POSTERN LABS — TECHNICAL DOCUMENTATION

The Postern Systems

A field-level technical description of every Postern subsystem: measured boot, remote attestation, the enterprise license spine, confidential-VM cloud, the SDK's real cryptography, vault apps, comms, phone-as-signer, and the Android bridge apps — with the exact structs, constants, protocol flows, and threat models pulled from the source, and an honest trust-tier label on every claim.

Forward-looking & unaudited — see §16 · Generated from `bloch-protocol` source, read-only · Every primitive named is a real dependency in the reviewed code

Contents

1 Status & trust-tier legend

Every claim below carries one of these tags. They are not marketing tiers — they describe exactly what has and has not been exercised, matching the vocabulary the project's own internal gap-analysis (`docs/gap/`) uses to grade itself. Read the tags before the prose; the prose says what a system *is designed* to do, the tag says *how far that has actually been run*.

TAG	MEANING
HOST-PROVEN	Builds and its test suite pass on an ordinary developer machine (no Nix, no TEE, no phone). Cryptographic logic is exercised, usually against committed fixtures — real math, not yet real hardware.
VM-GATED	Blocked only on a Linux + Nix (+ QEMU/flakes) host to build and boot — no special hardware required, just a box this project has not yet run the build on.
HARDWARE-GATED	Requires real measured-boot / TEE / device hardware — a SEV-SNP or TDX host, a PinePhone, a StrongBox-equipped Android device — to produce a genuine, hardware-rooted verdict. Cannot be faked in a VM.
DEPLOY-GATED	The logic is implemented and tested as a library/crate, but no shipping surface calls it yet — durable storage, a deployed endpoint, or a wired UI is missing.
DESIGN-ONLY	Specification, scaffold, or prose only. No running code implements the claim yet.
LIVE	A reachable, deployed endpoint that answers today. Verified by direct HTTP probe as of the date this document was generated, not merely asserted.
UNAUDITED	Applies to *everything* in this document without exception. No third-party security audit has been performed on any Postern system or on the underlying protocol. An audit is contracted, not delivered.

> The honesty discipline this document follows is inherited directly from the project's own internal posture: "*if a claim isn't true yet, phrase it as the plan — and a plan is all it is.*" Where a specific field, constant, or parameter could not be located in the reviewed source, this document says so explicitly rather than inferring or inventing it.

2 What Postern is

Postern Labs builds owned, commercially licensed privacy and security products — a hardened OS, a remote-attestation companion, an enterprise license and governance spine, a confidential-VM cloud edition, a cryptographic SDK, vault/comms apps, and an Android bridge tier — that sell **protection and support**, sold the way BlackBerry once sold a hardened device: never the underlying token. Every Postern product anchors, at its lowest layer, to **Bloch-SIS-PoW**, an *ownerless* protocol (SHAKE-256 hashcash proof-of-work behind a Module-SIS lattice gate, Falcon‖ML-DSA hybrid post-quantum signatures, a GhostDAG-style consensus) that has no owner, no foundation, and no official website by design — Postern consumes it only through its public RPC/API, exactly as any other user would, with no privileged access. Further reading on the protocol itself, including its own reproducible build and mainnet-beta status, lives outside this document at blochprotocol.com and in the protocol's own ownerless repository; this document covers only the **products** layered on top of it.

Everything described here is **pre-production, beta, and unaudited**. The sections below go deep — real struct and field names, real cryptographic parameters, ordered protocol flows, and an explicit threat model per system — but depth is not a claim of maturity. Where the reviewed source shows a placeholder, a fixture-only test, or a feature that has never touched real hardware, this document states that plainly, inline, next to the primitive it qualifies.

3 Postern OS — measured-boot images

VM-GATED HARDWARE-GATED (PCR/TPM2 PROOF) UNAUDITED — every image below is written and, per the project's own build tooling, reproducible-by-design; **none has been built on a Linux+Nix host in this review, and none has been hardware-booted.** Nix files were authored on a non-Nix macOS workstation; no `flake.lock` is committed.

3.1 Architecture: NixOS profiles layered on a shared base

Postern OS is a family of NixOS profiles (`os/*.nix`) that share a base appliance module (`os/configuration.nix`) and layer hardening upward: a **Node ISO** (x86_64 appliance, `.#iso`), a **Desktop ISO** (hardened GNOME/Wayland daily driver, `.#desktop-iso`), a **Mobile image** (aarch64 PinePhone via Mobile-NixOS, `.#mobile-image`), an **Attested image** (dm-verity + UKI + TPM2 measured boot, `.#attested-image`), and a **Cloud/Chiostro image** (confidential-VM guest, `.#cloud-image`, covered in §9). All are declared in `flake.nix` as `nixosConfigurations` pinned to `nixpkgs/nixos-24.05`.

3.2 The measured-boot chain

The attested profile (`os/attested.nix`) is the one that produces a verifiable `os_roothash`. Its `image.repart` block declares three partitions under the image name `bloch-os`:

`esp` — Type `esp`, format `vfat`, ≥128 MiB, holds the Unified Kernel Image (UKI).
`root` — Type `root`, format `erofs`, `Verity = "data"`, `VerityMatchKey = "root"`, `Minimize = "best"` — the read-only sealed rootfs.
`root-verity` — Type `root-verity`, `Verity = "hash"`, `VerityMatchKey = "root"` — the dm-verity hash tree paired to `root`.

Boot flow, as configured (`boot.initrd.systemd.enable`, `boot.loader.systemd-boot.enable`, `boot.initrd.systemd.tpm2.enable`, `fileSystems."/".options = [ "ro" ]`):

1. **Reproducible build** — `nix build .#attested-image` produces an immutable disk image from a pinned flake input set.
2. **systemd-repart** lays out esp / root / root-verity; the root partition's dm-verity roothash is computed over the erofs image.
3. The roothash is written onto the kernel command line as `roothash=<hex>`, sealing the rootfs to that exact byte content — any modification changes the hash and the verity device refuses to mount.
4. Kernel + initrd + cmdline are combined into one signed **Unified Kernel Image (UKI)** PE binary and **measured into TPM PCRs** as a single unit at boot (systemd-boot + `systemd.tpm2`).
5. Root mounts read-only (`ro`); mutable state lives off the sealed root under `services.bloch.dataDir` (default `/persist/bloch`).
6. *[cloud/CVM variant only, see §9]* — the SEV-SNP/TDX launch policy additionally writes an admission-policy hash `P` (admitting only image digest `D`) into SNP **HOSTDATA** or TDX **MR_CONFIG_ID**, so the TEE

launch measurement itself attests to the exact booted image, independent of the disk-level roothash.

> **Status of `os_roothash`.** The reviewed source confirms this chain is *configured*, but the concrete `os_roothash` value the verifier (§4) would check against is a **design-reference placeholder, not a value produced by an actual build** — the measured-boot chain above has never been exercised end-to-end in this review. No PCR measurement has been taken on real firmware. VM-GATED to first produce a real roothash; HARDWARE-GATED to bind it to a TPM2 PCR measurement.

3.3 Node ISO, Desktop ISO, Mobile image

Image	Config	What it configures	Status
Node ISO (x86_64)	`os/configuration.nix`, `.#iso`	`networking.hostName = "postern-os"`; `services.bloch = { enable = true; mine = true; }` — an unattended mining appliance, firewalled, `system.stateVersion = "25.05"`.	WRITTEN, NOT BUILT HERE · claimed QEMU x86_64 boot-validated off-repo, but **no x86_64 QEMU boot harness or committed boot evidence exists in this repo** — treat any such claim as unverified by this review · hardware boot pending.
Desktop ISO (x86_64)	`os/desktop.nix` + `os/browser.nix`	GNOME/GDM on Wayland (telemetry-stripping options disabled); LUKS+TPM2 at install time; deny-all inbound firewall; Tor client on `127.0.0.1:9050`; DNS-over-TLS + DNSSEC via `resolved`; AppArmor; `sudo.execWheelOnly`; hardened sysctls (`kernel.kptr_restrict=2`, `kernel.unprivileged_bpf_disabled=1`); coredumps off; mining off by default.	WRITTEN, NOT BUILT HERE · same off-repo-claim caveat as above · hardware boot pending.
Mobile image (aarch64, PinePhone)	`os/mobile.nix` + `os/telephony.nix`, Mobile-NixOS, `.#mobile-image`	Mining forced off (phones never mine); ships the wallet engine byte-identical to the node's; SSH and docs off for battery/privacy hygiene; telephony wired over D-Bus only (ModemManager + callaudiod as separate GPL processes, never linked in — a deliberate license boundary) via `crates/postern-telephony`.	WRITTEN, NOT BUILT HERE · **never booted on any platform, not even QEMU** — the single least-proven artifact in the OS tier · touch-wallet UI not shipped in this profile (engine only).

The Postern Browser layer (`os/browser.nix`) hardens Firefox via locked enterprise policy + preferences: telemetry, Pocket, and form history disabled; tracking protection (including cryptomining/fingerprinting lists) locked on; DNS-over-HTTPS on; uBlock Origin installed and locked; `privacy.resistFingerprinting = true`; total cookie protection

(`network.cookie.cookieBehavior = 5`); `dom.security.https_only_mode = true`; WebRTC local-IP leak closed (`media.peerconnection.ice.default_address_only` / `ice.no_host`); geolocation disabled.

3.4 postmarketOS community edition

LIVE ARTIFACT PUBLICATION HARDWARE-GATED (NEVER DEVICE-BOOTED) — `pmos-overlay/` builds three signed apks (`postern-wallet`, `postern-apps`, and a `postern` meta-package) for postmarketOS's phosh flavor, with nftables + Tor and first-run wiring. They are signed and published to a live aarch64 apk repository — but have **never booted on a real postmarketOS device**; the `postern` meta-package only resolves once a device is already running a booted pmOS. This tier makes **no reproducibility or attestation claim by design** — it uses a software keystore and is explicitly the non-attestable, community-maintained edition, distinct from the measured-boot Postern OS proper.

3.5 Boot-verification infrastructure and the reproducibility gap

An aarch64 QEMU boot harness exists (`os/boot-harness/boot-qemu-aarch64.sh` + `check-boot-log.sh`, exposed as `apps.aarch64-linux.boot-qemu` in the flake) with a marker ladder (bootloader → kernel → stage-1 → systemd → login) — but it has **never been run** in this review (no Nix/QEMU on the reviewing host). Its own header states it "proves the software boot chain only — QEMU ≠ hardware." No equivalent x86_64 harness exists in-repo. The project's headline claim — **reproducibility** — requires two independent builders producing a bit-for-bit-identical image; with no committed `flake.lock`, this has not yet happened for any image. REPRODUCTION IS VM-GATED ; a second, independent builder confirming a matching hash remains outstanding.

Threat model — Postern OS

Boundary	What is enforced there	What is explicitly out of scope
Disk / rootfs	dm-verity refuses to mount a rootfs whose content hash doesn't match the sealed roothash — tamper-evident at the block-device layer.	An attacker with physical access before first LUKS provisioning, or a firmware-level implant beneath the measured chain, is out of scope of dm-verity alone.
Firmware → kernel	UKI measured as one signed PE into TPM PCRs — a modified kernel/initrd/cmdline changes the PCR values.	No Secure-Boot signing chain is shipped yet (Microsoft-endorsed or self-enrolled keys) — a user must currently disable Secure Boot or enroll their own keys; "signed-shim support is planned, not shipped."
Network	Deny-all inbound firewall by default; Tor client bundled on desktop; DoT+DNSSEC.	Telephony (SMS/voice) is plaintext to the carrier and lawful-intercept-accessible by design — the phone profile hardens the local client only, never wire privacy to the tower; location/IMEI/IMSI remain visible to the carrier.

4 Postern Seal — remote attestation

HOST-PROVEN (DEFAULT BUILD + SNP CRYPTO VS. FIXTURE)

HARDWARE-GATED (NO GENUINE VERDICT WITHOUT REAL TEE/DEVICE HARDWARE) UNAUDITED — the crate `crates/postern-seal-companion` ships **two structurally distinct verifiers** that should not be conflated: a lightweight **pinned-device-key model** that compiles and is tested by default, and **hardware-TEE-root models** (SEV-SNP, TDX, Android StrongBox) gated behind Cargo features, each with real cryptography that has been exercised only against offline fixtures — never against a live TEE or device this project launched.

4.1 The default build: pinned-device-key attestation (`lib.rs`)

`AttestationReport` { os_roothash: [u8; 32], measurement: Option<Vec<u8>>, nonce: [u8; 32], pubkey: Vec<u8>, signature: Vec<u8> }
`Expected` { os_roothash: [u8; 32], measurement: Option<Vec<u8>>, challenge: [u8; 32], trusted_pubkey: Option<Vec<u8>> }
`Verdict` — enum: `Genuine` | `Mismatch(&'static str)` | `Unverified(&'static str)`

`fn verify_report(report, expected) -> Verdict` runs six ordered, fail-closed checks — the first failure short-circuits with a named reason:

1. Is a `trusted_pubkey` supplied at all? If not: `Unverified("no trust anchor…")` — the report can never authenticate itself against its own embedded key.
2. `report.os_roothash == expected.os_roothash`, else `Mismatch("os_roothash — not the published genuine image")`.
3. Measurement equality, if `expected.measurement` is present, else `Mismatch("tee measurement…")`.
4. `report.nonce == expected.challenge`, else `Mismatch("nonce — stale or replayed report")`.
5. `report.pubkey == anchor`, else `Mismatch("device key — not the pinned/trusted device")`.
6. Signature check via the hybrid post-quantum verifier (`bloch_crypto::crypto::verify` — Falcon‖ML-DSA), else `Mismatch("signature — attestation not signed by the trusted key")`. Only if all six pass: `Genuine`.

The signed transcript is produced by the shared `postern-core::seal` module, domain-separated so an attestation signature can never be replayed as some other Postern signature:

```
const ATTESTATION_DOMAIN: &[u8] = b"postern-seal/attestation/v1";
// wire layout:
DOMAIN(29B) ‖ os_roothash(32B) ‖ u32-LE measurement-length ‖ measurement bytes ‖ nonce(32B)
```

Channel binding (`binding.rs`) uses a fixed formula shared by both the SNP and TDX report-data fields (each 64 bytes on the wire):

```
report_data = SHA-256(nonce ‖ guest_pubkey) ‖ 0x00×32   // REPORT_DATA_LEN = 64
```

> This pinned-device-key path is a real, host-testable verifier — but it is a **different trust model** from a hardware TEE root: it authenticates a report against a key you already trust, not against silicon. The project's own attestation design document sketches a richer `image_digest` / `hostdata` / `tee`-bearing report type intended to unify the two — that richer type was **not found implemented anywhere in the reviewed Rust source**; treat it as aspirational design, not shipped code.

4.2 SEV-SNP verification (`snp.rs`, feature `sev-snp`)

Built on the `virtee/sev` crate with pure-Rust crypto (no OpenSSL). Key types:

> **`AmdGeneration`** — Milan | Genoa | Turin (3rd/4th/5th-gen EPYC)
> **`MinTcb`** { bootloader: u8, tee: u8, snp: u8, microcode: u8 } — component-wise anti-rollback floor
> **`SnpExpected`** { generation, measurement: [u8;48], host_data: Option<[u8;32]>, report_data: [u8;64], min_reported_tcb: Option<MinTcb>, vmpl: Option<u32> }
> **`SnpVerdict`** — Genuine | Rejected(&'static str)

The AMD root of trust is **compiled in** (vendored ARK/ASK from `virtee/sev`'s `certs::snp::builtin::{milan,genoa,turin}`) — "the wire never supplies the root." `verify_snp_report(report_bytes, vcek_der, expected)` runs eleven ordered checks:

1. Parse the report structurally.
2. Load the pinned AMD CA chain for the claimed generation.
3. Parse the presented VCEK certificate.
4. **VCEK → ASK → ARK** chain verification (ARK self-signature, ARK→ASK, ASK→VCEK).
5. VCEK signs the report body.
6. Guest policy forbids debug access.
7. VMPL matches the expected privilege level (if pinned).
8. Measurement equals the expected launch measurement.
9. HOSTDATA equals the expected admission-policy hash (if pinned).
10. **report_data** equals the channel-binding value (nonce/pubkey binding).
11. Reported TCB satisfies the pinned minimum-version floor.

An opt-in AMD-KDS CRL path (`KdsCrl`, `check_kds_crl`, `verify_snp_report_with_crl`) checks the VCEK's serial against a fetched CRL, verified against the same pinned ASK, with freshness enforced by the CRL's `next_update` field —

but the crate does **no network I/O itself**; a caller must fetch the CRL out of band from AMD's KDS endpoint. The module's own comment is explicit about the gap this leaves open: "*the pinned ARK/ASK are AMD's published 2020-era long-lived keys, so a plain* `verify_snp_report` *still accepts a since-revoked VCEK*" unless the CRL path is used.

Live report acquisition (`request_bound_evidence` / `request_ext_report` / `request_raw_report`) reads `/dev/sev-guest` via real ioctls on Linux, and fails closed everywhere else (`ReportError::UnsupportedPlatform` off-Linux, `DeviceUnavailable` on Linux without a real SNP guest).
THE CRYPTO PATH IS PROVEN OFFLINE AGAINST A COMMITTED, REAL MILAN REPORT+VCEK TEST FIXTURE — but THIS CODE HAS NEVER EXECUTED INSIDE A REAL SEV-SNP GUEST THIS PROJECT LAUNCHED, and neither have its documented negative controls (stale nonce, MITM pubkey, wrong generation, off-by-bit measurement, sub-floor TCB, debuggable policy).

4.3 Intel TDX verification (`tdx.rs`)

A DCAP v4 TD-quote structural parser ships by default (constants: `TEE_TYPE_TDX = 0x00000081`, `QUOTE_VERSION_V4 = 4`, `HEADER_LEN = 48`, `TD_BODY_LEN = 584` for TDX 1.0 — TDX 1.5's longer body, +64 bytes for `MRSERVICETD` and a second TCB SVN, is not decoded and deliberately fails the strict length check). This structural parser **authenticates nothing by design** — a forged quote parses fine.

The real chain-verification path lives behind feature `tdx-verify` (submodule `dcap`): PCK certificate → Intel processor CA → a pinned Intel SGX Root CA, QE-report binding, CRL freshness, and a TCB-SVN floor. Its central constant, as found in source:

```
pub const PINNED_INTEL_SGX_ROOT_CA_DER: &[u8] = &[];  // EMPTY placeholder
```

> **Because this constant is an empty byte slice, `verify_td_quote` and `verify_td_quote_bound` can structurally never return `Genuine` on this codebase today — every call fails closed with `Unverified`, even for a synthetic quote whose every pinned field matches (this is a load-bearing test in the crate: `well_formed_full_match_still_fails_closed`). The chain-verification crypto itself (`verify_collateral_anchored` — cert-chain walk, dual CRL checks, QE-report binding, ECDSA-P256 signature checks, TCB-SVN floor) is real and exercised in tests, but only against a synthetic root generated for the test, never Intel's actual root or real TDX silicon.**

4.4 Mobile key attestation (`mobile.rs`, feature `mobile-attest`)

Parses the Android Key Attestation X.509 extension (OID `1.3.6.1.4.1.11129.2.1.17`) to a caller-supplied list of pinned Google root certificates — `MobileExpected.roots` empty ⇒ fail closed by design. Checks, in order: nonce match, a `SecurityLevel` floor (`Software < TrustedEnvironment < StrongBox`, ordinal-comparable), hardware-enforced verified-boot state and device-lock flag (read only from the TEE-enforced authorization list, never the software-enforced one), OS patch-level anti-rollback, and app identity (package names + signing certificate digests). The module states plainly that it attests the phone, not the workspace running on it, performs no revocation/CRL check (a compromised-and-revoked device key would still chain), and that StrongBox itself is a classical P-256 hardware security module — it does not hold or sign with the Falcon‖ML-DSA key, only attests the device and wraps that key at rest. NEVER RUN AGAINST A REAL DEVICE ATTESTATION.

4.5 The seal-gate protocol (`gate.rs`) — wire format

A TEE-independent, hostile-input-tested wire codec used by the guest-side gate binary and its client (both feature-gated, real binaries: `postern-seal-gate`, `postern-seal-verify`):

```
Client → gate :  "PSG1" ‖ nonce(32B)                          // challenge, 36 bytes total
Gate  → client:  "PSE1" ‖ len(report) u32-LE ‖ report
                        ‖ len(vcek)   u32-LE ‖ vcek DER
                        ‖ len(pubkey) u32-LE ‖ guest pubkey      // evidence
```

Hard per-field caps: `MAX_REPORT_LEN = 16 KiB`, `MAX_CERT_LEN = 8 KiB`, `MAX_PUBKEY_LEN = 1 KiB` (a real SNP report is ~1.2 KiB and a VCEK ~1.3 KiB — anything near these caps is already garbage). Every length prefix is bounds-checked before the corresponding bytes are allocated or sliced; trailing bytes after the last field are rejected (`WireError::TrailingBytes`) to refuse smuggled data. On the guest side, `postern-seal-gate` listens on `0.0.0.0:16510` and, per connection: reads the 36-byte challenge, derives `report_data = SHA-256(nonce ‖ wg_pubkey)`, fetches the SNP extended report via `/dev/sev-guest` at VMPL 0, and replies with the encoded evidence — or, on any failure, closes the connection without replying ("do not reply with anything unverifiable"). `postern-seal-verify` is the client: it mints a fresh nonce, sends the challenge, decodes the evidence, and calls `verify_gate_evidence` — which overwrites `expected.report_data` with the freshly computed channel-binding value before delegating to `verify_snp_report`, so "the ONLY key it is sound to open the tunnel to is `evidence.guest_pubkey`" on a `Genuine` verdict.

Threat model — Postern Seal

Trust anchor	What it proves	What it does not prove
Pinned device key (default build)	The report was signed by a specific, previously-trusted key, is fresh (nonce-bound), and matches a specific claimed roothash/measurement.	Nothing about the underlying hardware — this model trusts a key, not silicon. No revocation of a compromised device key.
AMD VCEK → ASK → ARK (SEV-SNP)	A cryptographically real chain from AMD's root, binding the report to a specific platform TCB, measurement, HOSTDATA, and channel nonce.	Anything not yet run on real SEV-SNP hardware Postern launched; a plain (non-CRL) call still accepts a since-revoked VCEK by default.
Intel PCK → SGX Root CA (TDX)	The chain-verification crypto is real and tested against a synthetic root.	Any real verdict — the pinned root is an empty placeholder today, so no TDX quote can ever verify as `Genuine` on this codebase as reviewed.
Google Android root (mobile)	The device (not the workspace) holds a hardware-backed key at a given security level, with a given verified-boot state and patch level, for a given app identity.	Revocation of a compromised device; workspace/OS integrity above the key itself.

5 The enterprise license spine

MASTER-LICENSE GENERATOR: LIVE CRL, D&B KYC, PERSON-KYC REVIEW: CODE COMPLETE, NOT FULLY WIRED UNAUDITED — the license spine (`functions/license/*.js`, mirrored by a Node/TypeScript library at `enterprise/licensing/src/*.ts`) issues, signs, and revokes offline-verifiable licenses. Its central honesty finding: **the cryptographic logic is consistently ahead of the deployment wiring** — the hard parts are done and tested, the gaps are almost entirely "is a durable store or endpoint actually plugged in."

5.1 Token shapes

`MasterPayload` { v: 1, typ: "master", tier: "corporate-master"|"business-freemium", duns: string|null, seats: number|"unlimited", iat, exp, master_pub (base64 Ed25519 pubkey), jti?, kid?, iss?, dnb?: DnbDecision, attestation?, kyc? }
`SubPayload` { v, typ: "sub", master_pub, sub, iat, exp (≤ master.exp), jti?, clearance?, device?, iss? }

There is deliberately no standard-JWT vocabulary — no `aud`/`nbf`; `role` exists only nested under `attestation.role` ("owner"|"manager"), never as a top-level claim. Two envelope shapes coexist and a client must handle either: the TypeScript library's generic `SignedToken<T> = { payload: T; sig: string }`, versus the deployed Pages Functions' flat `{ master_token: <payload>, master_sig: "<base64>" }`.

5.2 Signing: canonical JSON over Ed25519

Canonical-JSON serialization (implemented independently three times in the codebase, kept in lockstep) recursively sorts object keys, preserves array order, and drops `undefined`:

```
function canonical(obj) {
  if (Array.isArray(obj)) return "[" + obj.map(canonical).join(",") + "]";
  if (obj && typeof obj === "object")
    return "{" + Object.keys(obj).sort()
      .map(k => JSON.stringify(k) + ":" + canonical(obj[k])).join(",") + "}";
  return JSON.stringify(obj);
}
```

The signing key (`VENDOR_PRIV_PKCS8_B64`) is a base64 PKCS8-DER Ed25519 private key held as a Cloudflare Pages secret, imported only via Web Crypto (`crypto.subtle.importKey("pkcs8", …, {name:"Ed25519"}, false, ["sign"])`) — no third-party JS crypto library anywhere in the signing path. The vendor public key is hard-coded raw base64 in the deployed code; `kid` is deterministic (first 16 hex chars of SHA-256 over the vendor pubkey's base64 form). Org master keypairs are minted per-issuance server-side and the private key returned to the caller exactly once, with an explicit "we do not store it" notice.

5.3 CRL / revocation

`CrlPayload` { v: 1, typ: "crl", iss, kid, version, issued_at, next_update, max_stale_secs, revoked_jti: string[], revoked_master_pub: string[] }

```
CRL_DEFAULT_NEXT_UPDATE_SECS = 24 * 3600         // publish daily
CRL_DEFAULT_MAX_STALE_SECS   = 14 * 24 * 3600     // 14-day soft-offline grace
```

`crlFreshness(crl, now)` returns one of three states: fresh (before `next_update`), ageing (soft-offline grace, up to 14 days past), or stale ("revocation can no longer be trusted — enterprise features degrade until refreshed"). `pickNewerCrl` is a strict version-only anti-rollback merge (ties favor the freshly fetched CRL); a corrupt or foreign CRL is treated as absent, not fatal — "it can never brick an otherwise-valid license." `checkRevocation(crl, master, sub?)` checks exactly three keys in order: `master.master_pub` (whole-org kill), `master.jti`, then `sub.jti` if present — never DUNS, never `kid`.

> Deployment gap. `checkRevocation` is exported and tested but, as reviewed, unreferenced by any deployed verifier — no shipping install currently fetches or enforces the CRL against a live license. The admin-write path additionally depends on an admin bearer token that is unset in the deployed configuration as reviewed, so admin-driven revocation returns an inert response; only renewal-driven auto-revocation (on a failed KYC re-check) can populate live CRL state today. State itself is an in-memory, keyed-by-token-signature-hash store — not yet a durable, server-signed store, matching the project's own "merged but not yet deployed" self-label.

5.4 D&B Direct+ KYC gate and the LoA ladder

OAuth2 client-credentials flow against D&B's real API surface:

```
DNB_AUTH_URL = "https://plus.dnb.com/v3/token"
DNB_DATA_URL = "https://plus.dnb.com/v1/data/duns"
GET `${DNB_DATA_URL}/${duns}?blockIDs=companyinfo_L1_v1`   // company info only — no principal/officer PII by de
```

Fail behavior is explicitly three-way: verified (proceed), not-found (D&B answered and the entity doesn't exist — hard refuse, and at renewal, revoke), and unavailable (unconfigured credentials, auth failure, network error — fails open to a `"format-only"` issuance, flagged for manual review at the 6-month renewal). The decision is bound directly into the signed token's `dnb` field so it cannot be silently inflated after the fact. As reviewed, the D&B client credentials are unprovisioned on the live deployment, so every issuance today runs the format-only path. DUNS itself is checked only by shape — `/^\d{9}$/` — no checksum, a documented "honor system" gap.

A Level-of-Assurance ladder (server-authoritative in `review.js::computeLoa`, mirrored on-device in `kyc-loa.mjs`) gates what a license can claim, hard-clamped 0–3:

LoA	Basis
0	Self-declared only.
1	Corporate-domain email proof (control of the organization's own email domain).
2	LoA-1 plus either a D&B principal-name match or on-device OCR field validation.
3	On-device OCR plus liveness/authenticity checks, or a full external identity-verification pass.

A privacy tripwire (`scanForPII`) runs before anything else touches an inbound request: a ~35-key field-name blocklist (image, document, ocrtext, name, ssn, taxid, cpf, cnpj, email, …), a 512-character string-length cap (long strings are presumed image/text blobs), and rejection of any `data:` URI — enforcing that "document images, raw OCR text, names, and document numbers never reach this server," an LGPD/GDPR-minimization design choice, not merely a policy statement.

5.5 End-to-end license flow

1. Client submits a KYC review request; `scanForPII` refuses anything carrying documents or raw personal data; the LoA is computed server-side and a verdict reference stored, TTL-bound.
2. Client requests a master license; DUNS format is checked; D&B is queried (fail-open/fail-closed as above); the stored KYC reference is resolved and clamps the token's asserted LoA — "a client cannot inflate its own LoA."
3. An org keypair is generated server-side; the payload is canonical-JSON-signed with the vendor's Ed25519 key; the private key is returned once, unstored.
4. Sub-licenses can be minted from the master (`mintSub`), each capped to the master's own expiry.
5. Clients periodically fetch the CRL, verify its signature against the hard-coded vendor public key, check freshness, and merge it with any cached copy via the anti-rollback rule.
6. At the 6-month renewal mark, D&B is re-checked; a `not-found` verdict revokes the master's own key going forward.

Threat model — license spine

The signing key is the sole point of failure for issuance integrity; it is a Cloudflare Pages secret, never present in client code. CRL freshness gracefully degrades ("ageing" then "stale") rather than failing hard, trading strict revocation guarantees for offline usability — by design, revocation only ever *removes* trust, never grants it, so a corrupted or missing CRL cannot forge a license. The live deployment gap (§5.3) means that, as reviewed, a revoked license is not actually rejected by any shipping verifier yet — this is the sharpest gap between "cryptographically sound" and "operationally enforced" in this subsystem.

6 Consiglio — the admin console

CRATE: HOST-PROVEN, 12 PASSING TESTS | PRODUCT WIRING: DEPLOY-GATED | UNAUDITED —

`crates/postern-consiglio-auth` implements wallet-as-login (hybrid post-quantum signature verification in place of a password) and dual-control governance for enterprise administration. As a crate it is real and tested; as a shipping product, nothing deployed currently drives it end-to-end.

6.1 Wallet-login verification

`server.rs::verify_response` runs five ordered, fail-closed checks (not-enrolled → wrong-key → expired → nonce-unknown/replay → signature) before delegating the actual cryptographic check to `bloch_crypto::crypto::verify` — the same verifier the underlying node runs, not a bespoke implementation. Critically, a bad signature does not retire the nonce (a failed attempt must not burn the user's challenge):

```
let ok = bloch_crypto::crypto::verify(&response.public_key, &transcript, &response.signature);
if !ok { return Err(AuthError::BadSignature); }  // nonce stays live on failure
```

The hybrid post-quantum scheme is implemented as ML-DSA-65 (`pqcrypto_mldsa::mldsa65`, NIST FIPS 204) and Falcon-1024 (`pqcrypto_falcon::falcon1024`), combined as a sequential, short-circuiting AND — both signatures must independently verify, behind a 2-byte suite envelope:

```
const SUITE_MAGIC: [u8; 2] = [0xB1, 0x0C];
const SUITE_MLDSA65_FALCON1024: u16 = 0x0001;   // today's hybrid
const SUITE_MLDSA65_ONLY:       u16 = 0x0002;   // the Falcon-removed suite
```

There is no dedicated `HybridSignature` struct in the reviewed crate — the combined signature is concatenated raw bytes behind that suite-id envelope. Approximate key/signature sizes per the crate's own documentation: ML-DSA-65 pubkey 1952 B / secret 4032 B / signature 3309 B; Falcon-1024 pubkey 1793 B, signature variable-length.

The domain-separated login transcript (`challenge.rs`):

```
const LOGIN_DOMAIN: &[u8] = b"consiglio:wallet-login:v1";
// SHA3-256( tag ‖ len(org)‖org ‖ len(user)‖user ‖ nonce(32B) ‖ issued_at:i64-LE )
```

Nonce lifecycle: `Challenge { org_id, user_id, nonce: [u8;32], issued_at, expires_at }`, default TTL 120 seconds; issued nonces move to a "consumed" set only on success, retained there for 240 seconds (2× the TTL) specifically to still catch a still-fresh-looking replay. A session is `{ id (128-bit hex), org_id, user, role, issued_at, expires_at }`, default TTL one hour, and is revoked by removal from the session map — an expired or unknown id always yields "no session" (fail-closed).

6.2 Dual control / Brewer-Nash separation-of-duties

Roles carry an explicit conflict relation:

```
fn sod_separated_from(&self, other: Role) -> bool { self.sod_conflicts().contains(&other) }
// ItAdmin and ComplianceOfficer are each other's sole SoD conflict
```

`approve_change` enforces, in order: self-approval refused, approver must hold an admin role, and the proposer/approver pair must be a separated pair (one IT-admin, one compliance-officer) — a same-role or non-admin approval is rejected. On any check failure the pending change is left untouched (fail-closed); only after all gates pass is it removed from the pending queue. Offboarding a user (`offboard_and_revoke`) removes their enrollment, kills every open session, and purges any outstanding challenge for them in one call.

> All Consiglio state — enrollments, nonces, sessions, pending changes — lives in in-memory `HashMap`s as reviewed; a snapshot/restore seam (`persist.rs`, round-trip tested) and a file-backed store (`store.rs`) exist, but no networked durable backend (Postgres/KV/Redis) is wired — a real multi-instance deployment has no shared state yet. The crate reaches JavaScript only through a Node bridge (`consiglio-verifier.mjs`) that spawns a compiled `consiglio-verify` binary and throws rather than falling back to a signature stub if that binary is not built — and the browser console currently ships a license-chain Ed25519 verify, not the wallet-login PQ verify (a browser cannot itself run the PQ verifier). WIRED INTO NOTHING THAT SHIPS TODAY.

7 Ezekiel 1 — the governance engine

HOST-PROVEN, ENFORCED & TESTED | PERSONAL HTML CONSOLE NOT YET DRIVEN BY THE CRATE | UNAUDITED

— `crates/ezekiel1` implements classical multi-level-security models (Bell-LaPadula, Brewer-Nash) plus a tamper-evident audit trail, gated by a Personal/Business edition split. The access-control logic itself is real, enforced, and tested; the shipping browser app is a separate, not-yet-connected demonstration surface.

7.1 Bell-LaPadula lattice (`lattice.rs`)

Levels are plain integers, not a dedicated enum: `Lattice { level: BTreeMap<String,u32>, parent: BTreeMap<String,Option<String>> }`.

```
fn may_read (subject_level: u32, resource_level: u32) -> bool { subject_level >= resource_level }  // no read-up
fn may_write(subject_level: u32, resource_level: u32) -> bool { subject_level <= resource_level }  // no write-d
```

`validate_no_widening` rejects any policy where a child layer's level exceeds its declared parent's — a structural, not just runtime, guarantee. This gating applies only for the Business edition.

7.2 Brewer-Nash Chinese walls (`walls.rs`)

`Wall { name, classes: Vec<ConflictClass>, bypass: bool }` — a policy declaring `bypass: true` is rejected at validation time, not merely discouraged. `may_access` denies access to a dataset if it shares a conflict class with a *different* dataset the same subject has already accessed. Lowering a wall is an explicit, audited action on the evaluator (`Engine::lower_wall`), not a config flag — it is per-owner, permanent for the process's lifetime, and always appends to the audit trail.

7.3 SHAKE-256 hash-chained audit trail

Shared with the rest of the suite via `postern_trail`. Exact chain formula:

```
preimage = canonical_json({ content fields…, prev_hash, v })   // "hash" itself excluded from its own preimage
hash     = SHAKE-256( prev_hash_bytes ‖ preimage_bytes )        // 32-byte output → 64 hex chars

fn chain_hash(prev_hash: &str, canonical_json: &str) -> String {
    let mut hasher = Shake256::default();
    hasher.update(prev_hash.as_bytes());
    hasher.update(canonical_json.as_bytes());
    let mut out = [0u8; 32];
    hasher.finalize_xof().read(&mut out);
    to_hex(&out)
}
```

`verify_chain` confirms every link is internally consistent but cannot detect a truncated tail (a valid prefix is itself a valid chain); `verify_chain_to_head` pins an externally-persisted head hash specifically to catch silent rollback or truncation. Every append calls `file.sync_all()` (fsync) before returning. The project states its own limits

plainly: "tamper-evident, not tamper-proof — the owner of the file can rewrite the entire trail and recompute every hash… what it guarantees is that a *partial* edit is caught."

7.4 Tenant isolation (userland, by construction)

`same_firmament(subject_tenant, resource_tenant) -> bool` is a plain equality check, enforced structurally at policy-validate time (no resource id shared across two tenants) and again at runtime for the Business edition. The project's own threat-model statement, carried verbatim in code comments: "*enforced by construction inside a single process… this is not an OS/kernel or hardware boundary… a memory-safety break, a side channel, or a bug outside this module could still bridge tenants.*" The evaluator's default posture is explicit deny: "where the policy is silent, the answer is no," and a grant without a parseable expiry field fails to parse at all, never defaults to "live."

Threat model — Ezekiel 1

Everything above is a userland policy engine inside one process, not an OS-level mandatory access control system — the boundary it draws is only as strong as the process boundary itself. It is honest about this in its own source comments, drawing an explicit parallel to the same caveat carried by the project's key-management layer. The audit trail defends against silent, partial tampering by someone who does not control the whole file; it does not defend against a fully compromised host that rewrites history wholesale.

8 Gramota — KYC onboarding

REFERENCE PROTOTYPE, NOT A DEPLOYED SERVICE UNAUDITED — two related surfaces: `apps/gramota/` is a registration chat-bot that collects DUNS/role/attestation and issues license tokens; `apps/gramota-kyc/` is a distinct on-device document/OCR KYC console. Both run entirely client-side against the license spine's endpoints (§5).

DUNS is checked only by shape (`/^\d{9}$/`) in both apps, consistent with the license spine. The on-device OCR pipeline is a lazily feature-detected integration point, not a bundled dependency:

```
async function runOcr(file) {
  if (window.Tesseract && window.Tesseract.recognize) {
    var res = await window.Tesseract.recognize(file, "eng");   // WASM worker, on-device
    return { text: (res && res.data && res.data.text) || "" };
  }
  return null;   // no engine present -> caller falls back to manual on-device entry
}
```

> No Tesseract.js/WASM build is bundled in the reviewed source — out of the box, "on-device OCR KYC" is manual on-device entry, not automated OCR, until an operator self-hosts a Tesseract build alongside the app. This is a mild but real divergence between the feature's name and its default behavior; both remain strictly on-device either way (document images and raw OCR text never leave the browser).

What crosses the wire is deliberately minimal: a resolve-mode request (`{duns, resolve:true}`) returning public D&B entity data, and a verdict-mode payload built from `buildVerdict()` — `{ duns, role, attestation, kyc: { ocr: { onDevice:true, match, fieldsMatched: ["legalName","country"], liveness } } }` — booleans and field names, never field values. The UI shows this exact payload to the user before it is sent. A separate, regex-based PII redaction path (distinct from the license spine's `scanForPII`) protects an optional bring-your-own-key Claude assist chat inside Gramota, masking DUNS-shaped digit runs and detected names before any text reaches the model.

9 Cloud / Chiostro — the confidential VM

CHIOSTRO CVM: WRITTEN, UNBUILT, UNBOOTED | LICENSE-VERIFICATION SERVICE: LIVE REFERENCE PROTOTYPE | MANAGED-NODE CONTROL PLANE: INTENT-RECORDING ONLY | UNAUDITED — "Cloud edition" is deliberately **two unrelated systems** sold under one name, and this document keeps them separate exactly as the source does.

9.1 Chiostro — the confidential-VM guest you can verify

`os/cloud.nix` layers onto the attested base (§3.2) with cloud-specific configuration:

- Guest kernel modules `sev-guest` and `tdx_guest` are loaded, exposing `/dev/sev-guest` / `/dev/tdx_guest`; virtio drivers are included in the initrd for cloud disks/NICs.
- Firewall admits exactly two ports: TCP 16510 (the seal gate — the *sole* pre-attestation listener) and UDP 51820 (WireGuard). Desktop/SSH access binds only to the WireGuard interface; password SSH auth is disabled.
- A fourth disk partition, `persist` (≥2 GiB, `linux-generic`), is LUKS-formatted on first boot: a 64-byte random key (`/dev/urandom`, base64-encoded) is generated in-guest, then TPM2-sealed to PCR 0 (firmware/UKI) and PCR 7 (secure-boot state) via `systemd-cryptenroll --tpm2-device=auto --tpm2-pcrs=0+7`. Subsequent boots unlock automatically only if those PCR values still match.
- WireGuard keys are generated in-guest on first boot (`wg genkey` / `wg pubkey`) and written to the encrypted persist volume — the host never sees the private half. A client allowlist file is treated as untrusted and deny-only: an empty or absent allowlist means the tunnel admits no client (safe default), and a malicious host can at most withhold legitimate access, never grant itself entry, because the tunnel terminates at the attested in-guest key.
- Explicitly and deliberately absent: any cloud-provider guest agent or cloud-init-style host-writable command channel — the only launch-time input the guest consumes is the (untrusted, deny-only) client public-key allowlist.

> Chiostro has never been built or booted in this review. Every element above is a Nix module written on a non-Nix host; it requires a Linux+Nix host to build and real SEV-SNP/TDX hardware to boot and produce a genuine launch measurement. Per-provider firmware/measurement/HOSTDATA details are correspondingly unverified — the project's own hardware runbook explicitly scopes a first attempt to GCP N2D Milan instances and states plainly that Azure's vTPM/paravisor path is not yet supported: "do not claim Azure end-to-end."

9.2 Hosted managed services — a separate, running system

These are Node.js reference-prototype services under `enterprise/cloud/services/`. Only the license-verification service is deployed and live; this document distinguishes it sharply from Chiostro because it performs zero TEE or attestation cryptography — it is a licensing convenience, not the confidential VM.

Endpoint	Behavior, as coded
`GET /v1/vendor-key`	Returns `{ vendor_public_key, fingerprint, note }` — `fingerprint` is the first 16 hex characters (8 bytes) of SHA-256 over the vendor key's base64 string.

Endpoint	Behavior, as coded
`POST /v1/verify`	Real Ed25519 chain verification of a submitted master/sub token pair; response includes an honest `disclosure` field and the same truncated vendor-key fingerprint.
`GET /v1/revocations`	Public, unsigned working-set of revoked hashes/jti/master_pub.
`POST /v1/revocations`	Admin-bearer-gated (constant-time token compare) write to the working set.
`GET / POST /v1/crl`	Serves or accepts a signed CRL; enforces monotonic `version` (rejects a rollback with HTTP 409) and validates `typ:"crl"` against the vendor public key before accepting a write.

The service refuses to start at all if its vendor public key is unconfigured (fail-closed by construction, not merely by convention). Revocation is checked at three granularities — a specific sub-token hash, a 128-bit serial (`jti`), or a whole `master_pub` (an org-wide kill switch) — against both the in-memory working set and any published signed CRL.

A companion relay service (`relay-service.mjs`, opaque end-to-end-encrypted transport for Postern Messenger bodies, sealed-sender mode, deliver-then-drop) exists in code but is not the default deployment target and is not deployed today.

9.3 Managed-node control plane — "a promise, not a proof"

The orchestrator (`orchestrator.mjs`) is explicit about its own honesty: absent an injected operator `driver` object, every `provision()` call records intent only —

```
let handle = null, endpoint = null, simulated = true, state = "pending";
if (driver && typeof driver.create === "function") {
  try { /* real driver call */ simulated = false; state = "running"; }
  catch (e) { state = "failed"; }
}
```

Every unwired node record therefore carries `simulated: true` — no real infrastructure is ever provisioned by the code as reviewed. A companion health API deliberately surfaces this fact (`allNodesSimulated`, `driverWired`) so operators cannot be misled by the API's own shape. Revocation still does real work here: a license revoke/expiry event triggers `deprovisionTenant()`, which tears down every (simulated or real) node and wipes the tenant's isolation namespace.

Threat model — Cloud / Chiostro

The critical distinction a reader must not miss: a live demo endpoint is not evidence the confidential VM is real. The deployed license service does real Ed25519 signature verification and nothing else — no TEE crypto, no attestation. The actual "TEE you can verify" claim rests entirely on Chiostro (§9.1), which has not been built or booted in this review. Running the managed edition at all makes Postern a data processor of customer traffic metadata even where message bodies stay opaque — the project's own disclosure calls this "a promise, not a proof" and states hosting is less private than self-hosting by design.

10 Postern SDK & the shared crypto core

HOST-PROVEN UNAUDITED — `crates/postern-sdk` is a single UniFFI facade over the whole product suite; `crates/postern-core` defines the one canonical AEAD seal and KDF that every other Rust crate in the suite consumes.

10.1 One UniFFI surface for the whole suite

A single macro call at the crate root — `uniffi::setup_scaffolding!()`, behind feature `native` — registers Android/iOS/desktop bindings for the entire suite (wallet, vault, courier, seal, hygiene, panic, deniable, keys) in one place, checked by CI to be the exactly-one registration in the workspace. Secret material is never exposed as a UniFFI `Record` field (whose auto-generated `toString()` would risk leaking it into logs); secrets live behind opaque handles wrapped in `Zeroizing`.

10.2 The canonical AEAD seal (`postern_core::aead`)

Wire format, version 2:

```
0x02 ‖ nonce(24B) ‖ commit(32B) ‖ AEAD-ciphertext(plaintext ‖ 16-byte tag)

const NONCE_LEN:  usize = 24;   // XChaCha20-Poly1305
const TAG_LEN:    usize = 16;
const COMMIT_LEN: usize = 32;
const VERSION_V2: u8    = 0x02;
```

Per-seal subkeys are derived via HKDF-SHA-256, salted by the fresh nonce and domain-labeled so encryption and commitment keys can never collide:

```
enc_key = HKDF(salt = nonce, ikm = key, info = "postern-seal/v2/enc"    ‖ context)
commit  = HKDF(salt = nonce, ikm = key, info = "postern-seal/v2/commit" ‖ context)
```

The 32-byte `commit` value is checked in constant time (`subtle::ConstantTimeEq`) before decryption — a key-commitment defense against "invisible salamander"-style cross-key ciphertext substitution attacks. Additional authenticated data binds `version ‖ AAD_LABEL ‖ len(context) as u64-LE ‖ context`, giving every product its own non-interchangeable context string (e.g. `postern-vault/v1`): a courier blob sealed under `postern-courier/v1` context will not open as a vault blob even under an identical 32-byte key — verified directly by a cross-context test in the reviewed source. A legacy pre-v2 format (bare `nonce ‖ ciphertext`, no context binding) remains openable for backward compatibility but is never written by current code.

10.3 Argon2id KDF — exact parameters

```
// crates/postern-core — kdf module
M_COST: u32 = 65536;   // 64 MiB
T_COST: u32 = 3;
P_COST: u32 = 1;
SALT_LEN: usize = 16;  // RFC 9106 recommendation

// a distinct, lower "floor" tier for memory-constrained targets:
M_COST_FLOOR: u32 = 19456;  // 19 MiB — the argon2 crate's own upstream default
T_COST_FLOOR: u32 = 2;
```

This is the single KDF definition every consumer (SDK, vault, wasm layer) re-exports rather than re-implementing — confirmed identical across `postern-vault::derive_key` and the wasm bindings' `kdf_label()`, which literally returns the string `"Argon2id (m=65536 KiB, t=3, p=1)"`.

10.4 The browser boundary — `wasm.rs`

Compiled only for `wasm32` under feature `wasm`, exposed to pages as `globalThis.PosternSDK`. `VaultKey` holds key bytes as `Zeroizing<[u8;32]>` inside wasm linear memory — never copied onto the JavaScript garbage-collected heap. Hybrid post-quantum *signing* (Falcon-1024‖ML-DSA-65) is structurally unavailable in the browser: the underlying PQClean C implementation does not target `wasm32-unknown-unknown`, so `has_pq_sign()` returns `false` by construction, with the reason surfaced verbatim via `pq_unavailable_reason()` rather than silently degraded. This is the design justification for phone-as-signer (§13) as the intended answer to browser-side post-quantum signing.

> The wasm bundle itself, as reviewed, is not built into the main source tree the apps ship from — it exists only in a separate build artifact location, gitignored from the main tree. Apps that want the real KDF/AEAD probe for an optionally-injected `globalThis.PosternSDK` at runtime and fall back to a labeled classical stand-in when it is absent (§11).

11 Vault apps

REFERENCE PROTOTYPES; CEREMONY LOGIC TESTED, REAL KDF NOT YET ADOPTED BY DEFAULT UNAUDITED — the single-file HTML apps (Chiave / `postern-login`, Sotto, Vedetta, Tolmach, Sarto / `postern-onboard`, Postern Store) share one wiring pattern: strict-CSP client-side crypto by default, upgrading to the real Argon2id/XChaCha20-Poly1305 core (§10) only when a host shell has injected `globalThis.PosternSDK`.

11.1 The classical stand-in, and its exact parameters

App	Default KDF stand-in	Iteration count
Chiave (`postern-login`)	PBKDF2-SHA-256	310,000
Sotto	PBKDF2-SHA-256	310,000
Vedetta	PBKDF2-SHA-256	310,000
Sarto (`postern-onboard`)	PBKDF2-SHA-256	310,000
Tolmach	PBKDF2-SHA-256	600,000 — deliberately higher; the app's own comment states Argon2id "would require a WASM dependency," so it compensates with more rounds

Every app upgrades to real Argon2id(64 MiB, t=3, p=1) + XChaCha20-Poly1305 when the SDK is present — this is a labeled, explicit fallback, never a silent one.

11.2 Chiave — the wallet-login ceremony, in detail

The login challenge is domain-separated and canonicalized before signing:

```
CH_KIND = "postern:login:v1"
CH_TTL_MS = 120000   // 2 minutes
challengeBytes(ch) = utf8( CH_KIND + "\n" + stableStringify(ch) )   // sorted-key JSON
```

The verifier keeps its own nonce book — never trusting client-supplied state — and checks, in order: (1) the nonce exists in the verifier's own pending map and is deleted immediately on lookup, whether the attempt passes or fails (single-use, no retry oracle); (2) expiry against the verifier's own recorded issue time, never a client-echoed timestamp; (3) origin binding — a challenge signed for the wrong origin fails even with an otherwise-valid signature, defeating a phishing relay; (4) only then, signature verification (Ed25519 preferred, ECDSA P-256 fallback — both explicitly labeled as classical stand-ins for the eventual Falcon-1024‖ML-DSA-65 pair). A KDF-downgrade clamp defends the at-rest seal itself: a stored iteration count read back from local storage is clamped to a floor of 310,000 and a denial-of-service-bounding ceiling of 3,100,000, so a local-storage-tampering attacker cannot downgrade the next unlock to a trivially brute-forceable iteration count.

11.3 Other apps and the store catalog

Sotto (voice assistant), Vedetta (markets lookout), and Tolmach (interpreter/minutes) are bring-your-own-Claude-key reference prototypes with disclosed privacy panels — voice input in Sotto goes through the browser's own speech recognizer (Google, on Chromium-based browsers) before reaching the app, not an on-device model; Vedetta's market-data providers see the symbols looked up. The Postern Store catalog (`catalog.json`) is, as reviewed, entirely self-declared unsigned: every entry's `signing.state = "unpublished"`, checksum and signature fields are null, and no app package has actually been built, mirrored, or published through it yet — a design roadmap (internally called "Depot") describes a future TUF-based upgrade using the same Falcon-1024‖ML-DSA-65 pair Consiglio's login uses, but that upgrade is not implemented.

12 Comms — Messenger & Courier

MESSENGER CORE: HOST-PROVEN, DEFAULT-ON | TOR/MATRIX TRANSPORTS: FEATURE-GATED OFF BY DEFAULT | UNAUDITED — two deliberately separate products, warned against conflation in the project's own interop spec: **Postern Messenger** (persistent group chat, vodozemac Megolm) and **Postern Courier** (serverless, single-shot file/message drop, hybrid post-quantum seal).

12.1 Postern Messenger (`crates/postern-messenger`)

The crate ships two independent Megolm code paths on vodozemac 0.10, and only one is the live default. `SecureChannel<S: SessionStore>` is the default, wired path — sessions are created with `GroupSession::new(SessionConfig::version_1())`, i.e. Matrix-interoperable Megolm v1. A second, older `megolm_reference.rs` path exists but its own module doc states it is "NOT wired to any transport… no new callers should adopt it" — superseded by `channel`.

`Frame` { session_id: String, ciphertext: String } — the wire envelope, base64 Megolm ciphertext, capped at `MAX_FRAME_LEN = 1 MiB`
`Decrypted` { plaintext: Vec<u8>, session_id: String, message_index: u32 }

Replay rejection is enforced directly in `SecureChannel::decrypt` against a per-session watermark, and — unlike many reference implementations — survives a restart because the watermark is itself persisted:

```
if !inbound.seen.insert(out.message_index) {
    return Err(ChannelError::Replay { session: frame.session_id.clone(), index: out.message_index });
}
self.persist()?;   // the "seen" watermark is part of the persisted state
```

At rest, sessions are protected by vodozemac's own encrypted pickle — ChaCha20-Poly1305 keyed by a caller-supplied 32-byte `PickleKey` (`Zeroizing<[u8;32]>`). The crate performs no key derivation of its own: in a shipped build that key is unwrapped from the Postern Vault (§10) at unlock. A wrong key produces a hard, explicit error — never a silent mis-decrypt. The crate's own documentation flags one caveat: the replay watermark is "NOT authenticated across a store swap," a rollback edge case.

Two transports are feature-gated off by default:

- `tor` feature — carries the same `Frame`s over a Tor circuit via an in-process Arti client. Content cryptography is unchanged (still classical Megolm — Tor hides metadata, not payload confidentiality); client-only, no onion hosting from the messenger side; circuit isolation is the caller's responsibility.
- `matrix` feature — real federation via `matrix-sdk` (feature `e2e-encryption`): password login, sync, tier-gated Megolm send/receive, and a fail-closed room classifier (`RoomEncryption::{Encrypted, NotEncrypted, Unknown}`) that refuses to send into anything but a clean, unbridged encrypted room — checked twice, once before send and once against the SDK's own post-send encryption info. Explicitly Phase-2, not implemented: cross-signing / device verification (emoji SAS). The crate's own comment: "that check does not exist yet." Session storage is in-memory only in this phase — a fresh device identity every run, no persistence, no Tor wiring for the federated path yet. The only interop test requires a live homeserver and is marked `#[ignore]`.

12.2 Postern Courier (`crates/postern-courier`)

Courier seals one payload to one recipient's public key — no session, no group state. The hybrid post-quantum combiner (labeled "X-Wing-style" in the crate — a custom HKDF combiner following the X-Wing/RFC 9180 pattern, not the literal X-Wing KEM):

```
ikm  = ss_MLKEM-1024(32B) ‖ ss_X25519(32B)
info = HKDF_LABEL ‖ WIRE_VERSION ‖ SHA-256(kem_ciphertext) ‖ SHA-256(recipient_public)
key  = HKDF-SHA256(salt=None, ikm, info)

HKDF_LABEL   = b"postern-courier/hybrid"
WIRE_VERSION = 2
```

The ML-KEM shared secret is zeroized immediately after use ("no early-return gap," per the crate's own test naming); X25519 rejects low-order/non-contributory points on both seal and open. The purpose, stated directly in source: binding the ciphertext to the KEM output defends against a property ML-KEM alone lacks (it is "not MAL-BIND-K-CT"). Payload encryption then delegates to the same canonical AEAD seal as everything else in the suite (§10.2) — XChaCha20-Poly1305, wire v2, key-committing.

An optional `onion-transport` feature (off by default) publishes an ephemeral, single-shot Tor onion service per drop: the sender serves exactly one connection then tears the service down; the recipient fetches once. Real-network delivery is exercised only by an `#[ignore]`d live test. The crate's own stated limits are blunt: no sender authentication ("anyone holding the recipient's public key can produce a valid sealed payload; a successful open proves nothing about who sealed it" — a signed-drop authentication story is planned, not implemented), and no cryptographic replay protection beyond a hard size cap:

```
pub const MAX_DROP_BYTES: usize = 64 * 1024 * 1024;  // 64 MiB, checked BEFORE allocation
```

A captured drop-frame can be re-served verbatim and will still successfully open for the correct recipient key — the crate's own doc states plainly: "no anti-replay/DoS hardening beyond a size cap — the seal, not the wire, is what protects the payload." The only authenticity anchor is an out-of-band recipient-key fingerprint (SHA-256 under a fixed domain tag, truncated to 10 bytes, shown as five hex groups) that must be verified through some other channel.

Threat model — Comms

Messenger's content confidentiality is real and default-on; its two opt-in transports (Tor, Matrix federation) are each individually honest about a specific gap — Tor hides metadata only, Matrix federation has no device verification yet. Courier's payload confidentiality is real and post-quantum-hybrid, but the product deliberately does not attempt sender authentication or replay defense at the protocol layer — both are explicit, stated design choices, not oversights, and any deployment that needs either property must add it out of band.

13 Phone-as-signer

PROTOCOL CRATE: HOST-PROVEN REFERENCE

REAL INTEROP DEMONSTRATED; SOFTWARE KEYSTORE, NOT HARDWARE | UNAUDITED — a phone holds signing keys and approves each transaction after seeing exactly what it signs (WYSIWYS), reached over a paired channel from a desktop that never holds the key. This is presented, and reviewed here, as a **v1 reference implementation with real interop** — not a unified, shipped feature: the desktop and phone ends are not yet a single wire protocol, and the default protocol signer is an explicit non-cryptographic stand-in used only for compile-time interop testing.

13.1 Message shapes (`integration/signer-protocol`)

`SignRequest` { session_id: [u8;16], req_nonce: [u8;32], counter: u64, payload: Vec<u8> }
`Approval` { session_id, req_nonce, counter, signature: Vec<u8> (hybrid Falcon‖ML-DSA), public_key: Vec<u8>, payload_digest: [u8;32] }

There is deliberately no prose/narration field — "the phone parses the payload itself (WYSIWYS), so a compromised desktop cannot narrate a lie." `SignPayload` is an enum of exactly two real intents: `Login(LoginChallenge)` and `Transfer(TransferIntent)`, each with its own domain-separation constant (`SIGNER_DOMAIN = b"postern:signer-protocol:v1"`, and the login path shares Consiglio's own `LOGIN_DOMAIN` from §6.1).

13.2 Replay guard

```
struct ReplayGuard { session_id: SessionId, last_counter: u64, seen_nonces: HashSet<[u8;32]> }
```

`check` runs three fail-closed tests before any human decision is even shown: session mismatch, a counter not strictly greater than the last seen (`StaleCounter`), or a previously-seen nonce (`ReplayedNonce`). State only advances in a separate `commit` step, called only after both the check passed and the human approved — a declined request signs nothing and does not advance the sequence, so a user who says no cannot be used to burn a legitimate later request. A reconnect watermark is authenticated (not merely stored) with a SHA3-256 MAC over a fixed context string, so a tampered watermark is detected rather than silently trusted — but the recommended reconnect model is a fresh forward-secret re-handshake per session, not watermark restoration.

13.3 WYSIWYS and the three consequence tiers

```
enum ConsequenceTier {
    SilentReadOnly,   // read-only, no key use — never prompts, never signs (currently unreachable — no read-only
    SingleTap,        // scoped, cannot move funds (wallet-login)
    ExplicitReview,   // irreversible / moves funds — full decode + deliberate confirmation
}
```

A pure rendering function maps every request to its tier and a fixed headline sentence: a login request always renders as `SingleTap` with `can_move_funds = false`; a transfer always renders as `ExplicitReview` with

`can_move_funds = true`. Both screens (desktop and phone) additionally show a short human-comparable confirmation fragment — a 6-hex-character digest of the session and nonce — so a user can visually cross-check the two devices are talking about the same request.

13.4 Grants and revocation

`GrantScope` has exactly one variant, `LoginTo { org }` — there is no blanket "approve everything" scope. Every grant carries a mandatory expiry; `revoke()` is a one-way flag with no un-revoke path. Value transfers are structurally excluded from ever being grant-covered — only a login request tagged `SingleTap` can ever match a grant. This is self-labeled "enforcement by construction" — a userland, in-process check, not an OS or kernel boundary.

13.5 Pairing (`integration/pairing`) — QR + spoken-code, step by step

1. Desktop generates an ephemeral keypair and a fresh 32-byte nonce, computes a channel-binding value, and renders it as a QR code.
2. The phone scans the QR out-of-band (camera).
3. Phone recomputes the channel binding before performing any key encapsulation, then encapsulates to the desktop's ephemeral public key (ML-KEM-1024 ‖ X25519 hybrid) and derives a short authentication string.
4. The KEM ciphertext and the phone's identity key are sent back to the desktop.
5. Desktop decapsulates and derives its own copy of the short authentication string.
6. A human compares a 6-digit code shown on both screens (`SAS_MODULUS = 1,000,000`, HKDF-derived under domain `b"postern-pairing/sas/v1"` — explicitly documented as not a full PAKE).
7. Only on explicit human confirmation does the channel open and a `PairingRecord` get written; a "no" produces nothing.

The resulting session key (`session_root`) is explicitly documented as "NEVER the wallet seed" — it is an ephemeral, pairing-derived key used only to secure the signer channel, kept separate by construction from the actual signing key material. Paired-terminal records are stored key-committing-sealed at rest; revoking a terminal wipes the record and cannot be undone. Reconnection defaults to a fresh forward-secret handshake each time rather than resuming old state.

13.6 `mobile/signer` — the real signing adapter

`WalletCoreSigner` is a real adapter forwarding to the underlying hybrid post-quantum wallet core (Falcon‖ML-DSA signing math lives one layer down, in the shared wallet crate, not stubbed here) — golden-tested end to end against the same verifier the node itself runs. Per-signature human presence is enforced structurally: a `PresenceProof` (biometric or device-PIN method, bound to the specific pending request nonce) is checked before `approve()` proceeds, and the type is deliberately non-serializable with no default value — "there is no blanket or ambient approval." For a raw transaction signing request, the phone decodes the real transaction bytes itself — never trusts desktop-supplied prose — and structurally refuses to describe or sign any multi-output transaction,

forcing exactly one recipient/amount pair per request. Every transaction confirmation screen carries a fixed disclosure: this is a software keystore, explicitly not a hardware wallet.

13.7 The structural key-denylist

Secret types (seed phrases, derived secret keys) implement no serialization trait at all — the only accessor is a borrow, never an owned copy that could be logged or serialized. A crate-sealed marker trait enforces this at compile time: only types the protocol crate itself blesses can be sent over the wire, and that blessing requires implementing `Serialize`, which the secret types structurally lack — so a secret type accidentally added to a wire message fails to compile, not merely fails a test. A separate runtime "golden" test independently confirms this empirically: it runs a full sign flow with a known test seed, serializes every protocol message the flow produces, and byte-scans the output in three encodings (raw, hex, JSON array) for any trace of the seed or its derived secrets — with a companion meta-test that deliberately plants a leak to prove the scanner would actually catch one.

Threat model — phone-as-signer

The strongest guarantee in this subsystem is the compile-time-enforced impossibility of a secret key crossing the wire, backed by an empirical scanner. The weakest link, stated plainly by the project itself: the phone-side keystore is software, not hardware-backed — a rooted phone can forge the approval UI a real user would see, and the desktop and phone reference implementations are not yet unified into a single production wire protocol.

14 Bridge apps — Porog, NoporIS, Sloboda

DESIGN-ONLY FOR ANDROID ISOLATION/ENFORCEMENT

CONSENT/DECISION KERNELS: HOST-PROVEN, UNIT-TESTED UNAUDITED — three provisionally named ("pending trademark clearance") folklore-tier products that extend reach onto stock Android without claiming to be Postern OS. The recurring, self-disclosed pattern across all three: **the pure decision/consent logic is real code with real tests; the Android-side enforcement (device-policy calls, container boot, live hardware attestation) is a stub, a NoOp, or entirely absent.**

14.1 Porog — the managed-profile container ("the door")

A hardened, self-enrolled Android managed-profile workspace on stock Android. `AttestationGateway.kt`, the intended StrongBox attestation bridge, is a pure interface plus a no-op stub — its own doc comment: "scaffolding to the interface; not an implementation. Design only, unaudited, never run on a device." Separately, `StrongBoxKey.kt` (in a different part of the tree) makes real Android Keystore calls: it generates a StrongBox-backed secp256r1 key with an attestation challenge, requests the certificate chain, and exposes the leaf-first DER chain for verification — but it is honestly documented as a classical P-256 hardware security module that does not hold the post-quantum wallet key in hardware, and it is not wired to the attestation gateway — the intended bridge class does not exist in the reviewed source.

The certificate-chain *validation* logic that would consume that output is real and fixture-tested Rust (§4.4's mobile-attestation module) — but it is not called from any Kotlin code in this review; it awaits FFI wiring. The cross-profile capability-reconciliation logic (which capabilities cross the personal/work boundary, under what user consent, mapped to real Android DevicePolicyManager primitives such as cross-profile intent filters and clipboard-copy restrictions) exists as a dependency-free, fail-closed JavaScript decision kernel with real logic — but no Kotlin device-policy controller in the repository reads that state or calls the underlying Android APIs yet. A separate Tor-egress glue crate implements a fail-closed "deny until a circuit is confirmed up" decision by construction, but its live Tor-circuit code path is not built in the reviewed environment.

14.2 NoporIS — the dual-persona bridge ("the crossing")

A per-crossing consent layer joining a personal ("Clean") persona and a second, Google-Play-enabled persona on one device — explicitly sold as separation, not privacy, since the Google-side persona is, by the project's own design docs, always non-attestable and phones home to Google.

`ConsentRecord` { id: [u8;16], session_id, capability, direction, scope, issued_at_unix, expires_at_unix, peer_device, signer_pub: [u8;32], signature: [u8;64] } — Ed25519-signed under a fixed domain tag
`CrossingRequest` { session_id, req_nonce: [u8;32], counter: u64, capability, from_persona, payload } — capped at 16 KiB, guarded by the same counter/nonce replay pattern as phone-as-signer (§13.2)

A `ConsentRecord`'s check runs five ordered, fail-closed steps (signature → session → direction → capability → time), and only a dedicated consent authority can mint one — a bare consent *request* from the far side of the crossing structurally has no signature field and can never mint its own approval. Capability enumeration is a fixed, closed set (clipboard, files, microphone, camera, location), each mapped to one of the two consequence tiers from §13.3. A hash-chained audit trail records every crossing.

> The actual isolation substrate — the code that would create a managed profile or start an Android container per persona — is explicitly stubbed in the reviewed source, with in-line comments stating plainly that the stand-in backends "isolate nothing: no managed profile is created, no container is started." The reconciliation logic that decides *when* isolation should change state is real; the backend that would *execute* that decision is not implemented. A companion on-device JavaScript state kernel is real and tested but its own documentation states the on-device wrapper that would consume it does not exist yet.

14.3 Sloboda — the Waydroid escape hatch ("the outsiders' quarter")

A full Android/AOSP userland running inside a Waydroid-based LXC container on the Linux host, for the Android apps a native port cannot cover — and, by explicit design, the one tier that is never covered by the Postern Seal. The NixOS module enabling it (kernel Binder support, Waydroid, LXC virtualization) is real, idiomatic configuration — but its own header states it has "never been evaluated or built… nothing has booted an Android container" in this review.

The non-attestable posture is enforced two ways, both confirmed in source: structurally, the module is simply never imported by any of the measured/attested NixOS profiles (§3), so it cannot appear in a sealed image by construction; and as a documented belt-and-suspenders check, the module carries a real NixOS assertion that fails the entire system evaluation if the root filesystem is ever configured read-only — the fingerprint of an attested profile — explicitly to catch an accidental future merge of the two. A companion shell CLI wraps the day-to-day container lifecycle (start/stop/status) and prints a fixed non-attestable-tier warning banner on every state-changing command; a fail-closed consent gate refuses to launch the container UI unless the user has separately opted in. An on-device egress policy kernel requires Tor-only egress before admitting any container traffic, fail-closed by default.

Threat model — the bridge tier

All three systems share one honest posture, stated in their own source rather than asserted here: every consent, kill-switch, or egress gate in this tier is a userland construction, explicitly not a kernel or TEE boundary, and every one of them is defeated by a rooted host. Where real cryptographic work exists (Ed25519-signed consent records, replay-guarded crossing requests, hash-chained audit trails) it sits entirely in the decision layer; the enforcement layer that would make a bypass actually costly — real managed-profile isolation, a real booted container, live StrongBox attestation reaching the Rust verifier — is design or stub in every case reviewed. None of the three claims to be "Postern OS," and the reviewed source consistently self-labels each gap rather than obscuring it.

15 Cross-cutting threat-model summary

One table, read straight down, for the systems above: where trust is actually anchored today, and what layer would have to fail for that anchor to be defeated.

System	Where trust is anchored today	Weakest disclosed link
Postern OS	dm-verity rootfs hash + UKI/TPM2 measurement, as configured	No image has been built or PCR-measured in this review; no Secure-Boot signing chain ships yet
Postern Seal	A pinned key (default) or the AMD/Intel/Google hardware root (feature-gated) — real crypto, fixture-tested	TDX's root is an empty placeholder (structurally cannot verify); SNP/StrongBox have never run against real hardware Postern launched
License spine	A single Ed25519 vendor key, held as a platform secret	Revocation is not enforced by any deployed verifier yet — a revoked license is not actually rejected client-side today
Consiglio	Real hybrid PQ (Falcon‖ML-DSA) signature verification, dual-control fail-closed logic	All state is in-memory; nothing deployed drives the real verifier end to end yet
Ezekiel 1	Enforced-and-tested BLP/Brewer-Nash logic inside one process	Explicitly not an OS/kernel boundary — a memory-safety bug elsewhere in the host process could bridge tenants
Cloud / Chiostro	The same TPM2/TEE anchors as OS/Seal, applied to a guest profile	Never built or booted; the live cloud endpoint is a separate, non-TEE licensing service, not the confidential VM
SDK / vault apps	Real Argon2id + XChaCha20-Poly1305 in Rust/wasm, key-committing	Shipping HTML apps default to a labeled PBKDF2 stand-in until a host shell injects the real core
Messenger / Courier	Real Megolm (messenger) and ML-KEM-1024‖X25519 hybrid (courier) content encryption	Courier has no sender authentication or replay defense by design; Matrix device verification is not implemented yet
Phone-as-signer	Compile-time-enforced impossibility of key material crossing the wire, plus WYSIWYS on real transaction bytes	Software keystore, not hardware-backed — a rooted phone can forge the approval UI
Porog / NoporIS / Sloboda	Real, tested consent/decision kernels	The Android-side enforcement (managed profile, container boot, live attestation) is stub or absent in every case reviewed

Read across the whole table, one pattern repeats: the cryptographic and decision-logic cores are consistently the most mature layer, and the deployment/enforcement wiring around them is consistently the least mature layer. This document has tried to keep that distinction visible at every level of detail rather than only at the summary.

16 Disclaimer

Everything described in this document is pre-production, experimental, forward-looking, and unaudited. No third-party security audit has been performed on any Postern Labs system, product, or on the underlying Bloch-SIS-PoW protocol. An audit is contracted but not delivered; until it is, nothing here is a certified security product, and internal review or a passing test suite is not a substitute for one.

This document is a **technical description, not a specification commitment and not a security assurance**. It was generated by reading the project's own source code and internal gap-analysis documents and reporting what was actually found there — including every placeholder, fixture-only test, unbuilt profile, and stub the source itself discloses. Where a specific field, struct, constant, or protocol detail could not be located in the reviewed source, this document says so explicitly rather than inferring or inventing it; where a status tag appears, it reflects the state of the code at the time of writing and may have changed since.

Do not rely on any system described here to protect real secrets, real funds, or real safety today. Every subsystem — measured boot, remote attestation, the license spine, Consiglio, Ezekiel 1, Gramota, Cloud/Chiostro, the SDK, the vault apps, Messenger, Courier, phone-as-signer, and the Android bridge apps — remains, without exception, **unaudited pre-production beta software**, provided "as is," without warranty of any kind, express or implied, including merchantability, fitness for a particular purpose, and non-infringement, consistent with the MIT and Apache-2.0 licenses that govern the underlying code.

Nothing in this document is legal, financial, or investment advice. Nothing in this document should be read as a claim that any coin, token, or protocol asset has value — the Bloch-SIS-PoW protocol this suite anchors to is a nascent, unaudited, low-hashrate mainnet beta, and its native coin carries no value claim, no token sale, and no listing effort. Postern Labs' revenue model is protection, support, and provable integrity — never the token, and never the sale of user data.

Generated from the `bloch-protocol` source tree, read-only, for publication as a downloadable technical reference on posternlabs.com.