BMB NEXUS · GENESIS PLATFORM

SERAPHUM: A Sovereign Post-Quantum Sealing Layer

The lattice cryptography beneath Genesis. A Category 1 LWE key-encapsulation mechanism that seals every perpetual license without a server in the trust path, and a Category 5 ML-KEM-1024 / ML-DSA-87 stack — implemented from specification, validated byte-for-byte against NIST vectors, with an explicit account of what SERAPHUM does not protect against.

Technical Specification · BMB Nexus
Updated 2026-08-04 · 10 pages
https://genesis.bmbnexus.ai/whitepapers/seraphum-seal

Cryptography · Post-Quantum · Lattice · Key Encapsulation · Licensing · Side Channels

SERAPHUM: A Sovereign Post-Quantum Sealing Layer

Technical Whitepaper v1.0

BMB Nexus · Genesis Platform
Specification v1.0 · 2026-08-04 · genesis.bmbnexus.ai/whitepapers/seraphum-seal
Security correspondence: care@bmbnexus.ai

Abstract

We present SERAPHUM, the lattice-cryptographic sealing layer beneath the Genesis platform. SERAPHUM exists to answer a question most software never has to ask: **how do you make a perpetual licence enforceable when there is no server in the trust path?** Genesis ships as a local-first product that must continue working when its vendor is unreachable — or gone — so the binding between a purchase and a machine cannot depend on a live authority. SERAPHUM performs that binding cryptographically. It comprises two layers. The **sealing layer** is a module-lattice key-encapsulation mechanism (module-LWE) operating at **NIST Category 1** parameters, producing an 800-byte encapsulation key, used to seal per-install material against a machine. The **communications layer**, SERAPHUM-PQ, operates at **NIST Category 5**: a hybrid ML-KEM-1024 ⊗ X25519 key agreement for confidentiality and a hybrid Ed25519 ⊕ ML-DSA-87 signature for authentication, with a continuous post-quantum ratchet. Both are implemented from specification with **zero third-party cryptographic runtime dependencies**. This document states the parameters exactly, describes the validation regime that makes the implementations checkable rather than merely asserted, reports a measured timing side-channel and its mitigation, and defines the trust boundary — including a candid section on what SERAPHUM does not defend against.

Keywords: Module-LWE, Key Encapsulation, Post-Quantum Cryptography, ML-KEM (FIPS 203), ML-DSA (FIPS 204), Hybrid Cryptography, NIST ACVP, Known-Answer Tests, Timing Side Channels, Perpetual Licensing, Local-First Software

---

1. Introduction

The dominant commercial model for software is the recurring subscription, and it carries an architectural consequence that is rarely stated plainly: **the vendor must remain reachable for the product to remain usable.** Entitlement is checked against a server, so the server becomes load-bearing. When it is withdrawn — through business failure, acquisition, policy change, or simple neglect — the software stops.

Genesis is sold once and owned permanently. That commitment is only meaningful if it survives our absence, which forces a different design: entitlement must be bound to the customer's own machine by mathematics rather than by an authority that answers requests. SERAPHUM is that binding.

SERAPHUM is one of four layers in AETHERION, the Genesis security shield. FORTRESS protects data at rest. VEIL protects code integrity. AETHER protects communications. **SERAPHUM protects the seal** — the cryptographic association between an entitlement, a per-install key, and a machine. It is the layer the other three rest on, and until now it was the only one without a public specification.

We publish it for the ordinary reason cryptography is published: a scheme whose security depends on its design being secret is not a secure scheme. Everything below is stated so that a reader can evaluate it, and the validation artefacts are structured so that a reader can re-run them.

---

2. Threat Model

SERAPHUM is designed against the following adversaries.

2.1 The offline adversary. An attacker in possession of a machine's stored artefacts — a stolen disk, a cloned image, a recovered backup — attempting to extract sealed material or transplant an entitlement onto different hardware. This is the primary adversary for the sealing layer.

2.2 The network adversary. An attacker who observes, records, modifies, replays, or blocks traffic between an install and any endpoint it contacts. SERAPHUM assumes the network is fully hostile and that transport security may be absent or defeated.

2.3 The harvest-now-decrypt-later adversary. An attacker recording ciphertext today against the prospect of a cryptographically relevant quantum computer later. This adversary motivates the entire post-quantum posture: material sealed now must remain sealed against decryption attempted decades from now.

2.4 The curious relay. An intermediary — including infrastructure we operate ourselves — that forwards material and would learn its contents if the scheme permitted it. SERAPHUM is designed so that our own infrastructure is not a party to be trusted.

2.5 The observer with a stopwatch. An attacker measuring the wall-clock duration of key operations to infer secret-dependent branching. Section 7 documents a real instance of this class that we found, measured, and mitigated.

Explicitly out of scope for this layer: an adversary with privileged code execution on an unlocked machine while the software is running. Binding an entitlement to hardware is not an anti-tamper function, and SERAPHUM does not claim to perform one. **That adversary is in scope for AETHERION as a whole** — it is addressed by FORTRESS and VEIL, which govern runtime secrets and executable integrity respectively. Section 10 states this boundary in full.

---

3. The Sealing Layer

The sealing layer is a key-encapsulation mechanism whose hardness rests on the **Module Learning With Errors** problem — the same foundation as the CRYSTALS-Kyber family. Recovering a secret from a public key requires solving a lattice problem for which no efficient classical or quantum algorithm is known.

3.1 Parameters. The scheme operates at the following parameters, which correspond to NIST Security Category 1 (comparable to the effort required to exhaust a 128-bit symmetric key):

- Polynomial degree **n = 256**
- Prime modulus **q = 3329**, chosen because it is prime and satisfies q ≡ 1 (mod 256), which admits the number-theoretic transform
- Module rank **k = 2**
- Centered-binomial noise bounds $\eta_1 = 3$, $\eta_2 = 2$
- Encapsulation key: **800 bytes**; keypairs are derived deterministically from a 32-byte seed

3.2 A precise statement of standing. These parameters are identical to those of ML-KEM-512 (FIPS 203). **The implementation is nonetheless not ML-KEM-512 and must not be described as such.** It is a sovereign module-LWE KEM sharing the parameter set, and it has not been submitted for FIPS validation. We state this prominently because the distinction is exactly the kind that vanishes in marketing language and then embarrasses a technical reader. The security claim is Category 1 lattice hardness; the compliance claim is none.

3.3 Construction. Sealing is hybrid. The KEM encapsulates a 32-byte symmetric key; that key seals the payload under **AES-256-GCM**. The lattice layer therefore protects a short key rather than bulk data, and the authenticated encryption layer provides integrity: a modified sealed blob fails its authentication tag rather than decrypting to plausible garbage.

3.4 Why Category 1 here. The sealing layer protects a binding, not a conversation. Its material is per-install, machine-scoped, and carries no user content. Category 1 lattice hardness places this far outside any tractable attack while keeping the operation light enough to run on modest hardware during startup. The communications layer, which carries actual user content and is exposed to harvest-now-decrypt-later recording, is deliberately built at Category 5 instead. **The asymmetry is a decision, not an oversight** — and Section 11 states the conditions under which we would revisit it.

---

4. The Seal Protocol

A seal request binds three independent factors: **a machine identity**, **a per-install public key**, and **a build attestation**. All three must be present and mutually consistent for a seal to be issued. Requests that fail this check are rejected with a specific error rather than silently downgraded to a weaker path.

4.1 Fail-loud resolution. A design rule governs the entire seal path: **an unresolvable factor terminates the request.** There is no fallback to a default, no permissive branch, no path by which a malformed or unrecognised request is honoured on weaker terms. Silent downgrade is the mechanism by which most licensing systems are defeated, and it is prohibited here by construction.

4.2 Continuity across releases. Superseded build attestations remain valid indefinitely. An install that was sealed under an earlier release continues to seal correctly after subsequent releases ship. **Nothing is ever revoked as a side effect of us shipping software.** This is a direct consequence of the perpetual-ownership commitment: a customer who never updates must never be stranded by our release cadence.

4.3 Ambiguity is not denial. A network failure, a timeout, or an unreachable endpoint is not evidence that an entitlement is invalid. The client distinguishes an explicit, authenticated rejection from an inconclusive result, and only the former affects local state. An earlier revision of this logic did not draw that distinction, and a transient network fault could clear a legitimate activation. The behaviour was corrected to require an explicit rejection, and the underlying principle is now a standing rule: **an ambiguous result must never be interpreted as a definitive denial.**

---

5. SERAPHUM-PQ: The Category 5 Layer

SERAPHUM-PQ is the cryptographic core of AETHER. It is post-quantum on **both** axes — confidentiality and authentication — which is less common than it sounds; many systems described as post-quantum protect only the former and leave signatures classical.

This section specifies the *primitives*. The protocol built on them — the two handshakes, the blind relay, sealed sender, group calls, federation, and the full communications threat model — is specified in the AETHER whitepaper. A reader evaluating the cryptography should read both: this document establishes that the primitives are correct, and that one establishes that they are correctly composed.

5.1 Confidentiality — hybrid ML-KEM-1024 ⊗ X25519.

- ML-KEM-1024 (FIPS 203) at n = 256, q = 3329, **k = 4**, $\eta_1 = \eta_2 = 2$, $d_u = 11$, $d_v = 5$
- Encapsulation key **1568 bytes**; shared secret 32 bytes
- **NIST Category 5** — the highest defined level

The lattice secret and the elliptic-curve secret are combined into a single AEAD key, which then seals the payload under AES-256-GCM. **The construction fails only if both lattice problems and elliptic-curve problems fall.** Pure post-quantum schemes stake everything on relatively young assumptions; pure classical schemes are defeated outright by a sufficiently capable quantum adversary. Hybrid is the conservative position, and it is the transitional posture NIST itself recommends — we are aligned with caution here, not ahead of it.

5.2 Authentication — hybrid Ed25519 ⊕ ML-DSA-87.

- ML-DSA-87 (FIPS 204, CRYSTALS-Dilithium), security level 5
- Public key **2592 bytes**, secret key **4896 bytes**, signature **4627 bytes**

- **Both** signatures are mandatory; verification requires both to pass

These sizes are asserted against their structural derivation when the module loads. A build in which a parameter has drifted does not start and produce subtly wrong signatures — **it refuses to load.**

5.3 Continuous post-quantum ratcheting. Beyond the handshake, SERAPHUM-PQ runs a double ratchet in which the root is seeded by the hybrid handshake, the continuous ratchet is X25519 (providing forward secrecy and post-compromise security), and **every ratchet step folds a fresh lattice secret into the root derivation.** The consequence is that post-quantum protection is a property of every chain rather than of the handshake alone — a distinction that matters precisely against the harvest-now-decrypt-later adversary of Section 2.3.

---

6. Validation

A from-scratch cryptographic implementation is a liability unless it is checkable. **We consider the validation regime the substantive claim of this document** — more so than the architecture, which is largely conventional. Sovereignty without verification is not a virtue.

6.1 Primitive-by-primitive gating. The signature implementation was built as an ordered sequence of primitives — modular reduction, the number-theoretic transform, sampling, rounding, packing, then the scheme itself — in which **each stage was proven correct before the next was written.** The intent is to make a defect localisable: an error surfaces at the stage that introduced it rather than as an inscrutable failure in an assembled system.

6.2 Proof rather than round-trip. Round-trip tests are weak evidence for transforms, because a transform and its inverse can be wrong in mutually cancelling ways and still round-trip perfectly. The number-theoretic transform is therefore validated against a **schoolbook negacyclic convolution** — an independent, obviously correct implementation — rather than against its own inverse. Modular reduction is checked for exact agreement against arbitrary-precision arithmetic.

6.3 Byte-for-byte agreement with an independent implementation. Key generation, signing, and verification are compared against a well-regarded independent library and required to match **byte for byte**, not merely to interoperate. Interoperability tolerates divergent-but-compatible behaviour; byte equality does not.

6.4 Official test vectors. The signature implementation is validated against **NIST ACVP** vectors for ML-DSA-87, and the KEM against published known-answer tests. Both vector sets are committed alongside the implementation so that validation is reproducible by a third party rather than asserted by us.

6.5 Adversarial tests. The suite includes tests that attempt to defeat the ratchet, to extract meaning from intercepted sealed blobs, and to confuse multi-device and group-key handling — tests written to fail the system rather than to confirm it.

6.6 Generated constants. Lattice schemes depend on large tables of precomputed roots of unity. A single mistyped entry produces an implementation that is wrong in ways ordinary testing may not reveal. **Every such table in SERAPHUM is computed at load from its defining root, never transcribed**, and derived scaling factors are asserted against their known values. There is no hand-entered constant available to be entered incorrectly.

7. Timing Side Channels

Lattice signature schemes sign by rejection sampling: candidates are generated and discarded until one satisfies a bound. The number of iterations depends on secret material, so the wall-clock duration of a signing operation leaks information about it.

7.1 The measurement. We measured this in our own implementation. On a single core, signing duration varied from approximately **9.8 ms to 130 ms — a factor of roughly thirteen.** That is not a theoretical margin; it is a swing an attacker can read with an ordinary clock.

7.2 The mitigation, stated precisely. First-contact key operations are wrapped so that they return at a **fixed wall-clock deadline** regardless of how long the underlying computation took. An observer timing the operation reads the same value whatever the secret.

7.3 What this is not. This is a deadline mask, **not constant-time arithmetic at the primitive level.** It removes the wall-clock signal available to a remote or coarse-grained observer. It does not claim to eliminate microarchitectural side channels — cache timing, branch prediction, or power analysis — available to an adversary executing code on the same physical machine. We draw this distinction sharply because a reader entitled to evaluate the claim deserves its exact shape.

7.4 Scope. The mask is applied to first-contact and key operations only, never to per-message operations. Masking imposes a fixed latency floor; paying it once per new peer is imperceptible against network establishment, while paying it per message would degrade ordinary use for no additional benefit.

---

8. Implementation Posture

8.1 No third-party cryptographic runtime dependencies. Both layers are implemented against platform primitives only. The reasoning is supply-chain exposure: a cryptographic dependency is a package that can be compromised upstream and shipped into the trust core of every install. That risk is traded for implementation risk, which we consider the more manageable of the two *only because* of the validation regime in Section 6. Absent that regime the trade would be indefensible, and we would not make it.

8.2 Frozen primitives. The sealing KEM is treated as frozen mathematics. It is not modified in place, and the copy shipped in the client is a byte-identical vendoring of the canonical implementation rather than a parallel edit. Cryptographic primitives that drift between copies are a well-documented source of subtle, dangerous failure.

8.3 Authenticated encryption everywhere. All sealing uses AES-256-GCM. There is no unauthenticated encryption path in SERAPHUM. Tampering produces an authentication failure rather than a plausible plaintext.

---

9. The Trust Boundary

SERAPHUM holds under the following conditions.

- **Secrets remain on the machine that generated them.** There is no escrow, and no mechanism exists by which we can recover a secret we never held.
- **Relaying infrastructure is not trusted.** Infrastructure carrying sealed material — including infrastructure we operate — handles ciphertext it cannot open. It is a router, not a party.
- **Network security is not assumed.** Transport protection is treated as a hygiene measure, not as a security boundary. SERAPHUM's guarantees hold on a fully hostile network.
- **Availability is not required.** An unreachable endpoint degrades to a deferred state, never to a denial. The product is designed to keep working when we are not there — that is the entire point.

---

10. What SERAPHUM Does Not Protect Against

We regard this section as the most important in the document. A security whitepaper without one is a marketing document.

10.1 Privileged execution inside a running endpoint — a layer boundary, not a platform boundary. SERAPHUM alone does not claim to defeat privileged execution on an unlocked, running machine. A sealing layer binds an entitlement to hardware; it is not an anti-tamper system, and it would be dishonest to present it as one.

That limit is a statement about *this layer*, and it should not be read as a statement about the platform. Within AETHERION, resistance to code inspection, runtime instrumentation, key extraction, and unauthorised modification is the responsibility of FORTRESS — which governs runtime secrets and protected state — and of VEIL, which protects executable identity and integrity on hardware the operator fully controls. The layers are deliberately separated so that each can state its own boundary precisely rather than every document making a vague claim about all of them.

10.2 The user's own decisions. A user who discloses a master password, or who authorises an action they did not understand, defeats the layer by legitimate means.

10.3 Microarchitectural side channels. As stated in Section 7.3, the timing mitigation addresses wall-clock observation. Co-resident cache, branch-prediction, and power analysis are not claimed to be defeated.

10.4 Novel cryptanalysis. Module-LWE is a comparatively young assumption. A mathematical advance against structured lattices would affect SERAPHUM as it would affect every deployed lattice scheme. The hybrid construction in the communications layer is precisely the hedge against this, and it is why the hybrid is not treated as optional.

10.5 Implementation defects. The validation regime is extensive, but extensive testing establishes the absence of the defects it tests for and nothing more. **SERAPHUM has not undergone independent third-party audit.** We state this plainly rather than allowing the breadth of Section 6 to imply otherwise.

10.6 Metadata. Sealing protects content. An adversary positioned to observe traffic may still learn that communication occurred, and approximately when. Content confidentiality and traffic-pattern confidentiality are different properties, and only the former is claimed.

---

11. Status and Future Work

11.1 Deployed. The sealing layer is in production and seals every install. The Category 5 communications stack is in production, validated against official vectors, and carries live traffic.

11.2 Under evaluation — parameter parity. The sealing layer operates at Category 1 while the communications layer operates at Category 5. Section 3.4 gives the reasoning. We are nonetheless evaluating migration of the sealing layer to ML-KEM-1024, principally so that a single statement covers the entire platform and no reader has to hold two levels in mind. Because superseded attestations remain valid indefinitely (Section 4.2), such a migration can be performed without stranding existing installs.

11.3 Sought — independent audit. The natural and correct next step for any self-implemented cryptographic stack is external review. We consider Section 6 a necessary precondition for that review, not a substitute for it.

11.4 A note on how we write about this. During development, an internal sealing routine was found to report success while producing output it had not actually sealed. It was detected and corrected before reaching a release. We record it here because it is the exact failure a security document should be built to prevent: **a cryptographic interface must never report a success it cannot honour**, and a cryptographic claim must never outrun the code that supports it. Every claim in this document was checked against the implementation before it was written down, and any claim we could not verify was removed rather than softened.

Appendix: SERAPHUM within AETHERION

AETHERION is the Genesis security shield. It is four layers with four distinct responsibilities, and each is specified in its own document so that no layer has to make vague claims on behalf of the others.

The thesis the four layers share: ARIA trusts the operator to use her. She does not trust any process — including the operator's — to dismantle her. The master password authorises ARIA to operate on the user's behalf and to open the user's own data through approved interfaces. It does not authorise extraction of raw key material, inspection of protected implementation, or modification of the security boundary. **Protecting the user's data and protecting the system's internals are different promises, and AETHERION keeps both.**

THE FOUR LAYERS OF AETHERION

FORTRESS — The Vault

Runtime secrets, protected state, and user-owned data at rest.

SERAPHUM — The Seal

Binds entitlement, installation, and machine cryptographically. *This document.*

AETHER — The Voice

Every message, call, and file that leaves the endpoint. Built on SERAPHUM-PQ (Section 5).

VEIL — The Skin

Executable identity and integrity on hardware the operator fully controls.

Companion reading: the AETHER whitepaper specifies the communications protocol built on the Category 5 primitives described in Section 5. The FORTRESS whitepaper specifies the data layer.

Cite as: BMB Nexus. “SERAPHUM: A Sovereign Post-Quantum Sealing Layer.” Genesis Platform Technical Specification v1.0, 4 August 2026. https://genesis.bmbnexus.ai/whitepapers/seraphum-seal