Aionda Mail Security Whitepaper

Zero-Knowledge · Post-Quantum · Made in Germany

Version 1.8 — 2026-10-13

Aionda GmbH
Stuttgart, Germany

contact-46epp9ba@contact.aionda.com
https://mail.aionda.com

PUBLIC DOCUMENT

Contents

1. Executive Summary . . . 3
2. Threat Model . . . 3
3. Architecture Overview . . . 5
4. Zero-Knowledge Authentication (OPAQUE) . . . 6
5. Vault Master Key & Shamir Secret Sharing . . . 8
6. Post-Quantum Hybrid KEM . . . 9
7. Email Encryption Pipeline . . . 10
8. Encrypted API Transport Layer . . . 12
9. Folder Sharing Cryptography . . . 14
10. Vault Drive — Encrypted Document Storage . . . 15
11. Aionda Chat — Post-Quantum E2EE Messaging . . . 21
12. Aionda Video Calls — Post-Quantum E2EE Conferencing . . . 26
13. Aionda Tasks - End-to-End Encrypted Task Management . . . 30
14. Side-Channel Protections . . . 33
15. Key Management & Lifecycle . . . 34
16. Recovery Mechanism . . . 35
17. Passwordless Authentication (Passkeys) . . . 36
18. Guardian: MITM Protection & Response Signing . . . 37
19. Enterprise Email Archive (Blockchain) . . . 40
20. What the Server Sees — and What It Does Not . . . 44
21. Algorithm Reference . . . 45
22. Comparison with Other Providers . . . 46
23. Limitations & Honest Boundaries . . . 47
24. Roadmap . . . 49
25. QR-code device sign-in . . . 49
Document History . . . 50
Contact . . . 53

1. Executive Summary

Aionda Mail is a zero-knowledge, post-quantum encrypted email service operated by Aionda GmbH in Stuttgart, Germany. The service combines disposable email addresses (DEAs) with a fully encrypted mailbox — a combination not offered by any other provider.

Core security properties:

- **Zero-Knowledge Architecture:** Client-side decryption of stored mailbox content takes place on the user's device, in the browser or the native app.
- **Post-Quantum Security:** Hybrid Key Encapsulation Mechanism (X25519 + ML-KEM-1024) protects all data against both classical and quantum computer attacks.
- **Zero-Knowledge Authentication:** OPAQUE protocol (RFC 9807) ensures passwords are never transmitted to or stored on the server — not even as hashes.
- **Shamir Secret Sharing (2-of-3):** The vault master key is split into three shares protected by password, passkey, and recovery key. Any two shares reconstruct the master key.
- **Perfect Forward Secrecy:** Every API request uses a unique, one-time cryptographic keypair. Compromising one request does not affect any other.
- **MITM Protection (Guardian):** The browser extension independently verifies all server responses via Ed25519 signatures and detects man-in-the-middle attacks through TLS certificate verification — even against corporate proxies and compromised CDNs.
- **GoBD-Compliant Email Archive:** Enterprise accounts benefit from a tamper-evident hash chain (SHA3-256 blockchain) with end-to-end encrypted content (Hybrid KEM), complete audit trail, legal hold, and configurable retention — compliant with German GoBD regulations.
- **No Password Recovery:** If the password and all recovery methods are lost, the data is irrecoverable. This is by design — it proves the server cannot access user data.

Jurisdiction: German law (DSGVO/GDPR), no data sharing with foreign intelligence agencies.

---

2. Threat Model

2.1 What Aionda Mail Protects Against

Threat	Protection
Server compromise (database leak, insider access)	All email content encrypted with keys the server never possesses
Network eavesdropping (ISP, Wi-Fi, CDN)	End-to-end encrypted API transport via Hybrid KEM
CloudFlare inspection	API requests are encrypted before leaving the browser; CloudFlare sees only ciphertext. Guardian extension detects response tampering via Ed25519 signatures
Corporate MITM proxies (ZScaler, Fortinet, etc.)	Guardian extension detects proxy certificates via issuer blocklist (Firefox)
Quantum computer attacks ("harvest now, decrypt later")	ML-KEM-1024 (NIST FIPS 203) provides post-quantum resistance
Password database theft	OPAQUE stores only cryptographic records, not password hashes

Threat	Protection
Offline brute-force attacks on passwords	OPAQUE prevents offline attacks; server-side rate limiting prevents online attacks
Email size analysis	Bucket padding obscures actual email sizes
Compression side-channels (CRIME/BREACH)	Bucket padding applied after compression
User enumeration	Deterministic fake responses for non-existent accounts

2.2 What Aionda Mail Does NOT Protect Against

Limitation	Explanation
Compromised device	If malware controls the browser, it can read decrypted content
Metadata to external recipients	Emails to Gmail/Outlook travel unencrypted after leaving our servers (unless PGP is used)
Email metadata on our servers	Timestamps, IP addresses, and encrypted email sizes are visible to the server
Web delivery trust	The browser downloads JavaScript from our servers on each visit (see Section 20 for mitigations)
Rubber-hose cryptanalysis	No cryptographic system protects against physical coercion
Key substitution by the server	The server distributes public keys. A malicious server can hand a sender a key of its own for a contact the sender has never shared with before (trust on first use). For later changes, the design in section 15.3 (signed key transitions) lets a pinning client distinguish a legitimate change from a substituted one — once the server part is deployed and the clients verify chains. Until then the Apple client stops at every change (since 4 October 2026), and the web client encapsulates to whatever key the server names. A server can also steer a client to an older, legitimately signed intermediate key set; this only matters if that intermediate key is compromised. When the signing-key-aware fingerprint is rolled out, pinning clients will stop once for every contact that has a signing key; a substituted key is visible in that comparison only if the user actually compares.

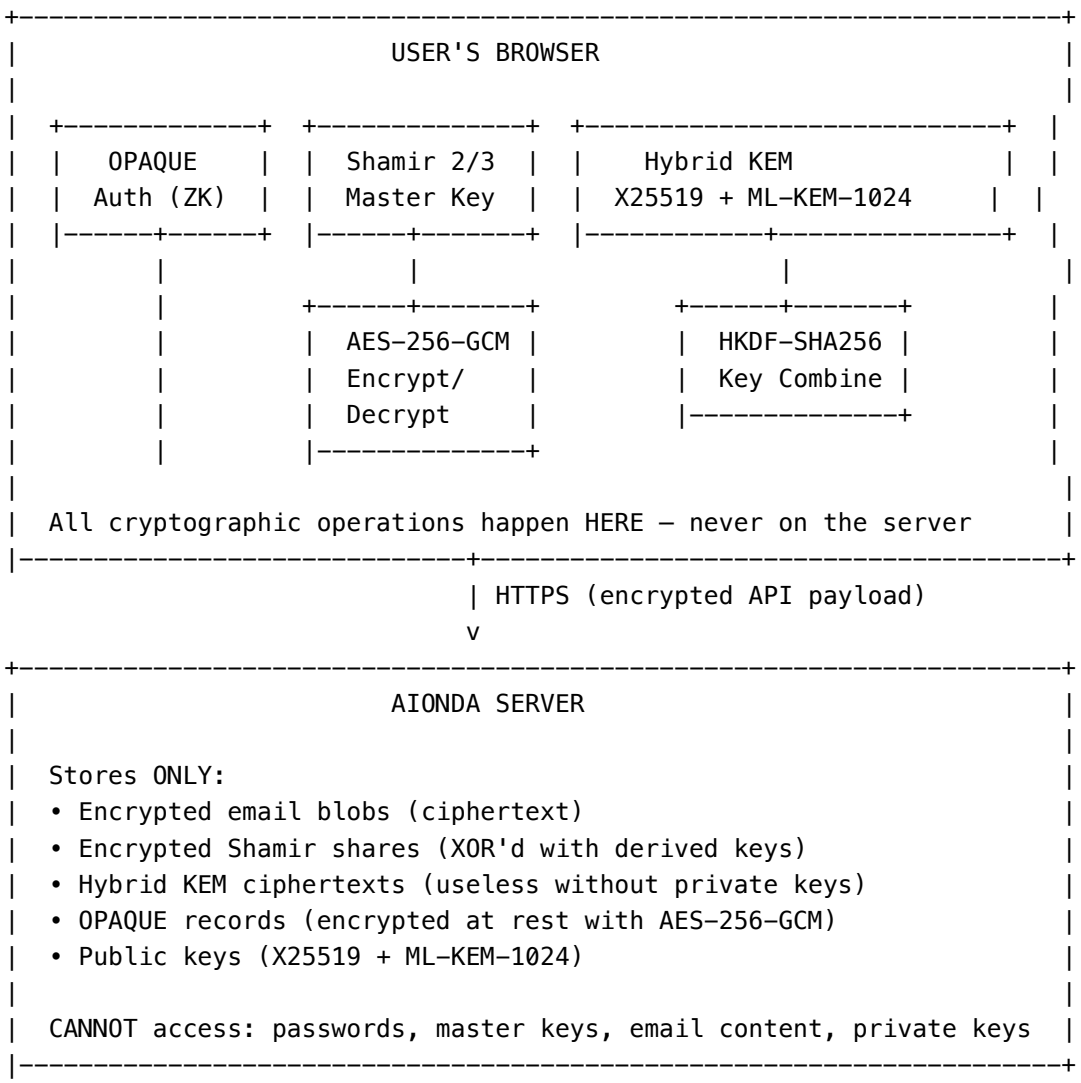
2.3 Design Principle

Aionda Mail follows the **“honest server” model**: the system is designed so that even a fully compromised server — or a malicious operator — cannot decrypt user data. Security does

not depend on trusting us. It depends on mathematics.

---

3. Architecture Overview



3.1 Component Overview

Component	Purpose	Location
OPAQUE (RFC 9807)	Zero-knowledge password authentication	Client + Server
Shamir Secret Sharing	Vault master key protection (2-of-3 threshold)	Client only
Hybrid KEM (X25519 + ML-KEM-1024)	Post-quantum key encapsulation for emails	Client + Server
AES-256-GCM	Authenticated symmetric encryption	Client + Server

Component	Purpose	Location
HKDF-SHA256	Key derivation from hybrid shared secrets	Client + Server
BIP39	Recovery key encoding (24-word mnemonic)	Client only
WebAuthn PRF	Passkey-based vault unlock	Client only
Bucket Padding	Side-channel protection	Client + Server

3.2 Native clients for iOS and macOS

Aionda Mail provides native applications for iOS (iPhone and iPad) and macOS. Both use a shared Swift/SwiftUI codebase, with platform-specific application entry points and interface adaptations. The shared `AiondaMailCore` package implements core functionality, while `AiondaMailUI` supplies the SwiftUI interface.

The native application interface is delivered as part of the installed app. This differs from the browser client, which loads its application code through web delivery. References to browser APIs, JavaScript and sessionStorage in this document describe the web implementation; they do not specify the native clients' local storage or runtime.

Native does not mean that every component avoids web technology: embedded WebKit views render HTML email, support HTML composition and provide the video-call media interface. The HTML email reader disables content JavaScript and uses a non-persistent website data store. These reader settings must not be generalized to the editor or video-call views, which require JavaScript.

The native architecture alone is not a security guarantee. Trust in the installed app, its updates and the operating system remains necessary; a compromised endpoint can access decrypted content. Web delivery remains part of the trust boundary for embedded components that load web content.

---

4. Zero-Knowledge Authentication (OPAQUE)

4.1 Why Not Password Hashing?

Traditional services store password hashes (bcrypt, Argon2). While better than plaintext, this approach has fundamental weaknesses:

- The server sees the password during login (even if only briefly in RAM)
- Password hashes can be brute-forced offline if the database is stolen
- The server could be modified to log passwords

OPAQUE eliminates all three problems. The password never leaves the browser — not as plaintext, not as a hash, not in any form.

4.2 How OPAQUE Works

OPAQUE (RFC 9807) is an Asymmetric Password-Authenticated Key Exchange (aPAKE) protocol. It uses a cryptographic challenge-response mechanism where the server can verify the user knows the correct password without ever learning what that password is.

Registration (one-time):

```
Browser                                    Server
--------                                   ------
1. Generate OPAQUE registration     ->
   request from password

                                           2. Create OPAQUE registration
                                              response (blind evaluation)
                                    <-
3. Finalize registration            ->      4. Store encrypted OPAQUE
   (export_key available)                      record (AES-256-GCM at rest)
```

Login (every session):

```
Browser                                    Server
--------                                   ------
1. Generate login request (KE1)     ->      2. Look up encrypted OPAQUE record
   from password                              Decrypt with server key
                                              Generate response (KE2)
                                    <-
3. Verify server response
   Compute session key + KE3        ->      4. Verify KE3 (cryptographic proof)
   Password verified locally!                 If valid: session authenticated
                                              If invalid: reject (max 3 attempts)
```

Key properties:

- Password is verified **client-side** in step 3 — the server never sees it
- The server stores an **OPAQUE record**, which is not a password hash and cannot be brute-forced offline
- OPAQUE records are additionally **encrypted at rest** with AES-256-GCM using a server-side key
- **User enumeration protection**: Non-existent accounts receive deterministic fake responses with identical timing
- **Rate limiting**: Maximum 3 authentication attempts per session, 120-second session timeout

4.3 Implementation

- **Library:** `@serenity-kit/opaque` (WASM-based, production-grade)
- **Server component:** Dedicated microservice for OPAQUE cryptographic operations
- **Base64 format:** base64url (URL-safe, no padding) for protocol compatibility
- **Audit logging:** All authentication events logged with timestamps and IP addresses

4.4 SRP Migration

Legacy accounts using SRP-6a are automatically migrated to OPAQUE upon next login. After migration, the SRP verifier is permanently deleted. Migration is one-way — accounts cannot revert to SRP.

---

5. Vault Master Key & Shamir Secret Sharing

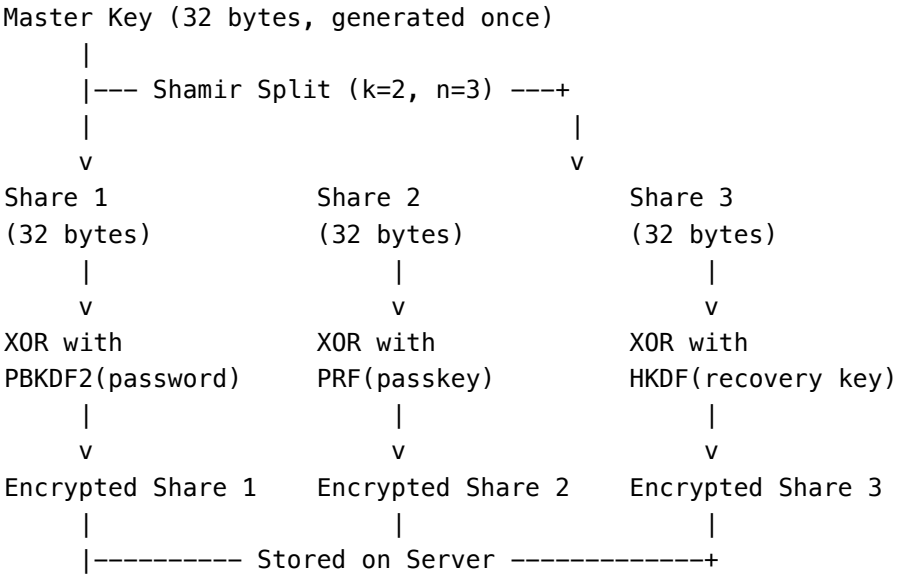
5.1 Master Key Generation

When a user activates the encrypted mailbox, a **256-bit (32-byte) master key** is generated using the browser's cryptographically secure random number generator (`crypto.getRandomValues`).

This master key is the root of all encryption. It never leaves the browser in plaintext. It is never stored anywhere — not in the browser, not on the server, not in any form.

5.2 Shamir Secret Sharing (2-of-3)

The master key is split into three shares using **Shamir's Secret Sharing** scheme over the Galois Field GF(2^8) with the AES irreducible polynomial (x^8 + x^4 + x^3 + x + 1).



Threshold property: Any 2 of the 3 shares are sufficient to reconstruct the master key via Lagrange interpolation. The server stores only the encrypted shares — and cannot decrypt any of them.

5.3 Share Protection

Each share is XOR'd with a key derived from a different authentication factor:

Share	Protected By	Key Derivation
Share 1	Password	PBKDF2-SHA256, 600,000 iterations, 32-byte random salt
Share 2	Passkey (FIDO2)	WebAuthn PRF extension, hardware-bound
Share 3	Recovery Key	HKDF-SHA3-256 with account-bound salt

Reconstruction scenarios:

- **Normal login:** Password (Share 1) + Passkey (Share 2) -> Master Key
- **Lost passkey:** Password (Share 1) + Recovery Key (Share 3) -> Master Key
- **Password change:** Passkey (Share 2) + Recovery Key (Share 3) -> Master Key

5.4 Session Storage Protection

Even within a browser session, the master key is never stored in plaintext:

1. An ephemeral AES-256 key is generated
2. The master key is encrypted with this ephemeral key
3. Only the encrypted blob is placed in `sessionStorage`
4. The ephemeral key exists only in JavaScript memory (garbage-collected on tab close)

---

6. Post-Quantum Hybrid KEM

6.1 Why Post-Quantum?

Quantum computers running Shor's algorithm could break classical public-key cryptography (RSA, ECDH, X25519) in polynomial time. While large-scale quantum computers do not yet exist, the threat of **"harvest now, decrypt later"** attacks is real: adversaries could store encrypted data today and decrypt it when quantum computers become available.

6.2 Hybrid Approach

Aionda Mail uses a **hybrid Key Encapsulation Mechanism** combining:

- **X25519** (Curve25519 ECDH) — proven classical security, 128-bit security level
- **ML-KEM-1024** (NIST FIPS 203, formerly Kyber-1024) — post-quantum security, NIST Security Level 5

The hybrid approach provides **defense-in-depth**: the combined key is secure as long as **at least one** of the two algorithms remains unbroken.

6.3 Encapsulation Process

```
Sender (encrypting an email):

1. Generate ephemeral X25519 keypair
2. X25519 key agreement with recipient's public key
   -> x25519SharedSecret (32 bytes)

3. ML-KEM-1024 encapsulation with recipient's public key
   -> mlKemSharedSecret (32 bytes) + mlKemCiphertext (1568 bytes)

4. Combine secrets:
   combinedSecret = x25519SharedSecret || mlKemSharedSecret  (64 bytes)

5. Derive final key:
   sharedSecret = HKDF-SHA256(
       ikm = combinedSecret,
       salt = nil,
       info = "trashmail-hybrid-kem-v1",
       length = 32
   )

6. Use sharedSecret to wrap the email's ephemeral AES-256 key
```

6.4 Decapsulation Process

```
Recipient (decrypting an email):

1. X25519 key agreement:
   x25519Shared = X25519(recipientPrivateKey, ephemeralPublicKey)

2. ML-KEM-1024 decapsulation:
   mlKemShared = ML-KEM-1024.Decapsulate(mlKemCiphertext, recipientPrivateKey)

3. Combine and derive (identical to sender):
   sharedSecret = HKDF-SHA256(x25519Shared || mlKemShared, "trashmail-hybrid-kem-v1")

4. Unwrap email's ephemeral AES-256 key using sharedSecret
5. Decrypt email content with ephemeral key
```

6.5 Key Sizes

Parameter	Size	Standard
X25519 public key	32 bytes	RFC 7748
X25519 private key	32 bytes	RFC 7748
ML-KEM-1024 public key	1,568 bytes	NIST FIPS 203
ML-KEM-1024 private key	3,168 bytes	NIST FIPS 203
ML-KEM-1024 ciphertext	1,568 bytes	NIST FIPS 203
Combined shared secret	32 bytes	HKDF-SHA256 output

6.6 Library

- **ML-KEM-1024:** `@noble/post-quantum` (audited, pure JavaScript implementation)
- **X25519:** WebCrypto API (`crypto.subtle.deriveBits`)
- **HKDF:** `@noble/hashes` (RFC 5869 compliant)

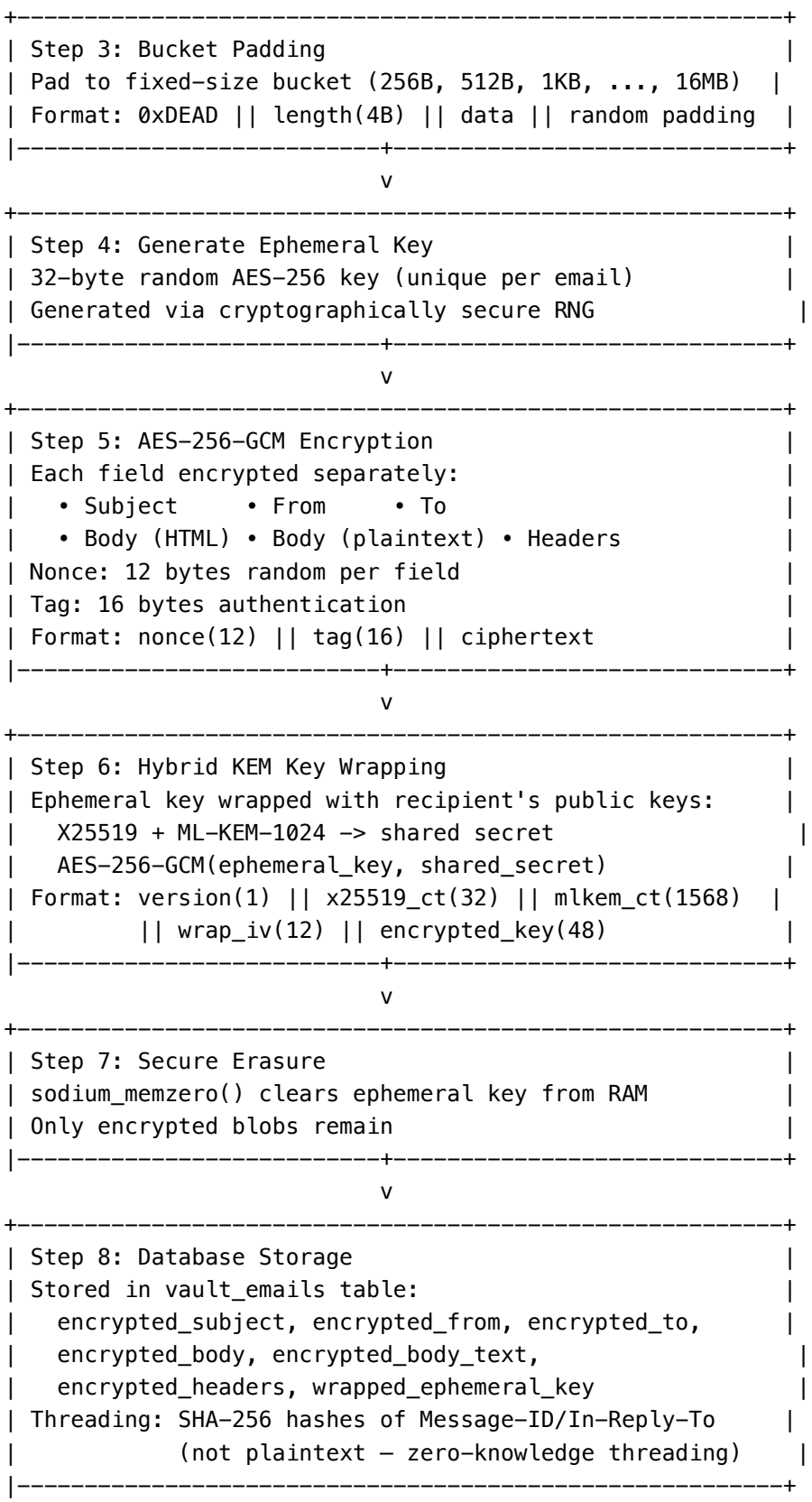
---

7. Email Encryption Pipeline

7.1 Incoming Email (SMTP -> Encrypted Storage)

When an email arrives at Aionda Mail's SMTP server:

```
+-----------------------------------------------------------+
| Step 1: SMTP Reception                                    |
| Mail server receives email -> passes to SMTP handler      |
|-------------------------+---------------------------------+
                          v
+-----------------------------------------------------------+
| Step 2: Compression (optional)                            |
| gzip level 6 for text content                             |
| Skipped for: JPEG, PNG, MP4, ZIP, PDF, DOCX, etc.         |
|-------------------------+---------------------------------+
                          v
```



7.2 Reading Email (Browser Decryption)

The reverse process happens entirely in the browser:

1. Fetch encrypted email from server via encrypted API
2. Parse wrapped ephemeral key (extract X25519 ciphertext + ML-KEM ciphertext)
3. **Hybrid KEM decapsulation** using vault private keys -> shared secret
4. Unwrap ephemeral AES-256 key
5. **AES-256-GCM decryption** of each field (subject, from, to, body, headers)
6. Reorder bytes: server format (nonce || tag || ct) -> WebCrypto format (nonce || ct || tag)
7. Remove bucket padding (detect 0xDEAD magic bytes)
8. Decompress if gzip (detect 0x1F8B magic bytes)
9. UTF-8 decode to plaintext

7.3 Sending Email

When composing and sending an email:

1. Client encrypts email content with a challenge-response protocol
2. Server receives encrypted payload, decrypts ephemerally (in-memory only), sends via SMTP
3. Server returns generated MIME headers to client (encrypted)
4. Client encrypts a copy with vault master key and stores in Sent folder
5. Ephemeral server-side plaintext is immediately discarded — never written to disk

7.4 Attachments

Each attachment is encrypted independently:

- Separate ephemeral AES-256 key per attachment
- Separate Hybrid KEM key wrapping per attachment
- Filename and MIME type encrypted separately
- No compression for already-compressed formats (JPEG, ZIP, PDF, etc.)

7.5 Email Threading (Zero-Knowledge)

Email threading (grouping related emails) uses only **SHA-256 hashes** of Message-ID and In-Reply-To headers. The server never sees the actual Message-ID strings — it can group emails by hash equality without knowing the content.

---

8. Encrypted API Transport Layer

8.1 Problem

Even with HTTPS, certain intermediaries can inspect traffic:

- **CloudFlare** (CDN/DDoS protection) terminates TLS and can see plaintext requests
- **Corporate proxies** may perform TLS inspection
- **API parameters** (like `?cmd=read_email&id=123`) leak metadata

8.2 Solution: End-to-End Encrypted API

All API communication is additionally encrypted end-to-end between the browser and the application server, inside the HTTPS tunnel:

```
Browser                                    Server
```

```
--------                                        ------

Phase 1: Key Exchange (once per session)
----------------------------------------
GET /get_encryption_keys          ->          Return 20 pre-generated
                                             Hybrid KEM keypairs
                                  <-         {uuid, x25519_pub, mlkem_pub}

Phase 2: Encrypted Request (every API call)
-------------------------------------
1. Pick random keypair from cache
2. Hybrid KEM encapsulate -> shared secret
3. gzip compress request payload
4. Bucket-pad compressed data
5. AES-256-GCM encrypt with shared secret
6. Generate ephemeral response keypair
7. POST /e {                      ->          8.  Validate key ownership
     encrypted_payload,                       9.  Hybrid KEM decapsulate
     key_uuid,                                10. AES-256-GCM decrypt
     x25519_ciphertext,                       11. Decompress
     mlkem_ciphertext,                        12. Route to API controller
     response_x25519_pub,                     13. Execute business logic
     response_mlkem_pub                       14. Encrypt response with
   }                                              client's response keys
                                  <-         15. Return encrypted response
16. Hybrid KEM decapsulate response
17. AES-256-GCM decrypt
18. Decompress -> plaintext response
```

8.3 Key Properties

- **One-time use:** Each API keypair is used exactly once, then permanently invalidated
- **Perfect Forward Secrecy:** Compromising one request key does not affect any other request
- **Session-bound:** Keys are claimed by a specific session and cannot be reused by another
- **Key pool:** Server maintains approximately 100,000 pre-generated keypairs
- **Auto-refetch:** Client automatically requests new keys when cache drops below 10
- **Key TTL:** Claimed keys expire after 24 hours
- **Bidirectional:** Both request AND response are encrypted — the server never returns plaintext

8.4 What CloudFlare Sees

With this architecture, CloudFlare (or any TLS-terminating proxy) sees only:

- `POST /e` — a single, opaque endpoint
- A binary blob of encrypted data
- No API command names, no parameters, no email IDs, no user data

---

9. Folder Sharing Cryptography

9.1 Sharing Model

Users can share encrypted folders with other Aionda Mail users. The sharing mechanism uses the Hybrid KEM to encrypt a folder-specific key for each recipient.

9.2 Sharing Flow

```
Folder Owner                                          Recipient
------------                                          ---------

1. Derive folder key from master key:
   folderKey = HKDF-SHA256(masterKey, folderUuid)

2. Fetch recipient's public keys:
   recipient.x25519_pub (32 bytes)
   recipient.mlkem_pub (1568 bytes)

3. Hybrid KEM encapsulate:
   hybridEncapsulate(recipient.x25519_pub, recipient.mlkem_pub)
   -> {x25519Ciphertext, mlKemCiphertext, sharedSecret}

4. Encrypt folder key:
   wrappedKey = AES-256-GCM(folderKey, sharedSecret, nonce)

5. Store on server:
   {x25519_ct, mlkem_ct, nonce, wrappedKey, permissions}

                                                      6. Fetch sharing record from server

                                                      7. Hybrid KEM decapsulate:
                                                         hybridDecapsulate(x25519_ct, mlkem_ct,
                                                           own_x25519_priv, own_mlkem_priv)
                                                         -> sharedSecret

                                                      8. Decrypt folder key:
                                                         folderKey = AES-GCM-decrypt(
                                                           wrappedKey, sharedSecret, nonce)

                                                      9. Decrypt emails in folder using folderKe
```

9.3 Permission Model

Permission	Capability
`readonly`	Read folder emails (decrypt only)
`einliefern`	Submit new emails into folder
`bearbeiten`	Edit folder contents
`antworten`	Reply to emails within folder
`vollzugriff`	Full access including ownership transfer

9.4 Cross-Vault Copy (Re-Wrapping)

When a recipient copies an email from a shared folder to their own vault, the email key must be **re-wrapped** for their own Hybrid KEM keypair. This is performed entirely client-side:

1. Decrypt shared folder key using recipient's private keys
2. Decrypt email's ephemeral key using folder key
3. Re-encapsulate ephemeral key with recipient's own public keys
4. Store re-wrapped copy in recipient's vault

The server facilitates the transfer but never sees any plaintext key material.

---

10. Vault Drive — Encrypted Document Storage

Vault Drive is the zero-knowledge document storage of Aionda Mail. Unlike emails, which use per-message ephemeral keys, Drive uses a **per-file-key architecture**: every document receives its own 32-byte AES-256-GCM key, generated client-side and wrapped with the master key.

10.1 Why per-file keys?

The per-file-key architecture offers three key advantages:

1. **Granular sharing:** Individual documents can be shared without exposing the master key. The server simply re-wraps the file key for the recipient — the content remains unchanged.
2. **Compromise isolation:** If a single file key is compromised (e.g. via a shared link), all other documents remain protected.
3. **Efficient sharing without re-encryption:** When sharing, the content does not need to be re-encrypted — all recipients read the same ciphertext with a re-wrapped key.

10.2 Upload flow

```
Client                                                  Server
------                                                  ------

1. Generate random file key:
   fileKey = randomBytes(32)

2. Encrypt content, name, MIME type:
   encContent = AES-256-GCM(content, fileKey, nonce1)
   encName    = AES-256-GCM(name,    fileKey, nonce2)
   encMime    = AES-256-GCM(mime,    fileKey, nonce3)

3. Wrap file key with master key:
   wrappedKey = AES-256-GCM(fileKey, masterKey, nonce4)

4. Send encrypted package:
   POST /e {
     encContent, encName, encMime,
     wrappedKey, wrappedKeyNonce
```

```
   }
                                                    5. Store in vault_files + vault_file_chunk
                                                       (pure ciphertexts, no key material)
```

10.3 Download flow

```
1. Fetch chunk + wrapped_key from server
2. Unwrap file key:
   fileKey = AES-256-GCM-decrypt(wrappedKey, masterKey, nonce)
3. Decrypt content, name, MIME:
   content = AES-256-GCM-decrypt(encContent, fileKey, nonce1)
```

Decryption happens inside a **Web Worker** that holds the master key only temporarily in memory. After the operation, the memory is overwritten.

10.4 Sharing — key rewrap only

The central advantage of per-file keys shows when sharing: the content is **not** re-encrypted. Only the 32-byte file key is re-wrapped using the recipient's hybrid KEM.

```
Owner                                               Recipient
-----                                               ---------

1. Unwrap file key:
   fileKey = AES-GCM-decrypt(wrappedKey, masterKey)

2. Fetch recipient's public keys:
   recipient.x25519_pub, recipient.mlkem_pub

3. Hybrid KEM encapsulation:
   hybridEncapsulate(recipient.x25519_pub,
                     recipient.mlkem_pub)
   -> {x25519_ct, mlkem_ct, sharedSecret}

4. Wrap file key with sharedSecret:
   shareWrappedKey = AES-256-GCM(fileKey,
                                 sharedSecret, nonce)

5. Store share record:
   INSERT INTO vault_file_shares {
     file_uuid, recipient_account_id,
     x25519_ephemeral, mlkem_ciphertext,
     share_wrapped_key, permissions
   }
                                                    6. Fetch share record + chunk
                                                    7. Hybrid KEM decapsulation:
                                                       hybridDecapsulate(x25519_ct, mlkem_ct,
                                                         own_x25519_priv, own_mlkem_priv)
                                                       -> sharedSecret
                                                    8. Unwrap file key:
                                                       fileKey = AES-GCM-decrypt(
                                                         shareWrappedKey, sharedSecret)
```

```
9. Decrypt SAME ciphertext:
   content = AES-GCM-decrypt(
     encContent, fileKey)
```

The owner does not need the recipient's master key — only the public hybrid KEM keys, which the server stores in plaintext.

10.5 Folder sharing (recursive)

When a folder is shared, the client processes all contained documents and subfolders recursively. Each file receives its own share record containing the respective document's file key, re-wrapped for the recipient. The folder itself has no folder key — the structure is linked via parent UUIDs.

Important: The KEM encapsulation step is performed **per operation**, not per file. All files in a batched share use the same `sharedSecret` — making the operation efficient without reducing security (each file still has its own file key).

10.6 Permission model

Permission	Can perform
Read	Download, preview, read metadata
Full access	+ rename, upload new version, move inside the shared folder, upload new documents
Not possible	Delete (owner only), move out of shared folder (owner only)

Permissions are stored in the `vault_file_shares` record and enforced server-side. Cryptography protects the content — access control protects the operations.

10.7 Rename, overwrite and metadata updates

For shared documents, rename and overwrite operations are performed directly on the `vault_files` record — not on individual share records. All recipients immediately see the new name or content, as they reference the same ciphertext.

On **overwrite** (new version), a **new file key** is generated, the old ciphertext is replaced, and all existing share records are re-wrapped client-side with the new key. The owner must load the public keys of all recipients for this operation.

10.8 Preview and in-memory decryption

Images, PDFs and other documents are decrypted **exclusively in memory** for preview. There is no disk cache with plaintext data. The client uses the **VaultDataLayer**, an encrypted IndexedDB cache that persistently stores ciphertexts and only decrypts them on demand.

Thumbnails are **never** generated server-side — the server never sees content. Thumbnails are generated client-side from the decrypted original and cached encrypted as well.

10.9 What the server sees

Visible to server	Not visible
Encrypted content (random bytes)	Plaintext content
Encrypted filename (random bytes)	Plaintext filename
Encrypted MIME type (random bytes)	File type (image, PDF, text…)
File size (padded to bucket boundaries)	Actual original size
Upload timestamp	File key, master key
Owner account ID	Contents of subfolders
Parent folder UUID (for structure)	Folder names (also encrypted)
Share records with KEM ciphertexts	Recipient master key, unwrapped file key

10.10 Quota enforcement

Quota enforcement happens server-side based on the **encrypted file size** (including bucket padding). The server does not know the actual plaintext size — the padding overhead is intentionally paid by the user to prevent size leakage.

Limits: - **Free:** 100 MB total storage, max. 10 MB per document - **Plus:** 5 GB total storage, max. 100 MB per document

10.11 External Sharing — public share links

Vault Drive documents can be shared with recipients who **do not have an Aionda Mail account** via public share links served from the isolated domain `mail.aionda.com`. The feature preserves zero-knowledge by treating each share as an **independent crypto envelope**, decoupled from the owner's vault master key.

Protection modes:

Mode	Trust factor	Use case
`link_only`	URL fragment (never sent to server)	Convenience — anyone with the link can access
`password`	Argon2id-derived key	Out-of-band password transfer (SMS, phone call)
`recipient_pubkey`	Hybrid KEM (X25519 + ML-KEM-1024)	Post-quantum secure when recipient has pre-registered public keys

10.11.1 Share creation (owner client)

```
1. fileKey := AES-GCM-decrypt(vault_files.wrapped_key, masterKey)
2. shareKey := randomBytes(32)                          // ephemeral, per share
3. wrappedFileKey := AES-GCM(fileKey, shareKey)         // new envelope
4. unlockVerifier := HMAC-SHA256(shareKey,
                     "unlock:" || shareUuid)            // proof-of-knowledge
5. If password protection:
     salt := randomBytes(16)
     pwKey := Argon2id(password, salt, t=3, m=64MB, p=1)
     wrappedShareKey := AES-GCM(shareKey, pwKey)
     -> Link: https://mail.aionda.com/s/<shareUuid>
```

```
   Else (link_only):
     -> Link: https://mail.aionda.com/s/<shareUuid>#<base64(shareKey)>
       (URL fragment — browsers never transmit fragments to the server)

6. Encrypted metadata (per-share, with shareKey):
     encFilename, encMime, encMessage  // all AES-GCM
7. POST /e  (encrypted API transport, see §8)
   -> server stores share record — never sees plaintext
```

Link delivery is user-chosen, not routed through Aionda Mail. After creation, the owner decides which channel transports the link: copy-to-clipboard, QR code, a pre-filled `mailto:` draft opened in the user's own mail client, Signal, SMS, AirDrop, or any other out-of-band channel. Aionda Mail never sees, sends, or logs the outgoing delivery — the server only learns which share-UUIDs exist. This matters for two reasons: (a) the owner can select the most trusted channel available to them (a corporate compliance policy, a preferred messenger, a verified face-to-face QR scan), and (b) for `password`-protected shares it enables **channel separation** — link via channel A, password via channel B — so a single compromised channel never grants access on its own.

10.11.2 Share access (recipient, no account required) The recipient visits `https://mail.aionda.com/s/<s` The Share Page is a **separate bundle** from the Manager, with its own reproducible build, Subresource Integrity manifest, and a `/.well-known/integrity.json` endpoint for offline verification.

```
1. Share Page bootstrap (client generates ephemeral hybrid KEM keypair)
2. share_fetch_meta -> { wrapped_file_key, salt?, encrypted_message, ... }
3. Unlock phase — produces a one-time unlock_token:
     password mode:    server verifies Argon2id derivation -> unlock_token
     link_only mode:   client computes HMAC-SHA256(shareKey, "unlock:"||uuid)
                       server verifies against stored unlockVerifier
                       -> unlock_token
4. fileKey := AES-GCM-decrypt(wrappedFileKey, shareKey)
5. share_download_chunk (per chunk, unlock_token required)
6. Client hashes plaintext, calls share_confirm_download
```

The `unlock_token` unifies the flow across all three protection modes and enables centralized rate-limiting and audit-logging — link-only access requires proof of knowledge of the fragment, so bots and crawlers without the fragment cannot issue downloads.

10.11.3 Enforced policies Every share must declare the following at creation time (all enforced server-side):

- **`expires_at`** — mandatory, default 7 days, maximum 90 days
- **`max_downloads`** — mandatory counter, default 10
- **Rate limiting** — Argon2 attempts on password shares are throttled per IP hash and per share
- **`revoked_at`** — the owner can revoke any share with a single click; future unlock attempts and downloads return a unified "share unavailable" response (same error as expired or exhausted — no enumeration oracle)

10.11.4 Tamper-evident audit chain All relevant access events are appended to the existing `enterprise_audit_log` hash chain (SHA3-256, see §18.2). The following action types are reserved for external sharing:

`EXT_SHARE_CREATED`, `EXT_SHARE_EMAIL_SENT`, `EXT_SHARE_VIEWED`, `EXT_SHARE_UNLOCKED`, `EXT_SHARE_PREVIEWED`, `EXT_SHARE_DOWNLOAD_STARTED`, `EXT_SHARE_DOWNLOAD_CONFIRMED`, `EXT_SHARE_INTEGRITY_BROKEN`, `EXT_SHARE_PASSWORD_FAIL`, `EXT_SHARE_REVOKED`, `EXT_SHARE_ROTATED`, `EXT_SHARE_EXPIRED`.

IP and user-agent hashes use a **daily rotating salt** (`vault_drive_external_share_daily_salts`, pruned after 30 days) — historical hashes become non-reversible after salt rotation for GDPR compliance, while short-term correlation remains available for the owner.

10.11.5 Key rotation (share rewrap) The owner can rotate a share at any time without re-uploading the content:

```
1. new_shareKey := randomBytes(32)
2. new_wrappedFileKey := AES-GCM(fileKey, new_shareKey)
3. INSERT new share row; old.linked_to_uuid := new.share_uuid
4. old.revoked_at := NOW()
```

Recipients using the old link receive “share unavailable”. The owner forwards the new link. All access events from both shares remain linked via `linked_to_uuid` in the audit chain.

10.11.6 What revocation does — and does not Revocation **stops future server-side delivery** of the share. It does **not** retroactively revoke share keys, file keys, or chunks already delivered. A recipient who has already downloaded the plaintext — or who captured the link fragment — can continue to use that material locally. For use cases with higher assurance requirements, combine: short `expires_at`, low `max_downloads`, password protection, and the Guardian browser extension for the recipient.

10.11.7 What the server sees

Visible to server	Not visible
`share_uuid`, `file_uuid`, `account_id` (owner)	Plaintext filename, MIME type, content, message
`wrapped_file_key`, `wrapped_share_key` (ciphertexts)	`share_key` (link-only: lives only in the URL fragment, client-side)
`unlock_verifier` (HMAC output, not reversible)	Password, Argon2-derived key
`protection_mode`, `expires_at`, `max_downloads`, `allow_preview`	Recipient identity (only salted IP/UA hash, pruned after 30 days)
`encrypted_filename`, `encrypted_mime`, `encrypted_message` (ciphertexts)	Their plaintexts
Plaintext `sender_display_name` (recipient sees it before unlock)	
Audit-chain entries for every access event	

11. Aionda Chat — Post-Quantum E2EE Messaging

Aionda Chat is the integrated real-time messaging surface of Aionda Mail. Unlike email — which uses one-shot ephemeral keys per message — chat conversations are long-lived and require **forward secrecy** and **post-compromise security** for every individual message. To achieve this we designed a dedicated protocol: **AAR (Aionda Async Ratchet)** — a post-quantum variant of Signal's X3DH + Double Ratchet, where every Diffie–Hellman step is replaced by a hybrid KEM (X25519 + ML-KEM-1024).

11.1 Why a Dedicated Protocol?

Email and chat have fundamentally different security profiles:

Property	Email	Chat
Cadence	Bursty (minutes/hours apart)	Real-time (seconds apart)
Session length	Single message	Continuous, days to years
Forward secrecy unit	Per message	Per message **within** a long session
Post-compromise recovery	Not required (one-shot key)	Required — each new message heals from past compromise
Asynchrony	High (recipient often offline)	High (recipient often offline)
Group dynamics	Static recipient list	Dynamic add/remove members

A chat conversation that simply re-used the email pipeline would either (a) burn a fresh ephemeral keypair every keystroke (unacceptably slow), or (b) share a single static conversation key (no forward secrecy). Neither is acceptable. AAR resolves this by maintaining a continuously ratcheting key state per peer — every message advances the state irrevocably.

11.2 Building Blocks

```
AAR uses exactly four primitives:

   Hybrid KEM         X25519 + ML-KEM-1024  (key encapsulation)
   AEAD               AES-256-GCM           (message encryption)
   KDF                HKDF-SHA256           (key derivation)
   Signatures         Ed25519               (identity-key binding, SPK signing)

No new cryptographic assumptions are introduced — every primitive is also
used elsewhere in the system and has been independently audited.
```

11.3 KeyBundle — The Asynchronous Handshake Material

Every user publishes a **KeyBundle** to the server when chat is first activated. The bundle enables peers to start sending messages even while the user is offline.

```
KeyBundle (per account, hybrid throughout):
```

```
Identity Key (IK)          — long-lived
   |-- IK.x25519_pub
   |-- IK.mlkem_pub
   |-- IK.ed25519_pub      — used to sign the SPK

Signed Pre-Key (SPK)       — rotated weekly
   |-- SPK.x25519_pub
   |-- SPK.mlkem_pub
   |-- Ed25519 signature   — signs the concatenation, binds SPK to IK

One-Time Pre-Keys (OPK)    — pool of ~100, consumed atomically
   |-- OPK.x25519_pub
   |-- OPK.mlkem_pub
```

Server storage: Only public halves of each key are stored, plus signatures and metadata. Private halves never leave the originating browser. The server enforces an atomic `UPDATE … LIMIT 1 SET consumed_ts = NOW()` when an OPK is fetched, guaranteeing single-use semantics under concurrent requests.

When the OPK pool falls below 20, the client tops up the pool with a fresh batch — this prevents the asynchronous handshake from degrading to a degenerate mode lacking one-time entropy.

11.4 PQXDH-Style Handshake (X3DH-PQ)

To start a new conversation with Bob, Alice fetches Bob's bundle (`IK_B`, `SPK_B`, one `OPK_B`) and Bob's Ed25519 signature on `SPK_B`. Alice verifies the signature, then performs four hybrid KEM encapsulations:

```
1. Verify Ed25519 signature on SPK_B    — binds SPK to long-term identity

2. Generate ephemeral Alice key (EK_A):
   EK_A.x25519, EK_A.mlkem  (one-time, discarded immediately after handshake)

3. Four hybrid KEM encapsulations:
   ss1 = HKEM(IK_A_priv  ⊗ SPK_B_pub)    — Alice identity -> Bob SPK
   ss2 = HKEM(EK_A_priv  ⊗ IK_B_pub)     — Alice ephemeral -> Bob identity
   ss3 = HKEM(EK_A_priv  ⊗ SPK_B_pub)    — Alice ephemeral -> Bob SPK
   ss4 = HKEM(EK_A_priv  ⊗ OPK_B_pub)    — Alice ephemeral -> Bob OPK

   (each ss_i is itself the HKDF combination of an X25519 secret
    AND an ML-KEM secret — see Section 6.4)

4. Root key derivation:
   SK = HKDF-SHA256(
          IKM  = ss1 || ss2 || ss3 || ss4,
          info = "AAR-X3DH-v1" || consumed_OPK_id
        )

5. Forget every private piece of ephemeral material; SK seeds the ratchet.
```

The construction guarantees:

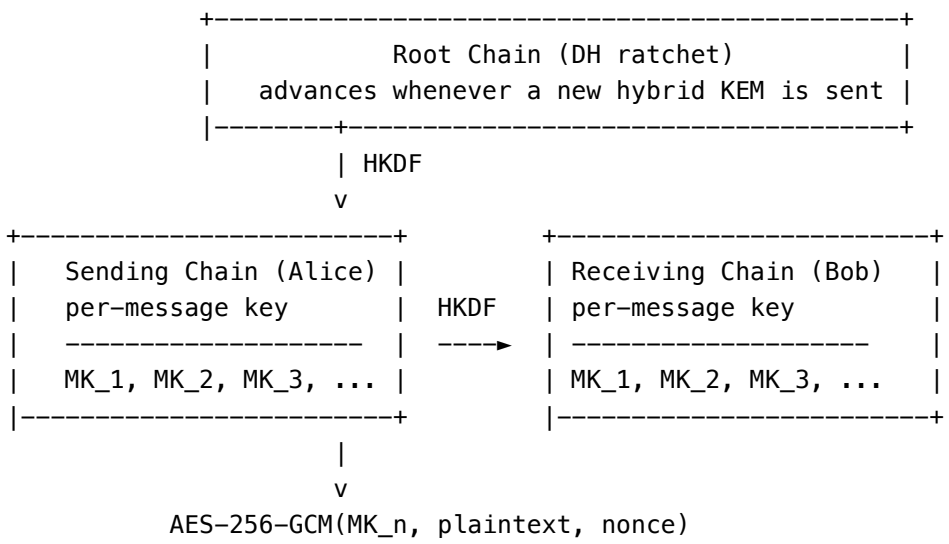
- **Mutual authentication** via the Ed25519-signed `SPK_B` and `IK_A` binding

- **Forward secrecy** even if `IK_B` is later compromised — `ss3` and `ss4` use ephemeral keys on both sides
- **Post-quantum security** because every `ss_i` includes an ML-KEM term — an attacker harvesting today's traffic for a future quantum computer still cannot reconstruct any of the four secrets
- **Replay resistance** through atomic OPK consumption — Bob's server-side trigger refuses to release the same OPK twice

The OPK ID consumed by Alice is bound into the `info` parameter of HKDF — Bob's side can therefore reproduce the same root key only with the same OPK, and only once.

11.5 Double Ratchet — Per-Message Keys

After the handshake, both sides advance a **Double Ratchet** state for every message:



Every chat event carries the sender's current **ratchet public key** (hybrid pair, ~1.6 KB). When the recipient receives a message whose ratchet key differs from the last one seen, both parties perform a fresh hybrid KEM, derive a new root key, and reset their chains. This is the **DH ratchet** in Signal terminology — except every step is a full hybrid KEM, not a single Diffie–Hellman.

Per-message guarantees:

- **Forward secrecy:** Deletion of `MK_n` makes message `n` undecryptable, even given the full long-term state at the time of capture
- **Post-compromise security:** Once a fresh DH-ratchet step has occurred after a compromise, all subsequent messages are protected from the attacker
- **Out-of-order delivery:** Message keys are cached for late-arriving messages (capped at 1000 keys per session to bound memory)

11.6 Message Container

Every chat event uploaded to the server is a self-contained ciphertext:

```
MessageContainer (base64-encoded, stored in chat_events.payload):

  +-------------------------------------------------------------+
  | ratchet_pub      :  Hybrid KEM public key (current step)    |
```

```
| prev_chain_len   :  Number of messages in previous chain     |
| msg_number       :  Counter within current chain             |
| ciphertext       :  AES-256-GCM(MK_n, plaintext || header)   |
| tag              :  AES-256-GCM authentication tag           |
|--------------------------------------------------------------+
```

The header inside the AEAD plaintext contains: sender login string, content_type (text/quote/file-reference), timestamp, optional reply_to_event_uuid.

The container is opaque to the server — including the message type, the sender's display name format, and any quoted reply context. Aionda's database stores the container as-is in `chat_events.payload`.

11.7 Group Conversations — Per-Recipient Sender Keys

For room conversations (3+ participants), AAR uses a **per-recipient fan-out** model. The sender derives one sender-key chain per peer pair and encrypts a message once per recipient. Each recipient therefore receives a personally-addressed ciphertext, decryptable only with the keys derived from their own AAR session with the sender.

Trade-off: Fan-out cost is `O(N)` in recipients — acceptable for typical team sizes (≤25 participants). For larger groups a future MLS-based mode is planned (see Roadmap). The current model is preferable to a single shared group key because: (a) it preserves forward secrecy on a per-pair basis, (b) it does not require key-rotation on member removal beyond the bilateral level, and (c) it inherits the post-quantum guarantees of pairwise AAR without modification.

11.8 Initial Conversation Container

When Alice starts a new room with Bob, Carol, and Dan, the first message to each participant is wrapped as an **initial-handshake container** that carries (a) the X3DH handshake material consumed against that participant's bundle and (b) the first message itself. The receiver dispatches on the discriminator field:

```
{
  "type": "initial",
  "initialMessage": { /* handshake metadata, OPK id consumed */ },
  "encryptedMessage": "<base64 MessageContainer>"
}
```

Subsequent messages in the same room reuse the established AAR session and only contain the `encryptedMessage` portion.

11.9 Transport — Encrypted API + Mercure SSE + WebSocket

The chat plane uses three orthogonal channels:

Channel	Direction	Purpose	Plaintext seen by server
`/e` Encrypted API	Client -> Server	Send message, fetch history, key-bundle ops (11 endpoints)	None — all 11 chat endpoints use the same hybrid-KEM transport documented in Section 8
Mercure SSE	Server -> Client	Push delivery of new events (`chat.event_received`, `chat.read_receipt`)	None — the server pushes the same opaque MessageContainer it stored
WebSocket `/chat/ws`	Bi-directional	Reconnect backfill (`sync_request`), heartbeat, presence	Presence boolean (online/offline) and login-string only — never message content

Presence is held in an in-memory map on `aionda_chat_realtime`; a single intranet-only HTTP endpoint exposes a `login -> online` lookup for the team picker so users can see whether a peer is reachable. No history, no read state, no content ever traverses the presence channel.

11.10 Read Receipts

Read receipts are an opt-in per-account setting (`chat_participants.send_read_receipts`). When enabled, marking a message as read publishes a `chat.read_receipt` event containing:

```
{ conversation_uuid, last_read_event_uuid, account_login }
```

Note that `last_read_event_uuid` is **not** the message content — it is a server-allocated identifier already known to the server. No additional information leaks beyond "Alice has now seen messages up to event X." Recipients who disable read receipts (`send_read_receipts = false`) never emit such events, and the server enforces the toggle.

11.11 Audit Chain Integration

For Enterprise accounts, every chat operation is appended to the existing tamper-evident audit log (`enterprise_audit_log`, SHA3-256 hash chain — see §18). The reserved action types are:

`CHAT_CONVERSATION_CREATED`, `CHAT_PARTICIPANT_ADDED`, `CHAT_PARTICIPANT_REMOVED`, `CHAT_MESSAGE_SENT`, `CHAT_MESSAGE_READ`, `CHAT_KEYBUNDLE_PUBLISHED`, `CHAT_KEYBUNDLE_ROTATED`, `CHAT_OPK_TOPPED_UP`, `CHAT_CONVERSATION_LEFT`.

The audit record contains only the conversation UUID, the actor's login, and a timestamp — **never** the encrypted payload, ratchet keys, or message contents.

11.12 What the Server Sees

Visible to server	Not visible
`conversation_uuid`, list of participants (by `mail_account.name`)	Plaintext message content

Visible to server	Not visible
Encrypted `MessageContainer` payloads (opaque ciphertexts)	Ratchet keys, chain keys, message keys
Public halves of IK, SPK, OPK; Ed25519 signatures on SPK	Private halves of any keypair
OPK consumption counter and timestamp	Which OPK was consumed for which conversation (correlation prevented by HKDF-info binding only existing client-side)
Conversation cadence (timestamps of `chat_events` rows)	Quoted reply chains, file attachments inside the container
Presence boolean (online/offline)	Typing indicators (peer-to-peer only, never reach the server)
Audit-chain entries for Enterprise accounts	Plaintext content of audit entries

11.13 Cryptographic Audits

The AAR protocol was implemented from scratch in TypeScript and underwent three independent AI-assisted audits documented in `typescript/manager/chat/crypto/AUDIT.md`, `AUDIT_SECOND_OPINION.` and `AUDIT_THIRD_OPINION.md`. Findings from each round were incorporated before public deployment. The primary surfaces audited were: handshake symmetry, OPK reservation under concurrency, deep-cloning of immutable ratchet state, and the AEAD framing of the `MessageContainer`.

11.14 Implementation Boundaries

The chat code base lives in `typescript/manager/chat/` (~9,000 lines of TypeScript) and `includes/classes/Ap` (PHP service layer). Key audit artifacts:

```
crypto/aar-types.ts            — type-only declarations, no logic
crypto/aar-keybundle.ts        — KeyBundle creation, rotation, serialization
crypto/aar-x3dh.ts             — Handshake (initiator + responder paths)
crypto/aar-double-ratchet.ts   — Root chain, sending chain, receiving chain
```

The clean separation between protocol logic (`crypto/`) and transport/UI (`chat-store.ts`, `chat-panel.ts`, …) ensures that the cryptographic core can be re-audited without UI scope creep.

---

12. Aionda Video Calls — Post-Quantum E2EE Conferencing

Aionda Mail includes end-to-end encrypted group video calls (cipher suite `AIONDA_PQ_CALL_V1`). The media relay never sees plaintext audio or video — it forwards only ciphertext frames. Group keys are agreed with a **post-quantum hybrid** handshake.

12.1 Security Goals & Trust Model

In scope — the protocol provides:

- **Confidentiality** of audio and video media, end to end.
- **Authenticity** of the group — every participant is cryptographically a member of the same MLS group, verified out-of-band through the Short Authentication String (Section 12.5).
- **Forward secrecy** — past media stays secret even if current keys are later compromised.

- **Post-compromise security** — the group recovers confidentiality after a compromised member is removed and the group rekeys.
- **Post-quantum confidentiality** — group establishment uses the X-Wing hybrid KEM.
- **Server independence** — a malicious or compromised SFU, signaling server or storage layer cannot read or inject media.

Out of scope (explicitly *not* defended against):

- Traffic analysis — packet timing, packet sizes, participant count and call duration are observable (Section 12.9).
- Endpoint compromise (see trust assumption below).
- Operating-system-level malware or physical device seizure.

Endpoint trust assumption. The model assumes an uncompromised browser and operating system. Frames are sealed inside the browser media pipeline (`RTCRtpScriptTransform`), but a compromised endpoint can read plaintext media *before* encryption or *after* decryption. The web-delivery trust model and its mitigations (Guardian response signing) are covered in Section 18 and Section 23.

12.2 Why a Dedicated Protocol?

Real-time conferencing has two requirements that the email and chat layers do not:

1. **Per-frame encryption at line rate.** Audio/video frames must be encrypted individually so a Selective Forwarding Unit (SFU) can route, drop and re-order them without ever decrypting. This is the role of **SFrame** (RFC 9605).
2. **Dynamic group key agreement.** Participants join and leave mid-call. Each membership change must rekey the group with forward secrecy and post-compromise security. This is the role of **MLS** (RFC 9420).

The server-side SFU (`mediasoup`) is treated as **untrusted infrastructure**: it is a zero-knowledge packet router, exactly like the storage layer for email.

12.3 Cipher Suite — `AIONDA_PQ_CALL_V1` (`0xFF01`)

The MLS profile substitutes the classical primitives of a standard MLS suite with post-quantum hybrids.

Role	Algorithm	Notes
HPKE KEM	**X-Wing** (ML-KEM-768 + X25519)	Hybrid; `draft-connolly-cfrg-xwing-kem`
Signatures	**ML-DSA-65**	NIST FIPS 204 (Dilithium), Level 3
KDF	HKDF-SHA-256	RFC 5869
AEAD (HPKE + SFrame)	**AES-256-GCM**	AES-256 provides a comfortable margin against known quantum search (Grover) attacks

No downgrade. There is no classical-only fallback. If the post-quantum handshake cannot be completed, the call fails closed — a downgrade is a hard error, never a silent weakening.

X-Wing maturity. X-Wing is currently an IETF/CFRG draft (`draft-connolly-cfrg-xwing-kem`). The current profile adopts it as the preferred hybrid KEM; future protocol revisions may migrate to a finalized CFRG/NIST-standardized hybrid KEM while preserving the MLS and SFrame layers unchanged.

Cryptographic agility. The cipher-suite identifier (`0xFF01`) fully determines the KEM, signature algorithm, KDF and AEAD. A future `AIONDA_PQ_CALL_V2` can replace any primitive without changing the higher protocol layers; suites are selected by identifier and never silently negotiated downward.

12.4 Group Key Agreement (MLS, RFC 9420)

Every participant is represented in the MLS ratchet tree by a **KeyPackage** (X-Wing public KEM key + ML-DSA-65 credential). The group evolves through *epochs*:

- **Welcome / Commit** — a joining member is added with an X-Wing-encapsulated path secret; the ratchet tree advances to a new epoch.
- **Rekeying on every membership change.** Any change — a join, a leave, a device replacement, or an identity-key rotation — advances the group to a fresh epoch. This is what delivers forward secrecy (a new joiner cannot read past media) and post-compromise security (a removed member cannot read future media).
- **Remove-Commit.** When a participant leaves (or a stale/zombie peer is detected), the owner issues a Remove-Commit that rekeys the group, so a departed member cannot decrypt subsequent frames even if it retained old key material.

The hybrid construction is designed so that compromising only **one** component — ML-KEM-768 *or* X25519 — does not by itself reveal the group secret.

12.5 Identity & Authentication

- **Credentials.** Each member carries an MLS **Credential** that binds a participant identity to its ML-DSA-65 signature key. Every MLS handshake message (Welcome / Commit / Update) is signed with that key, so the SFU or signaling server cannot forge group operations.
- **Short Authentication String (SAS).** The room link carries a high-entropy **pre-shared secret** in the URL fragment, which is never sent to the server in the clear. The MLS group exporter and this secret are folded together into a Short Authentication String, whose fingerprint the client **pins**. A mismatch — for example a signaling server or SFU attempting to splice in a rogue participant — triggers a **hard abort with no soft fallback**. This binds the cryptographic group to the out-of-band room secret and defeats server-side man-in-the-middle.
- **Authentication factor.** For a guest room, *possession of the room link is equivalent to possession of the room credential* — the room secret is the sole authentication factor. Authenticated Aionda participants additionally redeem a signed, account-scoped signaling token (Section 12.8).

12.6 Media Encryption (SFrame, RFC 9605)

Each media track is encrypted frame-by-frame **before** it reaches the SFU. The key lifecycle:

```
MLS group secret
      |  MLS Exporter (RFC 9420 §8)
      v
```

```
SFrame base key
      |  HKDF(base_key, KID = sender || epoch)
      v
per-sender key
      |  AES-256-GCM(nonce = frame counter, aad = sframe_header)
      v
per-frame ciphertext || tag      ->  wire = sframe_header || ciphertext || tag
```

- The SFrame key is derived from the **MLS Exporter**, so it exists only on authenticated group members — never on the server.
- Encryption runs in a dedicated `RTCRtpScriptTransform` Worker (the standardised Encoded-Transform API, available in Chromium and Safari/WebKit), so frames are sealed on the browser's media pipeline.
- The codec is VP8; the SFU performs keyframe routing on metadata only — it never inspects the encrypted payload.

12.7 Replay Protection

Each encrypted frame carries a monotonic SFrame **counter** inside its authenticated header, and is bound to the current MLS epoch. Receivers track the highest counter seen per sender (a persisted receive-side counter store). A frame with a duplicate or regressed counter, or one bound to a stale epoch, is rejected — a captured ciphertext cannot be replayed into the session.

12.8 Signaling & Access Model

- **Signaling** runs over an authenticated WebSocket (`aionda_signaling`). Each session is gated by a short-lived **JWT** minted by the application server after the client redeems the room secret via the encrypted `/e` API (see Section 8). The token is symmetric (HS256) and validated only between the application server and the single signaling service; it authorizes session setup and never carries media or group secrets. The signaling server relays MLS Welcome/Commit messages but cannot read them.
- **Room link = bearer credential.** Anyone holding the full link (room UUID + fragment secret) may join — the model is intentionally "whoever has the link". Database IDs are never exposed; rooms are addressed only by unguessable UUIDs.
- **Owner control.** The room owner (verified server-side by account ownership) may mark a room *ended* to block future joins, and may remove a present participant via an MLS Remove-Commit (Section 12.4).

12.9 What the SFU / Server Sees — and Metadata Limitations

The infrastructure CAN see	The infrastructure CANNOT see
Encrypted frame sizes, timing, routing metadata	Audio or video content (frames are AES-256-GCM sealed)
MLS handshake ciphertext (Welcome/Commit)	Group secrets or SFrame keys (derived client-side from the MLS exporter)
Room UUID, participant count, JWT subject	The room link secret (lives in the URL fragment)
ICE candidates / transport addresses	Any plaintext that would let it join or decrypt the session

Metadata the system intentionally does not hide: IP addresses exposed to ICE / STUN / TURN infrastructure, packet timing, packet sizes, participant count, and call duration. Traffic-analysis resistance is not a design goal (Section 12.1).

12.10 Security Properties (Summary)

Property	Guarantee
Confidentiality	Only current MLS group members possess the SFrame keys.
Authentication	All MLS messages are signed with ML-DSA-65; the group is bound to the room secret via the pinned SAS.
Forward secrecy	Compromise of current secrets does not reveal past media.
Post-compromise security	Rekeying restores confidentiality after a member is removed.
Server independence	Compromise of the signaling server, SFU or storage does not expose media.
Post-quantum	Group establishment relies on the X-Wing hybrid KEM (ML-KEM-768 + X25519).

These properties describe the protocol *design*. An independent third-party cryptographic audit of the call subsystem is planned; until it is published, the guarantees above are a self-assessment.

---

13. Aionda Tasks - End-to-End Encrypted Task Management

Aionda Mail includes an encrypted task manager: task lists, subtasks, notes, due dates, reminders and recurring tasks. It follows the same zero-knowledge rules as the mailbox. Everything a user writes is encrypted in the browser; the server stores ciphertext plus a small, deliberately chosen set of plaintext scheduling fields.

13.1 Per-Task Keys

Every task and every task list receives its own freshly generated 32-byte key (AES-256-GCM), created in the browser with the WebCrypto random number generator. Content fields are encrypted with that key; the key itself is then wrapped with the vault master key:

```
random 32-byte task key
      |  AES-256-GCM, fresh 96-bit IV per field
      v
encrypted_title / encrypted_notes / encrypted_steps / encrypted_metadata

vault master key
      |  HKDF-SHA-256(salt = 32 random bytes, info = "trashmail-tasks-v1")
      v
wrapping key
      |  AES-256-GCM
      v
```

```
wrapped_key = version(1) || salt(32) || IV(12) || ciphertext || tag
```

The server stores only `wrapped_key` and the ciphertext columns. It never receives the task key, the master key or any plaintext. Without the master key, which never leaves the browser, the wrapped key cannot be opened.

Domain separation. The HKDF label `trashmail-tasks-v1` is specific to the task module. Without such a label, a wrapped key produced by another module (calendar, contacts) would be interchangeable with a task key. The label binds each wrapped key to the module that created it.

Independent decryption. Tasks and lists are decrypted one by one. A single unreadable record is logged and skipped instead of failing the whole view, so one damaged ciphertext cannot lock a user out of the remaining tasks.

13.2 What Is Encrypted

Field	Contents	Protection
`encrypted_title`	Task title	AES-256-GCM with the per-task key
`encrypted_notes`	Free-text note	AES-256-GCM with the per-task key
`encrypted_steps`	JSON array of subtasks (text and done state)	AES-256-GCM with the per-task key
`encrypted_metadata`	JSON, including the recurrence rule (RRULE)	AES-256-GCM with the per-task key
`encrypted_name`	Name of a task list	AES-256-GCM with the per-list key

Every field is sealed with its own fresh 96-bit IV, so identical plaintext never produces identical ciphertext.

List names are encrypted as well. A list called “Acquisition Contoso” is content, not metadata, and is treated as such. The server sees that an account owns five lists, not what they are called.

13.3 Deliberate Plaintext Metadata

A small set of scheduling fields is stored in plaintext, so the server can filter, sort, count and schedule without decrypting anything:

Plaintext field	Purpose
Due date	The “Planned” smart list, overdue detection, sorting
Reminder time	Triggering the reminder at the right moment
Important flag	The “Important” smart list
Completed flag and completion time	Separating open from completed tasks
“My Day” date	The “My Day” smart list, which is valid for the current day only
Position	Manual ordering inside a list
List assignment (UUID)	Grouping tasks under their list

Plaintext field	Purpose
Linked email (UUID)	Reference to the email a task was created from

The server learns when something is due, never what it is. It can determine that an account has fourteen open tasks, three of them due on 3 August, one of them marked important. It cannot read a single title, note, subtask, recurrence rule or list name.

This is the same trade-off already made for calendar event times, and it is documented rather than hidden. The alternative - encrypting the due date as well - would force the client to download and decrypt the complete task history for every filter, every counter and every reminder. The scheduling fields carry no content, and the objects they belong to are addressed only by UUID.

13.4 The “Flagged Email” List

The “Flagged email” list is not a second data store. It is a view over the star flag that already exists on mailbox emails, so no additional plaintext is created for it.

Subjects shown in that list are decrypted exclusively on the client, through the normal email decryption path (Section 7.2): the per-email key is unwrapped in the browser and the subject is decrypted there. The task module never handles an email key and never asks the server for a subject line.

13.5 Transport, Addressing and Supervision Mode

- **Encrypted transport.** All task endpoints run over the encrypted `/e` transport layer (Section 8). Request and response are sealed with the hybrid KEM, so a CDN or any other intermediary sees only `POST /e` with an opaque payload - not which task was created, renamed or completed.
- **UUIDs only.** Tasks, lists and linked emails are addressed exclusively by UUID. Internal database identifiers are never exposed, which removes enumeration as an attack path.
- **Supervision mode.** Where an owner administers an employee or AI-agent mailbox, the task module applies the same key separation as the rest of the vault: task keys are wrapped with the master key of the supervised context, not with the owner’s key. If that context key is unavailable, the operation fails hard. There is deliberately no fallback to the owner key - a silent fallback would write an employee’s tasks under the wrong key. That is a cryptographic downgrade, and a downgrade must fail loudly rather than succeed quietly.

13.6 Input Validation Against Silent Truncation

Every ciphertext blob is length-checked on the server before it is written:

Field	Limit
Task title, list name	2,048 bytes
Note	1 MiB
Subtasks	256 KiB
Metadata	256 KiB
Wrapped key	60 to 1,700 bytes

Oversized values are rejected with an explicit error. This is an integrity control with a confidentiality consequence: depending on its SQL mode, the database would otherwise truncate an oversized value silently. A truncated AES-GCM ciphertext fails authentication, and the content is then permanently unreadable - for the user as well. Refusing the write is the only safe behaviour.

13.7 What the Server Sees - and What It Does Not

The server CAN see	The server CANNOT see
Due dates, reminder times, “My Day” dates	Task titles, notes, subtasks
Important and completed flags, completion time	The name of any task list
Number of tasks and lists, their order and grouping	Recurrence rules and all further metadata
The UUID of a linked email	The content or subject of that email
Ciphertext sizes, creation and modification times	Any key material - task keys, list keys, master key

---

14. Side-Channel Protections

14.1 Bucket Padding

Problem: Encrypted email sizes can leak information. An attacker observing ciphertext lengths could infer content (e.g., a 50-byte email is likely “OK, thanks” while a 500KB email contains attachments).

Solution: All data is padded to fixed-size “buckets” before encryption:

```
Bucket sizes: 256B, 512B, 1KB, 2KB, 4KB, 8KB, 16KB, 32KB,
              64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB

Format: [0xDEAD magic][4-byte length][actual data][random padding to bucket boundary]
```

Example: A 523-byte email is padded to 1,024 bytes. An observer sees only “1KB email” — not the actual 523-byte size.

14.2 Compression-Before-Encryption

Data is compressed with gzip (level 6) **before** encryption. This is the only correct order:

- Compression after encryption would fail (encrypted data has maximum entropy)
- The bucket padding after compression prevents CRIME/BREACH-style attacks that exploit compression ratios

14.3 Threading Privacy

Email threads use SHA-256 hashes of Message-ID headers instead of plaintext identifiers. The server can group related emails by hash equality without knowing the actual message identifiers.

---

15. Key Management & Lifecycle

15.1 Key Hierarchy

```
Vault Master Key (32 bytes, generated once per account)
   |
   |--- Vault Keypair (Hybrid KEM)
   |    |-- X25519 public key (32 bytes) — stored plaintext on server
   |    |-- X25519 private key (32 bytes) — encrypted with master key
   |    |-- ML-KEM-1024 public key (1568 bytes) — stored plaintext on server
   |    |-- ML-KEM-1024 private key (3168 bytes) — encrypted with master key
   |
   |--- Per-Email Ephemeral Keys (32 bytes each)
   |    |-- Wrapped with recipient's Hybrid KEM public keys
   |
   |--- Per-Attachment Ephemeral Keys (32 bytes each)
   |    |-- Wrapped independently per attachment
   |
   |--- Folder Keys (derived via HKDF per folder)
   |    |-- Shared via Hybrid KEM encapsulation per recipient
   |
   |--- Signature Encryption Key (derived from master key)
        |-- Encrypts email signature templates
```

15.2 Key Storage

Key	Storage Location	Protection
Master Key	Nowhere (reconstructed on-demand from Shamir shares)	Shamir 2-of-3
Vault private keys	Server (encrypted)	AES-256-GCM with master key
Vault public keys	Server (plaintext)	Not sensitive — public by definition
Email ephemeral keys	Server (wrapped)	Hybrid KEM encapsulation
OPAQUE records	Server (encrypted at rest)	AES-256-GCM with server key
Encrypted Shamir shares	Server	XOR with password/passkey/recovery derived keys
API transport keys	Server (pre-generated pool)	One-time use, 24h TTL

15.3 Key Fingerprints and Signed Key Transitions

Corrected on 7 October 2026. Earlier versions of this section claimed a server-stored fingerprint with “detection of unauthorized key changes” and “client-side verification of key integrity”. Measured on 4 October 2026: the column `vault_keys.key_fingerprint` existed in the schema

but **no code wrote or read it**, and no client compared a delivered key with anything remembered. What the design now specifies (contract `docs/VAULT_SCHLUESSELWECHSEL_SIGNATUR.md`, v2; server implementation in review, **not yet deployed** as of this edit):

Fingerprint (client-side, v2). `SHA-256("aionda-postfachschluessel-fingerabdruck-v2" ‖ len ‖ X25519 ‖ len ‖ ML-KEM-1024 ‖ len ‖ ML-DSA-65 signing key)`, where a mailbox without signing key contributes a zero length. Shown as the first 128 bits in groups of four hex characters (`1A2B-3C4D-…`). Two people compare it out of band. Because the signing key is part of it, a substituted signing key is visible in the comparison; the server never computes it for trust decisions.

Signed key transitions (server). Each vault gets an ML-DSA-65 (FIPS 204) signing key; its seed is stored AES-256-GCM-encrypted under the master key with purpose- and account-bound additional data. Installing or replacing that key through its dedicated command requires a proof of the master key: the server encapsulates a challenge to the vault's own hybrid-KEM public keys, and only a client that can open it signs the installation record. The legacy vault re-setup path still accepts a whole new key set without that proof; a pinning contact then sees a visible stop, not a silent change (named as a boundary in the contract). Once a signing key is installed and verification is available, the server rejects unsigned rotations. When the KEM key pair is rotated, the client signs a canonical transition record (old fingerprint, new X25519 + ML-KEM public keys, new signing key, login, timestamp) with the **old** signing key. The server verifies before storing anything (rejects on invalid; rejects — rather than silently accepting — when verification is unavailable) and delivers the chain of transitions together with the public keys. A rotation without a signature is still accepted, but the chain entry is visibly unsigned.

What a pinning client is specified to do. Remember a contact's KEM keys and signing key together on first contact (trust on first use); accept a changed key only if a chain of valid signatures leads from the remembered set to the delivered one; stop and ask the user to compare fingerprints otherwise — including when a signing key appears for the first time for a contact that was remembered without one. The Apple client pins KEM keys since 4 October 2026 (every change stops); chain verification and the v2 fingerprint are a follow-up ticket there (`4a91f154`), the web client does not pin yet (`b5243fcc`).

16. Recovery Mechanism

16.1 Recovery Key (BIP39 Mnemonic)

During vault setup, the user is presented with a **24-word recovery phrase** generated from 256 bits of entropy, encoded using the BIP39 standard:

```
Example: apple river mountain sunset golden bridge falcon ocean
         crystal thunder meadow silver dolphin forest marble castle
         velvet compass harbor window ancient pepper rocket shield
```

16.2 Recovery Key Derivation

```
1. Generate: 256 bits random entropy
2. Encode:   BIP39 mnemonic (24 words, 11 bits per word)
3. Derive:   verificationKey = HKDF-SHA3-256(
                 entropy,
                 salt = accountId,
                 info = "trashmail-recovery-verify"
             )
```

```
4. Hash:     verificationHash = SHA3-256(verificationKey)
5. Store:    Server stores ONLY verificationHash (32 bytes)
```

16.3 What the Server Stores

The server stores **only the SHA3-256 hash** of a derived verification key. It does not store:

- The recovery words
- The entropy
- The verification key itself

16.3.4 Recovery Flow

1. User enters 24-word recovery phrase
2. Client derives entropy -> HKDF-SHA3-256 -> SHA3-256 -> verification hash
3. Client sends verification hash to server (never the plaintext key)
4. Server compares with stored hash
5. If match: All 2FA methods are disabled, user sets up fresh authentication
6. Recovery key is revoked after single use

16.3.5 Rate Limiting

- Maximum 3 verification attempts per hour
- 60-minute lockout after exceeding limit
- One-time use: recovery key is permanently revoked after successful use

16.3.6 No Password Recovery

There is no password reset via email. If a user loses their password AND all other authentication factors (passkey + recovery key), their data is permanently inaccessible. This is the fundamental proof that zero-knowledge works — if the service could recover user data, it could also read it.

---

17. Passwordless Authentication (Passkeys)

17.1 WebAuthn PRF Extension

Aionda Mail supports FIDO2 passkeys (hardware security keys, biometric authenticators) for passwordless login and vault unlock.

The **WebAuthn PRF (Pseudo-Random Function) extension** provides a deterministic 32-byte output bound to the specific passkey and credential. This output is used to protect Shamir Share 2.

17.2 How It Works

```
1. Registration:
   - User creates passkey via navigator.credentials.create()
   - PRF extension generates hardware-bound output
   - Output XOR'd with Shamir Share 2 -> encrypted share stored on server

2. Authentication:
   - User authenticates with passkey (biometric/PIN)
```

```
   - PRF extension reproduces same 32-byte output
   - Output XOR'd with encrypted share -> Shamir Share 2 recovered
   - Combined with Share 1 (password) -> Master Key reconstructed

3. Vault Unlock (passwordless):
   - If both passkey (Share 2) and password (Share 1) available -> immediate unlock
   - Password verified via OPAQUE (separate from passkey auth)
```

17.3 Multiple Passkeys

Users can register multiple passkeys (e.g., MacBook Touch ID, iPhone Face ID, YubiKey). Each passkey independently protects its own copy of Share 2. Any single passkey combined with the password is sufficient to unlock the vault.

---

18. Guardian: MITM Protection & Response Signing

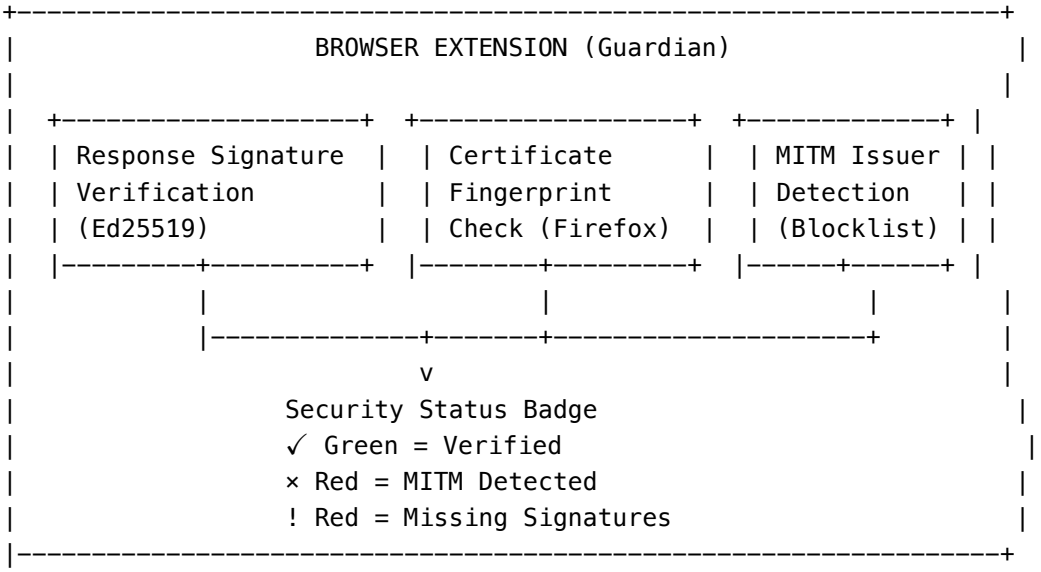
18.1 The Problem with Web Applications

Every web application has an inherent trust problem: the browser downloads JavaScript from the server on every visit. A man-in-the-middle (MITM) attacker — whether a compromised CDN, corporate proxy, or rogue ISP — could theoretically inject modified code that exfiltrates encryption keys.

Aionda Mail addresses this with the **Guardian module**, a browser extension component (available for Chrome and Firefox) that independently verifies server integrity.

18.2 Architecture Overview

The Guardian module operates inside the browser extension's Service Worker — completely independent from the web application's JavaScript. It performs three types of verification:



18.3 Response Signature Verification (Ed25519)

Every API response from Aionda Mail's server is cryptographically signed using **Ed25519** (Edwards-curve Digital Signature Algorithm).

Signing process (server-side):

```
1. Server generates API response body (JSON)
2. Construct signing input: responseBody + "|" + unixTimestamp
3. Sign with Ed25519 private key -> 64-byte signature
4. Attach HTTP headers:
   X-Aionda-Signature: <base64(signature)>
   X-Aionda-Timestamp: <unix_timestamp>
   X-Aionda-Key-Id: <key_identifier>
```

Verification process (browser extension):

```
1. Extract signature, timestamp, and key ID from HTTP headers
2. Look up Ed25519 public key by key ID (bundled in extension)
3. Verify key has not expired (valid_from / valid_until)
4. Check timestamp freshness: |now - timestamp| ≤ 300 seconds
5. Reconstruct signing input: responseBody + "|" + timestamp
6. crypto.subtle.verify("Ed25519", publicKey, signature, data)
7. If invalid -> MITM alert, red badge
```

Key properties:

- **Replay protection:** 5-minute timestamp window prevents replaying old responses
- **Tamper detection:** Any modification to the response body invalidates the signature
- **Key isolation:** Public keys are bundled inside the extension (not downloaded from the server)
- **Environment separation:** Dev keys (`dev-2026-01`) cannot be used on production URLs and vice versa

18.4 Ed25519 Public Key Management

Public keys are shipped with the browser extension in `public_key.json`:

```
{
    "keys": {
        "prod-2026-01": {
            "algorithm": "Ed25519",
            "public_key": "<base64 SPKI DER>",
            "valid_from": "2026-01-13T00:00:00Z",
            "valid_until": "2027-01-13T00:00:00Z"
        }
    }
}
```

- **Key format:** SPKI DER (Subject Public Key Info, Distinguished Encoding Rules)
- **Key size:** 32 bytes (256-bit Ed25519 public key)
- **Signature size:** 64 bytes (fixed)
- **Rotation:** New keys are added before old keys expire; extension updates deliver new keys
- **No server trust:** Keys are embedded in the extension binary, not fetched from the server

18.5 TLS Certificate Verification (Firefox)

On Firefox, the Guardian module performs additional TLS certificate verification using the `browser.webRequest.getSecurityInfo()` API (not available in Chrome due to Manifest V3 limitations).

Verification flow:

```
1. Browser extension intercepts HTTPS response
2. Extract TLS certificate chain from browser's security info:
   - Leaf certificate fingerprint (SHA-256)
   - Issuer Distinguished Name (O=, CN=)
   - Subject (CN=)

3. Check against known MITM issuers (hardcoded blocklist):
   ZScaler, Netskope, Fortinet, Palo Alto, Blue Coat,
   Check Point, Barracuda, Sophos, WatchGuard, Cisco Umbrella
   -> If match: MITM detected, show warning

4. Check against trusted issuers:
   Google Trust Services, Cloudflare, Let's Encrypt,
   DigiCert, Sectigo
   -> If match AND subject matches expected domain: OK

5. If unknown issuer: Fetch server's own certificate fingerprint
   - Server connects to itself via external routing (prevents spoofing)
   - Response is Ed25519 signed (prevents MITM from lying about cert)
   - Compare issuer organization with browser's certificate issuer
   -> If mismatch: MITM suspected, show warning
```

Why issuer-based validation instead of pinning? CloudFlare (used as CDN) rotates leaf certificates across edge servers. Traditional certificate pinning (matching exact fingerprints) would cause false positives. Issuer-based validation is more robust: the issuing CA is stable even when leaf certificates change.

18.6 Self-Certificate Fetching (Anti-Spoofing)

The server's certificate endpoint uses a clever anti-spoofing technique:

```
Server connects to cert.trashmail.com (or cert-subdomain.domain)
with SNI = mail.aionda.com

-> Forces external routing through CloudFlare
-> Receives the actual certificate that users see
-> Prevents localhost spoofing
-> Response signed with Ed25519 to prevent tampering
```

The server essentially asks "what certificate does the outside world see for my domain?" — and signs the answer so the extension can trust it.

18.7 Security Status Indicators

The extension displays a badge on the browser toolbar:

Badge	Color	Meaning
✓	Green	All responses verified — signatures valid
!	Orange	Using deprecated signing key (rotation pending)
×	Red	MITM detected — signature verification failed
!	Red	Missing signatures — responses not signed
[Shield]	Blue	Protected mode — no verification performed yet

18.8 Threat Coverage

Attack	Detection Method	Browser
Corporate MITM proxy (ZScaler, Fortinet)	Certificate issuer blocklist	Firefox
Modified API responses	Ed25519 signature verification	Chrome + Firefox
Replay attacks	5-minute timestamp window	Chrome + Firefox
CDN compromise (CloudFlare)	Response signature mismatch	Chrome + Firefox
Certificate substitution	Issuer comparison + server self-check	Firefox
Dev/prod key confusion	Environment-bound key IDs	Chrome + Firefox

18.9 Limitations

- **Chrome Manifest V3:** Cannot inspect TLS certificates — only response signature verification is available
- **Extension required:** Users without the extension do not benefit from Guardian protections
- **Ed25519 is not post-quantum:** Signature verification uses classical cryptography. A sufficiently powerful quantum computer could theoretically forge Ed25519 signatures.

---

19. Enterprise Email Archive (Blockchain)

19.1 Overview

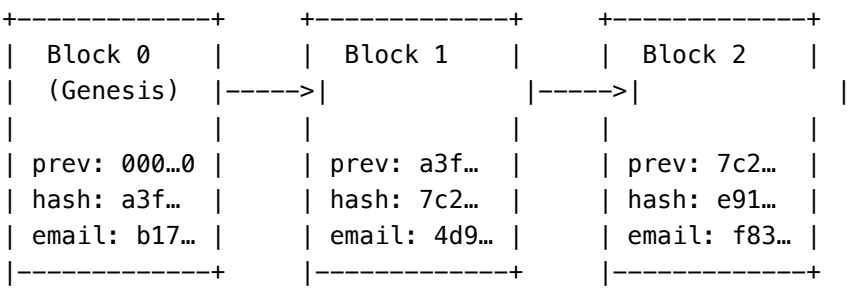
Aionda Mail's Enterprise plan includes a **GoBD-compliant email archive** secured by a cryptographic hash chain (blockchain). Every archived email becomes an immutable block in a per-company chain. Any tampering — modification, deletion, or insertion of blocks — is cryptographically detectable.

The archive combines two independent security layers:

1. **Hash chain (SHA3-256):** Guarantees integrity and immutability — proves no email was altered or removed after archival
2. **Hybrid KEM encryption (CAK):** Guarantees confidentiality — the server cannot read archived email content

19.2 Hash Chain Architecture

Each archived email becomes a block in a sequential, tamper-evident chain:



Block hash calculation:

```
block_hash = SHA3-256(
    prev_block_hash  || "|" ||
    timestamp        || "|" ||
    email_hash       || "|" ||
    direction        || "|" ||
    sender_domain    || "|" ||
    recipient_domain
)
```

Properties:

- **Hash algorithm:** SHA3-256 (NIST FIPS 202)
- **Genesis block:** `prev_block_hash` = 64 zeros, `block_number` = 0
- **Sequential numbering:** Enforced by database `UNIQUE KEY (company_uuid, block_number)`
- **One chain per company:** Complete isolation between enterprises
- **Email hash:** `SHA3-256(sender || recipient || timestamp || size)` — integrity proof of the original email data

19.3 Tamper Detection

The chain verification algorithm detects any form of tampering:

```
For each block (ordered by block_number ASC):

  1. Verify link:    block.prev_block_hash == expected_prev_hash
  2. Recalculate:    expected = SHA3-256(prev_hash | timestamp | email_hash | ...)
  3. Verify content: block.block_hash == expected
  4. Advance:        expected_prev_hash = block.block_hash

If ANY check fails -> chain is broken at block N
```

Tampering Attempt	Detection
Modify email content	`email_hash` changes -> `block_hash` recalculation fails
Modify metadata (sender, domain, timestamp)	Included in hash input -> `block_hash` mismatch
Delete a block	Next block's `prev_block_hash` becomes orphaned

Tampering Attempt	Detection
Insert a block	Breaks sequential `block_number` + `prev_block_hash` chain
Reorder blocks	`UNIQUE KEY` constraint + sequential verification prevents this
Replace entire chain	Genesis block hash would differ from any external backup

Verification result reports the exact block number where tampering was detected, with expected vs. actual hash values for forensic analysis.

19.4 Company Archive Key (CAK) — Zero-Knowledge Encryption

Archive contents are encrypted end-to-end using a **Company Archive Key** — a Hybrid KEM keypair (X25519 + ML-KEM-1024) generated client-side by the company owner.

```
Company Owner's Browser                          Server
──────────────────────                           ──────

1. Generate Hybrid KEM keypair (client-side):
   X25519 keypair (32 + 32 bytes)
   ML-KEM-1024 keypair (1568 + 3168 bytes)

2. Derive wrapping key from password:
   wrappingKey = HKDF-SHA256(
     password,
     salt = "trashmail-archive-{account_id}",
     info = "trashmail-archive-key-wrap",
     length = 32
   )

3. Wrap private keys:
   AES-256-GCM(x25519_priv || mlkem_priv, wrappingKey)

4. Send to server:                          ->   Store:
   • Public keys (plaintext)                     archive_x25519_pub
   • Wrapped private keys (encrypted)            archive_mlkem_pub
                                                 wrapped_archive_key
```

Key distribution to other employees (Admin, Compliance Officer):

1. Owner decrypts CAK private keys with their password
2. Owner re-wraps private keys with target employee's password-derived key
3. Server stores the re-wrapped copy on the employee's record
4. Each authorized employee has their own independently wrapped copy

The server never sees the CAK private keys in plaintext.

19.5 What Gets Encrypted

When an email is archived, two layers of encryption are applied:

Encrypted metadata (AES-256-GCM with Hybrid KEM):

```
{
  "d": "INBOUND",
  "s": "user@example.com",
  "r": "admin@company.de",
  "sd": "example.com",
  "rd": "company.de",
  "sz": 45000,
  "ts": "2026-02-27T10:30:00Z",
  "ha": true,
  "ac": 3,
  "en": "John Doe"
}
```

Encrypted email content (separate Hybrid KEM key wrapping):

```
{
  "subject": "Meeting notes",
  "body": "<html>...</html>",
  "from": "sender@domain.com",
  "to": "recipient@company.de"
}
```

Zero-knowledge enforcement: After encryption, the plaintext metadata fields in the database (sender_address, recipient_address, domains) are replaced with their SHA3-256 hashes. The server stores only hashes — the original values exist only inside the encrypted blobs.

19.6 Audit Trail

Every action on the archive is logged in an **independent audit chain** (also hash-chained with SHA3-256):

Action	When Logged
`EMAIL_RECEIVED` / `EMAIL_SENT` / `DRAFT_ARCHIVED`	Email archived
`VIEW_EMAIL` / `VIEW_ATTACHMENT`	Employee reads archived email
`SEARCH_ARCHIVE`	Search performed
`EXPORT_EMAIL` / `EXPORT_REPORT`	Data exported
`VERIFY_CHAIN` / `CHAIN_VERIFIED_OK` / `CHAIN_VERIFIED_BROKEN`	Integrity check
`LEGAL_HOLD_SET` / `LEGAL_HOLD_RELEASED`	Legal hold toggled
`ARCHIVE_DECRYPT`	CAK used to decrypt content
`ADMIN_ACCESS`	Administrative action

Each audit entry records: actor (UUID + role), IP address, session ID, target email hash, and whether the chain was valid at the time of access.

19.7 Legal Hold & Retention

- **Retention period:** Configurable per company (default: 10 years), calculated per email as `archived_at + retention_years`
- **Legal hold:** Individual emails can be placed under legal hold, preventing deletion until released. Includes reason, actor, and timestamp.
- **GoBD compliance:** The combination of immutable hash chain, complete audit trail, configurable retention, and legal hold satisfies the requirements of the German GoBD (Grundsätze zur ordnungsmäßigen Führung und Aufbewahrung von Büchern, Aufzeichnungen und Unterlagen in elektronischer Form sowie zum Datenzugriff).

19.8 Forensic Export

Authorized users (Owner, Admin) can export the complete chain state for independent verification:

- Full chain data with all block hashes
- Verification result (valid/broken, broken block number if applicable)
- Expected vs. actual hash values for forensic analysis
- Last 50 audit log entries
- JSON format for external re-verification with any SHA3-256 implementation

---

20. What the Server Sees — and What It Does Not

This section explicitly documents the zero-knowledge boundary.

20.1 The Server CAN See

Data	Why Visible	Mitigation
IP address	TCP/IP requirement	Use VPN/Tor if desired
Timestamps	Email reception time	Inherent to email protocol
Encrypted email blobs	Stored for retrieval	AES-256-GCM encrypted, key unknown to server
Padded ciphertext sizes	Storage requirement	Bucket padding hides actual sizes
Recipient DEA address	Routing requirement	DEA is disposable, not the real address
Account existence	Authentication flow	User enumeration protection deployed
Public keys	Required for encryption by server	Public by definition, not sensitive
Encrypted Shamir shares	Storage for user	XOR'd with keys server doesn't know
OPAQUE records	Authentication protocol	Not password hashes, encrypted at rest

20.2 The Server CANNOT See

Data	Why Invisible
Email content (subject, body, headers)	Encrypted with ephemeral keys wrapped via Hybrid KEM
User password	OPAQUE — password never transmitted
Master key	Reconstructed only in browser from Shamir shares
Vault private keys	Encrypted with master key before storage
Email ephemeral keys	Wrapped with Hybrid KEM, server lacks private keys
Recovery key / mnemonic	Only SHA3-256 hash of derived key stored
Passkey PRF outputs	Hardware-bound, never leave authenticator
Folder names	Encrypted with folder-specific keys
Email signatures	Encrypted with master key
API request content	Encrypted via /e transport layer
API response content	Encrypted before transmission

20.3 Cryptographic Guarantee

Even with full access to:

- The complete database
- All network traffic
- The server's source code and configuration
- All OPAQUE records and server keys

…an attacker **cannot** decrypt a single email without the user's password (or passkey + recovery key). This is not a policy — it is a mathematical impossibility enforced by the cryptographic design.

---

21. Algorithm Reference

21.1 Complete Algorithm Table

Component	Algorithm	Parameters	Standard
Password authentication	OPAQUE	RFC 9807, aPAKE	RFC 9807
Password key derivation	PBKDF2-SHA256	600,000 iterations, 32B salt, 32B output	NIST SP 800-132
Vault encryption	AES-256-GCM	256-bit key, 96-bit nonce, 128-bit tag	NIST SP 800-38D
Classical key exchange	X25519	Curve25519, 256-bit	RFC 7748
Post-quantum KEM	ML-KEM-1024	Kyber-1024, NIST Level 5	NIST FIPS 203
Hybrid key derivation	HKDF-SHA256	64B IKM, info="trashmail-hybrid-kem-v1"	RFC 5869
Secret sharing	Shamir SSS	k=2, n=3, GF(2^8)	Shamir (1979)

Component	Algorithm	Parameters	Standard
Recovery key encoding	BIP39	256-bit entropy, 24 words	BIP-0039
Recovery key derivation	HKDF-SHA3-256	Account-bound salt	NIST FIPS 202
Recovery key verification	SHA3-256	32-byte output	NIST FIPS 202
OPAQUE record encryption	AES-256-GCM	Server-side at-rest encryption	NIST SP 800-38D
Passkey vault unlock	WebAuthn PRF	HMAC-based, hardware-bound	WebAuthn Level 2
Compression	gzip	Level 6	RFC 1952
Bucket padding	Custom	17 sizes (256B–16MB), 0xDEAD magic	—
Response signing	Ed25519	256-bit key, 512-bit signature	RFC 8032
Archive hash chain	SHA3-256	Per-block hash, sequential linking	NIST FIPS 202
Archive key wrapping (CAK)	HKDF-SHA256 + AES-256-GCM	Password-derived wrapping key	RFC 5869 / NIST SP 800-38D
Certificate verification	SHA-256	TLS cert fingerprint comparison	—
Email threading	SHA-256	Hash of Message-ID	NIST FIPS 180-4
Video-call group keys	MLS + X-Wing	Suite `AIONDA_PQ_CALL_V1` (0xFF01), ML-KEM-768 + X25519	RFC 9420 / draft-connolly-cfrg-xwing-kem
Video-call signatures	ML-DSA-65	Dilithium, NIST Level 3	NIST FIPS 204
Video-call media	SFrame (AES-256-GCM)	Per-frame, key from MLS exporter	RFC 9605

21.2 Security Levels

Algorithm	Classical Security	Post-Quantum Security
X25519	128-bit	Broken by Shor's algorithm
ML-KEM-1024	256-bit equivalent	NIST Level 5 (≈AES-256)
AES-256-GCM	256-bit	128-bit (Grover's algorithm)
SHA-256	256-bit	128-bit (Grover's algorithm)
SHA3-256	256-bit	128-bit (Grover's algorithm)
Hybrid KEM (combined)	128-bit (X25519 bound)	Level 5 (ML-KEM bound)

---

22. Comparison with Other Providers

Feature	Aionda Mail	Tuta Mail	Proton Mail
Country	Germany (Stuttgart)	Germany (Hannover)	Switzerland
Zero-Knowledge	Yes (OPAQUE + client-side crypto)	Yes	Yes
Post-Quantum	Yes (ML-KEM-1024 + X25519 Hybrid)	Yes (Kyber-based)	In development
Password protocol	OPAQUE (RFC 9807) — password never leaves browser	bcrypt (password sent to server over TLS)	SRP-based
Subject encrypted	Yes	Yes	No
Headers encrypted	Yes	Partial	No
Contact names encrypted	Yes (in vault)	Yes	No
Disposable email addresses	Yes (core feature, unlimited for Plus)	No	Yes (via SimpleLogin)
Browser addon	Yes (Chrome + Firefox)	No	Via SimpleLogin
Folder sharing	Yes (Hybrid KEM per recipient)	Limited	Yes
Open source client	No	Yes	Yes
Security audit	Planned	Yes	Yes
Password recovery	No (by design)	No (by design)	No (by design)
Passkey support	Yes (FIDO2 + PRF)	Yes	Yes
PGP support	Yes (incoming + outgoing)	No (own protocol)	Yes (OpenPGP)
GoBD-compliant email archive	Yes (SHA3-256 hash chain + Hybrid KEM)	No	No
MITM detection (browser ext.)	Yes (Ed25519 signatures + TLS check)	No	No
Perfect Forward Secrecy (API)	Yes (per-request ephemeral keys)	Unknown	Unknown
Email size obfuscation	Yes (bucket padding)	Unknown	No

23. Limitations & Honest Boundaries

23.1 Web Application Trust Model

Aionda Mail is a web application. On every page load, the browser downloads JavaScript from our servers. A sophisticated attacker who compromises our servers could theoretically serve modified JavaScript that exfiltrates keys.

Current mitigations:

- Subresource Integrity (SRI) hashes on all script tags
- Content Security Policy (CSP) headers restrict script sources
- All critical cryptographic code is included in the main application bundle
- **Guardian browser extension** (Section 18): Ed25519 signature verification on all API responses detects server-side tampering; TLS certificate verification (Firefox) detects MITM proxies

Planned mitigations:

- Service Worker caching for offline operation (reduces trust-on-load frequency)

23.2 Metadata Visibility

While email content is fully encrypted, certain metadata is visible to the server:

- When emails were received (timestamps)
- Which DEA address received the email
- Approximate email size (within bucket boundaries)
- Account activity patterns

23.3 Email Processing Log

For diagnostic purposes, Aionda Mail includes an optional **email processing log** that can temporarily store the raw content of incoming emails. This feature is configurable per disposable email address (DEA) and can be enabled or disabled in the DEA settings ("Log email content").

When enabled (opt-in per DEA):

- The full raw SMTP message (headers + body) is stored in plaintext on the server
- Automatic deletion after a short retention period (less than 7 days)
- Accessible only to the account owner via authenticated API
- Purpose: troubleshooting delivery issues, verifying forwarding, reviewing spam filtering decisions

When disabled:

- No email content is stored in the processing log
- Only metadata is logged (sender address, timestamp, delivery status)
- Vault encryption remains the sole storage mechanism

Important: This processing log is independent of the encrypted vault. Emails stored in the vault are always encrypted with Hybrid KEM regardless of the log setting. The processing log exists as a legacy feature from the email forwarding system and provides operational transparency. Users who require strict zero-knowledge storage for all emails should disable this option.

23.4 External Email Security

Emails sent to or received from non-Aionda addresses travel through the standard email infrastructure (SMTP). While stored encrypted in the vault, the email content was visible during transit unless PGP encryption was used.

23.5 No Key Escrow

There is no master key, backdoor, or recovery mechanism available to Aionda GmbH. If a user loses their password and all recovery methods, their data is permanently lost. This is an intentional design decision that proves the integrity of the zero-knowledge model.

---

24. Roadmap

Milestone	Status	Target
Zero-Knowledge Architecture	Completed	—
Post-Quantum Hybrid KEM (ML-KEM-1024)	Completed	—
OPAQUE Authentication (RFC 9807)	Completed	—
Shamir Secret Sharing (2-of-3)	Completed	—
Encrypted API Transport Layer	Completed	—
Passkey/WebAuthn PRF Support	Completed	—
End-to-End Encrypted Calendar	Completed	—
End-to-End Encrypted Tasks	Completed	—
GoBD-Compliant Email Archive (Blockchain)	Completed	—
Guardian MITM Protection (Ed25519)	Completed	—
TLS Certificate Verification (Firefox)	Completed	—

---

25. QR-code device sign-in

25.1 User flow and lifetime

An already authenticated, unlocked client displays **Sign in a new device (QR code)**. The new device scans the invitation. Both devices display an eight-digit comparison code; the user must compare it and explicitly approve on the existing device before access is transferred. Web, iOS and macOS support displaying an invitation; the receiving client must support the pairing protocol and its trusted transport profile.

The invitation expires after **15 minutes (900 seconds)**. This is a fixed product choice, not an industry standard. Reading or retrying does not extend it. A completed invitation cannot provision another device. Cancellation or expiry invalidates an unfinished transfer; it does not log out a device that has already completed sign-in.

25.2 QR contents and encrypted transfer

The `AIONDA_DEVICE_PAIRING_V2` QR contains a pairing UUID, username, service origin, creation and expiry times, a SHA-256 fingerprint binding the ephemeral source public keys and invitation metadata, and the transport profile and authentication origin. It contains **no account password, recovery words, access token or vault master key**. The username is readable: do not publish or forward the QR image.

The recipient verifies the QR fingerprint against the public keys supplied by the relay. X25519 + ML-KEM-1024 establishes a shared secret. HKDF-SHA-256 derives role-separated keys bound to the protocol transcript; AES-256-GCM authenticates and encrypts the claim, readiness, approval and receipt messages. Account, environment, recipient and lifetime are bound to the exchange. Invalid fingerprints, transport bindings or authenticated messages cause an error rather than a plaintext fallback.

After confirmation, the source sends a **new device access token and the vault master key** inside the encrypted approval bundle, not the existing account password or recovery phrase. The token is registered using OPAQUE. The server relays ciphertext and manages lifecycle and registration records; the pairing bundle is decrypted by the recipient. The recipient receives real mailbox access, so approval is a security-sensitive action. Revoking its token blocks subsequent authentication but cannot erase data or keys already obtained by that device.

25.3 Browser risks and safeguards

Browser-originated pairing displays a localized risk warning and requires recent account authentication: the server accepts creation only within 120 seconds of successful account-bound OPAQUE or passkey verification. The mailbox must be unlocked. The browser source is bound to its session; the receiving client uses a fixed trusted authentication profile rather than an arbitrary endpoint from a QR code.

Prefer the native iOS or macOS app as the source. End-to-end encryption protects the exchange only while both endpoints execute trustworthy code. A TLS-inspecting proxy such as ZScaler, a malicious extension, or compromised web delivery may alter browser code and read secrets before encryption. HTTPS, the QR itself, and a matching comparison code do not make a compromised endpoint safe. Guardian can provide additional checks when installed and enabled, but is not a prerequisite and is not assumed in this security claim. Never approve an unexpected request; compare the numbers on the two devices you control.

Document History

Version	Date	Changes
1.0	March 2026	Initial publication
1.1	April 2026	New chapter 10: Vault Drive (per-file-key architecture, sharing via key rewrap)
1.2	April 2026	Section 10.11: External Sharing — public share links with hybrid KEM envelopes, unlock_token flow, Argon2id password mode, URL fragment for link-only, tamper-evident audit chain (EXT_SHARE_*), and user-chosen link distribution

Version	Date	Changes
1.3	May 2026	New chapter 11: Aionda Chat — post-quantum E2EE real-time messaging via AAR (Aionda Async Ratchet), a hybrid X25519 + ML-KEM-1024 variant of Signal X3DH + Double Ratchet. Per-message forward secrecy, post-compromise security, and integration with the Enterprise audit chain. Subsequent chapters renumbered 12-22. PT-BR and PT-PT translations added. Fixed double-numbering bug in PDF rendering (LaTeX auto-numbering disabled; chapter numbers now sourced from headings only).
1.4	June 2026	New chapter 12: Aionda Video Calls — post-quantum E2EE group conferencing. MLS (RFC 9420) group key agreement with the `AIONDA_PQ_CALL_V1` cipher suite (X-Wing = ML-KEM-768 + X25519, ML-DSA-65 signatures), SFrame (RFC 9605) per-frame AES-256-GCM media encryption keyed from the MLS exporter, SAS binding, and a zero-knowledge SFU relay. Subsequent chapters renumbered 13-23.

Version	Date	Changes
1.5	July 2026	New chapter 13: Aionda Tasks - end-to-end encrypted task management. Per-task and per-list AES-256-GCM keys wrapped with the vault master key (HKDF-SHA-256 with module-specific domain separation), encrypted titles, notes, subtasks, list names and recurrence rules, an honest account of the deliberately plaintext scheduling metadata, the flagged-email list built on the existing mail star flag, supervision-mode key separation, and server-side length validation against silent ciphertext truncation. Subsequent chapters renumbered 14-24.
1.6	October 2026	Section 15.3 corrected to what is measured (the server-stored fingerprint and “detection of unauthorized key changes” did not exist); design for signed key transitions described (per-vault ML-DSA-65 signing key, fingerprint v2 covering KEM and signing key, canonical transition record signed by the previous key, master-key proof for installing the signing key, server-side verification before storage, chain delivered with recipient keys) — server implementation in review, not deployed at the time of this edit. Section 2.2: key substitution by the server named as a boundary, including steering to an older signed key set.
1.7	13 October 2026	25: QR-code device sign-in (Web, iOS, macOS).
1.8	13 October 2026	3.2: Native iOS/macOS clients, shared Swift/SwiftUI architecture and embedded WebKit trust boundaries.

Contact

Aionda GmbH Stephan Ferraro Stuttgart, Germany

Email: contact-46epp9ba@contact.aionda.com Web: https://mail.aionda.com

This document describes the security architecture of Aionda Mail as of 13 October 2026. Cryptographic systems evolve — this document will be updated as the architecture changes.