QuantumLink

TECHNICAL WHITEPAPER

Protocol, Cryptography, and Systems Architecture

QuantumLink is a product of Dytallix, Inc. Please visit www.dytallix.com for more information.

DOCUMENT STATUS

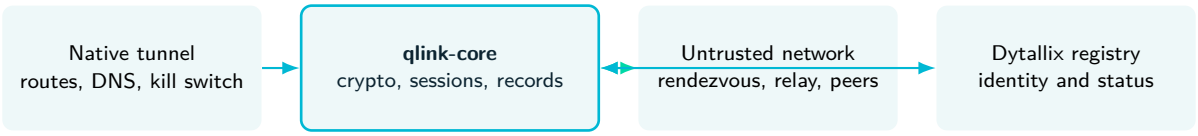
This paper reflects the current status of QuantumLink as of August 2026. The general architecture is expected to remain consistent, but implementation details may change before or after beta testing. Consult `github.com/exocognosis/QuantumLink` for the latest implementation status. Target-state properties are requirements, not evidence of completed production deployment; see Section 15B, Status vs. Spec.

01 / ABSTRACT

Engineering thesis and document map

QuantumLink is a server-minimized Layer 3 overlay in which a shared Rust core owns protocol state, post-quantum session establishment, device authentication, signed peer records, replay rejection, route validation, rendezvous, relay coordination, and platform FFI. Native platform components own packet capture, privileged route and DNS installation, local secret storage, user interaction, packaging, and release signing.

The central separation is architectural: **identity is on-chain; traffic is off-chain**. The Dytallix registry is consulted as a public-mesh admission input. It does not carry packet data, routes, DNS queries, endpoints, timing, or session keys. A signed, expiring peer record binds fresh reachability information to a persistent device identity. An authenticated peer session then protects packet frames over direct UDP or end-to-end encrypted relay-assisted paths.



Document map. The opening sections define the system, threat model, and protocol state machine. The cryptographic core then specifies the FIPS 203 handshake, transcript, credentials, symmetric data-path decision, frame proof obligations, replay state, and bounded fragmentation. Later sections cover registry admission, path selection, linkability, diagnostic disclosure, platform boundaries, quantitative failure injection, current status versus product specification, and implementation control flow.

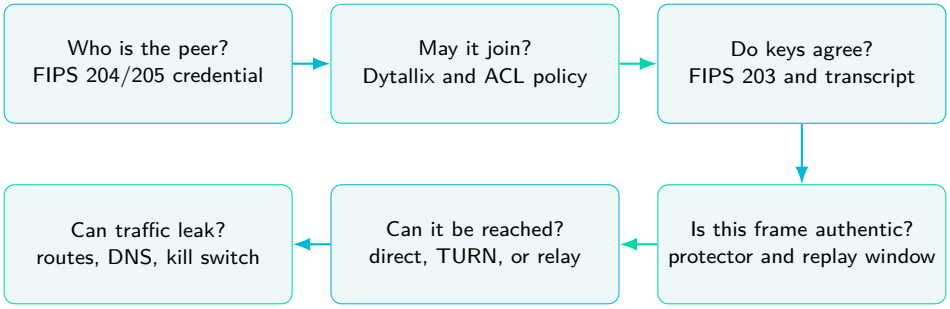
PRIMARY SECURITY CLAIM

If peer identity verification, FIPS 203 session establishment, transcript binding, directional key derivation, frame authentication, replay checks, and route policy all succeed, protected packets are delivered only through an authenticated QuantumLink peer session. Failure of a required stage produces a drop rather than transparent bypass.

Notation. I, R denote initiator and responder; s is a suite identifier; n_I, n_R are nonces; ek, dk are FIPS 203 encapsulation and decapsulation keys; ct is a ciphertext; ss is a shared secret; H_T is the transcript digest; and $K_{I\to R}, K_{R\to I}$ are directional traffic keys.

The security story in one page

QuantumLink separates six questions that are often collapsed into the word "VPN." Each question has a distinct mechanism and a distinct failure mode.



Threat	Plain-language risk	Required response
Harvest now, decrypt later	traffic captured today is attacked after quantum capability arrives	FIPS 203 session establishment and aggressive key rotation
Active intermediary	an attacker changes keys, suites, records, or frames	signed identity, transcript binding, authenticated frames, pinned registry policy
Malicious peer	an admitted device injects routes, replays, or oversized inputs	ACLs, route bounds, replay state, parser and resource limits
Metadata observer	a relay or network observer correlates stable identity and timing	relay-only public records, rotating aliases, padding options, explicit residual-risk disclosure
Local failure	service, route, DNS, or carrier state fails while applications transmit	protected-route kill switch, typed drops, failure-injection evidence

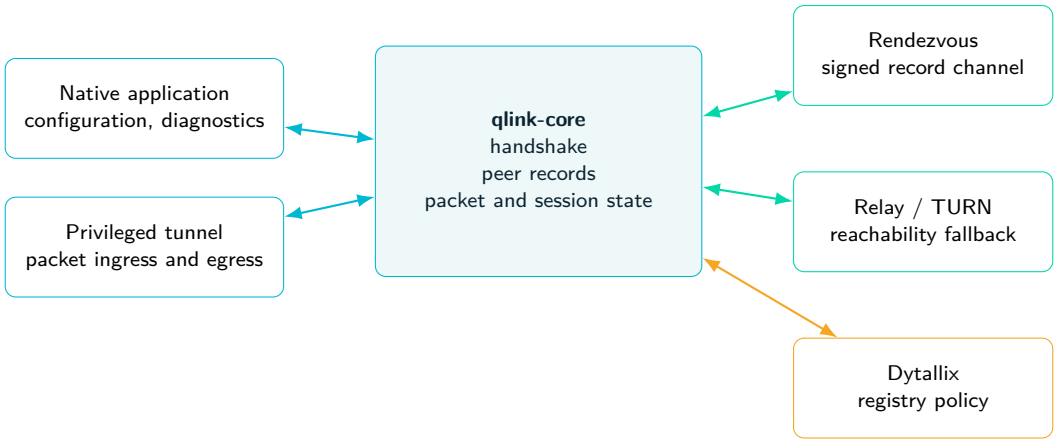
NON-CRYPTOGRAPHER SUMMARY

The chain decides whether a public identity is eligible. The handshake creates fresh secret material. The frame layer protects packet content. The carrier only moves opaque bytes. Native routing and the kill switch prevent bypass. A failure in any required layer must stop protected delivery rather than silently weaken it.

This decomposition also identifies review boundaries. A strong FIPS 203 handshake does not compensate for an unauthenticated frame construction, and a correct frame construction does not compensate for a leaking platform route.

Components, planes, and ownership

The final-release system is decomposed by responsibility rather than by platform. This prevents macOS, Windows, and SteamOS from drifting into distinct protocols.



Plane	Owned data	Security boundary
Identity	device public key, peer ID, registry binding, status	signatures and registry policy precede public-mesh admission
Control	signed record, expiry, sequence, routes, candidates, ICE material	discovery transports records but does not authenticate them
Session	suite, transcript, shared secret, directional keys, generation	FIPS 203 and SHAKE256 bind roles, suite, and messages
Data	normalized IP packet, frame header, packet number, tag	authenticated peer-session protector and replay window
Platform	tunnel interface, routes, DNS, secret store, IPC/FFI	native privilege and fail-closed leak control

Invariant S1 (single protocol owner). No platform adapter may independently implement suite negotiation, peer-record verification, session key derivation, or replay semantics. It may transport configuration and packets across a typed FFI or process boundary.

Invariant S2 (chain separation). Registry reads influence authorization only. Packet confidentiality and integrity never depend on chain availability after an admitted session is established.

Invariant S3 (untrusted helpers). Rendezvous, STUN, TURN, and relay responses are attacker-controlled until signatures, pins, bounds, and authenticated session checks succeed.

Adversaries, assets, and non-goals

Adversary	Capability	Required defense
Passive collector	records rendezvous, relay, STUN, direct traffic and timing	post-quantum session establishment; payload confidentiality; metadata minimization
Active MITM	substitutes records, candidates, certificates, ciphertexts, registry answers	signed records; transcript and suite binding; endpoint certificate binding; pinned registry policy
Malicious peer	owns a valid key and sends routes, malformed frames, replays, floods	ACL and registry admission; route validation; replay window; size and resource bounds
Sybil operator	creates many identities or manipulates a registry endpoint	active matching registration; chain/network pins; status and key binding; policy visibility
Compromised helper	controls rendezvous or relay ordering, availability, metadata	helper is not a trust anchor; signed data; end-to-end frames; cached verified state
Quantum attacker	later obtains cryptographically relevant quantum capability	FIPS 203 session establishment and FIPS 204/205 signatures; explicit crypto agility

Assets. Protected packet plaintext; FIPS 203 session material; FIPS 204 device seeds; peer-store keys; Dytallix wallet keys; signed records; ACLs; route and DNS policy; platform signing credentials; diagnostics; and release artifacts.

Trust assumptions. The endpoint OS, privileged tunnel mechanism, OS CSPRNG, local secret store, selected PQ primitives, and canonical serializers behave as specified. Operators configure registry pins, trust mode, routes, DNS, ACLs, and kill-switch policy correctly.

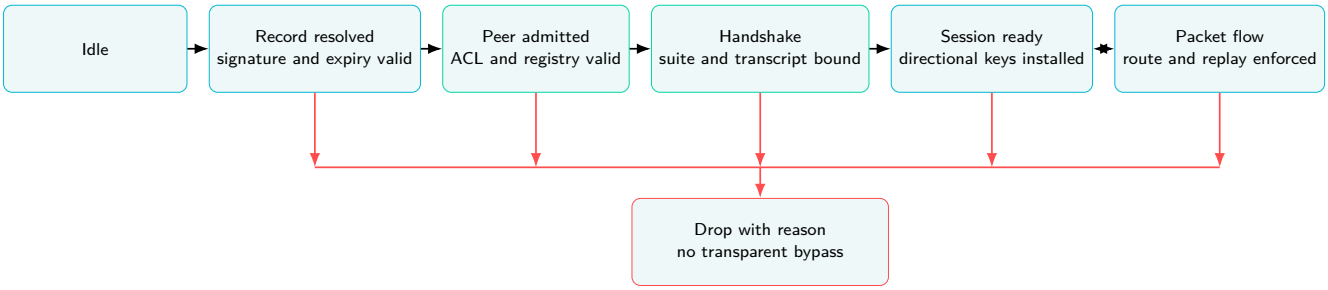
Non-goals. QuantumLink does not provide anonymity against a global observer, cover traffic, multi-hop unlinkability, endpoint compromise resistance, kernel compromise resistance, or hardware side-channel protection. Public identity is not equivalent to anonymity, and a relay can observe timing and source/destination metadata even when it cannot decrypt authenticated frames.

SECURITY SCOPE

The paper reasons about the protocol and final-release architecture specified by the public repository. Proof sketches reduce system claims to primitive assumptions and explicit invariants. They are not machine-checked proofs and do not replace independent cryptographic review.

Attack surface ordering. The highest-value review sequence is: canonical encoding; key generation and persistence; suite downgrade handling; transcript construction; record verification; registry binding; session installation and rotation; frame parsing and authentication; replay state; route policy; FFI/IPC memory ownership; and privileged platform leak control.

Connection lifecycle and fail-closed transitions



Let q be the connection state. The permitted transition relation $\rightarrow$ is constrained by predicates V_R (record valid), V_A (admission valid), V_H (handshake valid), and V_S (session live):

$$q = \text{Resolved} \rightarrow \text{Admitted} \quad \text{iff } V_R \wedge V_A, \tag{1}$$

$$q = \text{Handshake} \rightarrow \text{Ready} \quad \text{iff } V_H \wedge V_S, \tag{2}$$

$$q = \text{Flow} \rightarrow \text{Deliver}(p) \quad \text{iff } V_S \wedge V_{route}(p) \wedge V_{frame}(p) \wedge V_{replay}(p). \tag{3}$$

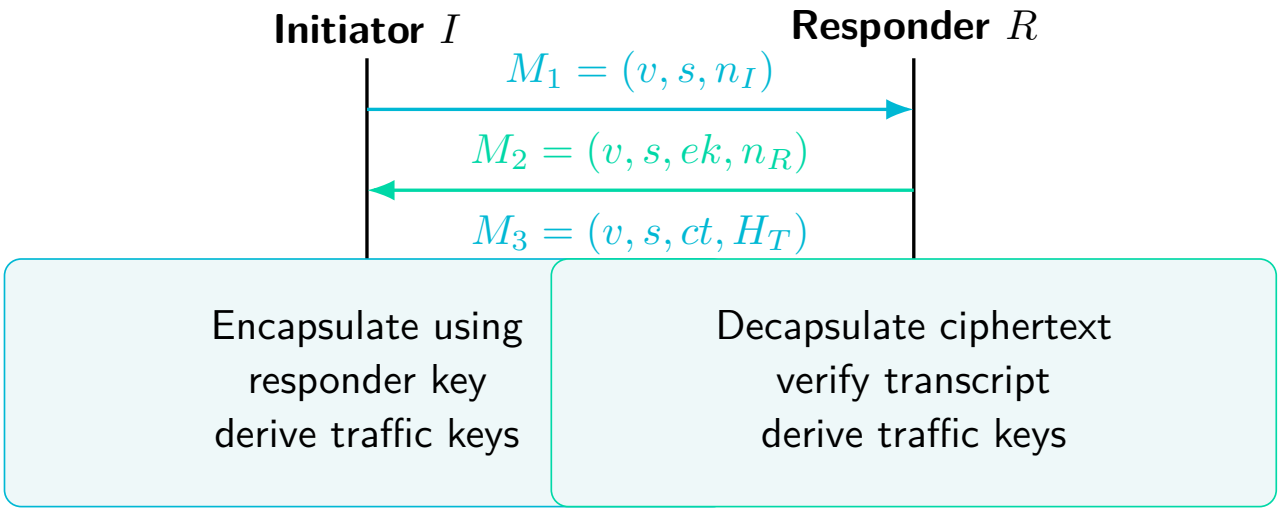
If any required predicate is false, the only permitted packet action is $\text{Drop}(reason)$.

Proof sketch. Assume the implementation exposes packet delivery only through the transition in Eq. (3). By construction, delivery requires all four predicates. Negating any predicate makes the conjunction false; therefore no delivery transition exists. If the transition function is total and maps every rejected input to a typed drop reason, protected traffic cannot silently escape through this state machine. The system-level claim additionally assumes the native adapter does not bypass the core. □

Typed failure reasons include missing registry state, revoked or suspended identity, key mismatch, registry unavailable, unsupported suite, transcript mismatch, missing peer session, expired session, rekey threshold, malformed frame, replay, and unprotected route.

Three-message FIPS 203 protocol

The v1 handshake uses FIPS 203 with parameter set 768 for session establishment and rejects the legacy hybrid classical/PQC identifier. Device authentication and signed peer discovery occur in adjacent protocol layers.



Responder key generation and initiator encapsulation are:

$$(dk, ek) \leftarrow \mathrm{FIPS203.KeyGen}_{768}(1^{\lambda}), \tag{4}$$

$$(ct, ss) \leftarrow \mathrm{FIPS203.Encaps}_{768}(ek), \qquad ss' \leftarrow \mathrm{FIPS203.Decaps}_{768}(dk, ct). \tag{5}$$

The authenticated-key experiment decomposes into primitive and binding failures:

$$\mathrm{Adv}^{ake}_{QL}(\mathcal{A}) \leq \mathrm{Adv}^{ind\text{-}cca}_{\text{FIPS203-768}}(\mathcal{B}_1) + \mathrm{Adv}^{coll}_{\mathrm{SHAKE256}}(\mathcal{B}_2) + \mathrm{Adv}^{auth}_{record}(\mathcal{B}_3) + \frac{q_h}{2^{256}}. \tag{22}$$

The accepted suite and protocol version must be identical in M_1, M_2, M_3. The initiator checks the responder suite before encapsulation. The responder recomputes H_T and rejects a changed suite or mismatched digest before installing keys.

Listing 1: Repository-derived handshake structures

```rust
pub struct InitiatorHello { version: u8, suite: String, initiator_nonce: [u8; 32] }
pub struct ResponderHello {
    version: u8, suite: String,
    responder_mlkem768_ek: Vec<u8>, responder_nonce: [u8; 32],
}
pub struct InitiatorFinish {
    version: u8, suite: String,
    mlkem768_ciphertext: Vec<u8>, transcript_hash: [u8; 32],
}
```

Proof sketch. Replace the real FIPS 203 shared secret with a random value. Distinguishing this game hop breaks IND-CCA security. Next replace transcript-derived values with independent random values; distinguishing breaks the SHAKE256 assumption or finds a transcript collision. Finally, an unauthenticated peer that causes acceptance forges the signed identity layer. Applying the union bound yields Eq. (22), plus the explicit 256-bit transcript-guessing term. □

Canonical binding and directional separation

Define the transcript digest over canonical encodings of the two hello messages:

$$H_T = \mathrm{SHAKE256}(\texttt{QLINK\_TRANSCRIPT} \parallel \mathrm{enc}(M_1) \parallel \mathrm{enc}(M_2), 32). \tag{6}$$

Directional material is derived from the FIPS 203 shared secret, transcript, suite, and role-separated labels:

$$K_0 \parallel K_1 = \mathrm{SHAKE256}(\texttt{QLINK\_KEYS} \parallel ss \parallel H_T \parallel s, 64), \tag{7}$$

$$(K^I_{tx}, K^I_{rx}) = (K_0, K_1), \qquad (K^R_{tx}, K^R_{rx}) = (K_1, K_0). \tag{8}$$

Proof sketch. FIPS 203 correctness gives $ss' = ss$ except with negligible probability. Both parties accept only if they hold identical M_1, M_2, version, and suite; hence both compute the same H_T in Eq. (6). SHAKE256 is deterministic, so Eq. (7) yields the same ordered pair (K_0, K_1). Role inversion in Eq. (8) then gives $K^I_{tx} = K^R_{rx}$ and $K^I_{rx} = K^R_{tx}$. □

Downgrade argument. Let an attacker transform accepted transcript T with suite s into T' with suite $s' \neq s$. Acceptance requires the explicit suite equality checks to pass and the responder-computed transcript hash to equal the transmitted value. Under canonical encoding and collision resistance,

$$\Pr[\mathrm{Accept}(T') \wedge s' \neq s] \leq \mathrm{Adv}^{coll}_{\mathrm{SHAKE256}} + \mathrm{Adv}_{parser} + \mathrm{Adv}_{logic}. \tag{9}$$

The final two terms isolate non-cryptographic risks: ambiguous serialization and an implementation path that omits validation.

Canonical length-prefixing is required to be injective over accepted messages:

$$\mathrm{enc}(x) = \mathrm{enc}(y) \Rightarrow x = y, \qquad \mathrm{absorb}(x) = |x|_{64} \parallel x. \tag{23}$$

Listing 2: Suite continuity and transcript verification

```
validate_suite(finish.version, &finish.suite)?;
if initiator.suite != self.hello.suite || finish.suite != self.hello.suite {
    return Err(QlinkError::Protocol("handshake suite changed across transcript".into()));
}
let expected_hash = hash_transcript(initiator, &self.hello)?;
if finish.transcript_hash != expected_hash {
    return Err(QlinkError::Protocol("transcript hash mismatch".into()));
}
```

FIPS 204 credentials and signed peer records

A device keypair supplies persistent identity. FIPS 204 with parameter set 65 is the default signature path; the suite model also includes the FIPS 205 SHAKE-128s parameter set. A peer record is a signed, expiring, sequence-numbered statement:

$$R = (mesh, peer, pk_D, routes, candidates, ice, cert, exp, seq, meta), \qquad \sigma_R \leftarrow \mathrm{Sign}(sk_D, \mathrm{enc}(R)). \tag{10}$$

Verification requires

$$\mathrm{Verify}(pk_D, \mathrm{enc}(R), \sigma_R) = 1, \quad peer = \mathrm{PeerId}(pk_D), \quad now < exp, \quad seq > seq_{max}. \tag{11}$$

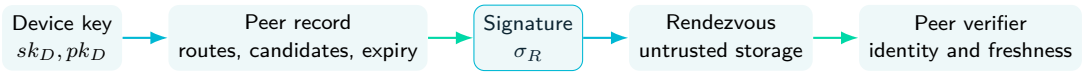
For q_s signing queries, accepted record forgery is bounded by

$$\mathrm{Adv}^{record}_{forge}(\mathcal{A}) \leq \mathrm{Adv}^{euf\text{-}cma}_{\text{FIPS204-65}}(\mathcal{B}_1) + \mathrm{Adv}^{coll}_{H}(\mathcal{B}_2) + \mathrm{Adv}_{enc} + \frac{q_s}{2^{256}}. \tag{24}$$

Proof sketch. Suppose an attacker changes any signed field of an accepted record without the device secret key. If canonical encoding is injective, the modified record produces a distinct message. Acceptance then yields either a signature forgery against EUF-CMA security or an encoding/hash collision. Expiry limits future acceptance; a persisted maximum sequence rejects an older but correctly signed record. □

Listing 3: Repository-derived record verification order

```
verify(device_key, canonical_bytes(record), signature)?;
require(peer_id(record) == peer_id(device_key))?;
require(now < record.expires_at)?;
require(record.sequence > persisted_max_sequence)?;
persist_max_sequence(record.sequence)?;
```



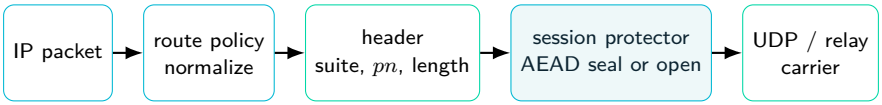
Inbound identity assertions bind peer ID, mesh ID, timestamp, nonce, and device public key to the connecting endpoint. Signed record verification answers what was published; the assertion answers whether the endpoint presenting the connection controls the expected device identity now.

METADATA LIMIT

Signed records necessarily expose selected discovery metadata to their recipients. Public-record minimization can prefer relay-family candidates so host and server-reflexive addresses are not silently advertised as public location metadata.

Packet framing, authentication, and replay state

The Layer 3 tunnel supplies IP packets to the Rust packet core. Route policy is checked and selected IPv4 metadata is normalized before transport framing. The negotiated peer session supplies the confidentiality and integrity boundary.



The production protocol profile selects ChaCha20-Poly1305 for packet confidentiality and integrity. The repository's current `PqcFrameProtector` instead contains a custom SHAKE256 encrypt-then-authenticate construction. That construction is documented here only as an **experimental compatibility profile**. It is not the production data-path protector and must not be presented as a standardized AEAD. For packet number pn and associated data A containing the suite and frame header, the experimental profile computes

$$S_{pn} = \mathrm{SHAKE256}(K \parallel H_T \parallel pn, |P|), \tag{12}$$

$$C = P \oplus S_{pn}, \tag{13}$$

$$\tau = \mathrm{SHAKE256}(K \parallel A \parallel C, t). \tag{14}$$

The receiver compares τ in constant time before replay acceptance and before releasing $P = C \oplus S_{pn}$. The stream and tag have distinct length-prefixed domain labels, but the current implementation uses the same directional 256-bit key for both functions. This is not a standardized AEAD and must not inherit an AEAD security claim without a dedicated reduction and independent implementation review.

Replay window. Let m be the maximum authenticated packet number and w the window width. A packet is fresh iff

$$\mathrm{Fresh}(pn) = (pn > m) \ \vee \ (m - w < pn \le m \ \wedge \ pn \notin W). \tag{15}$$

If the authentication tag has t bits and the adversary makes q_f verification attempts, then the frame-forgery term satisfies

$$\mathrm{Adv}^{\mathit{int\text{-}ctxt}}_{\mathit{frame}}(\mathcal{A}) \le \mathrm{Adv}^{\mathit{prf}}_{\mathrm{SHAKE256}}(\mathcal{B}) + \frac{q_f}{2^t} + \mathrm{Adv}_{\mathit{nonce}}. \tag{25}$$

Listing 4: Authenticated receive path

```
let header = parse_bounded_header(frame)?;
require(header.suite == session.suite)?;
verify_tag(session.rx_key, header, ciphertext, tag)?;
require(replay_window.is_fresh(header.packet_number))?;
let plaintext = unmask(session.rx_key, header.packet_number, ciphertext);
replay_window.commit(header.packet_number);
return deliver(plaintext);
```

Proof sketch. After a packet number is accepted it is inserted into W. A duplicate has $pn \in W$ and fails Eq. (15). A stale packet has $pn \le m - w$ and also fails. A new high packet advances m and shifts the bounded window. Thus no packet number is accepted twice within a session, assuming replay state is not rolled back. □

ChaCha20-Poly1305 production profile

The production path is a standard, constant-time ChaCha20-Poly1305 implementation with extensive public analysis. This matches the website's stated symmetric data-path choice and performs predictably on devices without AES acceleration. A 256-bit symmetric key retains approximately 128-bit security against generic quantum search, so using a classical symmetric primitive does not introduce the quantum-vulnerable public-key problem that FIPS 203 and FIPS 204 replace.

For a standard AEAD, derive independent traffic keys and a nonce base from the FIPS 203 secret and transcript:

$$K_{aead} \parallel N_0 = \mathrm{SHAKE256}(\texttt{QLINK\_AEAD} \parallel ss \parallel H_T \parallel s, 44), \tag{30}$$

$$N_{pn} = N_0 \oplus \mathrm{encode}_{96}(pn), \tag{31}$$

$$(C, \tau) = \mathrm{ChaCha20Poly1305.Seal}(K_{aead}, N_{pn}, P, A). \tag{32}$$

The counter must never repeat under K_{aead}; generation changes derive a fresh key and nonce base. Headers carrying version, suite, direction, generation, packet number, and length are authenticated as A.

Proof sketch. Assume FIPS 203 key indistinguishability, SHAKE256 key-derivation pseudorandomness, nonce uniqueness, and the AEAD's IND-CPA and INT-CTXT security. Replace the FIPS 203 secret with random, then replace derived traffic material with random. Under unique nonces, confidentiality reduces to AEAD IND-CPA and forgery reduces to AEAD INT-CTXT. Transcript, suite, direction, and generation are authenticated through key derivation or associated data, so cross-suite and cross-generation substitution requires a primitive break or implementation error. □

The experimental SHAKE256 profile may be retained only for research, compatibility testing, and comparative measurement. It must use a distinct, non-production suite identifier, must not be negotiated by a production build, and must be visibly labeled in diagnostics. A conservative experimental revision derives separate keys:

$$K_E \parallel K_A = \mathrm{SHAKE256}(\texttt{QLINK\_FRAME\_SUBKEYS} \parallel K \parallel H_T, 64), \tag{33}$$

then uses K_E only for the mask and K_A only for the tag. A complete proof must model length-prefix injectivity, domain separation, counter uniqueness, multi-user security, chosen-ciphertext behavior, truncation, state rollback, and the relationship between keyed SHAKE invocations.

Criterion	ChaCha20-Poly1305 production profile	SHAKE256 experimental profile
Standardization	RFC-standard AEAD with broad deployment	product-specific construction
Review surface	library integration, nonce and state discipline	construction proof, implementation audit, state discipline
Key separation	internal AEAD design	same key today; separate subkeys recommended
Performance	optimized across major platforms	architecture-dependent; benchmark required
Documentation status	specified for production	present in repository; not production-approved

08B / FRAME AUDIT

Experimental SHAKE256 proof obligations

Retaining a custom protector as an experimental profile requires evidence beyond round-trip, tamper, and replay tests. The review target is the exact encoded construction, not merely the SHAKE256 primitive. Completing these obligations would support further evaluation, but would not silently promote the profile into production; production eligibility would require an explicit protocol revision and independent review decision.

Let q_e be encryption queries, q_v verification attempts, ℓ the maximum absorbed input length, and $t = 256$ the tag length. Under independent keyed-XOF domains and unique counters, the intended bound is

$$\mathrm{Adv}^{aead}_{QL}(\mathcal{A}) \le \mathrm{Adv}^{prf}_{XOF,E}(\mathcal{B}_1) + \mathrm{Adv}^{prf}_{XOF,A}(\mathcal{B}_2) + \frac{q_v}{2^t} + \mathrm{Adv}_{enc} + \mathrm{Adv}_{state}. \tag{34}$$

Adv_{enc} captures ambiguous or non-injective encoding. Adv_{state} captures repeated counters, rollback, generation collision, and key reuse. Equation (34) is a proof target, not a completed proof for the current same-key implementation.

Listing 5: Current repository authentication order

```
let expected_tag = frame_tag(&self.suite, &self.rx_key,
                             counter, header, ciphertext);
if !constant_time_eq(tag, &expected_tag) {
    return Err(QlinkError::Crypto(
        "PQC frame authentication failed".into()));
}
if !self.rx_window.observe(counter) {
    return Err(QlinkError::Protocol("PQC frame replay rejected".into()));
}
Ok(xor_stream(&self.suite, &self.rx_key,
              counter, header, ciphertext))
```

Audit checklist.

- Verify that tag comparison, key derivation, parser rejection, and secret cleanup have no secret-dependent branches or memory access patterns.
- Prove counter uniqueness across crash, restore, sleep, generation rotation, and simultaneous transports.
- Confirm every variable-length field is length-prefixed and every allocation is bounded before cryptographic work.
- Test cross-suite, cross-direction, cross-generation, truncation, extension, reordered-fragment, and multi-user substitution.
- Benchmark throughput, latency, energy, and timing variance against ChaCha20-Poly1305 on supported CPUs.
- Require independent cryptographic review before assigning a production AEAD-equivalent claim.

PROFILE ENFORCEMENT

Production builds negotiate ChaCha20-Poly1305 only. The SHAKE256 protector remains experimental, uses a distinct suite identifier, and stays disabled in production builds. A future proposal to promote it would require a complete published proof, independent cryptographic review, constant-time audit, cross-platform measurements, and an explicit specification change.

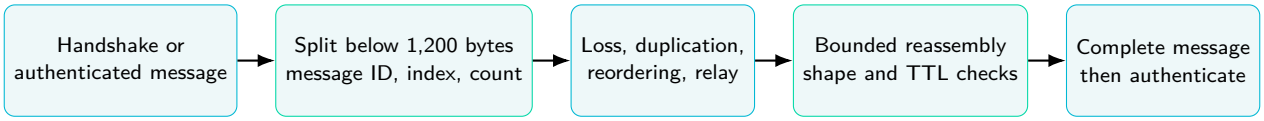
Fragmentation and reassembly under hostile networks

FIPS 203 parameter set 768 messages exceed the size of many conservative UDP payload budgets when combined with protocol headers and identity material. The native carrier therefore caps each UDP datagram at 1,200 bytes and fragments larger frame or authenticated-message payloads below IP fragmentation.

For carrier header h_c, fragment header h_f, datagram cap $M = 1200$, and message length L:

$$P_{max} = M - |h_c| - |h_f|, \tag{35}$$

$$n = \left\lceil \frac{L}{P_{max}} \right\rceil, \qquad 1 \le n \le 2^{16} - 1. \tag{36}$$



Listing 6: Repository-derived fragmentation policy

```
const MAX_UDP_DATAGRAM_LEN: usize = 1_200;
const MAX_REASSEMBLED_MESSAGE_LEN: usize = 1024 * 1024;
const MAX_REASSEMBLY_ENTRIES: usize = 64;
const REASSEMBLY_TTL: Duration = Duration::from_secs(30);

let chunk_count = payload.len().div_ceil(MAX_FRAGMENT_PAYLOAD_LEN);
for (index, chunk) in payload.chunks(MAX_FRAGMENT_PAYLOAD_LEN).enumerate() {
    let fragment = encode_fragment_payload(
        message_id, total_len, index as u16, chunk_count as u16, chunk)?;
    send_datagram(kind.fragment_datagram_kind(), fragment).await?;
}
```

Reassembly checks total length, count, index, per-position length, duplicate completion, maximum in-flight entries, and a 30-second lifetime. Tests cover large authenticated messages, large frames, out-of-order fragments, duplicates, poisoned entries, oversized datagrams, stale-entry reclamation, and receive-limit enforcement.

Remaining resilience work. Add selective retransmission or restart semantics for lossy paths, loss and jitter measurements across cellular and hostile NAT environments, message-level authentication before allocating large retained state where possible, and per-source quotas to limit fragment-cache denial of service.

Installation, rotation, expiry, and byte limits

An installed peer session is not merely a key pair. It binds peer identity, direction, generation, transcript, expiry, and rekey budget:

$$S = (peer, dir, gen, H_T, t_{exp}, B_{max}, K_{tx}, K_{rx}). \tag{16}$$

For the same peer and direction, replacement is accepted only when the generation increases:

$$\mathrm{Install}(S') = 1 \Rightarrow peer' = peer \wedge gen' > gen. \tag{17}$$

Traffic readiness additionally requires $now < t_{exp}$ and $bytes(S) < B_{max}$. Inbound replay state resets when a new inbound generation is installed.

$$Ready(S, t, b) = Installed(S) \wedge t < t_{exp} \wedge b < B_{max} \wedge SuiteAllowed(s) \wedge KeyBound(peer). \tag{26}$$

Listing 7: Repository-derived readiness gate

```
pub fn peer_session_ready(&self) -> bool {
    self.peer_session_block_reason(PeerSessionDirection::Outbound)
        .is_none()
}
```

Listing 8: Repository-derived monotonic session installation

```
pub fn install_peer_session(&mut self, session: InstalledPeerSession) -> bool {
    let (slot, bytes) = match session.direction {
        PeerSessionDirection::Outbound => (&mut self.outbound_peer_session,
                                           &mut self.outbound_peer_session_bytes),
        PeerSessionDirection::Inbound => (&mut self.inbound_peer_session,
                                          &mut self.inbound_peer_session_bytes),
    };
    if let Some(current) = slot.as_ref() {
        if current == &session { return true; }
        if current.peer_id == session.peer_id && current.generation >= session.generation {
            return false;
        }
    }
    let reset_replay = session.direction == PeerSessionDirection::Inbound;
    *slot = Some(session); *bytes = 0;
    if reset_replay { self.replay_window = ReplayWindow::new(); }
    true
}
```

Proof sketch. Assume generation is persisted or derived from a monotonic session lifecycle. Equation (17) rejects installation of any older same-peer generation. Because packet delivery requires the currently installed session and inbound replay state is reset only when the generation advances, a stale session cannot replace a newer one through this API. Rollback of persisted generation outside the API remains a platform threat. □

10 / CODE PATH

Handshake implementation cross-reference

The following expanded excerpt shows the security-critical order: validate suite, parse the encapsulation key, encapsulate, compute the transcript, derive directional keys, and return the finish message plus session state.

Listing 9: Initiator finish path, adapted from qlink-core/src/crypto.rs

```
pub fn finish(self, responder: &ResponderHello)
    -> Result<(InitiatorFinish, SessionKeys)>
{
    validate_suite(responder.version, &responder.suite)?;
    if responder.suite != self.hello.suite {
        return Err(QlinkError::Protocol(format!(
            "responder suite {} does not match initiator suite {}",
            responder.suite, self.hello.suite)));
    }
    let ek_bytes = responder.responder_mlkem768_ek.as_slice().try_into()
        .map_err(|_| QlinkError::InvalidKey(
            "invalid ML-KEM-768 encapsulation key size".into()))?;
    let ek = EncapsulationKey::new(&ek_bytes)
        .map_err(|_| QlinkError::InvalidKey(
            "invalid ML-KEM-768 encapsulation key".into()))?;
    let (ciphertext, mlkem_shared) = ek.encapsulate();
    let transcript_hash = hash_transcript(&self.hello, responder)?;
    let (tx_key, rx_key) = derive_directional_keys(
        mlkem_shared.as_slice(), &transcript_hash,
        &responder.suite, Direction::Initiator)?;
    let finish = InitiatorFinish {
        version: PROTOCOL_VERSION,
        suite: responder.suite.clone(),
        mlkem768_ciphertext: ciphertext.as_slice().to_vec(),
        transcript_hash,
    };
    Ok((finish, SessionKeys {
        suite: responder.suite.clone(), handshake_hash: transcript_hash,
        tx_key, rx_key,
    }))
}
```

Dytallix data model and admission predicate

The persistent registry record is compact and keyed by QuantumLink peer ID:

$$D = (peer, owner, h_{device}, h_{transport}, h_R, status, reputation, stake, updated, expires). \tag{18}$$

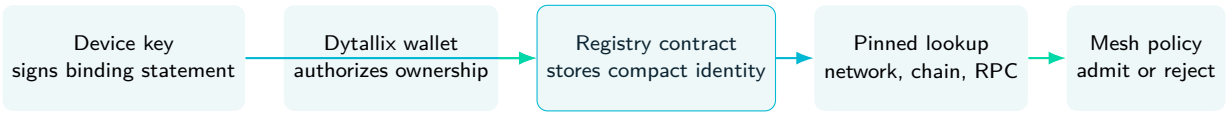
For a public mesh, define the admission predicate

$$\begin{aligned}\mathrm{Admit}_{public}(R, D) = {} & \mathrm{SigValid}(R) \wedge \mathrm{Active}(D) \wedge \mathrm{Unexpired}(D) \\ & \wedge (peer_R = peer_D) \wedge (H(pk_R) = h_{device}) \wedge \mathrm{NetworkPinned}(D).\end{aligned} \tag{19}$$

Registry freshness is explicit rather than implied by transport success:

$$Fresh_D(D, t) = \big(status_D = active\big) \wedge \big(expires_D = \bot \ \vee \ t < expires_D\big) \wedge (t - updated_D \leq \Delta_{policy}). \tag{27}$$

Private meshes may treat registry verification as preferred; development meshes may make it optional. Public meshes do not permit identity mode `Off`.



Proof sketch. Assume signature unforgeability, collision resistance of H, authentic pinned registry reads, and correct contract authorization. If Eq. (19) accepts, the fresh peer record is signed by the device key whose hash is bound to the active registry record for the same peer ID. Substitution requires a signature forgery, hash collision, compromised registry trust root, or policy implementation error. □

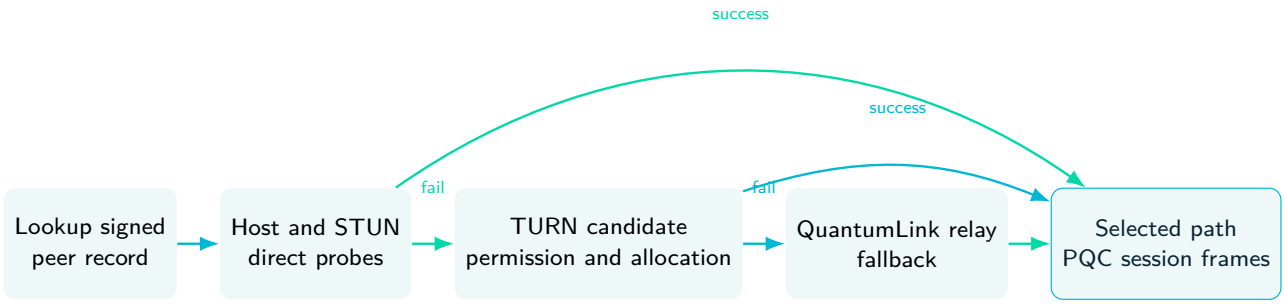
Listing 10: Repository-derived public-mesh admission checks

```
verify_peer_record_identity(peer_record)?;
let registry = registry_record.ok_or(MissingRegistry)?;
require(registry.status == Active)?;
require(registry.expires_at.map_or(true, |e| now < e))?;
require_match("peer_id", registry.peer_id, peer_record.peer_id)?;
require_match("device_key_hash", registry.device_key_hash,
              hash(peer_record.device_key))?;
```

The contract does not store packet contents, DNS activity, raw endpoints, route lists, timing, relay paths, or session material. The latest peer-record hash can bind short-lived discovery state to persistent registration without copying the record on-chain.

Direct probes, ICE, TURN, and relay fallback

Rendezvous is a publication and lookup mechanism for signed peer records. Reachability selection consumes candidates only after record verification and policy admission.



Carrier versus security boundary. Native UDP, TURN, and QuantumLink relay paths carry opaque session frames. Their availability and metadata properties differ, but none is permitted to replace signed identity or authenticated app-layer frame protection. A relay can drop, delay, reorder, or correlate traffic; it cannot forge a valid frame without session keys.

Path selection invariants.

- A candidate is eligible only if it came from an authenticated record or a locally configured trusted source.
- Direct paths are preferred when validated; relay is a reachability fallback, not a decryption point.
- Last-good path caching does not bypass record expiry, identity policy, or session generation checks.
- Network change clears or revalidates path state and preserves the protected-route fail-closed policy.

Listing 11: Illustrative path policy pseudocode

```
for candidate in verified_record.candidates() {
    if policy.permits(candidate) && direct_probe(candidate).authenticated() {
        return install_path(candidate, negotiated_session);
    }
}
if let Some(turn) = live_turn_allocation(verified_record) { return install_path(turn, session); }
if let Some(relay) = signed_quantumlink_relay(verified_record) { return install_path(relay, session); }
return Err(PathError::NoAuthenticatedPath);
```

12A / LINKABILITY

Stable identity, rotating aliases, and metadata controls

The long-term device key and registry identity provide accountability but also create correlation risk. A relay or network observer can combine a stable peer ID, endpoint changes, timing, record publication cadence, and public-wallet linkage into a longitudinal profile.

Let the observer's view in epoch e be

$$V_e = (alias_e, endpoints_e, timing_e, relay_e, wallet_e, record\_hash_e). \tag{37}$$

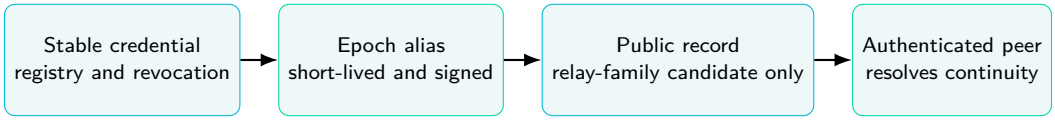
The goal is not anonymity against a global observer. It is to reduce unnecessary equality tests between V_e and V_{e+1} while preserving authorized continuity.

Rotating discovery alias. A short-lived alias can be derived and signed by the stable device credential:

$$alias_e = \mathrm{SHAKE256}(\texttt{QLINK\_ALIAS} \parallel mesh \parallel e \parallel pk_D, 16), \tag{38}$$

$$\sigma_e = \mathrm{Sign}_{FIPS204}(sk_D, mesh \parallel e \parallel alias_e \parallel exp). \tag{39}$$

Peers that are already authorized verify the binding without placing the stable peer ID into every public discovery record. The Dytallix record continues to bind the stable device credential and status. This reduces passive rendezvous correlation but does not hide transport IPs or timing from an on-path observer.



Public-mesh defaults.

- Publish relay-family candidates only. Host and server-reflexive candidates may be used locally for direct probing but are not placed in public records unless an operator deliberately changes policy.
- Rotate human-readable aliases and discovery pseudonyms by sequence or epoch. Keep the stable registry identity outside rendezvous records when Verified mode is sufficient.
- Offer bounded padding buckets for control messages and optional low-rate cover traffic only with explicit bandwidth and battery budgets. Neither feature creates a global-observer anonymity claim.
- Display exactly which endpoints, aliases, routes, ICE material, certificates, and wallet fields will be published before record submission.

The repository already documents sequence-rotating pseudonyms and relay-only public publication as defaults. Rotating cryptographic peer IDs, cover traffic, and padding remain design extensions until specified and implemented end to end.

12B / DIAGNOSTIC PRIVACY

Redaction, controlled disclosure, and auditability

Default support bundles redact raw peer IDs, wallet addresses, network addresses, routes, DNS values, SSIDs, external IPs, and packet content. A stronger production design treats unredacted export as a privileged disclosure ceremony rather than a convenience toggle.

$$ExportSensitive = OperatorApproval \wedge Reauth \wedge PurposeBound \wedge Expiring \wedge Audited. \tag{40}$$

For managed deployments, policy may additionally require a hardware-backed assertion or two independent approvers:

$$ExportSensitive_{managed} = ExportSensitive \wedge (HardwareAttested \ \vee \ Approvals \geq 2). \tag{41}$$

Export class	Default contents	Authorization
Standard	counters, typed failures, versions, redacted aliases, coarse path state	local operator action
Detailed	bounded configuration and timing with stable pseudonyms	reauthentication and explicit purpose
Sensitive	raw identifiers or endpoints required for a specific incident	managed policy, short-lived grant, immutable audit receipt
Packet capture	never part of a support bundle	separate security workflow outside QuantumLink diagnostics

Listing 12: Disclosure policy pseudocode

```
match export_class {
    Standard => export(redacted_allowlist()),
    Detailed if reauthenticated() && purpose_is_bounded() =>
        export(expiring_pseudonymous_view()),
    Sensitive if policy.approvals() >= 2 || hardware_attested() =>
        export_once(audited_sensitive_allowlist()),
    _ => return Err(ExportDenied),
}
```

Controls must be enforced in the privileged exporter, not only hidden in UI. Every grant records requester, approver, purpose, field allowlist, expiration, and artifact hash. Sensitive artifacts should be encrypted to an incident-specific recipient key and automatically deleted from staging after confirmed transfer.

STATUS BOUNDARY

Redacted-by-default exports are part of the repository design. Multi-party approval and hardware-attested sensitive export are hardening requirements proposed here and must not be represented as implemented until platform evidence exists.

Metadata inventory and information flow

Data class	Device	Control helpers	Dytallix chain
Private keys	protected secret store	never	never
Packet plaintext	tunnel and peer endpoint	never	never
Session keys	qlink-core memory	never	never
Routes / DNS	local policy and authenticated peer	minimized signed record where required	never
Reachability	local candidates	short-lived signed record / relay metadata	never
Identity	device key and peer ID	signed peer record	compact ownership and status binding
Diagnostics	redacted local bundle	operator-controlled export only	never

Linkability model. Verified identity mode checks active registry state without publishing the wallet address in rendezvous records. Public Wallet mode intentionally exposes wallet linkage for operators that want visible reputation or staking. Stable peer IDs and network timing still permit correlation; QuantumLink does not claim unlinkability against a global observer.

Let L be observable linkability. A qualitative decomposition is

$$L = L_{peer\_id} + L_{endpoint} + L_{timing} + L_{wallet} + L_{operator}. \tag{20}$$

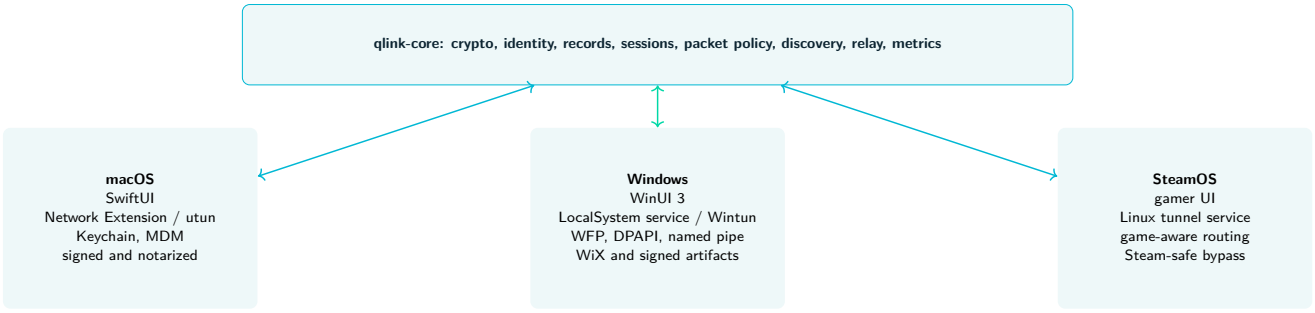
The architecture reduces $L_{endpoint}$ through relay-only public records, L_{wallet} through Verified mode, and $L_{operator}$ through local-first redacted diagnostics. It does not eliminate L_{timing} or stable-identity correlation.

DATA-MINIMIZATION RULE

No subsystem receives data merely because it is convenient. The registry receives identity and status, rendezvous receives a short-lived signed discovery statement, relay receives opaque frames and routing metadata, and the native endpoint alone handles packet plaintext and local policy.

Support bundles redact raw `qlink_*` peer identifiers and network addresses by default. Raw export is an explicit operator action. Wallet secrets remain in the Dytallix keystore and are excluded from tunnel configuration, provider messages, and diagnostic archives.

Native privilege with one shared protocol



FFI and IPC contract. Platform code passes validated configuration, packet bytes, peer IDs, and typed lifecycle commands. It must not implement an alternate cryptographic path. Memory ownership, pointer lifetime, buffer length, JSON bounds, error redaction, and crash behavior are security-critical.

Native responsibilities.

- Install protected routes and DNS without transient leaks.
- Store device seeds and peer-store keys using OS-protected storage.
- Maintain kill-switch behavior through sleep, wake, network change, service restart, and UI failure.
- Bind privileged mutations to authenticated local IPC and least-privilege policy.
- Surface peer-session readiness, replay drops, registry rejection, and path selection without exposing secrets.
- Verify platform signatures, update metadata, and release provenance before installation.

Cross-platform invariant. Given identical canonical inputs, all platforms must obtain identical peer IDs, record verification results, transcript hashes, directional keys, admission decisions, frame parsing outcomes, and replay decisions from the shared core.

Fail closed, bounded, and diagnosable

Failure behavior is part of the protocol. Each boundary emits a stable class and a safe action.

Failure	Packet action	Operator-visible meaning
`rejected_missing_registry`	drop before connect	peer lacks required public-mesh registration
`rejected_revoked`	drop before connect	registry status forbids admission
`rejected_key_mismatch`	drop before connect	record key does not match registered binding
`unsupported_suite`	abort handshake	peer proposed an unaccepted protocol suite
`transcript_mismatch`	abort handshake	messages or suite changed in transit or parsing
`peer_session_unavailable`	drop protected packet	no authenticated live session is installed
`dropped_replay`	drop frame	packet number was duplicate or outside the window
`no_authenticated_path`	keep protected routes blocked	direct, TURN, and relay selection failed

Define safety property $\mathcal{F}$:

$$\mathcal{F}: \quad \neg Ready(peer) \Rightarrow \forall p \in Protected,\ Action(p) = Drop(reason). \tag{21}$$

Listing 13: Repository packet disposition boundary

```
pub enum PacketDisposition {
    QueuedForTransport,
    DroppedUnprotected,
    DroppedPeerSessionUnavailable,
}
if require_peer_session && !peer_session_ready() {
    return Ok(PacketDisposition::DroppedPeerSessionUnavailable);
}
```

Proof sketch. At the core boundary, Eq. (3) requires readiness, route validity, frame validity, and freshness. At the platform boundary, protected routes and the kill switch must direct packets only to the core and must remain installed during failure. If both obligations hold, absence of readiness implies every protected packet is dropped with a reason. A platform route leak would violate the second obligation even if the core is correct, so release tests must cover the composed system. □

Resource bounds. Public helpers apply TLS, service admission, hot token rotation, revocation, request-size bounds, connection ceilings, idle timeouts, relay registration limits, payload limits, per-peer saturation quotas, metrics, alerting, and retention controls. Protocol parsers reject malformed lengths before allocation or cryptographic work where possible.

Observability. Metrics count packet ingress/egress, frames, unprotected drops, malformed drops, missing-session drops, and replay drops. Labels must remain low-cardinality and redacted. Diagnostics describe causes without recording packet payloads or raw identifiers by default.

15A / FAILURE INJECTION

Measuring the fail-closed property

"Fail closed" is measurable only when failures are injected into the composed system while protected applications continue sending identifiable test traffic. Unit tests of a drop enum are necessary but insufficient.

For scenario s, let N_s be protected probe packets emitted, D_s typed drops observed inside the tunnel boundary, R_s authenticated deliveries, and L_s cleartext packets captured on excluded physical interfaces. Define

$$LeakRate_s = \frac{L_s}{N_s}, \tag{42}$$

$$AccountingError_s = \left| N_s - (D_s + R_s) \right|, \tag{43}$$

$$Recovery_s = t(Ready_{restored}) - t(Fault_{removed}). \tag{44}$$

The required safety result is $L_s = 0$ and $AccountingError_s = 0$ for every scenario. If zero leaks are observed across N independent probes, the rough one-sided 95 percent upper bound is $3/N$; this does not prove zero leakage but makes the evidence quantitatively interpretable.

Injected fault	Required observation	Evidence artifact
Session absent or expired	protected packets become typed drops; no carrier frame emitted	core counters plus physical-interface capture
Tag or ciphertext corruption	no plaintext delivery; authentication-drop counter increments	mutation vector and tunnel capture
Replay and reordering	duplicate rejected; valid in-window reordering follows policy	packet-number trace and counter snapshot
Registry unavailable or revoked	public admission fails before dialing; private behavior follows policy	registry fixture, decision log, connection trace
Relay loss and hostile NAT	path changes or remains blocked without cleartext bypass	direct and relay captures with selected-path log
Service crash or restart	kill-switch rules persist or re-arm before routes permit traffic	OS filter inventory and packet capture
Sleep, wake, network switch	stale session and route state cannot leak during reconciliation	timestamped route, DNS, filter, and packet evidence

Listing 14: Failure-injection harness outline

```
start_protected_probe(flow_id, rate);
inject(fault);
capture(tunnel_interface, physical_interfaces, route_table, dns_state);
assert(cleartext_matches(flow_id) == 0);
assert(sent == authenticated_delivered + typed_drops);
remove(fault);
measure(time_to_authenticated_ready);
archive_signed_evidence_manifest();
```

Release evidence must report OS, build hash, configuration, fault timing, probe count, typed-drop totals, packet-capture hashes, route and DNS snapshots, recovery time, and any excluded scenario. Missing host evidence remains a blocked gate rather than a pass.

15B / STATUS VS. SPEC

Current evidence, product target, and release gates

The website describes the selected production profile, while the repository still contains implementation work that has not converged on that profile. This table separates observed evidence from target-state requirements. A target entry is not a claim of completed implementation or production deployment.

Property	Repository evidence	Website or product target	Required convergence
Session establishment	FIPS 203 parameter set 768, transcript and directional derivation	post-quantum key establishment	keep FIPS naming and publish vectors
Frame protection	custom SHAKE256 mask and 256-bit tag exists as an experimental implementation	ChaCha20-Poly1305 production profile	implement and default-enable the standard AEAD; assign SHAKE a distinct experimental suite and disable it in production
Large handshake transport	native UDP fragmentation, bounded reassembly, out-of-order and abuse tests	not explained publicly	document limits and measure lossy-path completion
Zero-classical profile	default core excludes dev QUIC and named classical dependencies; platform and optional tooling remnants remain	full-stack zero-classical operation is a target	define the profile boundary and remove all remaining classical dependencies before making a full-stack claim
Public record privacy	relay-only publication and sequence-rotating pseudonyms are documented defaults	traffic and metadata stay off-chain	add publication preview and verify defaults on every platform
Stable identity	long-term peer ID derived from device key	persistent reputation and Sybil resistance	specify rotating discovery aliases without weakening revocation
Diagnostic privacy	raw identifiers redacted by default; raw export is explicit	no logs and privacy-first operation	add privileged disclosure policy and auditable grants
Fail-closed behavior	typed drops, counters, kill-switch logic, and platform test plans	quantitatively verified leak-blocking behavior is a target	publish reproducible failure-injection captures and quantitative results before claiming measured production assurance
Registry trust	testnet model and policy paths; hosted production evidence remains gated	production on-chain admission is a target	label testnet evidence and require pinned, reproducible production-registry validation before claiming production deployment

Profile terminology. "Post-quantum data plane" means quantum-resistant key establishment and authentication plus a symmetric construction with an adequate quantum security margin. "ChaCha20-Poly1305 production profile" names the selected final-release protector. "SHAKE256 experimental profile" names the non-production construction retained for evaluation. "Full-stack zero classical" additionally requires classical algorithms to be absent from optional carriers, redaction helpers, platform signing, update plumbing, and build tooling. The current evidence does not yet establish that full-stack property.

RELEASE RULE

A property moves from target to implemented only when source, tests, platform behavior, public documentation, and reproducible evidence agree. A website statement alone is not implementation evidence; a repository unit test alone is not production system evidence.

Verification matrix and engineering conclusions

Property	Verification method	Failure detected
Handshake agreement	deterministic vectors and initiator/responder cross-check	divergent secrets or role assignment
Downgrade rejection	mutate version and suite at each message	missing suite continuity check
Record authenticity	field mutation, expiry, sequence rollback, key substitution	unsigned or stale discovery acceptance
Registry binding	wrong peer, key hash, network, chain, status, expiry	unauthorized public-mesh admission
Frame integrity	bit flips in header, ciphertext, tag, packet number	unauthenticated plaintext release
Replay safety	duplicates, reordering, window edge, generation rotation	double acceptance or stale replay state
Route safety	protected, excluded, malformed, IPv4 metadata cases	leak or route-policy bypass
Platform composition	sleep/wake, network change, crash, restart, upgrade	transient leak or stale session readiness

Let E be unauthorized protected-packet delivery. A union bound over the composed enforcement layers gives

$$\begin{aligned}\Pr[E] \leq\ & \mathrm{Adv}_{QL}^{ake} + \mathrm{Adv}_{forge}^{record} + \mathrm{Adv}_{frame}^{int\text{-}ctxt} + \Pr[\neg Fresh(pn)] \\ & + \Pr[PolicyBypass] + \Pr[PlatformLeak] + \Pr[StateRollback]. \end{aligned} \tag{28}$$

This is a reduction map, not a numerical claim. Each non-primitive term is an explicit engineering obligation with a corresponding test family in the matrix above.

Engineering conclusions.

1. The core security argument is compositional: FIPS 203 agreement, transcript and suite binding, FIPS 204/205 device signatures, registry admission, authenticated frames, replay state, and native route enforcement must all hold.
2. The blockchain is an identity and authorization input, never a traffic or session subsystem. This separation is the architectural differentiator.
3. Server minimization does not mean no services. Rendezvous, STUN, TURN, relay, updates, and entitlement verification remain bounded helper functions outside the packet plaintext boundary.
4. Carrier cryptography is not the product trust anchor. The QuantumLink app-layer session binds identity and post-quantum establishment across direct and fallback paths.
5. Release assurance must test the composed native system, not only the Rust primitives. Route installation, kill-switch persistence, FFI/IPC, secret storage, signing, and updates are part of the security boundary.

FINAL PROPERTY

For an admitted peer and a live session generation, a protected packet is delivered only if route policy accepts it, its authenticated frame verifies under the direction-specific key, and its packet number is fresh. Registry, discovery, and relay services never receive packet plaintext or session keys.

Executable security gates

The implementation reduces acceptance to explicit typed checks. For input x, session S, record R, and registry state D:

$$Accept(x) = ParseBounded(x) \wedge Verify(R) \wedge Admit(R, D) \wedge Ready(S) \wedge Authenticate_S(x) \wedge Fresh(x.pn). \tag{29}$$

No carrier-specific branch may weaken this conjunction.

Listing 15: Repository-derived suite and version gate

```
fn validate_suite(version: u8, suite: &str) -> Result<QlinkCryptoSuite> {
    if version != PROTOCOL_VERSION {
        return Err(QlinkError::Protocol(format!(
            "unsupported protocol version {version}"
        )));
    }
    suite_from_identifier(suite)
}

pub fn suite_from_identifier(suite: &str) -> Result<QlinkCryptoSuite> {
    match suite {
        SUITE_FIPS203 => Ok(QlinkCryptoSuite::Fips203MlKem),
        SUITE_FIPS204 => Ok(QlinkCryptoSuite::Fips204MlDsa),
        SUITE_FIPS205 => Ok(QlinkCryptoSuite::Fips205SlhDsa),
        _ => Err(QlinkError::Protocol(format!("unsupported suite {suite}"))),
    }
}
```

Listing 16: Repository-derived protected packet gate

```
if !self.policy.protects(destination) {
    self.metrics.dropped_unprotected += 1;
    return Ok(PacketDisposition::DroppedUnprotected);
}
if self.require_peer_session &&
   self.peer_session_block_reason(PeerSessionDirection::Outbound).is_some() {
    self.metrics.dropped_peer_session_unavailable += 1;
    return Ok(PacketDisposition::DroppedPeerSessionUnavailable);
}
normalize_ipv4_packet(&mut normalized_packet)?;
let frame = self.frame_codec.encode_transport_frame(
    self.next_packet_number, protocol_family, &normalized_packet)?;
```

The first gate rejects unknown versions and suites before cryptographic state is created. The second applies route policy and live-session readiness before framing. Together with authenticated receive processing, these are the executable counterparts of Eqs. (3), (26), and (29).

Implementation reference. The authoritative engineering surfaces are `qlink-core/src/crypto.rs`, `packet_core.rs`, `mesh_transport.rs`, `dytallix_identity.rs`, the Dytallix registry crate, platform architecture documents, the product specification, and the repository threat model.