit file key and encrypts the document once with AES-256-GCM; that file key is then wrapped separately for every authorized recipient, producing recipient-specific key envelopes while keeping a single encrypted copy of the document in storage. The primitives are the NIST standards ML-KEM-768 (FIPS 203) for key encapsulation and ML-DSA-65 (FIPS 204) for signatures, with HKDF-SHA-384 for key derivation and SHA-384 for digests, chosen against measured alternatives rather than assumed (Sections 4.2.2 and 6.6).

The specific contributions are:

1. **A post-quantum data room.** A governed multi-party sharing model built on ML-KEM-768 key wrapping for documents at rest, rather than for sessions, with the cost of that choice measured rather than asserted (Sections 6.4 and 6.5).
2. **Need-to-know sharing.** Access follows envelope possession rather than room membership,

and the server filters on that before disclosing filenames, counts, or even the existence of a file (Section 5.3.2).

3. **Quorum-sealed files.** Where a classification demands a two-person rule, the file key is split across a threshold of custodians using Shamir secret sharing [9], so no individual holds a complete capability, and the unseal ceremony keeps even a fully compromised backend a messenger rather than a participant (Section 5.3.3).
4. **Cross-organization sharing.** A room owned by one organization can be shared with another as a unit, after which the receiving organization's administrators grant their own members access up to a ceiling fixed by the share, without either side losing the guarantee that access follows envelope possession.
5. **A measured suite comparison.** Classical, post-quantum, and hybrid suites were implemented against the same envelope structure and measured under identical conditions, so the choice of suite rests on evidence and its cost is quantified (Section 6.6).

The prototype is packaged as a Windows desktop client; macOS and Linux are a packaging concern rather than a design constraint and are left to future work. Section 3 states the threat model in full, including what the design deliberately does not defend against.

2 Related Work

Quarom sits where four lines of work meet: encrypted storage on untrusted infrastructure, virtual data rooms, deployed post-quantum protocols, and the tamper-evident logging and key-transparency techniques that make an untrusted server's claims checkable. This section reviews each, then states precisely what none of them covers.

2.1 Encrypted Storage on Untrusted Infrastructure

The idea that a storage provider should not be trusted with plaintext predates the cloud. Plutus [10] encrypts file content on the client, groups files into "filegroups" sharing a key, and introduces lazy revocation, where a revoked reader is locked out of future versions without forcing an immediate re-encryption of everything they could already read. SiRiUS [11] layers confidentiality and integrity over an existing untrusted file system without changing the server, attaching to each file a metadata block holding the file key encrypted to each authorized reader. Structurally that is the same one-ciphertext, many-envelopes shape Quarom uses, and the same trade-off appears: per-reader metadata grows with the sharing graph rather than with the data.

Kamara and Lauter [12] set out the architecture such a system needs in a cloud setting, separating a data processor, a data verifier, a token generator, and a credential generator, and observe that the hard problems are key management and revocation rather than encryption itself. That prediction has held: every system reviewed here, Quarom included, spends most of its complexity on those two.

Commercially, Tresorit describes a zero-knowledge model in which each file is encrypted on the user's device with a random file key, and neither files, keys, nor password material are sent or stored in the clear [13]; Proton Drive follows a comparable model with per-file keys wrapped to account keys [14]. These systems demonstrate that client-side encryption works at consumer scale. Two things separate them from the problem addressed here. They wrap keys with classical primitives such as RSA-4096 or elliptic curves, which are exactly the primitives most exposed to harvest-now, decrypt-later attacks on long-lived documents. And they are built around personal file sync, so their sharing model is a link or a per-file grant rather than the governed structure a data room needs: per-room roles, sensitivity-driven policies, staged revocation, threshold access,

and an audit trail meant to be relied on afterwards.

2.2 Virtual Data Rooms

Data rooms such as Intralinks, Datasite, iDeals, and the lighter-weight Digify, DocSend, and Papermark are widely used for confidential exchange in due-diligence, legal, and healthcare work, and they are where the governance model is mature: granular permissions, watermarking, expiry, and detailed access reporting are standard. In their usual deployment, however, the provider is a trusted party. Files sit in server-side storage that the provider, or a compromised administrator, can in principle read in the clear, and the published descriptions consistently pair AES-256 at rest with TLS in transit and classical key management. That arrangement is at odds with a threat model in which the hosting infrastructure is itself a possible adversary, and none of the public documentation surveyed describes post-quantum key establishment for stored documents.

2.3 Post-Quantum Cryptography in Deployed Systems

Several production systems already use post-quantum key establishment, nearly always in a hybrid setup that combines a classical elliptic-curve scheme with a lattice-based KEM. Signal's PQXDH extends X3DH by mixing a lattice-based KEM into the opening handshake alongside X25519, so the session's first secret stays confidential as long as either primitive holds [15]; Bhargavan et al. verified this design formally [16]. Apple's PQ3 for iMessage takes a similar hybrid approach with periodic post-quantum rekeying, with a companion analysis by Stebila proving confidentiality against both classical and quantum adversaries [17, 18]. Tuta's TutaCrypt replaces an RSA-based scheme with a hybrid of X25519 and ML-KEM [19], and Cloudflare and the major browsers support the `X25519MLKEM768` hybrid for TLS 1.3, which by mid-2025 protected a large share of human-generated traffic to Cloudflare's edge [20, 21].

Two things follow for Quarom. Hybrid rather than post-quantum-only is the current norm, reflecting caution about trusting any single new algorithm family; Quarom departs from that norm deliberately, and Section 6.6 reports both the measured cost of that choice and the reasoning, while Section 7.5 records it as the system's most consequential known weakness.

Second, and more fundamentally, all of these systems protect *sessions*. A session key that is compromised expires with the session, and forward secrecy limits the damage to a window. Documents at rest are long-lived on purpose, so harvest-now, decrypt-later hurts far more: a recorded ciphertext stays valuable for as long as the document does. Applying session-oriented reasoning to stored objects is precisely the mistake this project set out to avoid, and it is why the forward-secrecy limitation of Section 7.4 is stated rather than glossed.

2.4 Standardized Post-Quantum Primitives

NIST began the standardization effort with a public assessment of the threat [3] and concluded the selection process in 2022 [4], publishing the first standards on 13 August 2024. FIPS 203 standardizes ML-KEM, a module-lattice KEM derived from CRYSTALS-Kyber [5, 22], whose security reduces to the Module Learning With Errors problem. FIPS 204 standardizes ML-DSA, a module-lattice signature scheme derived from CRYSTALS-Dilithium [6, 23], built on the Fiat–Shamir-with-aborts paradigm. FIPS 205 adds SLH-DSA, a stateless hash-based scheme that rests only on hash-function assumptions and serves as a conservative fallback [7].

Quarom uses ML-KEM-768 and ML-DSA-65, at NIST security category 3 (roughly AES-192 against a quantum adversary under Grover's algorithm [2]). This is the same parameter set as Signal's PQXDH and the widely deployed `X25519MLKEM768` TLS hybrid, giving a well-studied margin without the larger keys and ciphertexts of the category-5 variants. The supporting

constructions are equally standard: AES-256-GCM as the AEAD [24, 25], HKDF-SHA-384 for key derivation [26, 27], Argon2id for password-derived keys [28], and Shamir's threshold scheme for quorum access [9, 29]. Key roles and retention follow NIST guidance [30].

2.5 Tamper-Evident Logging and Key Transparency

An audit trail kept by the party it is auditing is worth little unless its integrity can be checked independently. Haber and Stornetta introduced the technique the field still uses: linking records so that each commits to its predecessor, making any alteration of the past evident [31]. Crosby and Wallach developed this into practical tamper-evident logs with efficient membership and consistency proofs, and made explicit the point that matters here, namely that a log must be anchored outside the logger's control for its guarantees to bind the logger itself [32].

Certificate Transparency applies this at Internet scale, requiring certificate issuance to be published in append-only logs whose heads are gossiped and audited, so a misissued certificate cannot stay hidden [33]. CONIKS carries the same idea to end-user key directories: a provider that substitutes a user's public key must either publish the substitution where the victim can observe it or present inconsistent views to different clients [34]. Quarom does not implement such a directory, and Section 7.3 analyses what its absence costs. The literature also supplies the standard against which the delivered audit chain must be judged, which Section 7.2 does explicitly and unfavourably.

2.6 Gap Addressed by Quarom

Table 1 places the systems above against the properties this project requires.

Table 1 Related systems against the properties Quarom requires. ✓ = provided, × = not provided.

	Provider untrusted	PQ key establishment	Protects data at rest	Governed sharing	Threshold access
Traditional VDRs	×	×	✓	✓	×
Tresorit, Proton Drive	✓	×	✓	×	×
Plutus, SiRiUS	✓	×	✓	×	×
Signal PQXDH, Apple PQ3	✓	✓	×	×	×
TLS `X25519MLKEM768`	—	✓	×	—	×
Quarom	✓	✓	✓	✓	✓

The division is clean and it leaves long-lived confidential documents unserved. Data rooms give such documents the governance they need but keep the provider inside the trust boundary. End-to-end encrypted storage moves that boundary to the client but is built for personal sync rather than governed multi-party disclosure, and wraps file keys with classical primitives. The deployed post-quantum protocols solve key establishment convincingly, but for sessions rather than for objects that must stay sealed for decades.

Quarom occupies the intersection none of them covers: a data room whose governance model is a data room's, whose trust model is zero-knowledge, and whose key wrapping is post-quantum from the start. Its contribution is threefold. Sharing becomes a cryptographic claim rather than an identity claim, so a member gets nothing without an envelope wrapped to their own key (Section 5.3.2). The assumption that some single party can always open a document is removed where the classification demands it, and the secrecy of individual shares rests on no computational assumption at all (Section 5.3.3). And the record of who did what is designed to be checkable

without trusting the party that keeps it, though Section 7.2 reports honestly how far the prototype falls short of that last goal.

3 Requirements and Threat Model

A zero-knowledge claim is only meaningful against a stated adversary. This section fixes the requirements the system is built to meet, names the assets it protects, the adversaries it protects them from, and the assumptions it makes, and then states plainly what it does not defend against. The organization follows standard practice for security requirements work [35], and the design principles it applies, in particular least privilege and open design, are those of Saltzer and Schroeder [36].

3.1 Functional and Security Requirements

Requirements are split into functional (**FR**) and security (**SR**). Each traces back to an objective from Section 1.3 and forward to the evidence in Section 6.

Table 2 Functional requirements.

ID	Obj.	Requirement
FR1	O3	A user can upload a document; it is encrypted on the device before any byte leaves it, and stored exactly once.
FR2	O3	A user holding access can download and decrypt a document, recovering it byte-for-byte.
FR3	O3	An authorized user can grant access to further recipients without re-encrypting or re-uploading the document.
FR4	O3	An authorized user can revoke a recipient's access without affecting any other recipient.
FR5	O4	Documents are organized into rooms with per-room roles of owner, administrator, contributor, and viewer.
FR6	O4	A room carries a sensitivity classification and a versioned policy constraining the operations permitted on its documents.
FR7	O5	A document whose classification requires it can be sealed to a k-of-n quorum of custodians and later unsealed by a ceremony among them.
FR8	O6	Every security-relevant action is recorded, and a room's history can be exported and verified offline.
FR9	O1	A room owned by one organization can be shared with another, whose administrators then grant access up to a ceiling fixed by the share.

3.2 Assets, Adversaries and Trust Assumptions

Assets. Table 4 lists what the system protects, in descending order of harm on disclosure. The ordering matters because the design spends its strongest mechanisms on the top of the list and deliberately accepts exposure at the bottom.

Adversaries. Table 5 names the adversaries the design considers, their assumed capabilities, and the mechanism that answers each. The first is the one that shapes the architecture: if the provider

Table 3 Security requirements.

ID	Obj.	Requirement
SR1	O1	The backend and storage provider never receive a plaintext document, an unwrapped file key, or an unencrypted private key.
SR2	O2	Every mechanism protecting a stored file key uses standardized post-quantum primitives at NIST category 3 or above.
SR3	O2	File content is encrypted under an AEAD whose associated data binds the ciphertext to its own identity and format, so a ciphertext cannot be reinterpreted as a different object.
SR4	O2	Each key envelope is bound to its room, its file, and its single intended recipient, so an envelope replayed into another room, or presented for another recipient, fails to open.
SR5	O4	Room membership alone confers no read access, and the server does not disclose the existence, name, size, or count of files a requester holds no envelope for.
SR6	O2	A ciphertext modified in storage is rejected rather than returned as plaintext.
SR7	O5	Fewer than k custodian shares reveal nothing about a sealed file key, and an attempt to open with fewer is refused rather than silently returning a wrong key.
SR8	O6	Insertion, deletion, or reordering of audit events is detectable by a verifier that does not trust the backend.
SR9	O1	A profile, protection level, or binding version may never be silently downgraded; an unknown or retired identifier fails closed.

Table 4 Protected assets and where they exist.

Asset	Harm if disclosed	Where it exists in the clear
Plaintext documents	Direct loss of the confidentiality the system exists to provide	Client device only, transiently
File encryption keys	Equivalent to disclosing every document they protect	Crypto process only, per operation
User private keys (ML-KEM, ML-DSA)	Opens every envelope ever wrapped to that user; permits forgery of that user's signatures	Crypto process only, while unlocked
Custodian shares of a sealed key	k of them reconstruct a file key	Crypto process only, during a ceremony
Room membership and role data	Reveals who is party to a transaction	Backend (by necessity)
Access policies and classifications	Reveals the sensitivity of a transaction	Backend (by necessity)
Audit records	Reveal the pattern of activity; their *integrity* is itself an asset	Backend (by necessity)
File metadata (sizes, timestamps, envelope counts)	Supports traffic analysis about a transaction	Backend (by necessity)

were trusted, almost none of this machinery would be needed.

Table 5 Adversaries considered, and the mechanism that answers each.

Adversary	Assumed capability	Primary defence
A1 Compromised backend or storage provider	Reads and modifies all stored ciphertext, envelopes, metadata and audit rows; observes all API traffic; may drop, delay, or reorder messages	Client-side encryption (SR1); AEAD rejects modification (SR6). The audit chain is intended to detect rewriting (SR8) but as delivered does not against this adversary (Section 7.2)
A2 Malicious or compromised administrator	Full legitimate administrative authority over rooms and membership	Cannot grant itself an envelope it cannot decrypt (SR5); quorum seal denies even a full administrator a unilateral read (SR7)
A3 Unauthorized room member	A valid account, legitimate membership of the room, and its own key pair	Access follows envelope possession, not membership (SR5); server-side filtering hides what they cannot open
A4 Compromised user account	The victim's credentials and private keys	Blast radius limited to files actually wrapped to that user (SR5); quorum files still require $k - 1$ other custodians (SR7)
A5 Network attacker	Intercepts, replays, or modifies traffic between client and backend	TLS in transit; envelopes bound to room, file, and recipient, so a replayed envelope fails (SR4)
A6 Future quantum adversary	Records ciphertext and envelopes today; runs Shor's algorithm [1] years later	Post-quantum key encapsulation (SR2); AES-256 content keys retain a meaningful margin under Grover [2]

Trust assumptions. The guarantees hold only where the following assumptions hold. Each is a genuine assumption, not a claim.

1. **The client binary is authentic.** The user runs a genuine build. Substituting the client defeats every client-side guarantee at once, because the client is what performs the encryption.
2. **The endpoint is trusted while keys are unlocked.** An attacker with code execution on the device during an unlocked session can read plaintext and key material from process memory. The design shortens that window through process isolation and single-use key handling; it does not eliminate it.
3. **The cryptographic libraries are correct.** ML-KEM-768 and ML-DSA-65 as implemented by `@noble/post-quantum` and `liboqs`, and AES-256-GCM and HKDF as implemented by the platform's WebCrypto, are assumed to be faithful and free of exploitable side channels. No independent review of these was performed here.
4. **The underlying problems are hard.** ML-KEM-768 and ML-DSA-65 are assumed secure at NIST category 3. The quorum mechanism is the one exception: its share secrecy is information-theoretic and rests on no computational assumption at all.

5. **The user checks fingerprints where the interface asks.** The defence against public-key substitution by the backend (Section 4.4.2) reduces, in the first-contact case, to a human comparing a fingerprint out of band. A user who accepts every warning is not protected by it.
6. **Randomness is sound.** File keys, nonces, salts, and Shamir coefficients come from the platform CSPRNG.

Security goals. Against the adversaries above, and under the assumptions above, the system aims for: **confidentiality** of document content from the provider and from any party without an envelope; **integrity** of documents, envelopes, and audit history, in the sense that modification is detected rather than prevented; **authenticity** of the origin of each envelope and each audited action; **access isolation**, so that access to one file implies nothing about access to another; and **tamper detection** for the record of who did what.

3.3 Limitations and Out-of-Scope Threats

The following are outside what the design defends against. They are listed because a threat model that names only what it defeats is not a threat model.

Endpoint compromise. Malware, a keylogger, or a hostile operating system on the user's device during an unlocked session defeats client-side encryption at its source. This is the single largest residual risk.

Availability. The backend can refuse service, delete ciphertext, or withhold envelopes. Nothing here prevents that; the design ensures such behaviour cannot become a *confidentiality* breach, and that withholding audit events is detectable rather than silent.

Copies already downloaded. Revocation prevents future access through the platform. A recipient who has already decrypted a document holds a plaintext copy the system has no further authority over. The report says so rather than implying otherwise.

Metadata and traffic analysis. The backend necessarily learns room membership, file sizes, timestamps, envelope counts, and access patterns. An adversary observing this can often infer the shape of a transaction without opening a single document. No padding, batching, or oblivious access is implemented.

Forward secrecy. Recipient KEM keys are long-lived. A recipient private key recovered in the future opens every envelope ever wrapped to it. Session protocols avoid this by rekeying; an object encrypted once and stored for years cannot, without periodic re-wrapping (Section 8.1).

Single-family cryptographic failure. The shipped suite is post-quantum only. If ML-KEM-768 or ML-DSA-65 is broken, there is no second mechanism underneath. Section 6.6 measures what the hybrid alternative would have cost and Section 7 states the consequence.

Insider exfiltration by a legitimate reader. A recipient authorized to read a document can photograph the screen or retype its contents. Watermarks raise the cost; they do not close it.

Formal verification. No machine-checked proof, external audit, or cryptanalysis was performed. The security evidence is a regression suite written by the same people as the code, which is a weaker form of assurance and is treated as such in Section 6.8.

4 System Design and Security Architecture

4.1 Architecture

Quarom is built as three separate parts that can only talk to each other through narrow, checked channels (Figure 1). The split follows least privilege [36]: each part holds the minimum it needs, so compromising one does not hand over the others.

The first is a hardened Electron desktop shell. The window that shows the user interface runs with `contextIsolation` on, `nodeIntegration` off, the Chromium sandbox enabled, and a Content-Security-Policy that blocks scripts from anywhere else. Extensions are disabled and each window gets its own session.

The second is a separate crypto process. Every secret lives only here: private keys, unwrapped file keys, and session-locking secrets. The interface cannot read them; it can only ask, over IPC, for a named operation. Each operation has a strict input schema, a permission check, and a fixed output shape, and a request that matches no operation, fails validation, or comes from a window without the right room permission is refused before any cryptography runs. Results that would expose a secret are never handed back: plaintext goes straight to the file dialog the crypto process opened, and unwrapped keys are discarded after a single use. JavaScript cannot guarantee that memory is wiped, so the design does not claim perfect zeroization; it keeps every plaintext key alive as briefly as possible.

The third is the backend. It stores ciphertexts, envelopes, versioned metadata, membership records, and the audit chain, but never sees a plaintext document, a raw file key, or an unencrypted private key (SR1). Its job is to authenticate users, enforce access rules at the API level, and store what the client has already made safe. It is adversary **A1** of Table 5, not a trusted component.

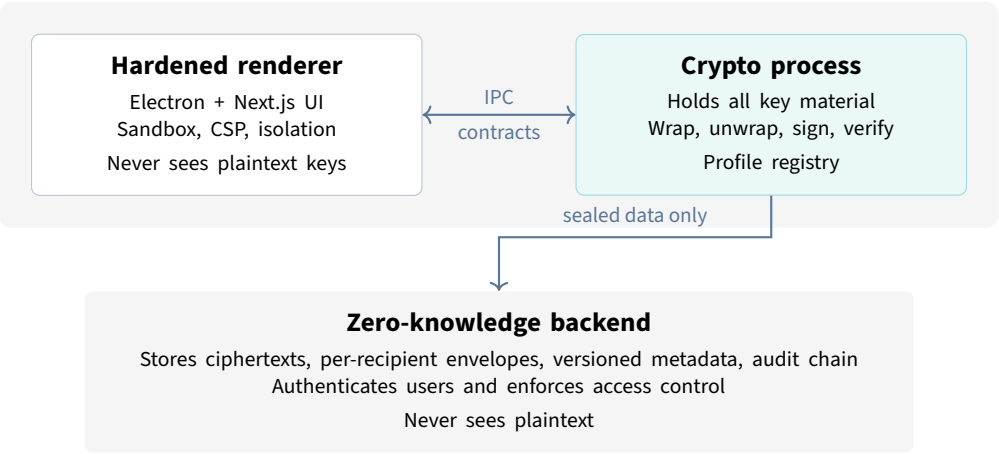


Figure 1 Quarom architecture. The renderer, crypto process, and backend form three isolation domains that communicate only through narrow, validated interfaces.

4.2 Cryptographic Protocol

The central idea is to keep *encrypting the file* separate from *sharing the key*. The file is encrypted once and stored once. Access is granted by handing out small per-recipient objects that can be added, revoked, or re-issued on their own (Figure 2).

On upload, the crypto process draws a fresh 256-bit file key and encrypts the document with AES-256-GCM [24]. That file key is then wrapped separately for each recipient. Sharing uses *key envelopes*, which hold the profile identifier, the file and room identifiers, owner and recipient identifiers, key versions, the ML-KEM ciphertext, the wrapped file key, a nonce, an HKDF salt, a reference to the ciphertext digest, an ML-DSA signature, and an access status. Adding a recipient

adds one envelope; revoking one marks that envelope revoked. The stored ciphertext changes in neither case.

4.2.1 What each layer authenticates

AES-GCM is an AEAD, so besides encrypting it authenticates an *associated data* (AAD) string that is not encrypted but must match exactly at decryption [25]. Choosing what goes into that string is what binds a ciphertext to its context, and Quarom uses it at two levels that must not be confused with one another.

The distinction matters because the two levels have opposite requirements. The *file ciphertext exists once and is read by every recipient*, so its AAD cannot mention any recipient: if it did, the ciphertext would decrypt for one person and fail for everyone else, and the "encrypt once, share many" property of FR3 would collapse into one copy per recipient. The *envelope exists once per recipient*, so its AAD is exactly where the recipient, and the room, are bound. Table 6 gives the three authenticated contexts as they are constructed in the code.

Table 6 The three authenticated contexts, as constructed in `web/src/lib/crypto/hpke.ts`. Every field is covered exactly once, at the level whose object it actually describes.

Object	Protected by	Bound fields
File ciphertext, one per file	AES-256-GCM, AAD	Domain tag `kaust-pqc-file`; file crypto version; file id; owner id; filename; size; cipher name. *No recipient, no room, no profile.*
Wrapped file key, one per recipient	AES-256-GCM, AAD, and the same bytes as HKDF-SHA-384 `info`	Domain tag `kaust-pqc-room-file-key-wrap`; room-binding version; canonical room id; file crypto version; file id; owner id; **recipient id**; KEM, KDF and cipher names
Envelope as a whole, one per recipient	ML-DSA-65 signature	Domain tag `kaust-pqc-room-file-envelope`; room-binding version; canonical room id; file crypto version; file id; owner id; filename; size; the file's AEAD nonce; **SHA-384 of the file ciphertext**; recipient id; all algorithm names; and every envelope byte field

Three consequences follow, and they are the substance of the security argument for sharing.

One ciphertext really is one ciphertext. Because the file AAD is independent of both recipient and room, the stored bytes are identical no matter how many people can read the document, which is what makes FR3 achievable at all.

The recipient binding is in the envelope, and it is doubled. The recipient identifier appears in the key-wrap context, which is used both as the HKDF-SHA-384 `info` that derives the wrapping key [26, 27] and as the AAD of the AES-GCM operation that wraps the file key. An envelope presented on behalf of a different recipient therefore fails twice over: a different `info` derives a different wrapping key, and the GCM tag does not verify against the altered AAD. It fails a third time at the ML-DSA-65 signature, which also covers the recipient identifier.

Moving a file between rooms fails at the envelope, not at the file tag. The room identifier is bound in the key-wrap context and in the envelope signature, not in the file AAD. Moving a ciphertext row to another room therefore leaves the file's own GCM tag verifiable, and this is unavoidable: the tag is over bytes that never mentioned a room. What breaks is access, which is the property that matters. No envelope in the destination room derives the right wrapping key, and every envelope carried across fails signature verification because it commits to the original room's identifier. The practical effect, that a relocated file cannot be read, is the same; the mechanism is the envelope layer, and the report states it that way because which layer fails determines what an attacker can and cannot do with the row they moved.

Finally, the envelope signature commits to the SHA-384 digest of the file ciphertext. That is the link back down: an envelope cannot be re-pointed at a different ciphertext, and a substituted ciphertext is detected when the digest is checked before decryption.

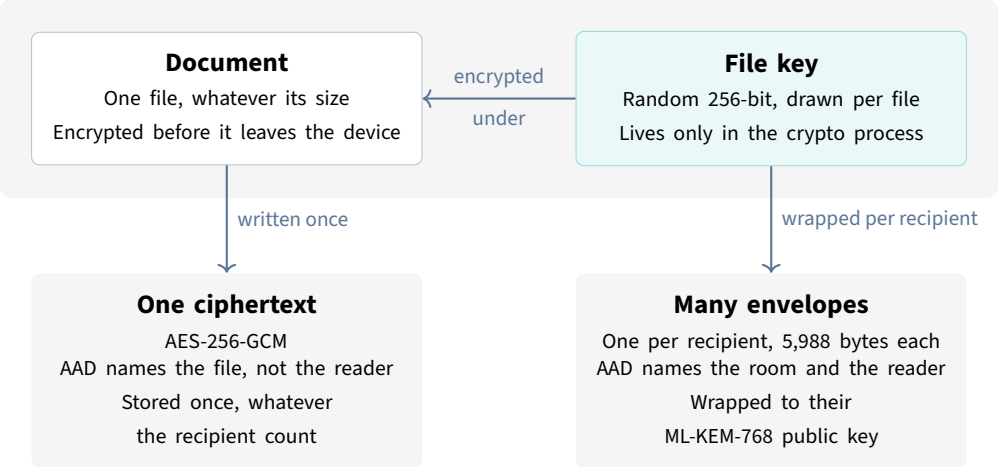


Figure 2 One ciphertext, many envelopes. The document is encrypted once under a random file key whose associated data describes the file alone. That key is wrapped separately for each recipient, and it is the envelope's associated data that names the room and the reader, so adding a recipient costs one small envelope instead of a second copy of the file.

Every cryptographic object also carries a version tag. This lets old objects stay readable after a format change, because a signature is checked against the algorithm named in its own tag and a ciphertext is decrypted according to its own recorded version.

4.2.2 The Protection Profile

Client-side cryptography sits behind a small provider interface that offers keypair generation, envelope wrap and unwrap, signing, and verification, all registered under one stable identifier naming the algorithm suite in use. Keeping the suite behind this seam means the code that calls cryptography never names an algorithm directly, and it gives every authenticated context a single value to bind to. Unknown or disabled identifiers are rejected outright rather than ignored.

Quarom registers one profile, `POST_QUANTUM`: ML-KEM-768 (FIPS 203) for key encapsulation and ML-DSA-65 (FIPS 204) for signatures, with AES-256-GCM as the file cipher, HKDF-SHA-384 for key derivation with a 32-byte per-envelope salt, and SHA-384 for digests. An envelope carries an ML-KEM ciphertext from which the recipient recovers the shared secret that unwraps the file key, and is authenticated by an ML-DSA-65 signature over its canonical form. The profile lists `mlkem768` and `mldsa65` as required parts, so an envelope missing either fails closed instead of being accepted in a weakened form.

The identifier deliberately carries no version suffix. A profile names an algorithm suite; every wire and storage format is versioned separately, on the object itself. Putting a version on the profile name as well would duplicate information held elsewhere and invite the two to disagree, so a format can change without implying a new suite and a suite could be retired without rewriting stored objects. Any value other than `POST_QUANTUM` is rejected as unknown rather than resolved to the registered one, which is what turns a substitution attempt into an error instead of a downgrade. Choosing a post-quantum-only suite, rather than the hybrid that comparable systems favour (Section 2.3), was done against measured alternatives rather than by assumption.

The profile also declares its own weaknesses, so they show up in documentation and in the interface rather than only in prose. Recipient KEM keys are static, so there is no forward secrecy: a recipient private key recovered in future opens every envelope ever wrapped to it. And a single-family suite is a single point of failure: if ML-KEM-768 or ML-DSA-65 is broken later, there is no second mechanism to fall back on. Both are accepted and documented consequences of the design rather than oversights.

On the Electron and Next.js client the primitives come from `@noble/post-quantum` 0.6.1, a pure-JavaScript library exposing `ml_kem768` and `ml_dsa65`. On the Python `pqc-service` backend they come from `liboqs-python`. Every call into a cryptographic library happens inside the crypto process, so key material never crosses the IPC boundary.

4.3 Data Rooms and Access Control

The unit of collaboration is a *room*: a named set of encrypted files with a member list, role assignments, and a versioned protection policy. Roles separate an *owner* (policy, membership, revocation) from an *administrator*, a *contributor* (upload and share) and a *viewer* (download and decrypt). The backend checks these rules on the server, so a tampered-with client cannot promote itself.

Rooms are enforced at two independent layers, and it is worth being precise about which does what. *Authorization* is server-side: the backend decides who may call which operation on which room. *Cryptographic room binding* is client-side: as Section 4.2.1 describes, the room identifier is bound into each envelope's key-wrap context and signature. The first can be bypassed by compromising the backend; the second cannot, which is why both exist. A backend that ignores its own authorization rules and hands an attacker every envelope in a room still hands them objects that only open with private keys it does not have.

Each room carries a sensitivity classification and a policy naming the weakest profile it will accept. Because the shipped platform registers only one profile (Section 4.2.2), this check runs but does not yet choose between live options. It rejects unknown and retired profiles instead (SR9).

Revocation. The normal tier is *future-access revocation*. The envelope is marked `revoked`, download links stop being issued, and the event is written to the audit chain with actor, reason, and time. The ciphertext is untouched, so other recipients are unaffected. This cannot erase a copy someone already downloaded, and Quarom says so plainly rather than implying otherwise. The stronger tier is *file-key rotation*. The crypto process decrypts the file internally, generates a new AES-256 key, re-encrypts it locally, and issues envelopes only to the remaining recipients. The old version is kept until the new one checks out. Plaintext never passes through the interface or the backend.

4.4 Key Management

4.4.1 User key lifecycle

User KEM and signing keys carry version tags and are managed inside the crypto process, and a new keypair becomes active only after it validates. Changing a password re-wraps the existing private keys under a new password-derived key rather than issuing new ones, so old envelopes still open. Retired *public* verification keys are kept forever so signatures on past events stay checkable, and envelopes wrapped under an out-of-date key version are flagged for re-wrap. The separation of key roles, and the retention of verification keys beyond the life of the signing key, follow standard key-management practice [30].

4.4.2 Recipient key authentication

Every envelope rests on one assumption not yet examined: that the public key it is wrapped to really belongs to the intended recipient. The backend is what hands out public keys. A dishonest one (**A1**) could publish its own KEM key under someone else's name and then read the file without ever touching a ciphertext. The bindings in Section 4.2.1 tie an envelope to a recipient *identifier*, not to a *verified key*.

Quarom answers this in three layers. Every published key travels in a *self-signed bundle* covering the user identifier, key type and role, key version, algorithm, profile, and creation time, so the backend cannot mint a key for a user whose signing key it lacks. That does not help a client meeting the user for the first time, since a fake identity can sign itself, so the client also derives a *fingerprint* (a SHA-384 digest of the member's published key set), shows it as a short comparable string, and pins it on first contact. Any later change raises an audited warning the user must accept.

These two layers make substitution detectable *after* first contact, and no more than that. A client that has already pinned a peer's fingerprint notices any later swap. A client meeting a peer for the first time has nothing to compare against except an out-of-band check by the user, and a self-signature proves only that whoever generated the key also signed the bundle, which a backend minting a fresh identity satisfies trivially. Closing the first-contact case needs a third layer the system does not have: an append-only directory of key publications, in the manner of Certificate Transparency [33] or CONIKS [34], against which clients could compare what they were told. Section 7.3 states the resulting exposure precisely and Section 8.1 treats it as work to be done.

4.5 Quorum Access

An ordinary envelope wraps the whole file key to one recipient, which makes that recipient a complete capability: their key alone opens the document. For documents governed by a two-person rule that is the wrong shape, because it means a single compromised account (**A4**), or a single coerced custodian, is enough.

Where a classification demands it, the file key is therefore not wrapped whole to anybody. It is split with Shamir secret sharing over $\mathrm{GF}(2^8)$ [9, 29] into n shares under a threshold k, and *one share* is wrapped to each custodian using the same envelope machinery as Section 4.2. Any k custodians can reconstruct the key; any $k-1$ learn nothing whatsoever about it, and that last claim is information-theoretic rather than computational, so unlike everything else in this report it rests on no hardness assumption that a quantum computer could disturb.

Opening such a file is a ceremony rather than an operation. The requester generates a throwaway ML-KEM keypair for that one request; k approvers each re-wrap their share to that ephemeral key and sign the approval with ML-DSA-65. The backend relays these messages and can drop, delay, or reorder them, but it cannot forge an approval and never sees a share in the clear,

so even a fully compromised backend remains a messenger rather than a participant. Section 5.3.3 describes what the implementation adds to make this safe in practice.

4.6 Auditing

Auditable events, such as uploads, shares, downloads, revocations, policy changes, and file-key rotations, go into a hash-chained log. Each event is serialized in one canonical way, so the same action always produces the same bytes. Each event also carries the actor, target, operation, a counter, a timestamp, and the hash of the event before it. Because every event commits to its predecessor, altering, inserting, deleting, or reordering anything forces every later hash to change, and verification catches it. Verification returns one of six explicit failure codes rather than a single true or false, and a portable export with an offline verifier lets an auditor confirm a room's history from a snapshot without access to the running system. The construction is the classical hash chain used for tamper-evident timestamping [31, 32].

The chain has one structural limit, and it is the reason SR8 is only partly met. Its hashes are *keyless*: nothing in them depends on a secret, so any party able to write rows can recompute a consistent chain. It therefore binds an attacker who can alter stored data but not recompute it, and does not bind the backend, which holds every row. Crosby and Wallach make the general point that a log must be anchored outside the logger's control before its guarantees constrain the logger itself [32]; this one is not. Section 7.2 sets out exactly what that costs, and Section 8.1 the two mechanisms that would close it.

5 Implementation

Section 4 described the design. This section reports what was actually built, written against the state of the repository rather than the plan. Where the two differ, the difference is stated, and Section 5.4 collects every such difference into one table so that no claim in this report has to be reconciled by reading the whole of it.

5.1 Technology Stack

The prototype is a four-part system in one repository.

Table 7 Delivered components and their test coverage, counted from the repository at the commit this report describes.

Component	Technology	Test files
`web/`	Next.js interface, provider abstraction, browser crypto	50
`desktop/`	Electron shell, IPC contract, crypto utility process	41
`pqc-service/`	FastAPI authentication, authorization, validation	93
`supabase/`	Schema, envelopes, audit chain, storage (77 migrations)	—

The cryptographic primitives come from two libraries. On the Electron and Next.js client they come from `@noble/post-quantum` 0.6.1, a pure JavaScript implementation exposing `ml_kem768` and `ml_dsa65`; AES-256-GCM, HKDF-SHA-384 and SHA-384 come from the platform WebCrypto. On the Python `pqc-service` backend the post-quantum primitives come from `liboqs-python`, used only for verification, never for key custody. Argon2id comes from a WebAssembly build. That the client library is pure JavaScript matters for Section 6.4: the measured costs are those of a portable implementation, not of optimized native code.

Client cryptography lives in `web/src/lib/crypto/` and is written without any tie to the DOM or the framework, so the same modules run unchanged inside the Electron crypto process. That is what lets the two modes share one set of wire formats while offering different isolation guarantees. Browser mode performs the same operations inside the page, and does not claim the process-level key isolation Electron mode provides.

5.2 Implemented Architecture

The Electron side is built as four roles rather than the two in Figure 1. **Main** owns the window lifecycle, the dedicated session, native file dialogs, IPC policy, and the session access token, which the interface never receives. The **preload bridge** exposes a small typed API. The **renderer** holds only account metadata, opaque capability identifiers, and status. The **crypto utility process** does derivation, key custody, envelope and AES work, and bounded file streams. Before dispatching any IPC request, main checks the expected WebContents, main-frame identity, exact renderer origin, deployment generation, session, and lifecycle state, and refuses unknown operations, malformed identifiers, replayed requests, oversized messages, and stale work. File selection and output use one-time opaque capabilities, so no file bytes or absolute paths ever reach the renderer.

Versioned objects and room binding. Room binding is itself versioned, and that is how files created before rooms existed were brought into them without re-encryption. Binding `0` covers personal envelopes and envelopes attached to a room as pure metadata, while binding `1` is a new room-bound envelope whose HKDF context and ML-DSA signature both cover the room UUID, exactly as Table 6 sets out. Because both bind the room, moving an envelope row to another room, or replaying it under a different room URL, causes unwrap or verification to fail. A binding-1 envelope is never retried as binding 0 after a failure, so there is no quiet downgrade path (SR9).

Password-derived protection. Argon2id runs at three iterations, 65,536 KiB of memory, parallelism four, and a 32-byte output [28]. HKDF-SHA-384 then derives two independent branches: an authentication key, whose Base64 proof FastAPI checks against a stored verifier, and a key-encryption key that wraps the ML-KEM and ML-DSA private keys under AES-256-GCM. The password, master key, key-encryption key, and unwrapped private keys are never sent anywhere.

5.3 Main Features

5.3.1 The protection profile as built

`POST_QUANTUM` is the only profile the platform accepts, enforced at four levels rather than by convention. The TypeScript and Python registries list it alone, so looking up any other identifier raises instead of falling back to something weaker. The envelope parser rejects unknown identifiers. Every database column that can hold a profile identifier, crypto version, or signature mode is constrained to `POST_QUANTUM`. And the account key lifecycle issues post-quantum key material only. Refusing an unknown identifier, rather than mapping it to the registered one, is what makes this an enforced property instead of a habit: a parser that quietly resolved an unknown label would accept whatever an attacker sent, so failing closed turns a substitution attempt into an error instead of a downgrade.

The registered parameters follow the standards directly, in bytes: ML-KEM-768 (FIPS 203) with a 1184-byte public key, 2400-byte secret key, 1088-byte ciphertext and 32-byte shared secret; ML-DSA-65 (FIPS 204) with a 1952-byte public key, 4032-byte secret key and 3309-byte signature; AES-256-GCM with a 32-byte key, 12-byte nonce and 16-byte tag; HKDF-SHA-384 with a 32-byte per-envelope salt; and SHA-384 with a 48-byte output.

5.3.2 Need-to-know sharing

The room model of Section 4.3 was changed during implementation in a way that strengthens the security claim: room membership no longer grants file access. An ordinary member lists and fetches only the files they hold an active signed envelope for. The server applies that filter before returning identifiers, filenames, counts, sizes, owners, or timestamps, and a member with no envelope gets an empty list with no placeholder from which a hidden file count could be worked out. Guessing a file ID runs into the same check. Owners and administrators keep an audited full-inventory permission, but still need their own envelope to download.

Moving from "member of the room" to "holder of an envelope" shrinks the damage a stolen recipient key can do (adversary **A4**): it now exposes only the files wrapped to it rather than every file in every room that user belongs to, and the gap grows the longer a room lives.

Sharing several files with several named recipients is committed all-or-nothing. It can be resumed using a UUIDv4 batch identifier plus a request hash the backend computes itself, so a partial failure is retried without creating duplicate envelopes. The API accepts at most 1,000 recipients per batch, and the crypto process serves at most four concurrent requests rather than an unbounded fan-out. That bound matters for the reason Section 6.4.3 measures: a full batch is tens of seconds of continuous cryptography, so the work has to be paced and reported rather than issued all at once.

5.3.3 Quorum-sealed files

Section 4.5 gave the design. Three implementation decisions are what make it safe in practice.

The Shamir primitive is implemented over the AES/Rijndael polynomial $x^8 + x^4 + x^3 + x + 1$. Multiplication and inversion are branch-free and table-free. The usual log/exp-table version is faster but looks up tables using secret-derived values, which leaks through cache timing, so that speed is given up on purpose. Section 6.4.4 shows what it cost, which is almost nothing.

A threshold below two is rejected, because then every share would be a complete copy of the secret. Duplicate custodians are rejected too, because merging them would hand one person two shares and quietly turn a k-of-n rule into one they could satisfy alone.

Because interpolating too few shares silently returns a plausible *wrong* key instead of failing, the file key and each share carry a SHA-384 commitment recorded at seal time and compared in constant time. This is what lets the system tell "you need one more custodian" from "a share was corrupted", a distinction Shamir cannot make on its own.

Each approver signs their approval with ML-DSA-65, which is non-repudiable evidence that they released their share to that specific request. The request identifier and a SHA-384 digest of the throwaway public key are bound into both the signature and the AES-GCM associated data, so an approval from one ceremony is meaningless noise in another.

5.3.4 Auditing as built

The canonical Version 1 event schema and its cross-language hashing are implemented in both TypeScript and Python against a shared fixture, so client and server agree byte-for-byte on an event's hash. Verification returns one of six explicit failure codes rather than a single true or false, and appends are serialized through a locked per-room chain head. A portable export and an offline verifier are implemented, so an auditor can check a room's history from a snapshot without access to the running system.

Events are emitted with the signature profile `UNSIGNED_V1` and report a signature status of `UNAVAILABLE` rather than `VALID`. The interface shows that status rather than hiding it, which is the difference between an honest absence and a false assurance. Section 7.2 analyses what an

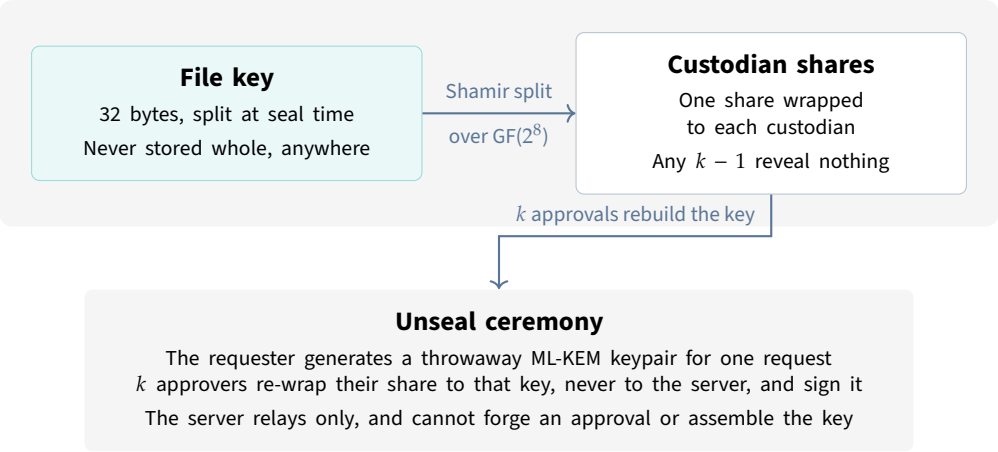


Figure 3 Quorum sealing. The file key is split into shares and one is wrapped to each custodian, so no single person holds a complete capability. To open the file, a requester makes a throwaway keypair and k approvers re-wrap their shares to it. The server only passes messages along.

unsigned, keyless chain does and does not constrain.

5.4 Implementation Status and Traceability

Table 8 maps every requirement of Section 3.1 to its implementation status and to the evidence in Section 6. Everything this report describes as part of the system is implemented; the table exists so a reader can tell at a glance which security claims it is entitled to make, and where one is qualified.

5.5 The Delivered Application

This section shows the delivered client. Every screen below is the Electron desktop application talking to the local `pqc-service` backend, with all cryptography performed in the separate crypto process described in Section 5.2.

Table 8 Requirement traceability. **Met** means implemented and covered by automated tests; **Partial** means implemented with a stated limitation.

Req.	Status	Evidence and qualification
FR1–FR4	Met	Upload, download, share, revoke covered by client and API tests (Section 6.2)
FR5, FR6	Met	Roles and versioned policies enforced server-side; only one protection profile is registered, so the policy check rejects unknown profiles rather than choosing among live ones
FR7	Met	Seal and unseal ceremony implemented and measured (Section 6.4.4)
FR8	Partial	Chain, canonical hashing, export, and offline verifier are built; live events are emitted *unsigned* and report `UNAVAILABLE` rather than `VALID` (Section 7.2)
FR9	Met	Cross-organization sharing implemented with a ceiling on the granted role
SR1–SR4	Met	Enforced by construction and covered by negative tests (Section 6.3)
SR5	Met	Server-side need-to-know filtering, verified at the API boundary
SR6, SR7	Met	Tamper and insufficient-quorum rejection covered by tests
SR8	Partial	Insertion, deletion and reordering are detected *against a party that cannot recompute the chain*. The chain is keyless and events are unsigned, so it constrains neither rewriting nor truncation by the backend itself, which is the adversary SR8 names (Section 7.2)
SR9	Met	Fail-closed profile and binding-version handling at four levels

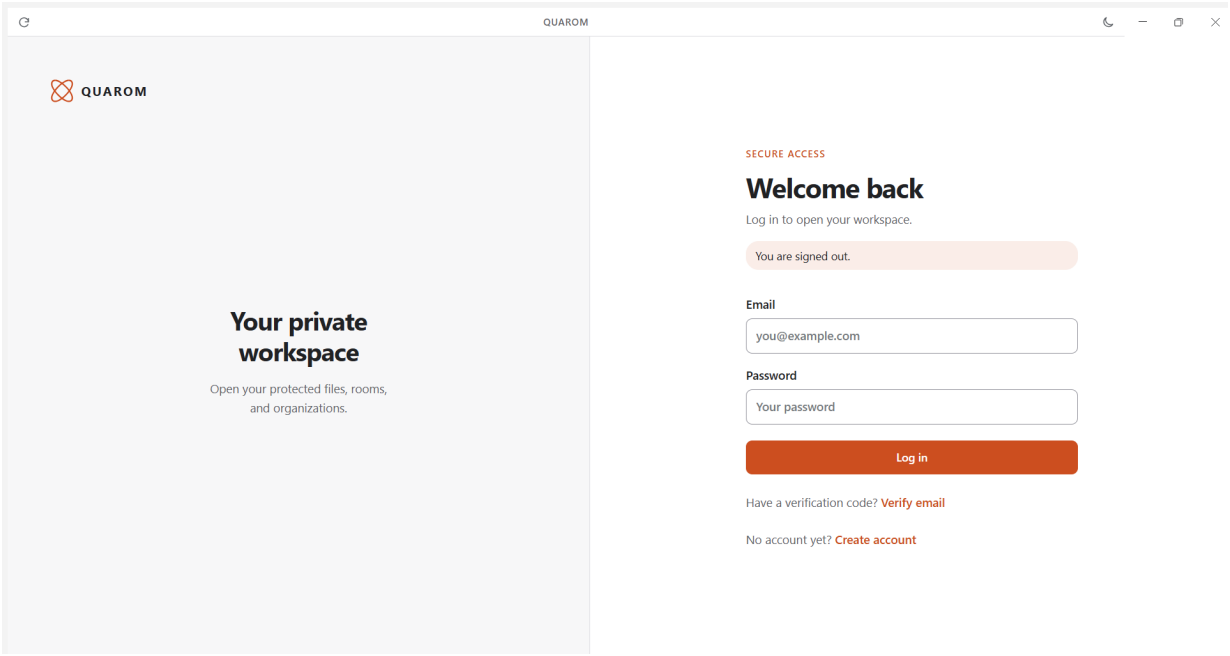


Figure 4 Sign-in. Private keys live only in the crypto process and are discarded on any lifecycle event, so the client returns to this screen whenever the renderer reloads or the application restarts, and signing in again derives the keys from the password rather than restoring a cached copy. The control at the top left refreshes the current view without reloading the document, precisely because a document reload would discard those keys.

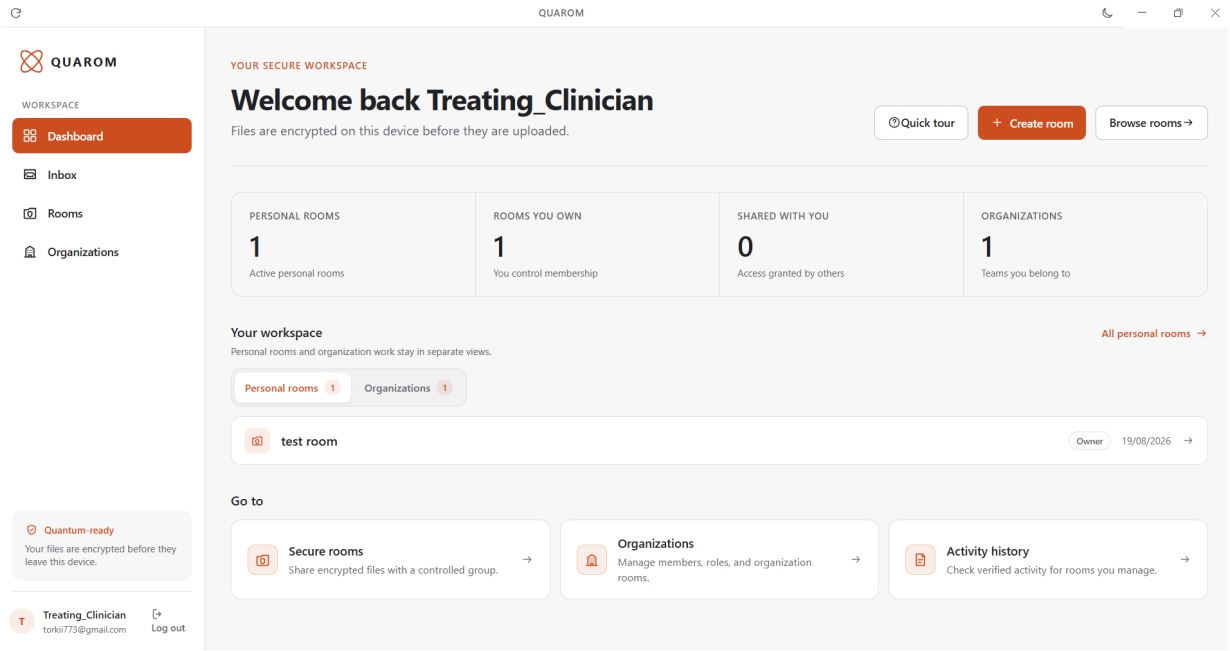


Figure 5 The workspace dashboard after signing in. Personal rooms and organization work are kept in separate views, and the counts distinguish rooms the user controls from rooms shared with them, which is the membership side of the access model. The reminder that files are encrypted on the device before upload is stated in the interface rather than only in documentation.

6 Evaluation and Results

Objective O7 was to evaluate the prototype for functional correctness, security behaviour, performance, and storage overhead, and to state plainly which claims the evidence supports. This section does that in that order. Section 6.8 then sets out what the evidence does *not* establish, which for a security system is the more important half.

6.1 Test Environment

All measurements were taken on one Windows host: an 11th-generation Intel Core i5-1155G7 at 2.50 GHz (4 cores, 8 logical), 7.8 GB RAM, Windows 11 Home (build 10.0.26200), Python 3.14.6, Node.js v24.18.0. Absolute times belong to that machine and to a pure-JavaScript cryptographic library (Section 5.1); optimized native code would be faster, and the comparisons between operations would be expected to survive that change while the absolute figures would not. The sizes in Section 6.5 are exact and machine-independent, since they follow from the FIPS parameter sets.

Cryptographic timings come from a harness that drives the same modules the shipped client uses, so the figures are the cost of the delivered code path rather than of a stand-in. It accompanies this report in `benchmark/` rather than the platform repository, and is run from inside the platform's `web` workspace as Appendix A.1 describes; its raw output is reproduced in Appendix A.5. Every figure is a median: of 200 rounds for the primitives, of 25, 5 or 3 rounds for the sharing batches according to size, and of 20 for the quorum ceremonies. The harness performs 30 untimed warm-up envelope creations before sampling, because without them the smallest batch is measured against a cold JIT and reports a higher per-envelope cost than the largest, which would misstate the very scaling the test exists to establish. Audit-chain timings come from `scripts/benchmark-audit-chain.py`.

6.2 Functional Tests

6.2.1 Automated verification

A full run on the host above passes **2,166 tests with none failing**: 1,390 in `pqc-service` under pytest (38 skipped, 49.4 s), 465 in `web` under vitest (1 skipped, 8.3 s), and 311 in `desktop` under `node:test` (14.8 s), across 184 test files.

Table 9 Test execution summary.

Suite	Files	Passed	Failed	Skipped
`pqc-service` (pytest)	93	1,390	0	38
`web` (vitest)	50	465	0	1
`desktop` (node:test)	41	311	0	0
Total	**184**	**2,166**	**0**	**39**

Line or branch coverage is not reported. The suite is organized around behaviours rather than lines, and a coverage percentage over a codebase that includes generated database migrations and framework glue would flatter the result rather than inform it. What is reported instead is which *requirement* each group of tests exercises, which is the question a reader of a security report actually needs answered.

6.2.2 Functional coverage by requirement

Table 10 maps each functional requirement to representative automated tests. The full flow, encrypt, upload, share, list, download, decrypt, revoke, is exercised end-to-end within the client and against the API, with the crypto path covered at the module level and the authorization path at the API boundary.

Table 10 Functional requirements and representative evidence.

Req.	Representative test
FR1, FR2	`hpke.test.ts`: "encrypts one blob and lets owner and recipient unwrap the same file key"
FR3	`roomSharing.test.ts`: one ciphertext, one envelope per named recipient, with duplicates collapsed
FR4	`test_admin_room_access.py`: `test_revoke_room_access_marks_removed`
FR5, FR6	`test_room_file_api.py` and the room policy suites: role enforcement and versioned policy evaluation server-side
FR7	`quorumEnvelope.test.ts`: seal to k-of-n, open k shares, reconstruct and verify against the commitment
FR8	`test_audit_chain.py`: `test_verifies_valid_chain_and_returns_next_page_cursor`; offline export verified by `scripts/verify-audit-export.py`
FR9	`test_cross_org_sharing.py`: share, accept, grant up to the ceiling, revoke, and re-share

6.3 Security and Negative Tests

Functional tests show the system works when used correctly. The security argument rests on what it does when used incorrectly, so the attack-resistance properties of Section 3 are written as executable regressions. Table 11 lists the negative tests that correspond to the security requirements, each of which asserts a *rejection*.

One attack in the threat model is *not* in this table, and its absence is the point: truncation of the audit chain by the backend (Section 7.2). There is no test asserting that it is rejected, because it is not.

6.4 Performance Benchmarks

6.4.1 Post-quantum primitives

Table 12 gives the cost of each primitive in isolation. The result that shapes everything else is the asymmetry between the two families: key encapsulation is cheap and signing is not.

ML-KEM-768 costs well under a millisecond in every direction. ML-DSA-65 signing costs *nineteen times* an encapsulation, and its signature is three times the size of a KEM ciphertext. The post-quantum penalty in this system is therefore not where it is usually assumed to be: it is not in key establishment, which is the part people expect lattice cryptography to make expensive, but in signatures. Section 6.5 shows the same asymmetry in bytes.

Table 11 Security requirements and the negative tests that enforce them. Every row asserts that the operation is refused.

Attack	Req.	Test and outcome
Tampered ciphertext	SR6	`hpke.test.ts`: "rejects a wrong recipient key, modified metadata, and modified ciphertext" — GCM tag fails, no plaintext returned
Wrong recipient	SR4	Same test: an envelope opened with another recipient's key fails at key derivation and at the tag
Envelope replayed into another room	SR4	`hpke.test.ts`: "rejects wrong-room and cross-file envelope replay" — the room id in the wrap context and signature does not match
Tampered binding or signature	SR4, SR9	`hpke.test.ts`: "rejects tampered binding, signature, and ciphertext" — no binding-0 retry after a binding-1 failure
Encrypted filename replayed onto another file	SR3	`hpke.test.ts`: "fails closed when the encrypted name is replayed against a different file id"
Unknown or downgraded profile	SR9	`envelope.test.ts`: "rejects an unknown profileId"; database constraints reject the value independently
Member without an envelope	SR5	`test_room_file_api.py`: `test_need_to_know_listing_and_known_id_fetch_fail_closed_at_api_boundary` — empty list, and a guessed file id is refused
Revoked recipient still holding an envelope	SR5, FR4	`test_revocation.py`: `test_assert_download_authorized_blocks_a_revoked_recipient`, and a companion asserting a revoked recipient is refused *identically* to an outsider, so the refusal leaks nothing about prior access
Fewer than k custodians	SR7	`quorumEnvelope.test.ts`: "$k-1$ custodians cannot open a file with a higher threshold" — refused before interpolation
Duplicate custodian, threshold below 2	SR7	`quorumEnvelope.test.ts`: "refuses a duplicate custodian", "refuses a threshold below 2"
Share envelope replayed or forged	SR7	`quorumEnvelope.test.ts`: "rejects a share envelope replayed into another room"; "refuses a share envelope signed by the wrong sealer"
Audit event modified	SR8	`test_audit_chain.py`: `test_detects_modified_event_content`
Audit row deleted or reordered	SR8	`test_audit_chain.py`: `test_detects_missing_row_and_reordering`
Audit event or sequence duplicated	SR8	`test_audit_chain.py`: `test_detects_duplicate_event_id_and_duplicate_sequence`
Audit event moved between rooms	SR8	`test_audit_chain.py`: `test_detects_wrong_room_and_wrong_previous_hash`
IPC from an unapproved origin or stale window	—	`ipc-validation.test.ts`, `ipc-authorization.test.ts`: refused before any cryptography runs

Table 12 Post-quantum primitive cost, median of 200 rounds, pure-JavaScript implementation.

Scheme	Operation	Median (ms)	Object size (B)
ML-KEM-768	Key generation	0.559	1184 public / 2400 secret
ML-KEM-768	Encapsulate	0.682	1088 ciphertext
ML-KEM-768	Decapsulate	0.546	32 shared secret
ML-DSA-65	Key generation	2.39	1952 public / 4032 secret
ML-DSA-65	Sign (1 KiB)	**13.23**	3309 signature
ML-DSA-65	Verify (1 KiB)	7.32	—

6.4.2 File encryption and decryption

Table 13 File operation cost by size. Decryption includes opening the recipient's envelope: one ML-DSA-65 verification, one ML-KEM-768 decapsulation, an HKDF derivation, and the AES-GCM unwrap.

	Encrypt		Decrypt (envelope + file)	
Size	Median (ms)	MiB/s	Median (ms)	MiB/s
1 KiB	0.565	—	12.84	—
64 KiB	1.10	57	12.38	5
1 MiB	8.24	121	19.91	50
10 MiB	70.63	142	88.45	113
100 MiB	821.8	122	769.7	130

Encryption is bandwidth-bound above about 1 MiB, settling at roughly 120–140 MiB/s. Decryption shows something more interesting: it has a floor of about 12 ms that does not depend on file size at all. A 1 KiB file and a 64 KiB file both take about 12.5 ms, and almost all of it is opening the envelope rather than decrypting the document, which the primitive timings account for directly: 7.32 ms of ML-DSA-65 verification plus 0.55 ms of ML-KEM-768 decapsulation plus key derivation.

This is why the "5 MiB/s" in the 64 KiB row must not be read as a bandwidth figure. It is a fixed cost divided by a small file. The practical consequence is that the post-quantum overhead is felt on *small* files, where it doubles or triples an otherwise instant operation, and disappears on large ones, where it is under 2% of the total. A data room holding many small documents pays a noticeably higher relative cost than one holding a few large ones, and the same pattern appears again in storage.

6.4.3 Sharing scalability

Table 14 Envelope creation against recipient count, measured after warm-up.

Recipients	Rounds	Median total (ms)	Per envelope (ms)	Envelopes/s
1	25	28.19	28.19	35.5
10	25	364.3	36.43	27.4
100	5	3,944.6	39.45	25.4
1000	3	38,498.7	38.50	26.0

Cost grows linearly with recipient count and per-envelope cost is flat across three orders of magnitude, settling at roughly 38 ms per envelope, or about 26 envelopes per second on a single

thread. ML-DSA-65 signing accounts for 13.2 ms of that; the remainder is one KEM encapsulation, a per-envelope HKDF derivation, an AES-GCM wrap, and the Base64 and JSON encoding of the canonical signature message, which were not profiled separately.

The operational consequence is concrete. Sharing one file with 1,000 recipients is about 38 seconds of continuous client-side cryptography on this hardware. That is long enough that the bounded concurrency and the progress reporting described in Section 5.3.2 are a correctness requirement for the interface rather than a nicety: without them the application would appear hung, and an impatient user retrying would double the work.

6.4.4 Quorum sealing and unsealing

Table 15 Quorum ceremony cost, median of 20 rounds. Sealing writes n share envelopes; unsealing opens k of them and interpolates.

Configuration	Seal (ms)	Per share (ms)	Unseal (ms)	Shamir split/combine (ms)
2-of-3	28.57	9.5	6.81	0.039 / 0.003
3-of-5	61.39	12.3	10.80	0.051 / 0.005
5-of-9	91.34	10.1	16.71	0.048 / 0.005

Sealing is linear in the number of custodians n, at roughly 10 ms per share envelope, and unsealing is linear in the threshold k. Both are fast enough that a two-person rule costs the user nothing perceptible: a 5-of-9 seal completes in under a tenth of a second.

The most useful result here concerns the deliberately slow implementation choice of Section 5.3.3. Splitting a 32-byte key takes 0.039–0.051 ms and recombining it 0.003–0.005 ms, so the branch-free, table-free field arithmetic chosen to avoid a cache-timing side channel accounts for well under one percent of the ceremony. The constant-time implementation was adopted on principle; the measurement shows the principle was free. Everything else is the envelope machinery the shares travel in, dominated once again by ML-DSA-65.

6.4.5 Audit chain verification

Chain verification is the one operation whose cost grows without limit as a room ages, so it was measured directly.

Table 16 Audit chain verification, full-page verification with `valid=True` on every run.

Events	Time (s)	Events/s
1,000	0.0641	15,603
2,000	0.1383	14,465
5,000	0.3837	13,032
10,000	0.7490	13,350

Verification is close to linear, between 13,000 and 15,600 events per second, with per-event cost rising only slightly across a tenfold increase in chain length. A room with 10,000 audited actions verifies its whole history in under 0.75 seconds, so full verification on every read stays affordable and need not be traded for incremental checking. Note that this measures *chain* verification only: because live events are emitted unsigned (Section 7.2), no signature verification is included in these times. Signing events, which Section 8.1 argues for, would add one ML-DSA-65 verification of about 7.3 ms per event unless that cost were amortized across a batch, so it is a change with a measurable price rather than a free one.

6.5 Storage Overhead

Storage is where the post-quantum choice shows most, and it can be computed exactly rather than measured. An envelope carries an ML-KEM-768 ciphertext, the wrapped file key with its GCM tag, a nonce, an HKDF salt, and an ML-DSA-65 signature, stored as Base64.

Table 17 Exact per-recipient envelope size. Stored column is $4\lceil n/3 \rceil$ per field, as persisted in Base64.

Field	Binary (B)	Stored (B)	Share of envelope
ML-KEM-768 ciphertext	1088	1452	24.2%
Wrapped file key (32 + 16)	48	64	1.1%
AES-GCM nonce	12	16	0.3%
HKDF salt	32	44	0.7%
ML-DSA-65 signature	3309	4412	**73.7%**
Total per recipient	**4489**	**5988**	100%

The file ciphertext itself carries only the AES-GCM overhead: a 12-byte nonce and a 16-byte tag, 28 bytes regardless of file size, which is negligible for any real document.

Two things follow from the envelope figure. Each extra recipient costs a flat 5,988 bytes *whatever the file size*, because the ciphertext is stored once, so ten recipients cost about 58 KiB of envelopes rather than ten copies of the document. And that flat cost is 73.7% ML-DSA-65 signature, the same asymmetry the timings showed.

Table 18 Envelope overhead as a proportion of stored data, for ten recipients.

File size	Ciphertext	10 envelopes	Overhead
10 KiB	10 KiB	58.5 KiB	585%
1 MiB	1 MiB	58.5 KiB	5.7%
10 MiB	10 MiB	58.5 KiB	0.6%
100 MiB	100 MiB	58.5 KiB	0.06%

The overhead is therefore tiny for large documents and severe for small ones. A data room full of small documents shared widely pays a storage penalty that has nothing to do with the size of its corpus and everything to do with its sharing graph: $O(\text{files} \times \text{recipients})$, at 5,988 bytes per pair.

6.6 Selecting the Protection Profile

The post-quantum suite was not adopted by assumption. Two alternatives were implemented against the same provider interface, envelope structure, and file cipher, so the only thing that varied was the key-establishment and signature machinery: a classical suite pairing X25519 with Ed25519, and a hybrid suite combining both families so that confidentiality and authenticity would survive the compromise of either.

That comparison necessarily ran against the tree at the commit just before the alternatives were removed, using an earlier harness that drove envelope creation at concurrency four. Its absolute timings are therefore *not* comparable with Table 14, which is single-threaded and measured on the current tree; the ratios between the three columns are what the comparison establishes, since all three were measured together under identical conditions. The envelope sizes are exact in both cases.

Table 19 The delivered suite against the two alternatives. Envelope size is exact; latency is the median of three runs at concurrency four for a 1,000-recipient batch on the earlier harness; key generation is the mean of 50 rounds for each suite's full keypair set.

	Classical	Post-quantum	Hybrid
Envelope, binary (B)	188	4,489	4,585
Envelope, stored (B)	256	5,988	6,120
1,000 envelopes (ms)	2,570.14	10,503.89	12,559.28
Throughput (env/s)	389.1	95.2	79.6
Keypair set (ms)	1.138	2.361	3.499

The classical suite wins on both cost axes, at 4.1× faster and 23.9× smaller per recipient, and is still unusable here: its security rests on the discrete-logarithm assumption Shor's algorithm is expected to break [1], and these documents have to outlive that event. It was ruled out on security grounds before cost was considered.

That left post-quantum against hybrid, where the measured differences are small: 2.1% more stored bytes and 19.6% more envelope-creation time for hybrid. On the numbers alone hybrid is affordable, and it is strictly the safer construction, the one Signal, Apple, and the TLS ecosystem have converged on (Section 2.3).

So the decision was not made on the measurements, which favour hybrid at a price the platform could have paid. It was made on the complexity hybrid forced on everything around the cryptography: two key hierarchies, two signature paths, a dual-signature policy mode, and every combination of those running through the envelope parser, policy evaluator, database constraints, and test suite. Post-quantum-only met the security requirement while collapsing that matrix to a single path. This is an engineering trade for a prototype, not a cryptographic finding, and what it gave up is concrete: no hedge if ML-KEM-768 or ML-DSA-65 is broken later, where hybrid would have kept one for 2% more storage.

6.7 Results and Discussion

Four findings are worth stating on their own.

The post-quantum cost is a signature cost. Both the time and the space penalty land almost entirely on ML-DSA-65: 13.2 ms of the 38 ms envelope, and 73.7% of its 5,988 stored bytes. ML-KEM-768, the part usually expected to be expensive, costs 0.68 ms and 1,452 stored bytes. Any future effort to reduce the overhead should target the signature scheme or the number of signatures, not key encapsulation.

The overhead is regressive in file size. Both the latency floor (about 12 ms per decryption) and the storage overhead (5,988 bytes per file–recipient pair) are fixed costs. They are invisible on a 100 MiB document and dominant on a 10 KiB one. A deployment holding many small, widely shared files is the case where this design is most expensive, and that is worth knowing before choosing it.

Encrypt-once-share-many is what makes it affordable at all. Because the ciphertext is independent of its readers (Section 4.2.1), adding a recipient costs 5,988 bytes and 38 ms rather than a second copy of the document. Without that property, a 100 MiB file shared with 100 people would cost 10 GiB instead of 100 MiB plus 585 KiB.

The security evidence is real but narrow. 2,166 automated tests pass, and the sixteen attack classes of Table 11 are refused. This shows the behaviour built matches the specification built. It is not evidence of cryptographic soundness, and one designed defence is not merely untested but absent.

6.8 Threats to Validity

Security properties are argued, not proven. There was no formal verification, independent audit, or cryptanalysis. The evidence is a passing regression suite covering the attacks the authors thought of, which is a weaker claim than resistance to attacks in general, and a suite written by the same people as the code shares its blind spots.

The audit and key-authentication guarantees are partial. The audit chain is keyless and its events are unsigned, so it detects tampering by a party that cannot recompute it but not by the backend that stores it, and audit events report `UNAVAILABLE` rather than `VALID`. Key authentication rests on first-use fingerprint pinning, which does not cover a substitution made at first contact. Neither guarantee should be read as more than partly realized; Section 7.2 analyses exactly what an adversary gains.

The measurements are single-host and single-threaded. All figures come from one consumer laptop running a pure-JavaScript cryptographic library. Medians are reported without confidence intervals, and the absolute throughputs are not typical of server hardware or of native implementations. The suite comparison additionally ran on a reconstructed tree with a different harness, so its absolute timings are not comparable with the rest of this section.

The end-to-end system was never benchmarked. Upload and download latency including network transfer, object storage, database writes, and audit persistence were never measured, so nothing here is claimed about how the complete system feels to a user. All measurements used small or synthetic datasets, leaving behaviour under concurrent multi-user load, large corpora, or realistic membership distributions unknown.

A version-skew warning is unresolved. The backend test run emits a warning that native liboqs 0.16.0 is paired with `liboqs-python` 0.14.1. The suite passes, but a mismatched cryptographic binding is not a configuration a security evaluation should wave through.

7 Security Analysis and Limitations

This section examines how the delivered system stands up against the adversaries of Section 3.2, and then treats in detail the two places where the delivered system falls short of the design. The second half matters more than the first: a report that only argues its strengths is not an analysis.

7.1 Analysis by Adversary

A1, compromised backend or storage provider. This is the adversary the architecture is built around, and against document *confidentiality* the answer is strong. The backend holds ciphertexts and envelopes; opening either requires an ML-KEM-768 private key that exists only inside a client's crypto process. Modification is detected rather than prevented: an altered ciphertext fails its GCM tag, and an altered envelope fails both key derivation and its ML-DSA-65 signature. The backend can deny service and can read metadata, both of which are out of scope by Section 3.3. Where this adversary is *not* adequately answered is the audit record, for the reasons set out in Section 7.2, and public-key distribution, for the reasons in Section 7.3.

A2, malicious administrator. An administrator can add themselves to any room and grant themselves any role, because those are legitimate administrative powers. What they cannot do is read a file, because reading requires an envelope wrapped to their key, and only a party who already holds the file key can create one. The administrator can compel a legitimate holder to share, which is a governance problem rather than a cryptographic one, but the act of sharing is itself an audited event. For a quorum-sealed file even total administrative authority is insufficient: no envelope anywhere contains the whole key.

A3, unauthorized room member. Need-to-know sharing reduces this adversary to what they have been given. The server filters on envelope possession before disclosing identifiers, filenames, counts, sizes, owners, or timestamps, so a member cannot enumerate what exists in a room they belong to, and guessing a file identifier meets the same check. This is a genuine strengthening over the original design, where room membership implied listing.

A4, compromised user account. The blast radius is the set of files actually wrapped to that user, not every file in every room they belong to. Quorum-sealed files survive the compromise entirely unless k custodians are compromised together. The limitation is that the compromise is permanent for already-issued envelopes: because recipient KEM keys are static, an attacker who obtains a private key can open every envelope ever wrapped to it, including those recorded before the compromise.

A5, network attacker. Transport security is TLS, and the envelope bindings of Section 4.2.1 defeat replay at the application layer: an envelope captured from one room and presented in another fails, and one presented for a different recipient fails. The negative tests in Table 11 exercise both.

A6, future quantum adversary. An adversary who records ciphertexts and envelopes today obtains AES-256-GCM ciphertexts, whose keys are wrapped under ML-KEM-768. Grover's algorithm [2] reduces the effective strength of AES-256 to roughly 128 bits, which remains far out of reach, and Shor's algorithm [1] does not apply to lattice problems. The residual exposure is not quantum but structural: it is the single-family risk of Section 7.5.

7.2 The Audit Gap

Section 4.6 noted that the audit chain meets SR8 only in part. This section is precise about why, because the gap is wider than the word "truncation" suggests.

What the chain proves as built. Each event stores a SHA-384 hash over its canonical content and the hash of its predecessor. Verification recomputes every hash and checks every link. This detects modification of an event's content, deletion or reordering of an interior row, a duplicated event identifier or sequence number, and an event moved between rooms; all four are covered by the tests in Table 11.

What the chain does not prove. The hash is *keyless*. Nothing in it depends on a secret, so any party who can write rows can recompute a consistent chain. Against the adversary the rest of this system is built around, adversary **A1**, which holds every row, this means the chain constrains nothing: the backend can truncate the log, or discard it entirely and fabricate a different internally-consistent history, and verification will report `VALID`. Truncation is usually named as the missing property because chaining plainly cannot detect a removed suffix, but a keyless chain held by the adversary does not detect wholesale rewriting either.

What the chain *does* defend against is a strictly weaker attacker: one who can alter stored rows but cannot recompute the chain. That covers accidental corruption, a partial-access attacker

such as one with row-level write access through an injection flaw, tampering with a backup or an exported snapshot, and a storage layer that silently drops or reorders records. These are real threats and the mechanism genuinely addresses them; they are simply not the threat the report's headline claim was about.

What would close it. Two mechanisms, neither of them exotic. First, signing each event with a key the backend does not hold would mean the backend can no longer fabricate events, only withhold them. Second, signed head checkpoints cached by clients outside the backend's reach would mean it can no longer roll back or truncate below the highest checkpoint any client has seen; a disagreement would then surface as a verification failure across the membership. That is the same construction that makes certificate issuance auditable [33], and its limits are the known ones: an event hidden from every client from the moment of creation cannot be missed by a client that never saw a later checkpoint. Both are treated as future work (Section 8.1) rather than as part of the delivered system.

What is deployed that helps. Within a single paginated read the client verifies cursor continuity across pages, so a chain that changes shape while it is being read is caught. There is no persistence of the highest observed head across sessions, so this does not extend to rollback detection.

Consequence for the report's claims. The honest statement of the delivered property is: *the audit chain is tamper-evident against any party that cannot recompute it, and is not tamper-evident against the backend that stores it.* Objective O6 asked for a record verifiable without trusting its keeper; the prototype does not meet it. Table 8 records SR8 as partial for this reason, and Section 8.1 places the two mechanisms first among the remaining work.

7.3 The Key-Substitution Gap

Section 4.4.2 describes the two layers the system has: self-signed key bundles and client-side fingerprint pinning. Together they give first-use trust, and no more.

A client that has already pinned a peer's fingerprint detects any later substitution, because the fingerprint changes and the change raises a warning the user must accept. A client meeting a peer for the first time has nothing to compare against but an out-of-band check by the user, and a self-signature proves only that whoever generated the key also signed the bundle, which a hostile backend minting a fresh identity satisfies trivially.

The exposure is therefore specific: a backend that controls key distribution from a user's very first interaction with a given peer, and whose substitution is never checked against an out-of-band fingerprint, can cause a sender to wrap an envelope to a key the backend controls. The file's confidentiality is then lost for that share, without any ciphertext ever being broken. An append-only key directory would not make this impossible either; it would make it detectable, because the backend would have to either publish the swapped key where its real owner can see it or present different views to different clients, and the comparison surfaces the inconsistency [34]. That is the route to closing this gap, and Section 8.1 treats it as such.

7.4 No Forward Secrecy

Recipient KEM keys are long-lived by design, because an object encrypted once and stored for years cannot renegotiate a key with a reader who arrives later. The consequence is that a recipient private key recovered at any future point opens every envelope ever wrapped to it, including ciphertexts recorded years earlier. Session protocols avoid this by rekeying; a document store cannot, short of periodic re-wrapping of every envelope under fresh keys, which is mechanically straightforward, since it touches envelopes and not ciphertexts, but is not implemented.

This is the one place where the harvest-now, decrypt-later argument cuts against the system rather than for it: post-quantum wrapping defeats an adversary who records ciphertext and waits for a quantum computer, but not one who records ciphertext and waits for a key compromise.

7.5 Single-Family Cryptographic Risk

The shipped suite is post-quantum only. If ML-KEM-768 is broken, every wrapped file key is exposed; if ML-DSA-65 is broken, envelope and approval signatures can be forged. There is no second mechanism underneath either.

Section 6.6 measured what avoiding this would have cost: 2.1% more stored bytes and 19.6% more envelope-creation time for the hybrid construction, which every comparable deployed system has chosen (Section 2.3). The decision to ship post-quantum-only was made on implementation complexity rather than on those numbers, and it is the most consequential known weakness of the delivered system. It is recorded here rather than defended.

7.6 Residual Risk Summary

Table 20 Residual risks, ordered by the authors' assessment of severity.

Risk	Status	Mitigation or route to closure
Endpoint compromise while unlocked	Out of scope by design	Process isolation and single-use keys shorten the window; nothing eliminates it
Audit record not tamper-evident against the backend	Open	Sign events off-backend; cache signed head checkpoints client-side (Section 7.2)
Key substitution on first contact	Partly mitigated	Append-only key directory; out-of-band fingerprint comparison (Section 7.3)
Break of ML-KEM-768 or ML-DSA-65	Accepted	Restore the hybrid suite at a measured 2.1% storage cost (Section 7.5)
Recipient key compromise opens all past envelopes	Accepted	Periodic key rotation with scheduled re-wrap (Section 7.4)
Metadata visible to the backend	Out of scope	Would require padding, batching, or oblivious access
Copies already downloaded	Out of scope	No cryptographic remedy exists
No formal verification or external audit	Accepted	Independent review; machine-checked proofs of the envelope protocol
liboqs version skew in the backend test environment	Open	Pin native and binding versions together

8 Conclusion and Future Work

Quarom set out to answer one narrow question, stated in Section 1.2: can a data room give an organization the sharing and governance it needs, with roles, sensitivity policies, revocation, and an auditable history, while keeping the hosting infrastructure outside the trust boundary and

protecting documents against someone who records ciphertext today and waits for a quantum computer?

For six of the seven objectives the answer is yes, and the evidence is in Section 6. The provider is outside the trust boundary (O1), key management is post-quantum throughout (O2), a document is encrypted and stored once however many people read it (O3), access follows envelope possession rather than membership (O4), and a classification requiring a two-person rule is enforced cryptographically rather than by policy (O5). The prototype runs, and 2,166 automated tests pass against it (O7).

Objective O6 is not met. The audit record is hash-chained, canonically serialized, exportable, and verifiable offline, and it detects tampering by any party that cannot recompute the chain. It does not constrain the backend that stores it, because the chain is keyless and no client anchors its head outside the backend's reach (Section 7.2). The report states this rather than presenting the chain as more than it is.

The measurements also produced a result worth carrying beyond this project. The cost of going post-quantum here is almost entirely a *signature* cost, not a key-establishment cost: ML-KEM-768 is sub-millisecond and contributes a quarter of the envelope, while ML-DSA-65 signing is 13.2 ms and contributes 73.7%. Work aimed at reducing post-quantum overhead in systems of this shape should target signature schemes, or the number of signatures, and not the KEM.

8.1 Future Work

The remaining work is ordered by how much of a stated claim it restores.

Close the audit gap. Signing events with a key the backend does not hold, and caching signed head checkpoints on clients, would together turn the audit chain from internally consistent into externally verifiable, and would let objective O6 be claimed without qualification. Signing needs a durable service-key lifecycle, which is the substantial part of the work. This is the most valuable item on the list, because it is the only one that closes an objective the project set for itself and did not meet.

Add an append-only key directory. This would close the key-substitution gap of Section 7.3, on which every envelope's guarantee ultimately rests, and the construction is well understood from Certificate Transparency and CONIKS [33, 34].

Restore a hybrid suite. Hybrid cost a measured 2.1% more storage and 19.6% more envelope-creation time than the suite that shipped (Section 6.6), in exchange for surviving the compromise of either algorithm family. Its absence is the most consequential known weakness of the delivered cryptography. Adding it means restoring a second key hierarchy and re-wrapping existing envelopes, which touches key management but not the stored ciphertexts.

Rotate recipient keys. Static recipient KEM keys are the related gap (Section 7.4). Periodic rotation with scheduled re-wrapping would bound the damage of a future key compromise, and the machinery to flag stale envelopes already exists.

Measure the complete system. Nothing here was measured end-to-end: network transfer, object storage, database writes, and audit persistence are all excluded, as is behaviour under concurrent multi-user load. The client-side figures are the floor, not the experience.

Seek external assurance. No formal verification, independent audit, or cryptanalysis was performed, and a test suite written by the authors of the code shares its blind spots. Machine-checked

analysis of the envelope protocol would be the highest-value form of that assurance, since it is the component every other guarantee depends on.

Complete the platform. macOS and Linux packaging remain unbuilt, and the liboqs version skew noted in Section 6.8 should be resolved rather than tolerated.

8.2 Closing Remarks

Harvest-now, decrypt-later is an unusual threat, because the defence has to be deployed long before the attack becomes possible. That makes it both easy to justify and easy to over-claim. Quarom is best read as a demonstration that the defence is affordable where it is usually assumed not to be, and that the same separation which makes it affordable, encrypting the file once and sharing the key many times, is what lets access be narrowed and a file key be split without ever touching a stored ciphertext. Its limits are stated in the body rather than a footnote, because a system whose purpose is long-term trust is poorly served by an evaluation that overstates itself.

References

[1] Peter W. Shor. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”. In: *SIAM Journal on Computing* 26.5 (1997), pp. 1484–1509. DOI: 10.1137/S0097539795293172.

[2] Lov K. Grover. “A Fast Quantum Mechanical Algorithm for Database Search”. In: *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC)*. 1996, pp. 212–219. DOI: 10.1145/237814.237866.

[3] Lily Chen et al. *Report on Post-Quantum Cryptography*. Tech. rep. NISTIR 8105. National Institute of Standards and Technology, 2016. DOI: 10.6028/NIST.IR.8105.

[4] Gorjan Alagic et al. *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*. Tech. rep. NISTIR 8413. National Institute of Standards and Technology, 2022. DOI: 10.6028/NIST.IR.8413-upd1.

[5] National Institute of Standards and Technology. *FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard*. Federal Information Processing Standards Publication 203. U.S. Department of Commerce, Aug. 2024. DOI: 10.6028/NIST.FIPS.203. URL: https://doi.org/10.6028/NIST.FIPS.203.

[6] National Institute of Standards and Technology. *FIPS 204: Module-Lattice-Based Digital Signature Standard*. Federal Information Processing Standards Publication 204. U.S. Department of Commerce, Aug. 2024. DOI: 10.6028/NIST.FIPS.204. URL: https://doi.org/10.6028/NIST.FIPS.204.

[7] National Institute of Standards and Technology. *FIPS 205: Stateless Hash-Based Digital Signature Standard*. Federal Information Processing Standards Publication 205. U.S. Department of Commerce, Aug. 2024. DOI: 10.6028/NIST.FIPS.205. URL: https://doi.org/10.6028/NIST.FIPS.205.

[8] Michele Mosca. “Cybersecurity in an Era with Quantum Computers: Will We Be Ready?” In: *IEEE Security & Privacy* 16.5 (2018), pp. 38–41. DOI: 10.1109/MSP.2018.3761723.

[9] Adi Shamir. “How to Share a Secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613. DOI: 10.1145/359168.359176.

[10] Mahesh Kallahalla et al. “Plutus: Scalable Secure File Sharing on Untrusted Storage”. In: *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*. 2003, pp. 29–42. URL: https://www.usenix.org/legacy/events/fast03/tech/kallahalla.html.

[11] Eu-Jin Goh et al. “SiRiUS: Securing Remote Untrusted Storage”. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2003. URL: https://www.ndss-symposium.org/ndss2003/sirius-securing-remote-untrusted-storage/.

[12] Seny Kamara and Kristin Lauter. “Cryptographic Cloud Storage”. In: *Financial Cryptography and Data Security, FC 2010 Workshops*. Vol. 6054. Lecture Notes in Computer Science. 2010, pp. 136–149. DOI: 10.1007/978-3-642-14992-4_13.

[13] Tresorit. *Tresorit Security*. 2026. URL: https://tresorit.com/security.

[14] Proton AG. *Proton Drive Security Model*. 2025. URL: https://proton.me/drive/security (visited on 08/20/2026).

[15] Ehren Kret and Rolfe Schmidt. *The PQXDH Key Agreement Protocol*. 2023. URL: https://signal.org/docs/specifications/pqxdh/.

[16] Karthikeyan Bhargavan et al. “Formal verification of the PQXDH post-quantum key agreement protocol for end-to-end secure messaging”. In: *Proceedings of the 33rd USENIX Security Symposium*. USENIX Association, 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/bhargavan.

[17] Apple Security Engineering and Architecture. *iMessage with PQ3: The new state of the art in quantum-secure messaging at scale*. 2024. URL: https://security.apple.com/blog/imessage-pq3/.

[18] Douglas Stebila. *Security analysis of the iMessage PQ3 protocol*. Tech. rep. Apple Inc., 2024. URL: https://security.apple.com/assets/files/Security_analysis_of_the_iMessage_PQ3_protocol_Stebila.pdf.

[19] Tuta. *TutaCrypt: Post-quantum cryptography for email*. 2024. URL: https://tuta.com/blog/post-quantum-cryptography.

[20] Bas Westerbaan. *State of the post-quantum Internet in 2025*. 2025. URL: https://blog.cloudflare.com/pq-2025/.

[21] Cloudflare. *Automatically secure: how we upgraded 6,000,000 domains by default to get ready for the quantum future*. 2025. URL: https://blog.cloudflare.com/automatically-secure/.

[22] Joppe Bos et al. “CRYSTALS-Kyber: A CCA-Secure Module-Lattice-Based KEM”. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018, pp. 353–367. DOI: 10.1109/EuroSP.2018.00032.

[23] Leo Ducas et al. “CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.1 (2018), pp. 238–268. DOI: 10.13154/tches.v2018.i1.238-268.

[24] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. Tech. rep. NIST SP 800-38D. National Institute of Standards and Technology, 2007. DOI: 10.6028/NIST.SP.800-38D.

[25] Phillip Rogaway. “Authenticated-Encryption with Associated-Data”. In: *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*. 2002, pp. 98–107. DOI: 10.1145/586110.586125.

[26] Hugo Krawczyk. “Cryptographic Extraction and Key Derivation: The HKDF Scheme”. In: *Advances in Cryptology – CRYPTO 2010*. Vol. 6223. Lecture Notes in Computer Science. 2010, pp. 631–648. DOI: 10.1007/978-3-642-14623-7_34.

[27] Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869, Internet Engineering Task Force. 2010. DOI: 10.17487/RFC5869.

[28] Alex Biryukov et al. *Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications*. RFC 9106, Internet Engineering Task Force. 2021. DOI: 10.17487/RFC9106.

[29] George Robert Blakley. “Safeguarding Cryptographic Keys”. In: *Proceedings of the 1979 AFIPS National Computer Conference*. 1979, pp. 313–317. DOI: 10.1109/AFIPS.1979.98.
[30] Elaine Barker. *Recommendation for Key Management: Part 1 – General*. Tech. rep. NIST SP 800-57 Part 1 Rev. 5. National Institute of Standards and Technology, 2020. DOI: 10.6028/NIST.SP.800-57pt1r5.
[31] Stuart Haber and W. Scott Stornetta. “How to Time-Stamp a Digital Document”. In: *Journal of Cryptology* 3.2 (1991), pp. 99–111. DOI: 10.1007/BF00196791.
[32] Scott A. Crosby and Dan S. Wallach. “Efficient Data Structures for Tamper-Evident Logging”. In: *Proceedings of the 18th USENIX Security Symposium*. 2009, pp. 317–334. URL: https://www.usenix.org/legacy/events/sec09/tech/full_papers/crosby.pdf.
[33] Ben Laurie, Adam Langley, and Emilia Kasper. *Certificate Transparency*. RFC 6962, Internet Engineering Task Force. 2013. DOI: 10.17487/RFC6962.
[34] Marcela S. Melara et al. “CONIKS: Bringing Key Transparency to End Users”. In: *Proceedings of the 24th USENIX Security Symposium*. 2015, pp. 383–398. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara.
[35] Adam Shostack. *Threat Modeling: Designing for Security*. Wiley, 2014. ISBN: 9781118809990.
[36] Jerome H. Saltzer and Michael D. Schroeder. “The Protection of Information in Computer Systems”. In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308. DOI: 10.1109/PROC.1975.9939.

A Appendix

This appendix holds the material a reader would need to check the report rather than take it on trust: how to repeat every measurement (A.1), the security test cases behind Table 11 (A.2), the exact cryptographic parameters in use (A.3), the authenticated context formats the protocol depends on (A.4), and the unedited output of the measurement runs (A.5).

A.1 Reproducing the Measurements

A.1.1 Harnesses

Table 21 Which harness produces which result.

Harness	Produces	Location
`benchmark.manual.ts`	Tables 12, 13, 14, 15	Supplied with this report in `benchmark/`
`benchmark-audit-chain.py`	Table 16	`scripts/` in the repository
`pytest`, `vitest`, `node:test`	Table 9; the evidence columns of Tables 10 and 11	Each package in the repository
`verify-audit-export.py`	Offline verification of an exported room history (FR8)	`scripts/` in the repository

A.1.2 Running the cryptographic harness

The harness imports the client's own modules from `src/lib/crypto/`, so it runs from inside the platform's `web` workspace where those modules and their dependencies live. Keeping it with the report rather than in the repository leaves the delivered tree unchanged, and the separate configuration means an ordinary `npm test` never picks it up: it takes minutes and reports numbers rather than asserting on them.

```
cp benchmark/benchmark.manual.ts        <platform>/web/src/lib/crypto/
cp benchmark/vitest.benchmark.config.ts <platform>/web/

cd <platform>/web
# writes benchmark-results.txt
npx vitest run -c vitest.benchmark.config.ts
```

A.1.3 Running the test suites

```
cd <platform>/pqc-service && pytest -q
cd <platform>/web         && npx vitest run
cd <platform>/desktop     && npm test

cd <platform>
python scripts/benchmark-audit-chain.py --events N
```

The envelope and ciphertext sizes in Section 6.5 need no harness at all: they follow arithmetically from the FIPS 203 and FIPS 204 parameter sets and the Base64 expansion $4\lceil n/3 \rceil$, and are therefore independent of any host.

A.2 Security Test Case Catalogue

Each case below asserts a *refusal*. Table 11 gives the test names; this table gives the case as it would be written for a test plan, with the requirement it defends and the observed outcome. All sixteen pass.

The attack this catalogue does *not* contain is truncation of the audit chain by the backend, for the reason given in Section 7.2: there is no test asserting it is refused, because it is not.

A.3 Cryptographic Parameters

Password-derived protection uses Argon2id with three iterations, 65,536 KiB of memory, parallelism four, and a 32-byte output [28]. Room binding is version 1 and the file crypto version is 2; both are carried in every object and in the authenticated contexts below.

A.4 Authenticated Context Formats

Table 6 describes the three authenticated contexts in prose. Each is a JSON array serialized to UTF-8, constructed in `web/src/lib/crypto/hpke.ts` exactly as follows. The distinction between the first and the second is the subject of Section 4.2.1.

1. File ciphertext: AES-256-GCM associated data.

```
["kaust-pqc-file", 2, <fileId>, <ownerId>, <filename>, <size>,
 "AES-256-GCM"]
```

Recipient, room and profile are deliberately absent: one ciphertext serves every recipient, so it cannot commit to any of them.

Table 22 Security test cases. Result column: ✓ = refused as expected.

ID	Req.	Condition and expected outcome	Result
TC-01	SR6	A stored ciphertext byte is flipped; decryption must fail on the GCM tag and return no plaintext	✓
TC-02	SR4	An envelope is opened with a different recipient's private key; key derivation and the tag must both fail	✓
TC-03	SR3	File metadata bound in the AAD is modified; decryption must fail	✓
TC-04	SR4	An envelope from room A is presented in room B; the wrap context and signature must not verify	✓
TC-05	SR4	An envelope is re-pointed at a different file; verification must fail	✓
TC-06	SR9	A binding-1 envelope fails; it must not be retried as binding 0	✓
TC-07	SR4	An envelope signature is stripped or altered; the envelope must be rejected	✓
TC-08	SR3	An encrypted filename is replayed against a different file id; the operation must fail closed	✓
TC-09	SR9	An unknown protection-profile identifier is supplied; the parser and the database constraint must each reject it	✓
TC-10	SR5	A room member holding no envelope lists files; the response must be empty and disclose no count	✓
TC-11	SR5	That member requests a file id they guessed correctly; the request must be refused	✓
TC-12	SR5, FR4	A revoked recipient requests a download; it must be refused, and identically to an outsider so the refusal leaks nothing	✓
TC-13	SR7	$k-1$ custodians attempt to open a sealed file; the attempt must be refused before interpolation rather than returning a wrong key	✓
TC-14	SR7	A seal is requested with a duplicate custodian, or a threshold below 2; both must be rejected	✓
TC-15	SR7	A share envelope is replayed into another room, or presented with the wrong sealer signature; both must be rejected	✓
TC-16	SR8	An audit event is modified, deleted, reordered, duplicated, or moved between rooms; verification must report the specific failure code	✓

Table 23 Registered parameters of the `POST_QUANTUM` profile, in bytes.

Primitive	Parameter	Size (B)
ML-KEM-768 (FIPS 203)	Public key	1184
	Secret key	2400
	Ciphertext	1088
	Shared secret	32
ML-DSA-65 (FIPS 204)	Public key	1952
	Secret key	4032
	Signature	3309
AES-256-GCM	Key	32
	Nonce	12
	Authentication tag	16
HKDF-SHA-384	Per-envelope salt	32
SHA-384	Digest	48
Shamir over GF(2^8)	Share of a 32-byte key	33

2. Room-bound key wrap: HKDF-SHA-384 `info`, and the GCM associated data of the wrapped file key.

```
["kaust-pqc-room-file-key-wrap", 1, <canonicalRoomId>,
 "file-crypto-version", 2, <fileId>, <ownerId>, <recipientId>,
 "ML-KEM-768", "HKDF-SHA-384", "AES-256-GCM"]
```

The same bytes serve as key-derivation context and as associated data, so a substituted recipient fails twice: once at derivation, once at the tag.

3. Room-bound envelope: the ML-DSA-65 signed message.

```
["kaust-pqc-room-file-envelope", 1, <canonicalRoomId>,
 "file-crypto-version", 2, <fileId>, <ownerId>, <filename>, <size>,
 b64(<aeadNonce>), b64(<ciphertextSha384>), <recipientId>,
 "ML-KEM-768", "ML-DSA-65", "HKDF-SHA-384", "AES-256-GCM",
 b64(<kemCiphertext>), b64(<wrappedFileKey>), b64(<keyNonce>),
 b64(<kdfSalt>)]
```

The digest of the file ciphertext is the link back down, so an envelope cannot be re-pointed at different bytes.

The personal, non-room variants replace the domain tags with `kaust-pqc-file-key-wrap` and `kaust-pqc-file-envelope` and omit the room-binding version and room identifier. A room-bound envelope is never retried against the personal context after a failure, so a downgrade from binding 1 to binding 0 is not reachable.

A.5 Measurement Output

The complete set of figures from the run reported in Section 6, gathered here so every number the report uses can be found in one place. Produced on the host of Section 6.1 under Node v24.18.0 on `win32/x64`; the harness writes them to `benchmark-results.txt`, which is supplied with this report.

Table 24 Post-quantum primitives in isolation, median of 200 rounds.

Scheme	Operation	Median (ms)
ML-KEM-768	Key generation	0.559
ML-KEM-768	Encapsulate	0.682
ML-KEM-768	Decapsulate	0.546
ML-DSA-65	Key generation	2.39
ML-DSA-65	Sign, 1 KiB message	13.23
ML-DSA-65	Verify, 1 KiB message	7.32

Table 25 AES-256-GCM file operations by size, median. Decryption includes opening the recipient's envelope, which is why the two smallest sizes cost almost the same.

	Encrypt		Decrypt	
Size	Median (ms)	MiB/s	Median (ms)	MiB/s
1 KiB	0.565	—	12.84	—
64 KiB	1.10	57	12.38	5
1 MiB	8.24	121	19.91	50
10 MiB	70.63	142	88.45	113
100 MiB	821.8	122	769.7	130

Table 26 Envelope creation by recipient count, measured after warm-up.

Recipients	Rounds	Median (ms)	Per envelope (ms)	Envelopes/s
1	25	28.19	28.19	35.5
10	25	364.3	36.43	27.4
100	5	3,944.6	39.45	25.4
1000	3	38,498.7	38.50	26.0

Table 27 Quorum ceremonies, median of 20 rounds, and the Shamir arithmetic inside them, median of 500 rounds on a 32-byte secret.

Configuration	Seal (ms)	Unseal (ms)	Split (ms)	Combine (ms)
2-of-3	28.57	6.81	0.039	0.003
3-of-5	61.39	10.80	0.051	0.005
5-of-9	91.34	16.71	0.048	0.005

Table 28 Audit chain verification, from `benchmark-audit-chain.py`. Every run reported `valid=True`.

Events	Seconds	Events/s
1,000	0.064088	15,603
2,000	0.138266	14,465
5,000	0.383672	13,032
10,000	0.749045	13,350