

Alien Lights Identity

A post-quantum, federated, publicly auditable identity system — built as an OAuth2/OIDC provider, not a replacement for one.

TECHNICAL WHITEPAPER · V3.0 · NOVEMBER 2026

This paper states one narrow, falsifiable claim, documents its own limitations in the same breath, and reports exactly what has and has not been validated so far: hybrid post-quantum signing, post-quantum key *custody* (not only signatures), per-identity Durable Object isolation, and a public transparency log with RFC 6962 Merkle inclusion proofs, all running in production on Cloudflare's edge today — not an unqualified claim to be the best identity system in the world. See §9 (threat model), §10 (comparison), and §11 (validation evidence).

Abstract

Alien Lights Identity gives every user their own **Durable Object** — not a row in a shared table — holding a UUID bound to a **hybrid cryptographic keypair**: a NIST-standardized post-quantum signature (ML-DSA-65, FIPS 204) running alongside a classical signature (Ed25519) over the same bytes, so that identity assertions remain verifiable whether or not a cryptographically relevant quantum computer arrives before the migration away from classical-only signatures is complete. Each identity's key rotates monthly, anchored to that identity's own signup date rather than a shared calendar epoch, and that identity's own object self-schedules its own rotation via a platform alarm, eliminating both a system-wide rotation sweep and the shared-table blast radius a sweep implies. Every private key in the system — identity keys, the identity provider's own signing key, and linked OAuth provider credentials — is wrapped at rest with a fresh **ML-KEM-768 + X25519** hybrid key encapsulation against a rotatable root key, rather than one static secret, so the custody chain itself, not only the signatures made with it, resists a "harvest now, decrypt later" adversary. Every key-affecting event — registration, rotation, revocation — is appended to a public, hash-chained, dual-signed **transparency log** extended with a full RFC 6962 **Merkle tree**, so any third party can fetch and verify the whole log, or confirm a single entry is included in a signed root without downloading the rest of it — the same inclusion-proof guarantee Certificate Transparency gives TLS certificates. The system interoperates with the existing OAuth2/OIDC ecosystem in both directions: it can link a user's identity to any standards-compliant external provider, and it can itself act as a standards-compliant OIDC provider for third-party applications, including token rotation on refresh. Verification is designed to keep working with no connectivity to Alien Lights at all — a cached public key is enough — and the live API degrades gracefully, rather than failing outright, through a brief database outage.

This revision is substantially longer than the paper's prior editions, and that length is deliberate: beyond describing the design, it documents exactly where each piece of state physically lives and why (§2), gives the full formal construction of the Merkle inclusion-proof math rather than just its guarantee (§6), adds a dedicated validation section reporting precisely what has been tested and in which environment — local development versus live production — rather than one blended claim (§11), and appends a complete API and data-model reference (Appendices A–B) so every claim in the prose is checkable against the actual route table and schema. Consistent with the project's own stated engineering values, it also states plainly what has *not* yet been done: independent security audit, hardware-backed key custody, and production-scale validation under real traffic.

1. Motivation

Two independent problems motivate this design. The first is cryptographic: a sufficiently capable quantum computer would break the discrete-logarithm and elliptic-curve assumptions underlying almost every identity token issued on the internet today (RSA, ECDSA, EdDSA). Traffic and tokens signed or encrypted today can be recorded now and attacked later — "harvest now, decrypt later" — which means the migration to post-quantum cryptography has to start before quantum computers are actually a threat, not after. NIST finalized its first post-quantum signature and key-encapsulation standards (FIPS 203, 204, 205) in August 2024; adoption in consumer identity systems has lagged well behind adoption in transport-layer security, where post-quantum key exchange is already common.

The second problem is structural: identity on the internet today is almost entirely *provider-attested, not user-verifiable*. A user has no independent way to confirm that the key material their identity provider is signing with today is the same key material it was signing with last month, or that a key rotation actually happened when the provider says it did, short of trusting the provider's own internal controls. Certificate Transparency solved an analogous problem for TLS certificates in 2013 by making every issued certificate publicly logged and auditable, with a compact proof that any one certificate is really in the log. No equivalent, publicly auditable log — let alone one with per-entry inclusion proofs — exists today for consumer identity provider key material.

A third, quieter problem motivates how the system is built, not just what it protects: a provider that keeps every user's state in one shared table has made every user's data available to whatever can compromise that table. Durable Objects make it practical to give each identity its own isolated storage and its own self-scheduled lifecycle instead, without reintroducing the operational cost of running one process per user by hand.

Alien Lights Identity addresses all three: hybrid post-quantum signatures and key custody for the first, a public Merkle-tree transparency log for the second, and per-identity Durable Object isolation for the third — while remaining a normal, interoperable participant in the existing OAuth2/OIDC ecosystem rather than asking the rest of the internet to adopt something new.

2. System Architecture & Data Model

Every registered identity consists of a UUID, a server-held `identity_secret` (stored only as a SHA-256 hash, shown to the user exactly once), and a current hybrid keypair. The identity can sign arbitrary messages with that keypair, link to any number of external OAuth2/OIDC providers, and present itself to third-party applications through Alien Lights' own OIDC endpoints. Figure 1 shows how these pieces relate.

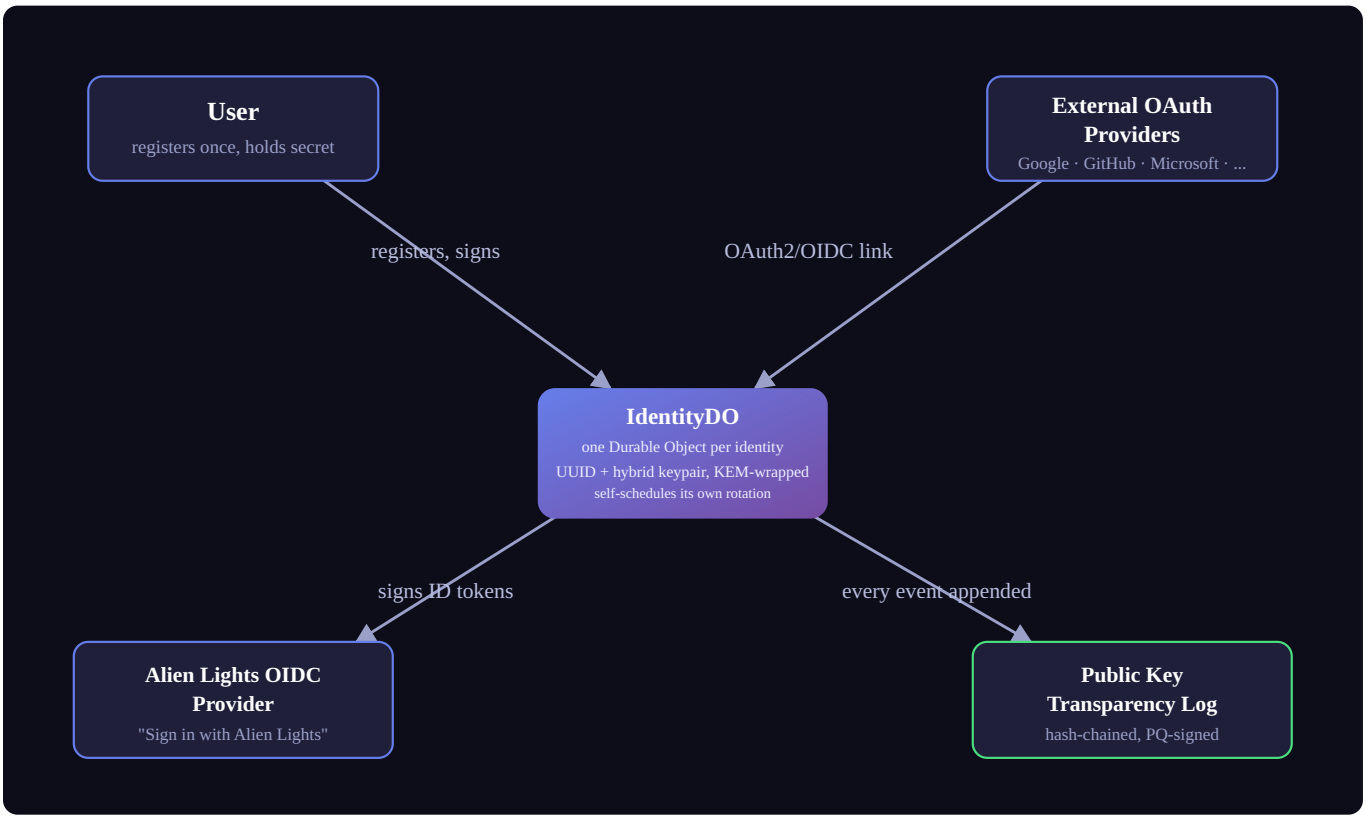


Figure 1 — Each identity's own Durable Object is the hub: populated by registration, enriched by linked external accounts, and the source of truth both for tokens issued under "Sign in with Alien Lights" and for every entry written to the public transparency log. No shared row connects one identity's object to any other's.

Per-identity state — the secret hash, the full key history, linked OAuth accounts, and a capped (500-entry) internal audit trail used only for operator debugging — lives inside that identity's own **Durable Object** (`IdentityDO`), one instance per registered identity, addressed by the identity's own UUID as the Durable Object name. Everything inherently global instead stays in Cloudflare D1 (serverless SQLite), mirroring the rest of the Alien Lights backend. Table 1 makes the placement of every piece of state, and the reason for it, explicit rather than leaving it implied by the code.

Table 1 — Where each piece of system state physically lives, and why

State	Location	Why it lives there

Secret hash, full hybrid key history, linked-provider list, internal audit trail	That identity's own `IdentityDO` (Durable Object storage)	Naturally shardable per identity; enables self-scheduled rotation (§4) and isolates blast radius (§4, §9)
Identity provider's own signing keypair, current and retired versions	`identity_idp_keys` (D1)	Exactly one IdP signing key is current at any moment — global by definition
Root key-encapsulation keypair, current and retired versions	`identity_root_kek_versions` (D1)	Exactly one root KEK wraps new keys at any moment — global by definition (§5)
Public transparency log (one shared, ordered, hash-chained sequence)	`identity_transparency_log` (D1)	A log's entire value is one shared total order across every identity and the IdP itself — cannot be sharded (§6)
External OAuth provider catalog and encrypted client secrets	`identity_oauth_providers` (D1)	Shared configuration, not per-identity data (§7)
"Which identity, if any, already owns this external account" index	`identity_oauth_link_index` (D1)	Answers a global uniqueness question without scanning every identity's Durable Object
Registered relying-party (RP) OAuth clients, authorization codes, and issued-token records	`identity_rp_clients` / `identity_rp_codes` / `identity_rp_tokens` (D1)	"Sign in with Alien Lights" provider-side state is inherently global, like the log (§7)

The short version of this table: anything that can be sharded per identity now is, and anything that by definition can't — a log needs one shared total order; a root key needs to be the same key for everyone it wraps — stays in the one shared place that makes sense for it. Appendix B gives the full field-level schema for both the D1 tables and the Durable Object storage keys.

3. Cryptographic Primitives: Hybrid Signatures

Every signature Alien Lights Identity produces — whether over a user-supplied message, an OIDC ID token, or a transparency log leaf — is actually *two* signatures over the identical bytes, computed and verified independently:

- **ML-DSA-65** (FIPS 204) — a lattice-based signature scheme standardized by NIST in August 2024 as part of its post-quantum cryptography program. Implemented via `@noble/post-quantum`, a pure-JavaScript, audited-primitive library with no native/WASM dependency, so it runs unmodified inside a Cloudflare Worker's V8 isolate.
- **Ed25519** (RFC 8032) — the classical elliptic-curve signature scheme already in wide production use, included so that any client or library that does not yet understand ML-DSA can still verify the signature today using ordinary, well-understood tooling.



Figure 2 — A single message produces two independent signatures over the same bytes, which travel together as one unit everywhere a signature is used.

For OIDC tokens specifically (full flow in §7), the classical Ed25519 signature is embedded as an ordinary EdDSA-signed JWT — any off-the-shelf JOSE/JWT library verifies it unmodified today — while the ML-DSA-65 signature over the identical `header.payload` signing input travels as a sibling `pq_signature` field, with the corresponding public key published as a vendor-extension `pq` member of the JWKS document's key entry. A quantum-safety-aware client checks both; every other client simply checks the JWT as it always has.

> **Why hybrid, not PQ-only?** ML-DSA has had roughly a decade of public cryptanalysis — far less than Ed25519's. Signing with both means a flaw discovered in either algorithm does not by itself break identity assertions: the classical signature holds if ML-DSA turns out to have an undiscovered weakness; the ML-DSA signature holds if a quantum computer breaks Ed25519. This is the same "belt-and-suspenders" posture major browsers and CDNs are already using for post-quantum *key exchange* in TLS — applied here to signatures, and in §5 to key custody as well.

4. Key Lifecycle: Anniversary-Anchored, Self-Scheduling Rotation

Regular key rotation limits the blast radius of a compromised key: an attacker who obtains a signing key can misuse it only until the next scheduled rotation. The naive implementation — every key expires 30 days after being issued, checked by a periodic sweep — has a hidden cost at scale: because most identities register at different times but a shared 30-day-from-now clock resynchronizes toward whatever moment the sweep runs, rotations tend to cluster, producing a system-wide spike in cryptographic work and transparency-log writes on a shared day rather than smooth, continuous load.

Alien Lights Identity instead anchors every identity's rotation schedule to that identity's own `created_at` timestamp. Key version *N* expires at `created_at + N months`, computed by `addMonthsClamped` (clamping to the last valid day of a shorter target month — an identity created on January 31 rotates on February 28/29, March 31, and so on). Because signups are naturally spread across the calendar, so are rotations.



Figure 3 — A shared rotation clock concentrates load on one day per month; anchoring to each identity's own signup date spreads the same total rotation work evenly across every day of the month.

What actually fires a rotation is a platform primitive, not a cron job. Each identity's own `IdentityDO` calls the Durable Object alarm API — `storage.setAlarm(expiresAt)` — every time it issues a key, pointing it at exactly that key's own expiry. That object's `alarm()` handler is the only thing that ever rotates that identity's key; there is no periodic job scanning a shared table looking for identities whose keys happen to be due. This removes the cost a naive sweep design has at scale regardless of clustering: a shared scan is one unit of work whose cost grows with the number of identities in the system, re-executed on a fixed clock regardless of how many of those identities actually need anything done; `N` independent, self-scheduled alarms firing only when they are each individually due has no such term. The one piece of

global rotation logic that remains — the identity provider's own signing key — is a single-row due-date check still run from the existing 30-minute cron, which is `O(1)` rather than a sweep, since there is only ever one IdP key to check.

A rotation can also be forced immediately via `POST /api/identity/:uuid/rotate`, which runs the identical rotation logic the alarm would, just without waiting for the expiry to actually arrive. Retired keys are never deleted — only marked non-current — so a signature made before a rotation remains verifiable by specifying its `key_version`, preserving a permanent, append-only history of every key an identity has ever held. A revoked identity's alarm is explicitly cancelled (`storage.deleteAlarm()`) so a revoked identity never generates a surprise new key it has no business holding; its secret hash is still checked (not erased) when revocation is requested a second time, so re-revoking an already-revoked identity is a harmless no-op rather than an error.

> **Why this also reduces blast radius, not just scheduling cost.** Moving rotation logic inside each identity's own Durable Object is inseparable from moving that identity's *state* there too (secret hash, key history, linked OAuth accounts, audit trail — see Table 1). The practical consequence: an infrastructure incident that disrupts wherever the platform happens to colocate one identity's object has no shared row or table it could also take out for any other identity. The previous, shared-D1-table design had the opposite property — every identity's state sat in the same physical rows, so whatever could take down one identity's data could take down all of them at once.

5. Full-Stack Post-Quantum Key Custody

§3's hybrid signatures protect the *act of signing* against a future quantum attacker — forging a new signature on a message or token requires breaking both algorithms. That is a narrower guarantee than it first appears: it says nothing about an attacker who, instead of forging a new signature, simply steals and later decrypts an *old* private key sitting at rest. If every private key in the system were encrypted with one AES key symmetrically derived from a single long-lived secret, recording the encrypted database today and later breaking that one secret — by brute force, by insider leak, or (for a sufficiently weak scheme) by a future quantum computer — would decrypt every private key the system has ever stored. This is "harvest now, decrypt later" aimed at *custody* rather than at the signature math, and hybrid signing alone does not close it.

Every private key in the system — every identity's hybrid signing key (held inside its own `IdentityDO`), the identity provider's own signing key, and every configured external OAuth provider's client secret — is instead wrapped using a **fresh ML-KEM-768 + X25519 hybrid key encapsulation** against a long-lived **root key-encapsulation keypair** (root KEK), not a single static symmetric secret. ML-KEM (FIPS 203) is NIST's standardized lattice-based KEM — the key-establishment counterpart to ML-DSA's signatures — paired here with a classical X25519 encapsulation via `@noble/post-quantum`'s hybrid combiner, for the same belt-and-suspenders reasoning §3 already applied to signing.

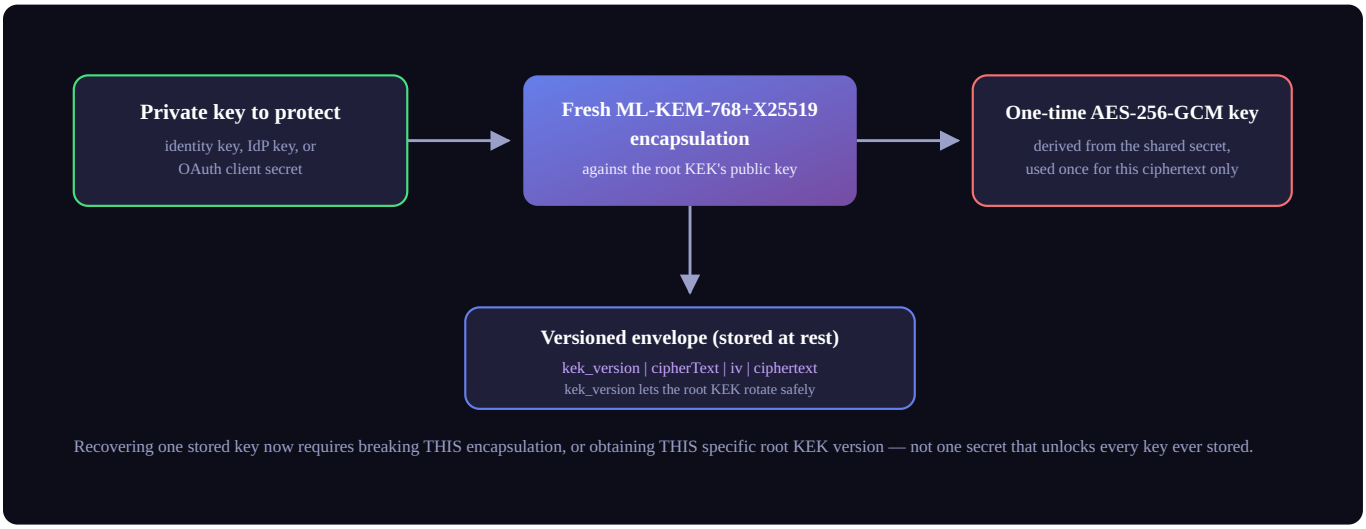


Figure 4 — Wrapping a private key at rest. Each wrap performs a brand-new KEM encapsulation, so compromising one stored ciphertext does not help against any other — contrast with a single static symmetric secret, where breaking it once decrypts everything.

Concretely, each wrap operation (1) encapsulates a fresh shared secret against the root KEK's public key, producing a KEM ciphertext (`cipherText`) and a 32-byte shared secret that is imported directly as an AES-256 key — the hybrid combiner already outputs a fixed-length, uniform SHA3-256 digest over both component shared secrets, so no additional key-derivation pass is needed; (2) generates a random 12-byte IV and encrypts the actual private-key bytes under that one-time AES key in GCM mode; (3) serializes the result into a single wire envelope: a 4-byte header (big-endian `uint16` KEK version, then `uint16` KEM-ciphertext length), followed by the KEM ciphertext, the IV, and the AES-

GCM ciphertext, all base64-encoded as one string. Unwrapping reverses exactly this: look up the root KEK version named in the header, decapsulate to recover the shared secret, re-derive the AES key, and decrypt.

The embedded **version** number is what lets the root KEK itself be rotated (an explicit administrative action, deliberately not on any automatic schedule, given how much depends on it, via `POST /api/identity/admin/root-kek/rotate`) without breaking anything already wrapped: old root KEK versions are retained, never deleted, purely so previously-sealed keys stay decryptable under whichever version originally sealed them.

> **Honest limitation: the root KEK's own bootstrap.** Every custody chain has to anchor in *something*, and here that something is the root KEK's own private key, which is the one secret in this entire chain still wrapped the old way — plain AES-256-GCM, keyed by a single Workers secret (`IDENTITY_KEK`) hashed into a 256-bit key, with no KEM involved — since there is no fourth-order key available to wrap a third-order KEK with. If `IDENTITY_KEK` is unset, the code falls back to a well-known, explicitly insecure development key so local `wrangler dev` work keeps functioning; that fallback logs a loud warning and must never be reached in a production deployment, which is why it is called out here rather than left to be discovered.

This was validated directly across a live root KEK rotation (detailed further in §11): an identity key wrapped under root KEK version 1 remained fully decryptable and usable for signing *after* rotating to version 2, and a key generated after the rotation correctly wrapped under version 2 and was immediately usable — confirming the versioned-envelope design does what it claims rather than silently breaking existing custody on rotation.

Client-held keys: an offline ceremony device

Everything above is a *server*-custody model: the system protects private keys it holds, but it still holds them. Table 2's "server-side signing custody" row names the natural alternative — the client holding its own private key and signing locally, so the server is never in the trust path for an individual signature at all. `apps/alien_mail` in this repository is a first, narrowly-scoped step toward that: a Flutter app that generates, stores, signs with, and verifies a FIPS 204 ML-DSA-65 keypair entirely offline on an Android phone — airplane mode is fine, no network call is required for any of those four operations.

It deliberately uses the *exact same* algorithm and wire format as §3: standard base64 encoding, the same empty-context hedged external `Sign`, the same 1952/4032/3309-byte key and signature sizes. That equivalence is proven rather than assumed — the app's `tool/interop_check.dart` cross-verifies signatures in both directions between its pure-Dart `pqcrypto` implementation and this backend's own `@noble/post-quantum` library, so a keypair generated on a phone with the network off produces signatures this system's own verifier would accept, byte-for-byte compatible, not merely spec-compatible.

Hardware-wrapped custody: Android Keystore, StrongBox, Knox Vault

As of v1.1.0, the app's secret key is wrapped at rest by an AES-256-GCM key generated directly in the Android Keystore, requesting **StrongBox** backing via the standard Android API (`KeyGenParameterSpec.setIsStrongBoxBacked`) — which, on Samsung devices that implement it, routes to **Knox Vault**, a discrete secure element, with no proprietary

Samsung SDK dependency required to reach it. The app's "Hardware key protection" card reports the *actual* level this specific device's Keystore returns — queried fresh via `KeyInfo` every time, not inferred from the phone's model name — as one of `strongbox` / `tee` / `hardware` / `software` / `unavailable`; devices without StrongBox fall back to a standard (usually still TEE-backed) Keystore key rather than failing. Deleting a keypair also destroys this Keystore wrapping key itself, not merely the encrypted blob, so a previously-exfiltrated copy of that blob becomes permanently unrecoverable.

> **Honest scope — read before treating a phone as an HSM.** "Hardware-wrapped at rest," "hardware-isolated during signing," and "wired into the live system" remain three different claims. Two to be precise about: (1) **the wrapping key can now be genuinely hardware-isolated** (Knox Vault/StrongBox, on a device that has it) — this is real, not aspirational, and is exactly what the protection card reports; (2) **signing itself is still not hardware-isolated** — the secret key still passes through normal application memory every time it signs, after being unwrapped, because no shipped secure element — Knox Vault included — runs ML-DSA operations internally yet. As of v1.2.0, the third item — wiring into this backend — is **done**: see the client-held-key section below.

Client-held keys, wired into the live backend

As of v1.2.0, a keypair generated by `identity_ceremony` can be registered with this backend without the server ever custodying the secret key — a second custody model alongside the server-side one described in §4, selected per identity at registration time and visible as a `custody` field (`"server"` or `"client"`) on every identity and key version. `POST /api/identity/register` accepts an optional `pq_public_key` (base64, the exact 1952-byte ML-DSA-65 public key the app generated locally); when present, the identity's `IdentityDO` stores only that public key — no classical/Ed25519 half (the app doesn't generate one yet), no KEM-wrapped secret key, nothing for the server to sign with. `POST /api/identity/{uuid}/sign` returns `409` for such an identity by construction, not by policy choice that could quietly be reverted — there is no secret key in that Durable Object's storage to retrieve. Rotation works the mirror-image way: `POST /api/identity/{uuid}/rotate` requires a new `pq_public_key` in the request body for a client-held identity (the app generates it locally first, via the same offline ceremony, then pushes it), rather than the server minting a replacement itself; there is also no self-scheduled monthly rotation alarm for a client-held identity; the device decides when, by actually re-running the ceremony. Verification is unchanged — `verifyHybrid` already treats a missing classical signature/key as "not checked," not "invalid," so a PQ-only client-held signature verifies exactly like a hybrid one, including through `POST /api/identity/verify`'s fully-offline inline-public-key path. Every registration and rotation for a client-held identity is still a leaf in the same transparency log described in §6, with `custody` recorded in that leaf's public data.

A prebuilt release APK, its SHA-256 checksum, and the full source are all in this repository and its infrastructure: `identity_ceremony-v1.2.2.apk` (~50 MB, SHA-256 `20399db821bd474ad63a2da3e520c0a737e0d1734a2c89dac11a591176118696`, debug-signed — this is a personal-use sideload build, not a Play Store release. v1.2.2 fixes a missing `INTERNET` permission that silently broke every network action in v1.2.0/v1.2.1 on real hardware — see the validation callout below; v1.2.1 itself was a cosmetic-only update to Alien Lights' yellow/black brand theme over v1.2.0).

6. Transparency Log: Hash Chain & RFC 6962 Merkle Inclusion Proofs

Every event that changes an identity's or the provider's own key material — registration, rotation (forced or scheduled), revocation, and the identity provider's own signing-key creation and rotation — is appended to `identity_transparency_log` as one leaf in a hash-chained, dual-signed sequence. This is intentionally a separate structure from each identity's internal audit trail (Table 1): the audit trail may contain anything useful for internal debugging and is capped and private; the transparency log contains *only* public key material, by construction, so that publishing the entire log carries no confidentiality risk.

Each leaf's `entry_hash` is defined as:

```
entry_hash = SHA256(leaf_index || prev_hash || event_type || identity_id || public_data || created_at)
```

signed with both ML-DSA-65 and Ed25519 using the identity provider's own current signing key, with that key's public key embedded directly in the leaf — so verification never depends on a live, mutable JWKS lookup, only on the leaf's own contents. `prev_hash` is the prior leaf's `entry_hash` (or 64 zero bytes for the genesis leaf), forming a linear chain in which altering any historical leaf changes its hash and breaks every subsequent link. The identity provider's own signing key is itself bootstrapped self-referentially: the very first leaf ever written (`idp_key_created`) is signed by the very key it announces — the same place any Certificate Transparency log's root ultimately has to start trusting itself.



Figure 5 — Four leaves from an actual local test run. An independent verifier re-derives every entry_hash from the leaf's own fields, confirms prev_hash matches the prior leaf's entry_hash, and checks both embedded signatures — exactly what `GET /api/identity/transparency/verify` does server-side, and exactly what anyone else can do without it.

This construction was validated directly: a controlled test that rewrote one leaf's `public_data` field in the underlying database was correctly reported by the verifier as `entry_hash does not match leaf contents`, identifying the exact tampered leaf (full results in §11).

From hash chain to Merkle tree: proving one leaf without the rest

A hash chain alone gives tamper-evidence and a total order, but checking it means downloading every leaf — fine for auditing the whole log, impractical for a lightweight client that only cares whether *one* specific entry (say, the registration event for a specific key it's about to trust) is really in the log. The system layers a full **RFC 6962 Merkle tree** — the same construction Certificate Transparency uses — over the same ordered sequence of leaves the hash chain already produces. Each leaf first gets a domain-separated **RFC 6962 leaf hash**, `sha256(0x00 || entry_hash)`; internal tree nodes are `sha256(0x01 || left || right)` — the differing leading byte (`0x00` for a leaf, `0x01` for an internal node) is what prevents a leaf hash from ever being mistaken for an internal node hash, defending against the second-preimage attack RFC 6962 explicitly calls out.

Formally, the Merkle Tree Hash of a leaf-hash range `[start, end)`, written `MTH(start, end)`, is RFC 6962 §2.1's exact recursive definition: `MTH(i, i+1) = leaf_hash[i]` for a single leaf, and for a larger range, split at `k`, the largest power of two strictly less than the range's size, and combine `MTH(start, start+k)` with `MTH(start+k, end)` under the node-hash prefix. This power-of-two split is the standard construction for an append-only tree that is not required to be a perfect binary tree at every size — exactly the unbalanced case Figure 6 below demonstrates with 5 leaves. The audit-path operation `PATH(m, start, end)` (RFC 6962 §2.1.1) walks the identical split recursively, at each level choosing whichever sibling subtree does *not* contain leaf `m` and appending its hash to the path, bottom level first. The implementation memoizes each `MTH` call per request so computing the root plus several audit paths from one snapshot does not redundantly re-hash the same subtrees.

The server publishes a **Signed Tree Head** (STH) — a `tree_size` and `root_hash`, signed by the identity provider's current key, from `GET /api/identity/transparency/root` — the same small, trusted checkpoint Certificate Transparency logs publish. Given that STH, a verifier requests an **inclusion proof** for one leaf (`GET /api/identity/transparency/proof?leaf_index=N`): the leaf's own hash, plus an audit path of `O(log n)` sibling hashes. Figure 6 shows this for a concrete 5-leaf tree — proving leaf 2 is included touches only two sibling hashes, never leaves 1, 3, 4, or 5's actual contents. Verifying the proof (`verifyInclusionProof`, or the convenience endpoint `POST /api/identity/transparency/verify-proof`) replays the same top-down split decisions used to build the path to learn, level by level, whether the target leaf's subtree was the left or right child — then walks the audit path bottom-up, combining the leaf hash with each sibling on the correct side, until a single candidate root falls out. If that candidate equals the STH's `root_hash`, the leaf is proven included; nothing else was ever read.

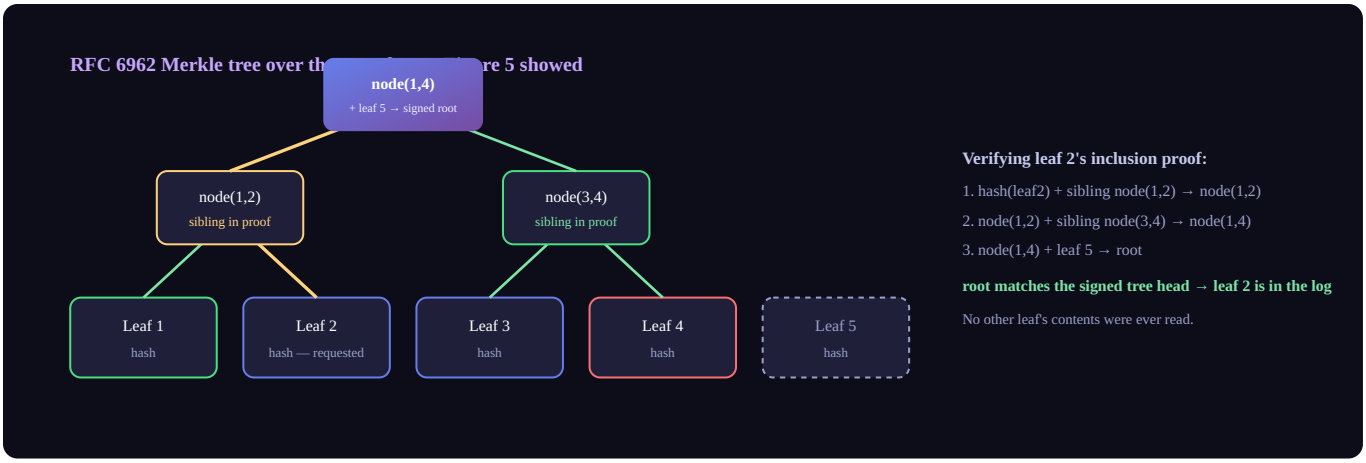


Figure 6 — An RFC 6962 inclusion proof for leaf 2 of a 5-leaf (deliberately unbalanced, not-a-power-of-two) tree. The verifier combines leaf 2's own hash with the two highlighted sibling hashes, bottom-up, and reconstructs the exact root the Signed Tree Head already committed to — without reading leaves 1, 3, 4, or 5.

This was validated end-to-end against an actual unbalanced 5-leaf tree — the hardest small case to get right, since it is not a power of two at any level: all 5 individual per-leaf inclusion proofs independently reconstructed the identical root computed separately by the full-chain verification endpoint. A single flipped byte in a submitted leaf hash was correctly rejected (the reconstructed root no longer matches); a request for an out-of-range `leaf_index` was correctly rejected with `400` (§11 has the complete results).

> **Honest limitation.** The server recomputes the relevant part of the tree from the underlying leaf hashes on every request — `O(n log n)` work per call via memoized recursion, `O(n)` just to load the leaves from D1 — rather than maintaining an incrementally-updated frontier the way a log operating at Certificate-Transparency billions-of-certificates scale would need to. That is the correct, defensible place to start for a log far below that scale; the verifier-side guarantee an inclusion proof provides does not depend on how the server happened to compute it, only on the proof reconstructing correctly, which it does.

7. Federation: OAuth2/OIDC In and Out

Alien Lights Identity is deliberately not a closed system. It participates in the existing federated identity ecosystem in both directions.

Linking out: a generic OAuth2/OIDC connector

Rather than hardcoding integration logic per provider, a single data-driven table (`identity_oauth_providers`) holds each provider's authorize/token/userinfo URLs, scopes, user-id/email JSON paths, and credentials; `handleOAuthLinkStart` and `handleOAuthLinkCallback` drive a standard authorization-code-plus-PKCE flow entirely from that data, with no code changes required to add a new provider. The schema ships pre-seeded with twelve providers most commonly requested — Google, GitHub, Microsoft, Discord, GitLab, Slack, LinkedIn, Twitch, Spotify, Reddit, X, and Apple — each disabled until an operator supplies real credentials through the admin endpoint, which stores the client secret already KEM-wrapped against the root KEK (§5), never in plaintext. A provider whose profile endpoint returns no separate `userinfo` URL (Apple's model) is handled by decoding the profile directly out of the token response's `id_token` instead.

Linking in: Alien Lights as an OIDC provider

Alien Lights also runs a complete, standards-compliant OpenID Connect provider: a discovery document at `/.well-known/openid-configuration`, a JWKS endpoint, an authorization-code grant with mandatory PKCE (`S256` challenge method — no implicit or plaintext-PKCE fallback), refresh tokens, and a userinfo endpoint — so third-party applications can offer "Sign in with Alien Lights" using ordinary OIDC client libraries. Relying-party applications are registered in advance via `POST /api/identity/admin/clients`, which returns a generated `client_id` and a one-time-shown `client_secret` (stored only as a hash) alongside an allow-list of exact `redirect_uris` the authorize and token endpoints both enforce. Figure 7 shows the full flow as implemented.

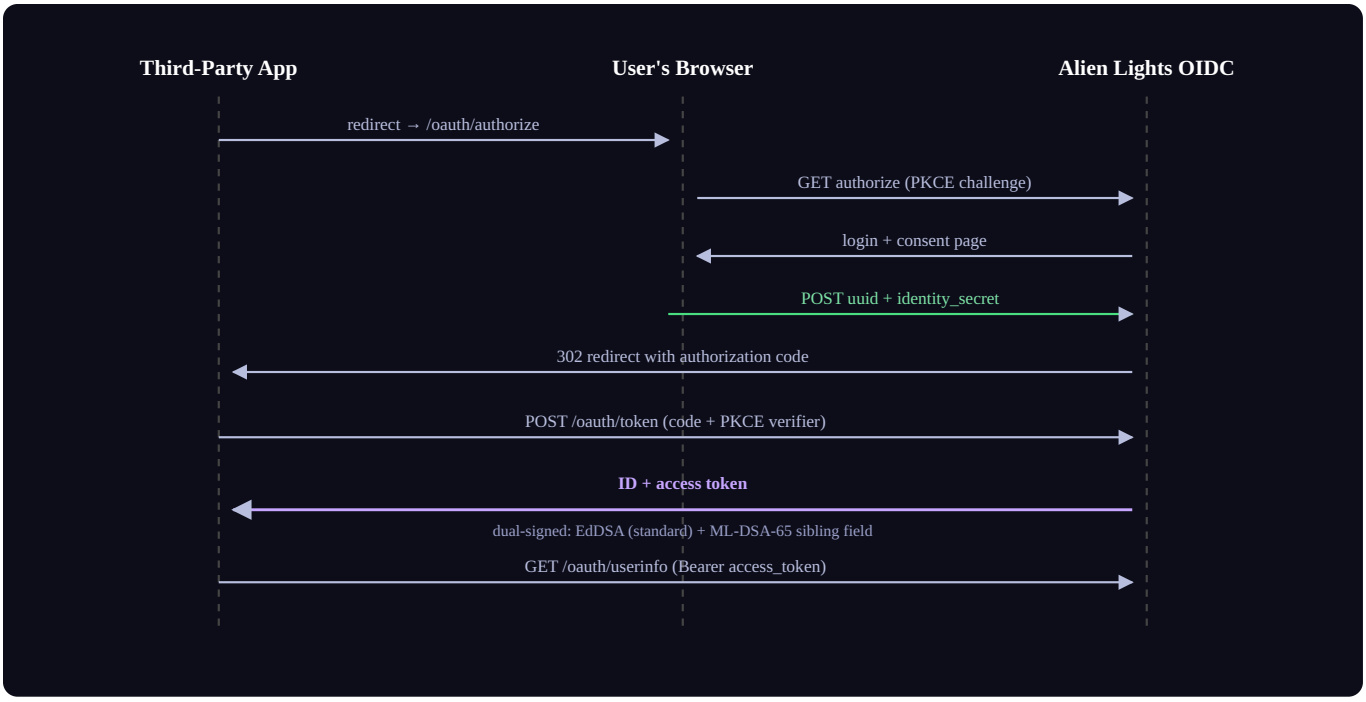


Figure 7 — A standard authorization-code + PKCE OIDC flow. The issued tokens are ordinary, interoperable JWTs with the ML-DSA-65 signature riding alongside as a vendor-extension field — a client that has never heard of post-quantum cryptography still works correctly.

A refresh grant (`grant_type=refresh_token`) revokes the presented refresh token and issues a brand-new access/refresh pair rather than simply extending the old one — single-use refresh tokens, so a stolen-and-replayed refresh token is detectable the first time the legitimate client tries to use its own copy and finds it already revoked. This flow, including the resulting access token, was independently verified outside the system's own verifier — decoding the issued JWT and checking its Ed25519 signature with an independent cryptography library against the key published at `/oauth/jwks.json` — confirming the issued tokens are standard, interoperable JWTs rather than something only Alien Lights' own code can read.

8. Resilience: Degraded and Disconnected Operation

Every endpoint described so far assumes the identity API and its backing database (Cloudflare D1) are both reachable. This section describes what happens when that assumption breaks — a transient D1 outage, a network partition, or a verifier with no connectivity to Alien Lights at all — and extends a property §6 already established for the transparency log (verification "never depends on a live, mutable JWKS lookup, only on the leaf's own contents") to live key lookups and signature verification generally.

Fully offline verification

`POST /api/identity/verify` accepts the signer's public key material (`pq_public_key` / `classical_public_key`) directly in the request body as an alternative to looking it up by UUID. When supplied, verification is a pure cryptographic computation — `verifyHybrid` over the message and signature — with no database query and no network call to Alien Lights at all. A verifier that cached a public key once, from any of three sources (a prior `GET /:uuid`, the OIDC JWKS document, or a transparency log leaf), can keep verifying signatures against it indefinitely with zero ongoing connectivity.

Cache-aside fallback for D1 outages

For the common case of looking a key up *by identity* rather than supplying it inline, three read paths — `GET /:uuid`, `GET /oauth/jwks.json`, and the by-UUID branch of `POST /api/identity/verify` — go through a cache-aside helper (`cachedRead`) backed by Workers KV. Every successful D1 read refreshes a cache entry; if D1 throws, the handler serves the most recent cached value instead of failing the request, and marks the response `degraded: true` with a `served_from_cache_at` timestamp so the caller can tell the difference. JWKS is the single highest-value endpoint to keep serving this way: the identity provider's signing key only changes on a monthly rotation, so a 15-minute-stale cache entry is essentially always still the current key anyway.

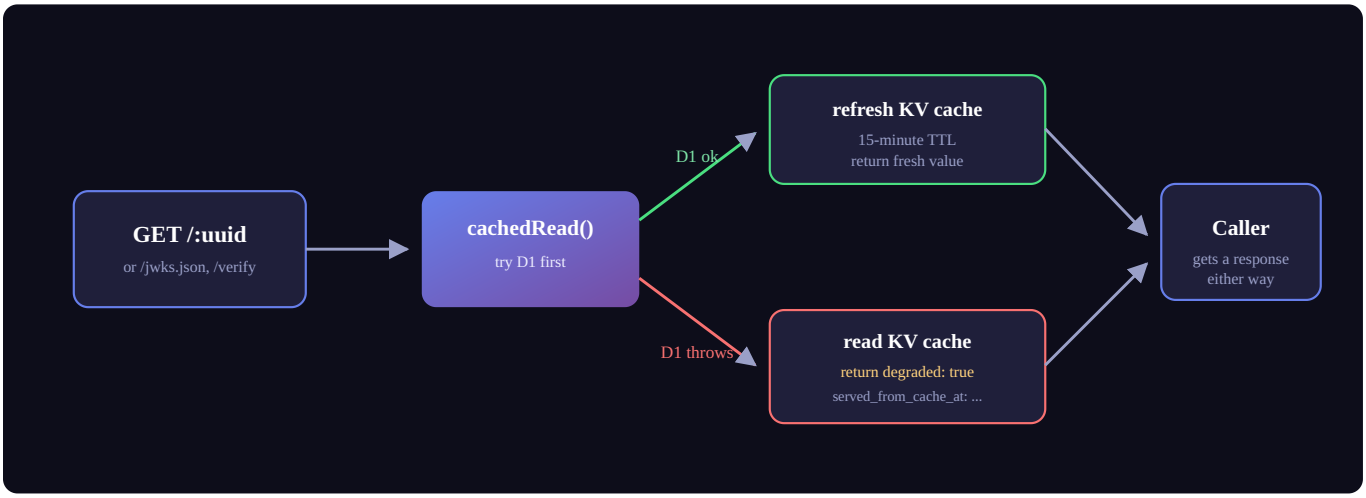


Figure 8 — `cachedRead()`: every public-key read refreshes a KV cache entry on success, and falls back to it on a D1 failure rather than returning an error. Writes (register/rotate/sign/revoke/token issuance) and the `userinfo` authorization check deliberately bypass this path entirely and stay fail-closed.

`GET /api/identity/health` performs a real `SELECT 1` against D1 and reports `{"d1": "ok"|"unreachable", "mode": "normal"|"degraded"}`, so a monitor or a client deciding how much to trust a `degraded: true` response elsewhere can check connectivity directly rather than only inferring it after the fact. This check covers D1 — the global tables identity.js still owns — and does not and cannot check every individual identity's own Durable Object.

> **Honest limitation.** The cache-aside fallback trades availability for revocation responsiveness, and that trade is not free. If an identity's key is revoked during the same window D1 happens to be unreachable, a verifier relying on the degraded-mode cache can keep accepting that now-revoked key for up to 15 minutes — the cache TTL — or until D1 recovers, whichever comes first. This window was chosen to ride through a brief D1 blip while keeping the exposure bounded and short, not to make it zero. Every write path, and the token-validity check in `userinfo`, are deliberately excluded from this fallback for exactly this reason — there is no safe degraded version of "is this credential still valid," only of "what was this public key the last time we successfully checked."

9. Threat Model and Honest Limitations

Consistent with the engineering values behind this project, this section states plainly what this system does *not* yet protect against, rather than leaving it implied.

Table 2 — Known limitations and their current mitigation status

Limitation	Detail	Status
Server-side signing custody	For a `custody: "server"` identity (still the default), the server signs on the identity's behalf rather than the client holding its own private key and signing locally — the server is in the trust path for every one of that identity's signatures, independent of how the key is protected at rest (§5).	As of v1.2.0, resolved for any identity that opts into `custody: "client"` at registration: `apps/alien_mail`'s offline-generated public key is registered via `POST /api/identity/register`, the server never receives or custodies that identity's secret key, and `/sign` returns 409 for it (§5). `custody: "server"` remains the default and is unaffected — this is an additional model, not a replacement.
No hardware security module	Key material is decrypted into application memory when used, not held in an HSM where it would never exist in plaintext, even momentarily. §5's ML-KEM wrapping protects the key at rest against a static-secret compromise; it does not protect against an attacker who compromises the running Worker process itself mid-operation.	Not yet implemented.
No independent security audit	No third-party penetration test, bug bounty, or formal protocol verification has been performed, for the cryptography, the Durable Object isolation model, or the Merkle tree implementation.	Not yet performed.
Merkle tree computed per-request, not incrementally	Inclusion proofs and the Signed Tree Head are recomputed from the underlying leaf hashes on every call (§6) rather than from a maintained incremental frontier. Correct and sufficient at current scale; would need revisiting at Certificate-Transparency-scale leaf counts.	Documented trade-off; not a correctness issue at today's scale.
No production-scale validation	The per-identity Durable Object, Merkle-proof, and key-custody features (§4–§6) have been verified end-to-end against `wrangler dev` and a local D1/Durable-	Not yet performed.

	Object instance (§11), not yet under real production traffic, concurrent-rotation races at scale, or a regional Cloudflare outage.	
Degraded-mode cache can briefly outlive a revocation	The cache-aside fallback for D1 outages (§8) can serve a cached public key for up to 15 minutes after that key was revoked, if the revocation happened during the same window D1 was unreachable.	Documented trade-off; bounded to a 15-minute window, excludes every write path and the `userinfo` authorization check.
Root KEK bootstrap is still a static secret	The root KEK's own private key is wrapped with a conventional AES-256-GCM secret (§5), not a further KEM — every custody chain has to anchor somewhere, and this is the one remaining link in this system that does.	Documented, narrow, and unavoidable without a hardware root of trust (see HSM row above).

Two items that appeared in this table in an earlier revision of this paper — the transparency log lacking per-leaf inclusion proofs, and a shared-database architecture without per-identity isolation — are resolved as of this revision (§6, §4) and have been removed from the table rather than left as stale "planned" rows.

10. Comparison to Industry Practice

This section makes one narrow, falsifiable comparison rather than a general claim of superiority. As of this writing, neither Google Identity nor Microsoft Entra ID has shipped post-quantum signatures on identity/OAuth tokens in production, nor does either give individual consumer users their own asymmetric signing keypair, nor does either publish a public, independently verifiable log of identity key-lifecycle events.

Table 3 — Capability comparison (as of November 2026)

Capability	Google Identity / Microsoft Entra ID	Alien Lights Identity
Post-quantum TLS key exchange	Yes (ML-KEM/Kyber hybrid, both)	Yes (inherited from Cloudflare's edge)
Post-quantum signatures on identity tokens	Not shipped in production	Yes — every token dual-signed
Post-quantum protection for key *custody* (private keys at rest)	Not publicly documented	Yes — ML-KEM-768+X25519 wrapping against a rotatable root key, not a static secret
Per-user asymmetric keypair	No — provider signs with its own key only	Yes
Per-identity infrastructure isolation (blast radius)	Not publicly documented at the individual-account level	Yes — one Durable Object per identity, self-scheduling its own rotation
Public, independently verifiable key-lifecycle log	No public consumer-facing equivalent	Yes
Per-leaf inclusion proofs (RFC 6962 Merkle tree)	N/A — no public log to begin with	Yes — the same guarantee Certificate Transparency gives TLS certificates
Independent security audit / compliance certifications	Extensive	None yet
Production scale and SLA history	Years of hardening	Unproven

The defensible claim is narrow: *first OIDC-compatible identity provider with hybrid post-quantum token signing, post-quantum key custody, per-identity infrastructure isolation, and a public key transparency log with per-leaf inclusion proofs, running entirely on Cloudflare's edge* — not a claim to be the best identity system in the world, which is a claim about institutional trust this system has not yet earned (§9).

11. Validation & Testing Evidence

This section reports exactly what has been exercised and in which environment, rather than one blended "it's been tested" claim — the distinction between local development validation and live production validation matters, and the Threat Model (§9) already flags production-scale load as not yet validated. Nothing below should be read as contradicting that.

Local development: per-identity Durable Objects, Merkle proofs, key custody (§4–§6)

Verified end-to-end against `wrangler dev` and a local D1/Durable-Object instance:

- Full lifecycle: registration → sign → verify (including signature-tamper rejection) → forced rotation → verifying a pre-rotation signature by explicit `key_version` → revocation.
- A revoked identity correctly rejected for both `sign` and `rotate`, while a second, repeated `revoke` call against an already-revoked identity returned success idempotently rather than erroring.
- An OAuth provider's configured client secret confirmed KEM-wrapped at rest — not plaintext — by reading the raw `identity_oauth_providers` row directly out of D1.
- A full root KEK rotation to a new version: a key wrapped under the prior version remained decryptable and usable for signing after rotation, and a freshly generated key correctly wrapped under the new version and was immediately usable (§5).
- A complete OIDC authorization-code + PKCE login (`authorize` → `token` → `userinfo` → `refresh_token`); the issued JWT was independently decoded and its Ed25519 signature checked against the published JWKS using a separate cryptography library, outside the system's own verifier (§7).
- OAuth link-start correctly rejecting a wrong identity secret (401) and an unconfigured provider (409).
- The full transparency log — registration, rotation, revocation, and IdP key creation leaves — verifying cleanly end to end, including its Merkle root; a direct database tamper to one leaf's `public_data` field correctly caught and reported with the offending leaf index (§6).
- All 5 individual Merkle inclusion proofs for a deliberately unbalanced 5-leaf tree independently reconstructing the same root as the chain-wide verification endpoint; a single flipped byte in a submitted leaf hash correctly producing a mismatched root; an out-of-range `leaf_index` correctly rejected with `400` (§6).
- Each `IdentityDO` confirmed to hold its own independent per-identity storage rather than sharing a table with any other identity's object.

Live production: resilience and connectivity (§8)

Deployed to production and confirmed end-to-end, including via a live browser session against the public `/identity` demo:

- `GET /api/identity/health` returning `{"d1":"ok","mode":"normal"}` against the live database.
- Registering a real identity through the production demo, signing a message, and then verifying that signature fully offline — supplying the public key inline so the check runs with zero D1 lookups — and separately checking API

health, both through the live "Works degraded, works disconnected" demo card.
- All three degraded-mode KV cache entry types (`identity_degraded_cache:identity:*`, `identity_degraded_cache:idp_key`, `identity_degraded_cache:key:*`) confirmed to actually exist in production KV, not just in the code path.
- The regenerated whitepaper PDF confirmed live and downloadable at `/static/papers/alienlights-identity-whitepaper.pdf`.
- The test identity used for this round of verification was revoked afterward as cleanup.

Offline companion app: cross-language crypto interop (§5)

`apps/alien_mail`'s compatibility with this backend's crypto was verified directly, not assumed from both implementations separately following the FIPS 204 spec:

- A signature generated by the backend's `@noble/post-quantum` `ml_dsa65` verified successfully by the app's pure-Dart `pqcrypto` implementation, and vice versa — both directions pass (`tool/interop_check.dart`).
- Key and signature sizes confirmed identical on both sides: 1952-byte public key, 4032-byte secret key, 3309-byte signature.
- Sign/verify round-trips, tamper rejection, cross-key rejection, fail-closed behavior on malformed input, and hedged (randomized) signing — two signatures over the same message differ but both verify — all covered by unit tests, plus an end-to-end widget test of the full generate → sign → verify flow.
- The Knox Vault/StrongBox wrapping layer's platform-channel contract (request/response/error handling, both success and native-side failure) and its envelope logic (wrapped vs. a graceful plain-storage fallback vs. reading the pre-v1.1.0 legacy format unchanged) are covered by dedicated unit tests against a mocked channel and an in-memory store. What those tests cannot cover — and what remains genuinely device-dependent — is which protection level a specific phone's Keystore actually grants; that is read live from the device by the app's own "Hardware key protection" card, not asserted by this paper.

Client-held keys end to end (§5, v1.2.0)

Exercised directly against a local `wrangler dev` instance with a real ML-DSA-65 keypair generated through `@noble/post-quantum` (the same library this backend runs in production):

- `POST /api/identity/register` with a `pq_public_key` correctly returning `custody: "client"`, a `null` classical key, and no `key_rotation_months` / `expires_at` (nothing for a self-scheduled rotation to apply to).
- `POST /api/identity/verify` correctly validating a genuine signature and rejecting a forged one from a different keypair, both via the server-side key lookup and via the fully-offline inline-`pq_public_key` path — an earlier pass caught and fixed a null-pointer bug in the server-side lookup path specific to a `classical: null` key row, now covered by this backend's own request/response shape, not just the client.
- `POST /api/identity/{uuid}/sign` correctly returning `409` for the client-held identity, instead of attempting to unwrap a secret key that was never stored.

- `POST /api/identity/{uuid}/rotate` correctly rejecting a client-held identity's rotation with `400` when no new `pq_public_key` is supplied, and correctly accepting one and bumping the key version when it is; a subsequent `verify` against the new key version succeeded while the old key version's own signature remained independently verifiable (and flagged `revoked: true`) when checked explicitly by version number.
- The transparency log and its whole-chain Merkle verification both confirmed clean after a client-held registration and rotation, with `custody: "client"` correctly present in each leaf's public data.
- Regression-checked the pre-existing `custody: "server"` path (registration, sign returning both signature halves, and a no-body forced rotation) unaffected by any of the above.

What real-device testing caught that the above didn't. Everything in the list above exercised the backend directly (`curl`, a Node script, or `wrangler dev`) — not the actual `identity_ceremony` APK making the request from a phone. When it was tested that way, on a real Galaxy device, every network action in the app (register/rotate/check-status/revoke) failed with `UnknownHostException: ... No address associated with hostname`, on every network, regardless of connectivity. The cause: v1.2.0 added this app's one network feature without adding `android.permission.INTERNET` to `AndroidManifest.xml` — a permission the app never needed before that (through v1.1.0 it was airplane-mode-safe by never touching the network at all). This passed `flutter test`, a `flutter run` debug session, and a Flutter web build without ever surfacing, because Flutter's debug/profile build variants carry their own `INTERNET` override for the Dart VM service — only the actual release build (the only kind ever published) was missing it. Fixed in v1.2.2; confirmed by decompiling the release APK itself (`aapt dump permissions`) before and after, not just inferred from the source diff. See `apps/alien_mail/RELEASES.md`'s v1.2.2 entry. The lesson this leaves standing: none of this paper's "verified against the live backend" claims for this app should be read as "verified from an actual installed APK on real hardware" unless stated that specifically — those are two different claims, and only the second one would have caught this.

First real-device, real-production run. A client-held identity was registered directly from the installed release APK on a Samsung Galaxy S25 FE against the *live production* API — not `curl`, not `wrangler dev`. The app's "Hardware key protection" indicator reported green — hardware-backed — meaning the device's StrongBox-backed Keystore (Knox Vault on this hardware) accepted and is holding the AES-256-GCM wrapping key rather than falling back to plain storage, and an on-device sign → verify round trip over that same key passed. This is the first entry in this paper that is simultaneously (a) from an actual installed APK on physical hardware and (b) against production rather than a backend-only or local-`wrangler dev` check — exactly the combination the callout above flags as untested by everything else in this section. It confirms the mechanism works as designed on at least one real device; it is not, on its own, evidence that every Android device this app targets reaches the same StrongBox-backed protection level (some will fall back to TEE-backed or software-only Keystore implementations depending on hardware).

What this section does not claim. "Deployed to production" and "validated under production *scale*" are different statements. Every item above was exercised correctly at low, manual-testing volume. None of it exercises concurrent-rotation races across many identities at once, sustained real-world traffic, or a genuine regional Cloudflare outage rather than a simulated D1 error — which is exactly why "No production-scale validation" remains its own row in Table 2 rather than being considered closed by this section.

12. Roadmap

An earlier revision of this paper (v1.1) listed three items here as the direct next steps: per-identity Durable Objects, Merkle inclusion proofs, and full-stack post-quantum key custody. All three are now implemented, described in this revision (§4, §5, §6), and exercised in local testing (§11) rather than promised — this section accordingly only lists what remains undone.

1. **Independent security review.** Publish the protocol specification openly and pursue third-party cryptographic and penetration review — of the hybrid signature and KEM-custody design, the per-identity Durable Object isolation model, and the Merkle tree implementation — before this system is trusted with production user data.
2. **Hardware-backed key custody.** Move from application-memory decryption of wrapped private keys toward an HSM or equivalent isolated execution environment, so the plaintext key material never exists outside a boundary even a compromised Worker process cannot read, and so the root KEK's own bootstrap secret (§5, §9) is no longer the weakest link in the custody chain. A phone's secure element (e.g. Samsung Knox Vault) is a plausible consumer-grade candidate: `apps/alien_mail` (§5) now wraps its own secret key with a genuine Knox Vault/StrongBox-backed key where the device supports it — confirmed engaged, not just implemented, on a real Galaxy S25 FE (§11) — proving the hardware-isolation half for at-rest storage. What remains open — for that app, and a fortiori for this backend's root KEK — is the harder half: the signing/decryption *operation itself* happening inside the secure element, which no shipped element (Knox Vault included) does for ML-DSA yet.
3. **Wire client-held keys into the API — done, as of v1.2.0.** `apps/alien_mail` (§5) can now register its locally-generated public key via `POST /api/identity/register`, push a locally-generated rotation via `POST /api/identity/{uuid}/rotate`, and have both verified through the existing `POST /api/identity/verify` endpoint unchanged — with the server never custodying that identity's secret key and refusing (`409`) to sign on its behalf. What remains open: this app only speaks ML-DSA-65 today, so a client-held identity has no classical/Ed25519 half (fine for this backend's own verify logic, which already treats a missing half as "not checked" rather than "invalid," but a gap versus a server-custody identity's full hybrid key); and there's no proof-of-possession step beyond the bearer `identity_secret` on rotate, matching — not exceeding — the trust model a server-custody identity's own forced rotation already uses.
4. **Production-scale validation.** Exercise the system under real traffic, concurrent-rotation races at scale, and a simulated regional Cloudflare outage — all currently validated only against local `wrangler dev` testing and limited live manual checks (§11).
5. **Incremental Merkle tree frontier.** If and when the transparency log's leaf count grows large enough that §6's per-request recomputation becomes a real cost rather than a documented simplification, move to an incrementally-updated tree frontier — the standard approach at Certificate-Transparency scale — without changing anything about the inclusion-proof guarantee itself.

Appendix A — API Reference

All routes are mounted under `/api/identity/*`. This is a complete list of the dispatcher's routes, grouped by concern, so every claim made above is checkable against the actual surface area rather than a paraphrase of it.

Table 4 — Personal identity

Route	Purpose
`POST /register`	Create a new identity (returns the secret exactly once)
`GET /:uuid`	Public identity summary + current key (degraded-mode eligible)
`POST /:uuid/rotate`	Force an immediate key rotation
`POST /:uuid/sign`	Sign a message with the identity's current key
`POST /verify`	Verify a signature, by UUID lookup or fully offline with an inline public key (degraded-mode eligible)
`POST /:uuid/revoke`	Revoke an identity (idempotent)
`GET /:uuid/transparency`	List this identity's own transparency-log leaves
`GET /health`	D1 connectivity check

Table 5 — "Sign in with Alien Lights" (Alien Lights as an OIDC provider)

Route	Purpose
`GET /.well-known/openid-configuration`	OIDC discovery document
`GET /oauth/jwks.json`	Published signing keys, classical + PQ vendor extension (degraded-mode eligible)
`GET` / `POST /oauth/authorize`	Authorization endpoint (renders/accepts login)
`POST /oauth/token`	Authorization-code and refresh-token grants
`GET /oauth/userinfo`	Bearer-token-authenticated profile endpoint

Table 6 — Linking an identity to an external OAuth2/OIDC provider

Route	Purpose
`GET /oauth/providers`	List configured external providers

`GET` / `POST /:uuid/oauth/:provider/start`	Begin the link flow (requires the identity secret)
`GET /oauth/:provider/callback`	Provider redirect target that completes the link
`POST /:uuid/oauth/:provider/unlink`	Remove a linked provider account

Table 7 — Public transparency log

Route	Purpose
`GET /transparency/log`	Paginated raw leaves since a given index
`GET /transparency/root`	Latest leaf + Signed Tree Head
`GET /transparency/verify`	Server-side full-chain + Merkle verification convenience endpoint
`GET /transparency/proof?leaf_index=N`	RFC 6962 inclusion proof for one leaf
`POST /transparency/verify-proof`	Server-side inclusion-proof verification convenience endpoint

Table 8 — Administrative (requires `ADMIN_API_KEY` bearer token)

Route	Purpose
`GET` / `POST /admin/providers`, `POST /admin/providers/:slug/credentials`	Configure external provider credentials (KEM-wrapped at rest)
`POST /admin/clients`	Register a new OIDC relying-party client
`GET /admin/root-kek`	Public summary of the current root KEK
`POST /admin/root-kek/rotate`	Rotate the root KEK to a new version

Appendix B — Data Model Reference

Field-level detail behind Table 1's placement summary.

Per-identity Durable Object storage (`IdentityDO`)

Storage key	Contents
`identity`	UUID, secret hash, display name, status, current key version, timestamps
`key:<version>`	One row per key version ever issued: both public keys, both KEM-wrapped private keys, created/expires/revoked timestamps — never deleted
`link:<provider>`	One row per linked external OAuth provider: provider user id, email, linked-at timestamp
`audit`	Capped (500-entry) ring buffer of internal lifecycle events for operator debugging — distinct from, and not published as part of, the public transparency log (§6)

Global D1 tables (`THREATS_DB`)

Table	Contents
`identity_idp_keys`	The identity provider's own signing keypairs, current and retired, each KEM-wrapped against the root KEK
`identity_root_kek_versions`	The root key-encapsulation keypair, current and retired versions, private key wrapped with the bootstrap AES-256-GCM secret (§5)
`identity_transparency_log`	Every leaf: hash-chain fields, RFC 6962 leaf hash, dual signature, signing key reference (§6)
`identity_oauth_providers`	External provider catalog: endpoints, scopes, user-id/email JSON paths, client id, KEM-wrapped client secret, enabled flag
`identity_oauth_link_index`	Global (provider, provider_user_id) → identity_id uniqueness index
`identity_rp_clients`	Registered "Sign in with Alien Lights" relying parties: client id, hashed secret, name, redirect URI allow-list
`identity_rp_codes`	Short-lived authorization codes awaiting exchange, with PKCE challenge
`identity_rp_tokens`	Issued access/refresh token records (hashed), used for refresh-grant lookup and revocation

References

1. National Institute of Standards and Technology. *FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA)*. August 2024.
2. National Institute of Standards and Technology. *FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM)*. August 2024.
3. Josefsson, S., Liusvaara, I. *RFC 8032: Edwards-Curve Digital Signature Algorithm (EdDSA)*. IETF, 2017.
4. Hardt, D. (ed.) *RFC 6749: The OAuth 2.0 Authorization Framework*. IETF, 2012.
5. Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., Mortimore, C. *OpenID Connect Core 1.0*. OpenID Foundation, 2014.
6. Laurie, B., Langley, A., Kasper, E. *RFC 6962: Certificate Transparency*. IETF, 2013.
7. Jones, M., Bradley, J., Sakimura, N. *RFC 7519: JSON Web Token (JWT)*. IETF, 2015.
8. Jones, M. *RFC 7517: JSON Web Key (JWK)*. IETF, 2015.
9. Lochter, M. et al. *RFC 7636: Proof Key for Code Exchange by OAuth Public Clients (PKCE)*. IETF, 2015.

Source code: `infra/cloudflare/src/identity.js`, `infra/cloudflare/src/identity-crypto.js`, `infra/cloudflare/src/identity-transparency.js` (IdP key lifecycle, Merkle tree, transparency log), `infra/cloudflare/src/identity-root-kek.js` (root KEK custody), `infra/cloudflare/src/durable-objects/identity-do.js` (per-identity Durable Object), `infra/cloudflare/0004_identity_system.sql`. Companion engineering document: `IDENTITY_SYSTEM.md` (see "Degraded / disconnected-environment behavior" for §8's implementation details). Interactive demo: `/identity` on alienlights.net. Connectivity check: `GET /api/identity/health`.