ZelEn

A Quantum-Safe Governed Object-Security Architecture for the Zelfire / ZelC Ecosystem

Technical Whitepaper, ZELEN-PQ5 Profile, Version 1.0

Haja Mo
Rocheston / Zelfire / ZelC
architecture@rocheston.com

Abstract

ZelEn is the quantum-safe encryption, identity, certificate, and secure object architecture of the Zelfire and ZelC ecosystem. It is not a new public-key primitive, and it is not presented as unbreakable, military grade, or as a replacement for NIST cryptography. ZelEn is a *governed, misuse-resistant, enterprise object-security architecture* that operationalizes NIST-standardized post-quantum cryptography—specifically ML-KEM (FIPS 203), ML-DSA (FIPS 204), AES−256−GCM (NIST SP 800-38D), and the KMAC256 / cSHAKE256 family (NIST SP 800-185)—into concrete key files, certificate files, encrypted object containers, compiler-enforced policy, and an optional advanced structural encoding layer.

The central thesis of this paper is the following:

> *ZelEn does not replace NIST post-quantum cryptography; it operationalizes it into a governed, misuse-resistant, enterprise object-security architecture for the Zelfire ecosystem.*

NIST builds the cryptographic engines. ZelEn builds the armored vehicle around them.

We present (i) the cryptographic suite ZELEN-PQ5 (Suite ID `0x0101`); (ii) the four native object classes `.zkey`, `.zpub`, `.zcert`, and `.zelen`, with type-level role separation; (iii) a fixed binary container with the ASCII magic `ZELEN` at offset `0x00` and the ASCII brand mark `ZELC` at offset `0x7D`, both bound into the authenticated transcript; (iv) a domain-separated key schedule built on KMAC256; (v) AEAD payload encryption with AES−256−GCM under a strict 96-bit nonce regime; (vi) an authenticated transcript signed under ML-DSA with optional SLH-DSA diversity; (vii) multi-recipient and chunked profiles; (viii) parser safety rules; (ix) ZelC compiler-level enforcement; (x) an optional finite-site/topos-inspired encoding extension (ZelEn-MTE) presented as non-load-bearing for confidentiality; (xi) a formal security model with a concrete advantage bound that reduces ZelEn object security to its underlying primitives modulo encoding and parser failure probabilities; and (xii) a non-conformant illustrative Python prototype.

We make explicit what ZelEn does and does not claim, develop the threat model, present formal definitions and proof sketches against an adaptive chosen-ciphertext adversary, anticipate criticism from cryptographers and CISOs in a hostile question-and-answer section, and document a compliance and validation roadmap that distinguishes *using FIPS-standardized algorithms* from *having a FIPS-validated cryptographic module.*

Keywords: post-quantum cryptography; ML-KEM; ML-DSA; AES-256-GCM; KMAC; authenticated encryption; secure object container; certificate; key lifecycle; governance; algorithm agility; FIPS 140-3; misuse-resistant cryptography.

Document class: Architecture Whitepaper. **Cryptographic profile:** ZELEN-PQ5 (Suite ID `0x0101`).

Foundation standards: FIPS 197, FIPS 203, FIPS 204, FIPS 205, NIST SP 800-38D, NIST SP 800-90A/B/C, NIST SP 800-108 Rev. 1, NIST SP 800-185, NIST SP 800-227. **This document is a specification and analysis; it is not a substitute for a FIPS 140-3 module validation, an independent cryptographic review, or a Common Criteria evaluation.**

Revision notes (v1.0, this draft): Round 1: (i) updated SP 800-227 citation to its September 2025 final; (ii) updated SP 800-90C from draft to its September 2025 final; (iii) added SP 800-108 Rev. 1 as the foundation for the ZelEn KMAC-based KDF profile; (iv) reframed the choice of ML-KEM-1024 as a conservative archival selection alongside NIST's ML-KEM-768 general-use recommendation; (v) replaced concrete 2^{-256} primitive-advantage figures with NIST Category-5 strength language; (vi) separated the AES-256 confidentiality target from the 128-bit GCM tag authentication target; (vii) made all KMAC256 invocations explicit in the SP 800-185 form KMAC256(K, X, L, S) with L in bits; (viii) used full FIPS 205 parameter-set names (SLH-DSA-SHA2-256s, SLH-DSA-SHAKE-256s); (ix) made the v1 SHA−256 transcript-hash choice explicit, with collision-margin commentary; (x) tightened the hybrid combiner with length-prefix injectivity and a robustness statement; (xi) defined the sourcing rule for policy_hash (computed from the verified certificate's policy block); (xii) defined the multi-recipient header authentication keying explicitly; (xiii) replaced the under-specified chunk-nonce XOR mode with a single KMAC256-derived construction; (xiv) resolved the chunked-mode plaintext-release tension via explicit Mode S (strict, default) and Mode P (provisional, opt-in); (xv) relabeled the Python listing as a non-conformant illustrative prototype.
Round 2: (xvi) replaced the truncated-PRF chunked-nonce construction with the injective 32-bit prefix ‖ 64-bit counter form, removing any uniqueness overclaim; (xvii) defined recipient_set_hash for the multi-recipient interpretation of the offset-`0x31` field, with explicit AAD and signature-transcript binding; (xviii) brought every remaining KMAC256 invocation (header table, multi-recipient wrap, notation appendix) into the canonical SP 800-185 KMAC256(K, X, L, S) form with L always in bits; (xix) strengthened the hybrid combiner per SP 800-227 composite-KEM guidance to include component ciphertexts, encapsulation keys, suite ID, and parameter-set identifiers; (xx) replaced every remaining "Python reference implementation" phrase with "illustrative prototype"; (xxi) made decryption-algorithm sourcing explicit for fp_S, vk_S, n, and the $C \,\|\, T$ split; (xxii) gave k_{sigctx} an operational role via the SIG_BIND keyed binding under which the object signature is computed; (xxiii) reattributed the ML-KEM-768 default recommendation to FIPS 203; (xxiv) added an ML-DSA signing-mode subsection specifying pure-mode signing of digest-as-message with context strings `"ZELEN-CERT-v1"` and `"ZELEN-OBJECT-v1"`; (xxv) reframed $\varepsilon_{\text{parse}}$ as an explicit implementation-assurance assumption rather than as a cryptographic-assumption term.

Contents

1 Introduction and Motivation

1.1 The Primitive-to-Platform Gap

The post-quantum migration is no longer a research exercise. The U.S. National Institute of Standards and Technology has published FIPS 203 (Module-Lattice-Based Key-Encapsulation Mechanism Standard, ML-KEM), FIPS 204 (Module-Lattice-Based Digital Signature Standard, ML-DSA), and FIPS 205 (Stateless Hash-Based Digital Signature Standard, SLH-DSA). Enterprises are now responsible for translating these primitives into deployable file formats, certificate models, identity systems, audit trails, key lifecycle controls, and governance frameworks. The cryptographic community has provided correct, conservative, peer-reviewed primitives. It has not provided—and was never intended to provide—an enterprise-ready data-at-rest architecture.

History furnishes a sobering catalogue of failures that arose not from broken primitives but from broken *compositions*. The BEAST, CRIME, Lucky 13, POODLE, ROBOT, and Bleichenbacher families of attacks did not undermine AES or RSA as primitives; they exploited padding, error handling, metadata authentication, and version negotiation. The same pattern would be expected, with greater severity, in the post-quantum era for at least three reasons:

(i) Post-quantum primitives are larger, parameter-sensitive, and impose more demanding transcript discipline than their classical predecessors. An ML-KEM-1024 ciphertext is 1568 bytes; an ML-DSA-87 signature is 4627 bytes. Naive concatenation of these into ad-hoc envelope formats is a substantial and largely under-explored attack surface.

(ii) The transition forces wholesale rewriting of cryptographic envelopes that have been stable for two decades. New code is, on average, less reviewed and less battle-tested than the code it replaces.

(iii) Hybrid migration modes, combining classical and post-quantum primitives, multiply the composition surface.

A correctly executed ML-KEM encapsulation does not, by itself, produce an enterprise-grade encrypted document. It produces a 32-byte shared secret. Everything between that shared secret and a usable, auditable, signed, identity-bound, policy-controlled, forensically traceable file is engineering. Engineering decisions made carelessly are where systems fail. ZelEn is the engineering layer.

1.2 What This Document Is and Is Not

This document specifies and analyzes the ZelEn architecture and the ZELEN-PQ5 cryptographic profile. It is intended for cryptographic reviewers, security architects, enterprise CISOs, regulatory auditors, and implementers. The document combines:

- normative specification of object formats, binary layouts, and cryptographic algorithms;
- formal definitions of security goals and a concrete advantage bound;
- a non-conformant illustrative Python prototype;
- a compliance roadmap;
- a hostile question-and-answer section that anticipates criticism.

This document is *not* a substitute for: an independent cryptographic review of any specific implementation; a FIPS 140-3 module validation; a Common Criteria evaluation; or jurisdiction-specific regulatory certification.

1.3 Notation and Conventions

Throughout, $\{0,1\}^n$ denotes the set of n-bit strings. We write $x \xleftarrow{\$} S$ for the assignment of a uniformly random element of S to x. Concatenation is written $a \,\|\, b$ or, equivalently, $a\|b$. The operator $\mathsf{Trunc}_n(x)$ denotes truncation of x to its first n bits. Hash and PRF outputs are byte strings unless otherwise noted. Hexadecimal byte values are written `0x..`. Binary offsets are written in hexadecimal, e.g., `0x7D`,

with byte ranges denoted `[a..b]` (closed at both ends). The symbol $\xleftarrow{\$}$ atop $\leftarrow$ indicates randomized sampling; the symbol $\leftarrow$ alone denotes assignment or deterministic computation. Algorithm calls are typeset in sans-serif. Object instances of native ZelEn types are typeset in `typewriter`.

We denote canonical encoding by CE, defined informally as a deterministic, injective mapping from a structured tuple to a byte string. The injectivity of CE is a load-bearing assumption in our security argument; it is discussed formally in section 23.

1.4 Document Structure

Section 2 motivates the architecture with reference to recurring deployment failures. Section 3 states the three principles. Section 4 enumerates claims and non-claims. Section 5 gives the threat model. Section 6 fixes the cryptographic suite. Sections 7 to 9 specify the objects and binary container. Sections 10 to 13 specify the cryptographic flow. Sections 14 to 18 cover lifecycle, multi-recipient, chunked mode, parser discipline, and the optional MTE extension. Sections 19 to 22 cover agility, performance, compliance, and conformance. Section 23 develops the formal security model. Section 24 answers anticipated criticism. Section 25 compares ZelEn with related standards. Sections 26 and 27 address implementation and conclude. Appendices provide the illustrative Python prototype, header layout, and test vector format.

2 Why ZelEn Exists: Ten Recurring Failure Modes

This section enumerates the ten recurring failure modes that motivate ZelEn. Each is documented with reference to a class of historical incident or anti-pattern, followed by the ZelEn response.

F1. Primitive misuse. Generic libraries (OpenSSL, BoringSSL, libsodium, liboqs) expose KEM, KDF, AEAD, and signature primitives directly. Application developers compose them in incompatible ways: forgetting the KDF, reusing a KEM shared secret as an AEAD key without domain separation, signing the plaintext but not the algorithm identifier, or omitting nonce uniqueness checks. ZelEn fixes the composition: the encryption flow is a single procedure that consumes a recipient public bundle, a sender private bundle, and a plaintext, and emits a `.zelen` object. The composition is not a parameter.

F2. Metadata tampering. Many encrypted file formats authenticate the payload but not the header. An attacker who cannot decrypt may still alter algorithm identifiers, recipient lists, key usage flags, or version fields. ZelEn authenticates the canonical fixed header as AEAD associated data, computes an additional KMAC256-derived header tag, and signs a transcript that covers the header.

F3. Key confusion. A single asymmetric key used for both encryption and signing is a known anti-pattern; it has produced concrete attacks against PGP-style envelopes and against early TLS designs. ZelEn separates roles at the type level: `.zkey` holds private KEM and signing material with policy; `.zpub` holds public KEM and verification keys with usage flags; `.zcert` binds an identity to a `.zpub` via an issuer signature; `.zelen` is the encrypted object.

F4. Identity ambiguity. A ciphertext that does not bind sender and recipient is vulnerable to surreptitious-forwarding and re-encapsulation attacks. ZelEn includes 32-byte sender and recipient fingerprints in the authenticated header and within the signed transcript. A surreptitiously-forwarded ZelEn object fails fingerprint verification.

F5. Nonce misuse. AES−256−GCM in Galois/Counter Mode is catastrophically fragile under nonce reuse: two ciphertexts under the same key and nonce reveal the XOR of the plaintexts and permit forgery. ZelEn mandates a fresh, uniformly random 96-bit nonce per object key, derived from a CSPRNG meeting NIST SP 800-90A/B/C. In chunked mode, per-chunk nonces are derived from a domain-separated KDF.

F6. Downgrade attacks. ZelEn authenticates the suite identifier, version, flags, policy hash, and certificate type. The signature transcript covers the suite identifier. An attacker cannot transparently substitute a weaker profile.

F7. Governance and policy gap. Raw cryptographic libraries expose primitives but do not enforce organizational policy: who can decrypt, with which suites, under what circumstances, and with what audit trail. ZelEn enables policy enforcement through ZelC at compile time, through SDKs at runtime, and through `.zcert` and `.zkey` policy fields at the object level.

F8. Forensics and incident response gap. Generic ciphertext is hard to identify in DLP, SIEM, EDR, and forensic recovery contexts. The fixed `ZELEN` magic at offset `0x00` and the fixed `ZELC` brand marker at offset `0x7D` give the format reliable signatures for scanners, recovery, and triage tools. Crucially, these markers do not provide cryptographic authentication; they are forensic anchors that happen to be transcript-bound.

F9. Post-quantum migration gap. Enterprises need a native, file-level post-quantum container that does not depend on rebuilding TLS or rewriting application protocols. ZelEn is that container. A `.zelen` file can sit on a network file share, be replicated to cold storage, traverse email, and be processed by DLP tools, all while preserving its post-quantum security properties.

F10. Algorithm agility gap. Cryptographic guidance evolves. ZelEn reserves suite identifiers for hybrid modes (combining a classical KEM with ML-KEM), SLH-DSA-anchored archival modes, and post-standardization KEMs such as HQC. The agility surface is itself authenticated.

3 Design Philosophy: Standards Core, Governed Architecture

ZelEn rests on three principles, stated formally below.

Principle 1 (Standards Core). The formal confidentiality and authenticity claims of ZelEn must reduce to NIST-standardized primitives. Specifically, confidentiality reduces to ML-KEM (FIPS 203), AES−256−GCM (NIST SP 800-38D), and the KMAC256 pseudo-random function and extendable-output function (NIST SP 800-185). Authenticity reduces to ML-DSA (FIPS 204), with optional SLH-DSA (FIPS 205) for hash-based diversity. *ZelEn does not invent a new public-key hardness assumption.*

Principle 2 (Governed Architecture). A standards core is necessary but not sufficient. ZelEn supplies governance: the rules, structures, and constraints that turn primitives into a deployable platform. Governance includes object semantics, fixed binary structure, transcript binding, certificate identity model, key lifecycle, compiler enforcement, parser safety rules, and policy authority.

Principle 3 (Misuse-Resistant Deployment). ZelEn is designed so that the easy path is the correct path. The native API expresses operations in terms of objects (`.zkey`, `.zpub`, `.zcert`, `.zelen`) rather than primitives. The single primary suite removes algorithm-selection foot-guns. The fixed nonce length removes a common AES−256−GCM hazard. The transcript-bound signature removes detachment between identity and ciphertext.

The engine/vehicle analogy. NIST builds the cryptographic engines: tested, peer-reviewed, parameterized, conservatively designed. ZelEn builds the armored vehicle around them: chassis, frame, doors, locks, seatbelts, dashboard, telemetry. An engine without a vehicle is not a transportation system. A primitive without a deployment architecture is not enterprise security. The engines retain their warranty; the vehicle's job is to ensure the engines are operated correctly, in a controlled environment, with logging, with defined failure modes, and with replaceable parts.

4 Claims and Non-Claims

ZelEn is offered for review against precisely what it claims. The list below is intended to be reviewable by cryptographers, auditors, and security engineers.

4.1 Claims

C1. ZelEn employs NIST-standardized post-quantum primitives, namely ML-KEM (FIPS 203), ML-DSA (FIPS 204), AES−256−GCM (NIST SP 800-38D), and KMAC256 / cSHAKE256 (NIST SP 800-185), with optional SLH-DSA (FIPS 205).

C2. ZelEn provides a governed object encryption format with a fixed binary structure, defined parsing rules, and authenticated metadata.

C3. ZelEn binds ciphertext to identity, policy, suite identifier, and certificate context through an authenticated header and a signed transcript.

C4. ZelEn reduces misuse compared with raw cryptographic libraries by exposing operations in object terms rather than primitive terms.

C5. ZelEn supports algorithm agility through a suite identifier registry that authenticates negotiated parameters.

C6. ZelEn provides Zelfire with a native encrypted object format, enabling DLP, SIEM, EDR, and recovery tools to identify, validate, and reason about post-quantum encrypted files.

C7. ZelEn provides a concrete advantage bound (theorem 23.9) reducing object security to the underlying primitives plus canonical-encoding and parser-failure probabilities.

4.2 Non-Claims

N1. ZelEn is not unbreakable. No deployable cryptographic system is unbreakable, and ZelEn explicitly inherits the assumptions of its primitives.

N2. ZelEn does not invent a new post-quantum hardness assumption. The hardness rests on ML-KEM and ML-DSA (and optionally SLH-DSA).

N3. The ZELC brand marker does not authenticate files by itself. It is a forensic marker and parser sanity checkpoint. Authenticity comes from AEAD authentication and ML-DSA signatures.

N4. The ZelEn-MTE extension is not an independent quantum-safe hardness layer. ZelEn confidentiality must hold even if the MTE design is fully public.

N5. ZelEn does not protect plaintext after decryption. Once decrypted, plaintext is governed by the host operating system, application, and operator behavior.

N6. ZelEn does not stop endpoint malware. Compromise of the host machine compromises any system on it.

N7. ZelEn does not automatically provide forward secrecy for stored files encrypted to long-term recipient public keys. If a long-term recipient decapsulation key is compromised, files previously encrypted to it may require re-encryption.

N8. ZelEn requires strong randomness, secure key storage, validated implementation, and strict policy enforcement. ZelEn cannot rescue a system whose RNG is broken or whose private keys are exfiltrated.

N9. ZelEn does not, by virtue of using FIPS-standardized algorithms, automatically inherit FIPS module validation. Production deployments targeting regulated environments should use or pursue FIPS 140-3 validated cryptographic modules where required.

These non-claims are not weaknesses. They are the boundary conditions of the design, stated honestly.

5 Threat Model

5.1 Adversary Capabilities

We consider a probabilistic polynomial-time adversary $\mathcal{A}$ with the following capabilities.

- $\mathcal{A}$ can observe and modify `.zelen` objects in transit and at rest.
- $\mathcal{A}$ can submit chosen ciphertexts to any decryption oracle exposed by an honest recipient.
- $\mathcal{A}$ can submit chosen plaintexts to any encryption oracle exposed by an honest sender.
- $\mathcal{A}$ may possess a quantum computer of cryptographically relevant scale; this is the assumed threat that motivates the post-quantum primitives.
- $\mathcal{A}$ may attempt downgrade, replay, splicing, truncation, malformation, and parser exploitation.
- $\mathcal{A}$ may issue malformed objects to probe error channels.
- $\mathcal{A}$ may publish any portion of the ZelEn specification, including the optional MTE extension. ZelEn confidentiality must survive full design disclosure.

5.2 Adversary Limitations

- $\mathcal{A}$ does not control the recipient host operating system or its memory.
- $\mathcal{A}$ does not have access to recipient `.zkey` material protected by hardware-backed storage or sealed policy.
- $\mathcal{A}$ cannot break ML-KEM, ML-DSA, AES−256−GCM, or KMAC256 within the security parameters of ZELEN-PQ5.
- $\mathcal{A}$ cannot influence the host CSPRNG used to generate fresh nonces and ephemeral material.
- $\mathcal{A}$ does not observe side channels (timing, power, electromagnetic) of the host implementation, unless the deployment context explicitly addresses such threats.

5.3 Out-of-Scope Threats

The following threats are explicitly out of scope for ZelEn:

- Side-channel attacks on host hardware, unless ZelEn is deployed within a side-channel-resistant module.
- Compromised endpoints; once plaintext is released to a calling application, it is governed by host-level controls.
- Social engineering against key custodians.
- Coercion of a key holder.
- Compromise of the underlying CSPRNG.
- Compromise of the certificate issuer's signing key (this is a trust-anchor failure that no certificate-bearing system survives).

5.4 Security Goals

Definition 5.1 (ZelEn Security Goals)**.** A ZelEn deployment with honest sender S, honest recipient R, and trusted issuer I achieves:

- **Confidentiality:** $\mathcal{A}$ without the recipient decapsulation key cannot recover plaintext, including under adaptive chosen-ciphertext attack, except with negligible advantage.
- **Authenticity:** $\mathcal{A}$ without the sender signing key cannot produce a `.zelen` object that verifies under the sender certificate, except with negligible advantage.

- **Integrity:** any modification to the ciphertext, header, certificate, or signature is detected.
- **Identity binding:** the sender and recipient are cryptographically bound to the object via fingerprints, certificates, and signed transcripts.
- **Misuse resistance:** standard usage of the SDK or compiler-enforced ZelC pathways produces objects that satisfy the above goals; deviating from the standard usage produces objects that either fail to verify or fail to encode.

6 Cryptographic Suite: ZELEN-PQ5

6.1 Suite Parameters

ZELEN-PQ5 is the primary and default ZelEn cryptographic profile.

Table 1: ZELEN-PQ5 Suite Parameters

Parameter	**Value**
Suite identifier	`0x0101` (big-endian, 16 bits)
Suite name	ZELEN-PQ5
Key encapsulation mechanism	ML-KEM-1024 (FIPS 203)
Primary signature algorithm	ML-DSA-87 (FIPS 204)
Optional diversity signature	SLH-DSA-SHA2-256s or SLH-DSA-SHAKE-256s (FIPS 205)
Authenticated encryption	AES−256−GCM (NIST SP 800-38D); 128-bit GCM authentication tag
AEAD nonce length	96 bits (12 bytes), uniformly random
Key derivation function	SP 800-108 Rev. 1 KMAC-based KDF profile, instantiated with KMAC256 and explicit domain-separation customization strings (NIST SP 800-185)
Extendable output function	cSHAKE256 where an XOF interface is required
Transcript hash	SHA−256 in v1 (engineering choice, see Remark below); SHAKE256 or SHA3−512 permitted in extended profiles for full Category-5 collision-resistance margin.
Random bit generation	NIST SP 800-90A/B/C compliant CSPRNG
Symmetric security target	256-bit (NIST Category 5 / AES-256-class)
Payload model	object encryption (data at rest)

Remark 6.1. The suite name ZELEN-PQ5 reflects alignment with NIST Category 5 / AES-256-class security targets. The PQ5 designation is a category label; it does not, and is not intended to, claim 512-bit symmetric security. The earlier name ZELEN-512 is deprecated to avoid this ambiguity.

6.2 Justification of Choices

Why ML-KEM-1024 for the default ZelEn profile. FIPS 203 recommends ML-KEM-768 as the default parameter set for general use, with ML-KEM-1024 specified for use cases requiring higher security; SP 800-227 provides additional general guidance on KEM use. ZelEn is targeted at archival and at-rest object protection, where stored ciphertext lifetimes may exceed twenty years and where

re-encryption is operationally expensive. We therefore select ML-KEM-1024 as the default for ZELEN-PQ5, accepting the additional bandwidth for a conservative archival posture, while retaining the option for a future ZELEN-PQ3 profile based on ML-KEM-768 for general-use deployments where bandwidth or latency considerations dominate. ML-KEM-1024 also aligns with AES−256−GCM-class symmetric strength, removing asymmetry between the public-key and symmetric layers in this profile.

Why ML-DSA-87 as the default signature. ML-DSA-87 is the strongest parameter set defined in FIPS 204. For sender authentication on objects that may be retained for long periods, the additional signature size is acceptable in exchange for a higher security target.

Why AES−256−GCM rather than ChaCha20-Poly1305 or AES-GCM-SIV. AES−256−GCM is universally available, FIPS-validated where applicable, hardware-accelerated on essentially every modern processor (AES-NI / VAES), and aligned with the symmetric target of ZELEN-PQ5. The AES−256−GCM key gives a 256-bit confidentiality target. The GCM authentication tag is 128 bits, which provides 128-bit per-tag forgery resistance subject to the usage bounds in NIST SP 800-38D (in particular, the data-volume and invocation-count limits). We reject AES-GCM-SIV for the primary profile because its misuse-resistance properties are largely subsumed by the ZelEn nonce discipline and header authentication; the cost is implementation availability and review depth, both of which favor AES−256−GCM.

Why a strict 96-bit nonce. AES−256−GCM is parameter-fragile with respect to nonce length: nonces shorter than 96 bits are derived through an alternative GHASH-based construction, introducing complexity. ZelEn pins the nonce length to 96 bits and the nonce source to a CSPRNG-driven uniform sample, eliminating a known foot-gun and aligning with the most-reviewed AES−256−GCM variant. NIST SP 800-38D explicitly recommends restricting GCM IV support to 96 bits for interoperability, efficiency, and simplicity.

Why KMAC256 / cSHAKE256 for the KDF. Domain separation is essential when deriving multiple keys from a single shared secret. ZelEn instantiates the SP 800-108 Rev. 1 KMAC-based KDF profile, which provides explicit, unambiguous domain separation through its customization-string argument and is a NIST-approved KDF construction over the 32-byte ML-KEM shared secret. We avoid HKDF-SHA-256 in the default profile because the customization-string discipline of KMAC256 is more directly verifiable in audit and because it avoids a separate HMAC dependency.

Remark 6.2 (v1 transcript hash). ZELEN-PQ5 v1 uses SHA−256 as the transcript hash on pragmatic engineering grounds: ubiquity, hardware acceleration, and validated implementations. SHA−256 provides 128-bit collision resistance and 256-bit preimage resistance under standard assumptions, which is asymmetric to the AES−256−GCM-class symmetric target of the profile. The transcript hash binding is not the load-bearing security primitive; the AEAD AAD authentication and the ML-DSA signature provide the operational integrity guarantees. Implementations targeting full AES−256−GCM-class collision-resistance margin in the transcript path *should* adopt the extended profile that uses SHAKE256 or SHA3−512 as the transcript hash. The successor v2 profile is expected to make this the default.

7 The Object Model: `.zkey`, `.zpub`, `.zcert`, `.zelen`

ZelEn defines exactly four native object classes. Their separation is a deliberate misuse-prevention design enforced at the type level by ZelC and at runtime by SDK invariants.

7.1 The Public Key Bundle (`.zpub`)

A `.zpub` holds public material: the ML-KEM encapsulation key ek_R, the ML-DSA verification key vk_R, optionally an SLH-DSA verification key vk_R^{slh}, the suite identifier, a subject identifier, key-usage rules, a policy hash, and a fingerprint. The fingerprint is computed canonically from the body of the `.zpub` and acts as the identity anchor for transcript binding.

Mathematically, the recipient's KEM keypair is generated by

$$(ek_R, dk_R) \leftarrow \mathsf{ML\text{-}KEM.KeyGen}(), \tag{1}$$

where ek_R is the encapsulation key and dk_R is the decapsulation key. The signature keypair is generated by

$$(vk_R, sk_R^{\text{sig}}) \leftarrow \mathsf{ML\text{-}DSA.KeyGen}(), \tag{2}$$

with verification key vk_R and signing key sk_R^{sig}. The `.zpub` is then

$$Z_{\text{pub},R} \;=\; \big(\text{version},\, \mathsf{suite\_id},\, \text{subject},\, ek_R,\, vk_R,\, [vk_R^{\text{slh}}],\, \text{key\_usage},\, \text{policy\_hash},\, \text{fingerprint}\big). \tag{3}$$

The fingerprint is

$$\text{fingerprint} \;=\; \mathsf{SHA{-}256}\big(\mathsf{CE}(Z_{\text{pub},R} \setminus \{\text{fingerprint}\})\big), \tag{4}$$

where $Z_{\text{pub},R} \setminus \{\text{fingerprint}\}$ denotes the body with the fingerprint field removed.

7.2 The Private Key Bundle (`.zkey`)

A `.zkey` holds private material: the ML-KEM decapsulation key dk_R, the ML-DSA signing key sk_R^{sig}, the corresponding `.zpub`, local policy, hardware-binding flags, export rules, expiry data, and lifecycle metadata.

$$Z_{\text{key},R} \;=\; \big(dk_R,\, sk_R^{\text{sig}},\, [sk_R^{\text{slh}}],\, Z_{\text{pub},R},\, \text{local\_policy},\, \text{hardware\_binding},\, \text{export\_rules},\, \text{lifecycle}\big). \tag{5}$$

> A production `.zkey` *must* be encrypted at rest and *should* be hardware-backed (HSM, TEE, secure enclave) or policy-sealed. The illustrative Python prototype in appendix C stores raw private material for demonstration only and is not suitable for production deployment.

7.3 The Certificate (`.zcert`)

A `.zcert` binds a subject identity to a `.zpub` bundle and a policy. It is signed by an issuer using ML-DSA-87 (and optionally counter-signed with SLH-DSA for diversity). A `.zcert` is not a private key; it is a signed public identity and trust object.

The to-be-signed body is

$$\text{TBS}_R \;=\; \mathsf{CE}\big(\text{subject},\, \text{issuer},\, Z_{\text{pub},R},\, \text{validity},\, \text{key\_usage},\, \mathsf{suite\_id},\, \text{policy}\big), \tag{6}$$

and the issuer's signature is

$$\sigma_I \leftarrow \mathsf{ML\text{-}DSA.Sign}\big(sk_I^{\text{sig}},\, \mathsf{H}(\text{TBS}_R),\, \text{ctx}=\texttt{"ZELEN-CERT-v1"}\big). \tag{7}$$

The certificate is

$$Z_{\text{cert},R} \;=\; (\text{TBS}_R,\, \sigma_I). \tag{8}$$

Certificate verification reduces to

$$\mathsf{ML\text{-}DSA.Verify}\big(vk_I,\, \mathsf{H}(\text{TBS}_R),\, \sigma_I,\, \text{ctx}=\texttt{"ZELEN-CERT-v1"}\big) = 1, \tag{9}$$

where vk_I is recovered from the issuer's certificate or trust anchor.

7.4 The Encrypted Object (`.zelen`)

A `.zelen` contains a fixed binary header H, an ML-KEM ciphertext block c_{kem}, an AES−256−GCM-encrypted payload C, an authentication tag T, the sender certificate chain $Z_{\text{cert},S}$, and an object signature Σ:

$$Z \;=\; (H,\, c_{\text{kem}},\, C,\, T,\, Z_{\text{cert},S},\, \Sigma). \tag{10}$$

7.5 Type-Level Role Separation

The four object classes are deliberately disjoint:

- A `.zpub` cannot be confused with a `.zkey` at the type level.
- A `.zcert` cannot be used to decrypt or to encapsulate.
- A `.zelen` cannot be created from a public key alone without also producing a signature transcript over the sender's signing key.
- A `.zkey`'s private KEM material cannot be used as a signing key, and its private signing material cannot be used as a decapsulation key.

This is enforced at three layers: by the type system in ZelC, by SDK API surfaces that take typed object references, and by the canonical encoding which includes a mandatory `type` discriminator.

8 The `.zelen` Binary Format

The `.zelen` file begins with a fixed binary header. The header is canonical: its fields are at fixed offsets, in fixed byte orders, with fixed lengths in the v1 primary profile. Variable-length cryptographic blocks follow the fixed header in length-prefixed form.

8.1 Fixed Header Layout

Table 2: ZELEN-PQ5 v1 Fixed Header Layout

Offset	Size	Field	Value / Purpose
`0x00`	5	Magic	ASCII `"ZELEN"`; identifies the file as a ZelEn object.
`0x05`	1	Version	`0x01` for v1.
`0x06`	1	Flags	Bitfield: signature-present, compression, multi-recipient, MTE, chunked-payload.
`0x07`	1	Security tier	`0x05` for PQ5.
`0x08`	1	Header length	v1 fixed-byte; future profiles use varint or 32-bit.
`0x09`	2	Suite ID	big-endian; `0x0101` for ZELEN-PQ5.
`0x0B`	2	Certificate type	big-endian; `.zcert` trust profile selector.
`0x0D`	4	Payload length	v1 32-bit; chunked mode supports 64-bit.
`0x11`	32	Sender fingerprint	SHA−256 of canonical sender `.zpub` body.
`0x31`	32	Recipient identifier	Single-recipient: SHA−256 of canonical recipient `.zpub` body. Multi-recipient: recipient_set_hash as defined in section 15.3.
`0x51`	32	KEM ciphertext digest	SHA−256(c_{kem}); binds variable KEM block to fixed header.
`0x71`	12	AEAD nonce	96-bit uniformly random nonce.
`0x7D`	4	ZelEn brand mark	exactly the bytes `0x5A 0x45 0x4C 0x43` (ASCII `"ZELC"`).
`0x81`	16	Header authentication tag	KMAC256(k_{hdr}, H_0, L=128, S=`"ZELEN v1 header tag"`).

The fixed header is exactly $0x91 = 145$ bytes. Following it are length-prefixed blocks with 32-bit big-endian length headers:

1. the ML-KEM ciphertext c_{kem};

2. the canonical encoding of the sender certificate $Z_{\text{cert},S}$;
3. the AEAD ciphertext-and-tag block $C \,\|\, T$ (with the 16-byte AES−256−GCM tag at the end);
4. the ML-DSA object signature Σ.

In multi-recipient profiles, a canonical recipient list is inserted prior to the payload block. In chunked profiles, a canonical chunk index manifest is inserted prior to the chunk stream.

8.2 Final-Form Recommendations

Remark 8.1. Production specifications targeting $\geq 4\,\text{GiB}$ payloads, deeply extensible headers, or future suite negotiation should adopt: (i) a varint or 32-bit header-length field replacing the v1 single-byte form; (ii) a 64-bit payload-length field; (iii) a defined binary canonical encoding (e.g., a constrained CBOR profile) for variable-length and structured fields. The offset semantics of `ZELEN` at `0x00` and `ZELC` at `0x7D` are fixed across all v1 profiles and are intended to be invariant in v2.

8.3 Endianness, Alignment, and Encoding Discipline

All multi-byte integer fields in the fixed header are big-endian. The canonical encoding CE used for variable-length structured content (certificates, recipient lists, chunk manifests) is required to be:

(i) **Deterministic:** for any structured input, CE produces a unique output byte string.

(ii) **Injective:** for distinct inputs, CE produces distinct outputs.

(iii) **Total:** CE is defined on every legal input and rejects illegal inputs.

(iv) **Bounded:** the encoded length is bounded by a polynomial in the size of the structured input.

A non-canonical encoding compromises the security argument (theorem 23.9) by introducing the canonical-encoding ambiguity term $\varepsilon_{\mathsf{CE}}$.

9 The Fixed `ZELC` Brand Marker at Offset `0x7D`

9.1 Specification

Every `.zelen` object *must* contain the four ASCII bytes `"ZELC"` at offset `0x7D`. The bytes are exactly:

`0x5A 0x45 0x4C 0x43`. (11)

The marker is unconditional, fixed across all v1 profiles, and embedded in the canonical authenticated header transcript.

9.2 Operational Purposes

The marker serves five operational purposes.

P1. Forensic identification. Incident responders, recovery tools, and DLP systems can detect a ZelEn object by scanning for `"ZELEN"` at offset `0x00` in conjunction with `"ZELC"` at offset `0x7D`. The double-marker pattern reduces false positives on partial files and on adversarial files that copy only the leading magic.

P2. Binary file recovery. Carving tools that operate over corrupted media (failed disks, partial backups, deleted regions) benefit from a double-anchor signature spaced $0x7D$ bytes apart.

P3. DLP and SIEM detection. Enterprise scanners can reliably classify post-quantum encrypted Zelfire-native objects regardless of file extension.

P4. Parser sanity checkpointing. A parser that fails to find `"ZELC"` at offset `0x7D` rejects before performing any cryptographic work, eliminating a class of malformed-object denial-of-service vectors.

P5. Version brand stability. The marker is invariant. There is no version field embedded inside the marker. It is a Zelfire / ZelC brand mark and a parser sanity checkpoint.

9.3 What the Marker Is Not

The `ZELC` marker is not secret.
The `ZELC` marker is not authentication.
The `ZELC` marker does not prevent forgery.
An attacker can trivially write `"ZELC"` at offset `0x7D` in any file. The marker becomes security-relevant only because it is part of the canonical fixed header that is authenticated as AEAD associated data and signed by the ML-DSA object signature. A modified or absent `ZELC` would alter the authenticated header bytes, invalidating both the GCM tag and the signature.

In short: the marker is a brand and a forensic anchor, not a security guarantee. The security guarantee comes from the cryptographic transcript that includes the marker.

10 Authenticated Transcript and Key Derivation

10.1 KMAC Notation Convention

We follow NIST SP 800-185, in which KMAC256 has the signature

$$\mathsf{KMAC256}(K,\, X,\, L,\, S), \tag{12}$$

where K is the key, X is the message, L is the requested output length *in bits*, and S is the customization string. All KMAC invocations in ZelEn are written in this canonical form, with L specified explicitly in bits. ZelEn's KDF construction is an instantiation of the SP 800-108 Rev. 1 KMAC-based key derivation profile, with the ML-KEM shared secret as the key-derivation key.

10.2 Sourcing of `policy_hash`

The value policy_hash used in the key schedule and signature transcript is defined as

$$\text{policy\_hash} \;=\; \mathsf{SHA{-}256}\big(\mathsf{CE}(\text{policy\_block})\big), \tag{13}$$

where policy_block is the canonical encoding of the certificate's policy field (suite-allowance set, key-usage flags, role attributes, expiry alignment, export rules) extracted from the verified sender certificate $Z_{\text{cert},S}$. The encryptor computes policy_hash from the in-hand certificate prior to KDF invocation. The decryptor verifies the certificate first (signature, validity window, suite policy), then computes policy_hash from the verified certificate's policy block, and only then reconstructs the KDF. A mismatch between encryptor and decryptor policy_hash values yields a header-tag failure, which is reported uniformly per the constant-error policy.

10.3 Inputs to the KDF

ZelEn key derivation is domain-separated, transcript-bound, and standards-anchored. The inputs are:

- the shared secret $K \in \{0,1\}^{256}$ from ML-KEM.Encaps;
- the canonical preliminary header H_0 (the fixed header with the header-tag region zeroed);
- the canonical encoding of $\big(\mathsf{H}(c_{\text{kem}})$, fp_S, fp_R, suite_id, policy_hash$\big)$;
- a version-bound customization string.

10.4 Pseudo-Random Key Extraction

$$\text{PRK} \;=\; \mathsf{KMAC256}\big(K,\; \mathsf{CE}(H_0,\, \mathsf{H}(c_{\text{kem}}),\, \mathsf{fp}_S,\, \mathsf{fp}_R,\, \text{suite\_id},\, \text{policy\_hash}),\; L{=}512,\; S{=}\texttt{"ZELEN v1 key schedule"}\big). \tag{14}$$

($L = 512$ bits, i.e., 64 bytes; PRK is the seed for subsequent per-purpose expansions.)

10.5 Per-Purpose Key Expansion

$$k_{\text{enc}} = \mathsf{KMAC256}\big(\text{PRK},\ \varepsilon,\ L{=}256,\ S{=}\texttt{"payload encryption"}\big), \tag{15}$$

$$k_{\text{hdr}} = \mathsf{KMAC256}\big(\text{PRK},\ \varepsilon,\ L{=}256,\ S{=}\texttt{"header authentication"}\big), \tag{16}$$

$$k_{\text{sigctx}} = \mathsf{KMAC256}\big(\text{PRK},\ \varepsilon,\ L{=}256,\ S{=}\texttt{"signature transcript binding"}\big), \tag{17}$$

$$k_{\text{nonce}} = \mathsf{KMAC256}\big(\text{PRK},\ \varepsilon,\ L{=}256,\ S{=}\texttt{"chunk nonce derivation"}\big) \quad \text{(chunked mode)}. \tag{18}$$

Here ε denotes the empty message; per-purpose differentiation is provided entirely by S, which is the SP 800-185 customization string. All output lengths are stated in bits to match the SP 800-185 KMAC interface.

Remark 10.1. Each derivation uses a distinct, version-tagged customization string. Domain separation is necessary because reusing the same key across distinct purposes is a known structural error pattern. KMAC256, with its explicit customization-string argument, makes domain separation unambiguous and standards-anchored.

10.6 Header Authentication Tag

$$t_H \ = \ \mathsf{KMAC256}\big(k_{\text{hdr}},\ H_0,\ L{=}128,\ S{=}\texttt{"ZELEN v1 header tag"}\big). \tag{19}$$

Output length is 128 bits, matching the 16-byte header-tag field at offset `0x81` and the AES−256−GCM tag length. The KMAC256-derived header tag complements the AES−256−GCM AAD authentication: AES−256−GCM authenticates the header as part of the AEAD construction; the KMAC256 header tag provides a key-confirming check that does not depend on having attempted AES−256−GCM decryption. Insert t_H at offset `0x81` to produce the final canonical header H.

10.7 Signature Transcript and Binding

$$\boxed{\text{SIG\_INPUT} \ = \ \mathsf{CE}\big(\texttt{"ZELEN-SIG-v1"},\ \mathsf{H}(H),\ \mathsf{H}(c_{\text{kem}}),\ \mathsf{H}(C),\ T,\ \mathsf{fp}_S,\ \mathsf{fp}_R,\ \mathsf{suite\_id},\ \text{policy\_hash}\big).} \tag{20}$$

The signature transcript is constructed over hashes rather than raw blocks for compactness and to enable streaming verification on large objects. The canonical encoding CE is injective by construction; this is part of the formal security argument in section 23.

The signature itself is computed over the keyed binding

$$\text{SIG\_BIND} \ = \ \mathsf{KMAC256}\big(k_{\text{sigctx}},\ \text{SIG\_INPUT},\ L{=}256,\ S{=}\texttt{"ZELEN v1 signature binding"}\big), \tag{21}$$

so that the value signed by the sender is bound by the per-object pseudo-random key k_{sigctx} derived in eq. (17). Binding the signature input to a per-object KDF-derived key prevents replay of a Σ across distinct objects that happen to share a SIG_INPUT structure but differ in derived key material, and gives the signature a tag-like dependency on the KDF output that complements the AEAD authentication of H.

10.8 ML-DSA Signing Mode and Context Strings

ZelEn instantiates ML-DSA in its FIPS 204 *pure* mode, signing a digest-as-message rather than the raw message: the inputs $\mathsf{H}(\text{TBS}_S)$ for certificates and SIG_BIND for objects are treated as opaque byte-string messages by ML-DSA.Sign. ZelEn does *not* use HashML-DSA in v1; the digest is computed by ZelEn before invocation, and ML-DSA signs the resulting message directly.

ZelEn distinguishes the two signature roles through the FIPS 204 context-string argument:

- Certificate signatures use ctx = `"ZELEN-CERT-v1"`.
- Object signatures use ctx = `"ZELEN-OBJECT-v1"`.

Distinct context strings enforce role separation at the signature layer: a Σ produced under `"ZELEN-OBJECT-v1"` cannot be re-used as a certificate signature, and vice versa, even if the underlying message bytes were to collide. The context-string mechanism is part of the FIPS 204 signing interface and does not require any additional cryptographic assumption beyond ML-DSA EUF-CMA.

11 Encryption Algorithm

11.1 Inputs and Outputs

Inputs: plaintext $M \in \{0,1\}^*$; recipient public bundle $Z_{\text{pub},R}$ with KEM key ek_R and fingerprint fp_R; sender private bundle $Z_{\text{key},S}$ with signing key sk_S^{sig} and fingerprint fp_S; sender certificate $Z_{\text{cert},S}$; suite identifier $\mathsf{suite\_id} = \texttt{0x0101}$.

Output: a `.zelen` object Z.

11.2 Algorithm

Algorithm 1 ZelEn.Encrypt (single-recipient profile)

Require: M, $Z_{\text{pub},R}$, $Z_{\text{key},S}$, $Z_{\text{cert},S}$, $\mathsf{suite\_id}$

1: policy_hash $\leftarrow \mathsf{SHA{-}256}(\mathsf{CE}(\text{policy\_block from } Z_{\text{cert},S}))$
2: $(K, c_{\text{kem}}) \leftarrow \mathsf{ML\text{-}KEM.Encaps}(ek_R)$ ▷ $K \in \{0,1\}^{256}$
3: $n \xleftarrow{\$} \{0,1\}^{96}$ ▷ uniformly random AEAD nonce
4: $H_0 \leftarrow \mathsf{BuildHeader}\big(\texttt{"ZELEN"}, 1, \text{flags}, \mathsf{suite\_id}, \mathsf{fp}_S, \mathsf{fp}_R, \mathsf{H}(c_{\text{kem}}), n, \texttt{"ZELC"}\big)$ ▷ header tag region zeroed; offset `0x31` carries fp_R in single-recipient mode and recipient_set_hash in multi-recipient mode
5: $\text{PRK} \leftarrow \mathsf{KMAC256}(K, \mathsf{CE}(H_0, \mathsf{H}(c_{\text{kem}}), \mathsf{fp}_S, \mathsf{fp}_R, \mathsf{suite\_id}, \text{policy\_hash}), L{=}512, S{=}\texttt{"ZELEN v1 key schedul}$
6: $k_{\text{enc}} \leftarrow \mathsf{KMAC256}(\text{PRK}, \varepsilon, L{=}256, S{=}\texttt{"payload encryption"})$
7: $k_{\text{hdr}} \leftarrow \mathsf{KMAC256}(\text{PRK}, \varepsilon, L{=}256, S{=}\texttt{"header authentication"})$
8: $k_{\text{sigctx}} \leftarrow \mathsf{KMAC256}(\text{PRK}, \varepsilon, L{=}256, S{=}\texttt{"signature transcript binding"})$
9: $t_H \leftarrow \mathsf{KMAC256}(k_{\text{hdr}}, H_0, L{=}128, S{=}\texttt{"ZELEN v1 header tag"})$
10: $H \leftarrow \mathsf{InsertHeaderTag}(H_0, t_H)$ ▷ header now finalized
11: $(C, T) \leftarrow \mathsf{AES{-}256{-}GCM.Enc}(k_{\text{enc}}, n, M, \text{AAD}{=}H)$ ▷ T is 128 bits
12: $\text{SIG\_INPUT} \leftarrow \mathsf{CE}(\texttt{"ZELEN-SIG-v1"}, \mathsf{H}(H), \mathsf{H}(c_{\text{kem}}), \mathsf{H}(C), T, \mathsf{fp}_S, \mathsf{fp}_R, \mathsf{suite\_id}, \text{policy\_hash})$
13: $\text{SIG\_BIND} \leftarrow \mathsf{KMAC256}(k_{\text{sigctx}}, \text{SIG\_INPUT}, L{=}256, S{=}\texttt{"ZELEN v1 signature binding"})$
14: $\Sigma \leftarrow \mathsf{ML\text{-}DSA.Sign}(sk_S^{\text{sig}}, \text{ SIG\_BIND}, \text{ ctx}{=}\texttt{"ZELEN-OBJECT-v1"})$
15: **return** $Z = (H, c_{\text{kem}}, C, T, Z_{\text{cert},S}, \Sigma)$

11.3 Notes on Steps

Step 1 (KEM encapsulation). K is a 256-bit shared secret; c_{kem} is the KEM ciphertext stored in the `.zelen` object.

Step 2 (Nonce sampling). The nonce is sampled from a CSPRNG meeting NIST SP 800-90A/B/C. ZelEn implementations *must* reject any RNG that does not satisfy these requirements.

Step 3 (Preliminary header). $H_0[\texttt{0x00..0x04}] = \texttt{"ZELEN"}$ and $H_0[\texttt{0x7D..0x80}] = \texttt{"ZELC"}$. The header authentication tag region at offset `0x81` is initialized to 16 zero bytes for the purposes of H_0.

Step 9 (AEAD payload encryption). The header H, including the magic, version, suite ID, fingerprints, KEM digest, nonce, ZELC marker, and header tag, is bound as additional authenticated data.

Step 10–11 (Signature). The sender's signing key signs the canonical signature transcript. The signature does not authenticate the plaintext directly; it authenticates the ciphertext-plus-AAD context, which is itself bound to the plaintext through AES−256−GCM.

12 Decryption Algorithm

12.1 Inputs and Outputs

Inputs: a `.zelen` object Z; the recipient private bundle $Z_{\text{key},R}$ with decapsulation key dk_R and fingerprint fp_R; a trust store containing one or more issuer verification keys vk_I.

Output: plaintext M, or a uniform error code.

12.2 Algorithm

Algorithm 2 ZelEn.Decrypt

Require: $Z = (H, c_{\text{kem}}, C, T, Z_{\text{cert},S}, \Sigma)$, $Z_{\text{key},R}$, trust store $\{vk_I\}$

if $|Z| <$ HEADER_LEN **then return** $\perp$ ▷ uniform error
end if
$H \leftarrow Z[0..\text{HEADER\_LEN} - 1]$
if H[`0x00..0x04`] $\neq$ `"ZELEN"` **then return** $\perp$
end if
if H[`0x7D..0x80`] $\neq$ `"ZELC"` **then return** $\perp$
end if
if suite ID at `0x09..0x0A` not in local policy's accepted set **then return** $\perp$
end if
Parse $c_{\text{kem}}, Z_{\text{cert},S}, B_{\text{ct}}, \Sigma$ from length-prefixed blocks
if any block truncated, oversized, or trailing bytes remain **then return** $\perp$
end if
$n \leftarrow H$[`0x71..0x7C`] ▷ 96-bit AEAD nonce
if $|B_{\text{ct}}| < 16$ **then return** $\perp$
end if
$C \leftarrow B_{\text{ct}}[0..|B_{\text{ct}}| - 17]$; $T \leftarrow B_{\text{ct}}[|B_{\text{ct}}| - 16..|B_{\text{ct}}| - 1]$ ▷ split ciphertext and 128-bit tag
if $\mathsf{H}(c_{\text{kem}}) \neq H$[`0x51..0x70`] **then return** $\perp$
end if
$vk_I \leftarrow$ trust store entry for issuer($Z_{\text{cert},S}$)
if not ML-DSA.Verify(vk_I, $\mathsf{H}(\text{TBS}_S)$, σ_I, ctx=`"ZELEN-CERT-v1"`) **then return** $\perp$ ▷ certificate first
end if
if $Z_{\text{cert},S}$ validity, key usage, or suite policy fails **then return** $\perp$
end if
$Z_{\text{pub},S} \leftarrow$ subject_public(TBS_S) ▷ extract sender public bundle from verified cert
$vk_S \leftarrow$ ML-DSA.vk from $Z_{\text{pub},S}$
$\mathsf{fp}_S \leftarrow$ SHA−256(CE($Z_{\text{pub},S} \setminus \{\text{fingerprint}\}$))
if $\mathsf{fp}_S \neq H$[`0x11..0x30`] **then return** $\perp$
end if
$\text{rid}_H \leftarrow H$[`0x31..0x50`] ▷ single-recipient: fp_R; multi-recipient: recipient_set_hash
if single-recipient mode **and** $\text{rid}_H \neq \mathsf{fp}_R$ from $Z_{\text{key},R}$ **then return** $\perp$
end if
policy_hash $\leftarrow$ SHA−256(CE(policy_block from verified $Z_{\text{cert},S}$))
$K \leftarrow$ ML-KEM.Decaps(dk_R, c_{kem})
$H_0 \leftarrow$ ZeroHeaderTag(H)
PRK $\leftarrow$ KMAC256(K, CE($H_0, \mathsf{H}(c_{\text{kem}}), \mathsf{fp}_S, \text{rid}_H$, suite_id, policy_hash), L=512, S=`"ZELEN v1 key schedu`
$k_{\text{enc}} \leftarrow$ KMAC256(PRK, ε, L=256, S=`"payload encryption"`)
$k_{\text{hdr}} \leftarrow$ KMAC256(PRK, ε, L=256, S=`"header authentication"`)
$k_{\text{sigctx}} \leftarrow$ KMAC256(PRK, ε, L=256, S=`"signature transcript binding"`)
$t'_H \leftarrow$ KMAC256(k_{hdr}, H_0, L=128, S=`"ZELEN v1 header tag"`)
if $t'_H \neq H$[`0x81..0x90`] **then return** $\perp$
end if
SIG_INPUT $\leftarrow$ CE(`"ZELEN-SIG-v1"`, $\mathsf{H}(H), \mathsf{H}(c_{\text{kem}}), \mathsf{H}(C), T, \mathsf{fp}_S, \text{rid}_H$, suite_id, policy_hash)
SIG_BIND $\leftarrow$ KMAC256(k_{sigctx}, SIG_INPUT, L=256, S=`"ZELEN v1 signature binding"`)
if not ML-DSA.Verify(vk_S, SIG_BIND, Σ, ctx=`"ZELEN-OBJECT-v1"`) **then return** $\perp$
end if
$M \leftarrow$ AES−256−GCM.Dec(k_{enc}, n, C, AAD=H, T)
if $M = \perp$ **then return** $\perp$
end if
return M

12.3 Constant-Error Policy

Constant-error policy. ZelEn implementations *must* return a single uniform error code on any decryption failure. They *must not* distinguish, in observable behavior or error message, between KEM decapsulation failure, header authentication failure, signature verification failure, AEAD authentication failure, certificate validity failure, and policy failure. Distinguishing between failure modes leaks information to an attacker probing a decryption oracle and converts a chosen-ciphertext attack from a security-game distinction into a practical exploitation channel.

12.4 No-Plaintext-Before-Authentication

Invariant 12.1. No plaintext byte may be released to any caller, log, debug channel, or error message before all of the following have succeeded: (i) header sanity checks; (ii) KEM digest match; (iii) header tag match; (iv) certificate verification; (v) object signature verification; (vi) AEAD authentication.

13 Signature and Certificate Model

ZelEn distinguishes two signature roles: the *certificate signature*, which binds an identity to a public bundle, and the *object signature*, which binds a specific `.zelen` instance to its sender.

13.1 Certificate Signature

The certificate signature is computed by the issuer over the canonical to-be-signed body:

$$\sigma_I \leftarrow \mathsf{ML\text{-}DSA.Sign}\big(sk_I^{\mathrm{sig}},\, \mathsf{H}(\mathrm{TBS}_R),\, \mathrm{ctx}=\texttt{"ZELEN-CERT-v1"}\big). \tag{22}$$

The TBS includes subject identity, issuer identity, the subject's `.zpub`, validity window, key usage rules, suite identifier, and policy. The certificate is verifiable by anyone in possession of the issuer's verification key vk_I.

13.2 Optional SLH-DSA Counter-Signature

For long-horizon archival profiles or for diversity against any future cryptanalytic concern about lattice signatures, ZelEn permits a second signature using SLH-DSA. SLH-DSA is hash-based and rests on different cryptographic assumptions than ML-DSA. A counter-signed certificate or object provides assurance that survives the hypothetical compromise of either signature family.

The counter-signed certificate is

$$Z_{\mathrm{cert},R}^{\mathrm{div}} = (\mathrm{TBS}_R,\, \sigma_I^{\mathrm{ml}},\, \sigma_I^{\mathrm{slh}}), \tag{23}$$

with both signatures verified during certificate validation. A failure in either signature causes the certificate to be rejected.

13.3 Object Signature

The object signature is computed by the sender over the canonical signature transcript:

$$\Sigma \leftarrow \mathsf{ML\text{-}DSA.Sign}(sk_S^{\mathrm{sig}},\, \mathrm{SIG\_INPUT}). \tag{24}$$

The object signature does not authenticate the plaintext directly; it authenticates the ciphertext-plus-AAD context, which is itself bound to the plaintext through AES−256−GCM. This composition prevents an attacker who learns the recipient's decapsulation key from later forging objects in the sender's name: the attacker may learn the plaintext but cannot produce a new Σ.

13.4 Certificate Authority Models

ZelEn supports several `.zcert` trust profiles:

- **Self-signed** development profiles for prototype use.
- **Enterprise-issued** profiles where a Zelfire enterprise root signs subject certificates.
- **Hierarchical** profiles with intermediate certificate authorities.
- **Federated** profiles where multiple roots are trusted under named scopes.
- **Transparency-anchored** profiles where issued certificates are logged to an append-only structure (for example, a Rosecoin-anchored ledger or a Certificate-Transparency-style log) for audit.

13.5 Revocation

ZelEn supports both list-based revocation (CRL-style) and freshness-based status responses (OCSP-style). Production deployments *should* prefer short-lived certificates with frequent renewal over long-lived certificates with revocation, because revocation distribution remains an engineering hazard with well-documented failure modes (stale CRL caches, OCSP soft-fail policies, must-staple deployment friction).

14 Key Generation and Lifecycle

14.1 Lifecycle Phases

ZelEn defines a complete key lifecycle as part of governance:

L1. Generation. Keys are generated using a validated CSPRNG. ML-KEM and ML-DSA key generation must be performed by an implementation that is constant-time with respect to secret material and that conforms to FIPS 203 and FIPS 204 respectively.

L2. Activation. A key is not used for production encryption or signing until its certificate is issued and entered into the active trust store.

L3. Expiration. Each `.zcert` specifies a validity window. Each `.zkey` carries an aligned expiry. ZelEn tools must refuse to use expired keys for new encryption or signing.

L4. Rotation. Keys are rotated on a defined schedule and additionally on cryptographic events (parameter weakness, suite deprecation).

L5. Revocation. A compromised or suspect key is revoked. Revocation is published to all trust consumers.

L6. Renewal. Certificates are renewed before expiry under the same or a successor key bundle.

L7. Emergency recovery. ZelEn supports policy-controlled emergency recovery keys.

L8. Escrow. Escrow is a policy decision, not a cryptographic decision. ZelEn provides the policy fields.

L9. Hardware-backed storage. Production `.zkey` material should reside in HSMs, TEEs, secure enclaves, or sealed key managers.

L10. Zeroization. On rotation, revocation, or destruction, all copies of key material in software memory and storage must be cryptographically erased.

L11. Compromise response. A key compromise triggers revocation, re-issuance, re-encryption of affected objects, audit log analysis, and notification under applicable regulation.

L12. Re-encryption. When a long-term recipient decapsulation key rotates, stored `.zelen` objects encrypted to the predecessor key may be re-encrypted to the successor key under controlled conditions.

L13. Audit logging. All lifecycle events are logged with cryptographic integrity protection.

L14. Policy-controlled key export. Default policy is no-export.

14.2 Forward Secrecy: An Honest Statement

Forward secrecy note. ZelEn file encryption to a static recipient public key does not automatically provide forward secrecy for stored files. If the recipient's long-term decapsulation key dk_R is compromised, older files encrypted to dk_R may require re-encryption or recovery-policy mitigation. ZelEn supports rotation, re-encryption, and recovery policies, but at-rest encryption to a static key is not equivalent to a ratcheting messaging protocol such as Signal's Double Ratchet. Organizations that require forward secrecy for archival encryption can layer ephemeral-key envelopes over ZelEn or use ephemeral recipient keys with a recipient-key management service.

15 Multi-Recipient Encryption

15.1 Construction

ZelEn supports objects encrypted to N recipients without N-fold payload encryption. The construction encrypts the payload once under a uniformly random payload key, then wraps that payload key separately for each recipient using ML-KEM.

$$K_{\text{payload}} \xleftarrow{\$} \{0,1\}^{256}, \tag{25}$$

$$(C, T) \leftarrow \mathsf{AES{-}256{-}GCM.Enc}(K_{\text{payload}}, n, M, \text{AAD}{=}H), \tag{26}$$

$$\forall i \in [1, N]: \quad (K_i, c_{\text{kem},i}) \leftarrow \mathsf{ML\text{-}KEM.Encaps}(ek_{R_i}), \tag{27}$$

$$\text{wrap}_i \leftarrow \mathsf{Wrap}(K_i, K_{\text{payload}}, \text{ctx}_i), \tag{28}$$

where ctx_i binds K_{payload} to a per-recipient context (recipient fingerprint, role, position in list, suite identifier). The recipient list is canonical and authenticated: it appears in the AAD of AES−256−GCM and in the signature transcript.

15.2 Wrapping Patterns

Two implementation patterns are permitted:

Pattern A (KDF-wrap). A per-recipient context string $\text{ctx}_i = \mathsf{CE}(\mathsf{fp}_{R_i}, i, \mathsf{suite\_id}, \texttt{"ZELEN v1 wrap"})$ is used to derive a per-recipient wrapping key and nonce from K_i, and the resulting wrapping key encrypts K_{payload} under that nonce.

$$k_{\text{wrap},i} = \mathsf{KMAC256}\big(K_i,\ \text{ctx}_i,\ L{=}256,\ S{=}\texttt{"ZELEN v1 recipient wrap key"}\big), \tag{29}$$

$$n_{\text{wrap},i} = \mathsf{KMAC256}\big(K_i,\ \text{ctx}_i,\ L{=}96,\ S{=}\texttt{"ZELEN v1 recipient wrap nonce"}\big), \tag{30}$$

$$\text{wrap}_i = \mathsf{AES{-}256{-}GCM.Enc}\big(k_{\text{wrap},i},\ n_{\text{wrap},i},\ K_{\text{payload}},\ \text{AAD}{=}\text{ctx}_i\big). \tag{31}$$

Pattern B (AEAD-wrap). K_i is used directly as a key-wrap KEK with an AES−256−GCM construction over $(K_{\text{payload}}, \text{ctx}_i)$.

15.3 Multi-Recipient Header Authentication

The single-recipient profile derives k_{hdr} from the (sole) KEM shared secret, so the same header tag is reproducible by encryptor and recipient. In the multi-recipient profile each recipient R_i has a distinct KEM shared secret K_i, so a header tag derived per-recipient is not a workable global header field. ZelEn resolves this as follows.

The recipient identifier field at offset `0x31`. In single-recipient mode, the field at offset `0x31` carries $\mathsf{fp}_R = \mathsf{SHA{-}256}(\mathsf{CE}(Z_{\text{pub},R} \setminus \{\text{fingerprint}\}))$ as before. In multi-recipient mode, the field at offset `0x31` carries the canonical 32-byte recipient_set_hash:

$$\text{recipient\_set\_hash} \ = \ \mathsf{SHA{-}256}\big(\mathsf{CE}(\text{recipient\_list})\big), \tag{32}$$

where recipient_list is the canonical, sorted (lexicographic by recipient fingerprint) list of recipient entries, each entry comprising the recipient's fingerprint, the recipient's per-recipient KEM ciphertext digest, the recipient's wrapped payload-key block, and the recipient's per-recipient key-confirmation tag. The Multi-Recipient flag at offset `0x06` bit 1 disambiguates the two interpretations; conformant parsers *must* interpret offset `0x31` according to the flag bit and *must not* attempt single-recipient fingerprint comparison when the multi-recipient flag is set.

Authentication of the recipient identifier. The exact 32-byte value at offset `0x31` (whether fp_R or recipient_set_hash) is bound into both the AES−256−GCM AAD (via the canonical header) and the SIG_INPUT transcript (via $\mathsf{H}(H)$). A modification to the recipient set therefore alters recipient_set_hash, which alters $\mathsf{H}(H)$, which invalidates both the GCM header authentication and the ML-DSA object signature.

Per-recipient and global authentication keying.

- In multi-recipient mode, the global header authentication tag at offset `0x81` is keyed by a key $k_{\text{hdr}}^{\text{mr}}$ derived from the payload key, not from any per-recipient KEM secret:

$$k_{\text{hdr}}^{\text{mr}} = \mathsf{KMAC256}\big(K_{\text{payload}},\ \varepsilon,\ L{=}256,\ S{=}\texttt{"ZELEN v1 multi-recipient header key"}\big). \quad (33)$$

- Each recipient entry in recipient_list contains a per-recipient key-confirmation tag derived from K_i via KMAC256 with a per-recipient customization string. This gives each recipient an independent first-line authenticity check on her own entry without requiring a per-recipient global header field.
- The global header authentication tag therefore confirms the payload-key-bound integrity of the canonical header, and the per-recipient confirmation tag confirms the binding of K_i to K_{payload} and to the recipient's entry.

The single-recipient profile is a degenerate case in which K_{payload} may be set to the per-recipient KMAC256-derived k_{enc} key, recovering the earlier construction.

15.4 Recipient List Discipline

- A recipient list with mixed suite policies is rejected; all recipients in a single `.zelen` must share an acceptable suite.
- A recipient revoked at the time of encryption is excluded by the encryptor; an encryptor that does not check recipient revocation is non-conformant.
- The recipient list ordering is canonical: lexicographic by recipient fingerprint, ensuring deterministic encoding.

16 Chunked Large-File Encryption

16.1 Motivation

For payloads exceeding a configured threshold (default 1 GiB), ZelEn defines a chunked AEAD profile. Chunking enables: streaming encryption and decryption with bounded memory; per-chunk authentication that surfaces tampering early; resumable transfers; and compatibility with object stores that have per-object size limits.

16.2 Per-Chunk Nonce Derivation

GCM requires uniqueness of the 96-bit IV under each encryption key. ZelEn defines a single, injective, deterministic 96-bit nonce-derivation rule for chunked mode that does *not* rely on truncated-PRF collision

resistance for nonce uniqueness. Per-chunk nonces are constructed as a 32-bit per-object random nonce prefix concatenated with the 64-bit chunk index:

$$\text{nonce\_prefix} = \mathsf{KMAC256}\big(k_{\text{nonce}},\ \mathsf{CE}(H,\ \text{chunk\_manifest\_id}),\ L{=}32,\ S{=}\texttt{"ZELEN v1 chunk nonce prefix"}\big), \tag{34}$$

$$n_i = \text{nonce\_prefix} \,\|\, \text{enc}_{64}(i), \tag{35}$$

where $\text{enc}_{64}(i)$ is the big-endian 64-bit encoding of the chunk index $i \in \{0, 1, \ldots, N-1\}$ and k_{nonce} is derived per eq. (18).

Remark 16.1 (Why this construction). Concatenation is an injective map on the chunk-index domain: distinct $i \neq i'$ yield distinct $\text{enc}_{64}(i) \neq \text{enc}_{64}(i')$, hence distinct $n_i \neq n_{i'}$ *deterministically*, with no probabilistic argument required. This preserves GCM's nonce-uniqueness requirement under the same k_{enc} for up to 2^{64} chunks per object. The 32-bit nonce_prefix is bound to the object header H and to a per-object chunk_manifest_id, ensuring distinct objects sharing k_{enc} (which would be a key-management failure under the ZelEn key schedule, but which the construction tolerates) receive disjoint nonce prefixes with overwhelming probability. Earlier draft profiles that suggested either $n_i = n_{\text{base}} \oplus \text{enc}_{64}(i)$ or a truncated 96-bit KMAC256 output as the per-chunk nonce are deprecated and *must not* be used in conformant ZelEn implementations: the former is under-specified for 96-bit/64-bit width mismatch, and the latter relies on truncated-PRF collision resistance for what should be a deterministic uniqueness property.

16.3 Per-Chunk AEAD

Each chunk is encrypted as

$$(C_i, T_i) \leftarrow \mathsf{AES{-}256{-}GCM.Enc}(k_{\text{enc}}, n_i, M_i, \text{AAD}{=}H \,\|\, \mathsf{IndexBlock}_i), \tag{36}$$

where $\mathsf{IndexBlock}_i$ authenticates the chunk index i, the chunk size $|M_i|$, and a final-chunk flag. The complete chunk manifest is authenticated under the object signature transcript.

16.4 Chunked-Mode Rules

- Strictly 96-bit nonces.
- No nonce reuse under the same key.
- Chunk index, chunk size, and final-chunk marker must be authenticated.
- Reordered, missing, duplicated, or truncated chunks must be rejected during streaming verification.

16.5 Chunked-Mode Plaintext Release Discipline

The global no-plaintext-before-authentication invariant (invariant 17.1) requires that no plaintext be released until *all* of: header sanity, KEM digest match, header tag match, certificate verification, object-signature verification, and AEAD authentication, have succeeded. In chunked mode this creates a tension with streaming consumers, because the object signature covers the whole chunk manifest and is not verified until end of stream. ZelEn resolves this with a two-mode discipline that the implementation *must* make explicit at API entry.

Mode S (Strict). The implementation reads, verifies, and decrypts all chunks into a sealed buffer. No plaintext byte is released to any caller until the object signature has verified at end of stream. This mode is the default and the only mode permitted under invariant 17.1.

Mode P (Provisional, opt-in). The caller *explicitly* requests provisional streaming. Per-chunk plaintext is released to the caller after that chunk's AEAD authentication succeeds, and *before* the object signature is verified. The caller *must* treat provisional plaintext as untrusted-by-sender until the implementation issues an end-of-stream signature-verification confirmation, and *must* discard or revoke any provisional output if the final signature fails. Mode P is suitable for media-streaming and indexing

pipelines that can tolerate provisional consumption with a final commit step. Mode P is *not* suitable for any workflow in which sender authenticity must hold before any byte is acted upon.

The default Mode S preserves invariant 17.1. Mode P narrows the invariant to a documented exception at the boundary between AEAD authenticity and signature authenticity, made explicit at the API surface so that the trade-off is auditable.

17 Parser Safety and ZelC Enforcement

17.1 Parser Requirements

A cryptographic file format is only as safe as its parser. ZelEn imposes the following parser discipline.

- **Single-pass.** The parser must not require multi-pass speculative decoding.
- **Length-delimited.** All variable-length blocks carry explicit length prefixes; the parser does not search for sentinels.
- **Canonical.** Reject non-canonical encodings; for any field that has a unique canonical form, accept only that form.
- **Allocation-capped.** Per-block, per-object, and per-session memory caps are enforced.
- **Recursion-bounded.** No nested structure is permitted to exceed a small fixed depth.
- **Fail-closed.** On any irregularity, the parser rejects with a single uniform error code.
- **Fuzz-tested.** The parser is included in the conformance fuzzing corpus.
- **Constant-error.** Distinguishing between failure types is not exposed to callers or attackers.
- **Memory-safe.** Reference implementations are written in memory-safe languages (Rust strongly preferred; safe subsets of other languages permitted).

17.2 Resistance Properties

The parser must be resistant to: malformed headers, oversized fields, duplicate fields, ambiguous encodings, downgrade attempts, integer overflow in length fields, recursion bombs in nested structures, and trailing-byte attacks.

17.3 ZelC Compiler Enforcement

The ZelC programming language is a Zelfire-native language with Kinetic Semantics and compile-time policy enforcement. ZelEn integrates with ZelC at three points.

Type-level enforcement. `.zkey`, `.zpub`, `.zcert`, and `.zelen` are first-class types in ZelC. A program cannot, for example:

- pass a `.zkey` where a `.zpub` is expected;
- encrypt with a verification key;
- sign with a decapsulation key;
- use an expired certificate;
- omit certificate verification before decryption.

Policy enforcement. Suite policies, key-usage policies, and certificate policies declared in source are checked at compile time, and again at runtime via the ZelEn SDK.

Audit emission. ZelC emits structured audit events for each ZelEn operation. Events include suite ID, fingerprints (*not* key material), object size, success or constant-error result, and the policy decisions evaluated.

17.4 Critical Statement on Executable Metadata

MTE is not an executable parser language. It is a bounded finite-site schema. A `.zelen` file must *never* contain executable parsing logic. Any data-driven decoding instructions present in MTE metadata are interpreted as schema descriptors, not as code, and the interpreter is bounded and total. This is a deliberate and load-bearing design choice: a file format whose parser is partially driven by file content is, in effect, a programming language, and programming-language parsers in this position have a long history of producing arbitrary-code-execution vulnerabilities.

17.5 Security Invariants

The following invariants govern any conformant ZelEn implementation.

Invariant 17.1. No plaintext is released unless AEAD authentication succeeds.

Invariant 17.2. No mutable unauthenticated header field may influence decryption policy.

Invariant 17.3. No AES−256−GCM nonce may be reused under the same encryption key.

Invariant 17.4. No downgrade from a stronger suite to a weaker suite is accepted.

Invariant 17.5. No private key material is exportable unless policy explicitly permits it.

Invariant 17.6. No parser error reveals which check failed (KEM, header tag, signature, AEAD, certificate, policy).

Invariant 17.7. No MTE metadata is trusted until it is authenticated by the transcript.

Invariant 17.8. No `.zelen` object is accepted unless `"ZELEN"` at offset `0x00` and `"ZELC"` at offset `0x7D` are present.

Invariant 17.9. No certificate is trusted unless the chain, validity, key usage, policy, and issuer signature verify.

Invariant 17.10. No implementation is conformant merely because it can decrypt valid files; it must also reject invalid, malformed, downgraded, unauthenticated, non-canonical, and policy-violating files.

18 Optional ZelEn-MTE Extension

18.1 Position

ZelEn-MTE is an optional finite-site / topos-inspired structural encoding layer over the ZelEn Core. It is described here for completeness and to position it precisely.

ZelEn-MTE is not an independent post-quantum hardness assumption. It does not contribute to the formal confidentiality bound in theorem 23.9. It contributes proprietary Zelfire object semantics: structured metadata encoding, contextual binding to organizational scope graphs, object obfuscation against casual structural analysis, and policy-graph binding. ZelEn confidentiality must hold even if the MTE design is fully public.

18.2 Public Structure

Definition 18.1 (MTE Site)**.** An MTE site is a pair $\mathcal{S} = (\mathcal{C}, \mathcal{J})$ where $\mathcal{C}$ is a finite category (with a finite set of objects and a finite set of morphisms between any two objects) and $\mathcal{J}$ is a Grothendieck topology on $\mathcal{C}$ specifying valid covers.

A public sheaf-like object

$$\mathcal{F}\colon \mathcal{C}^{\mathrm{op}} \longrightarrow \mathbf{FinSet} \quad \text{or} \quad \mathcal{F}\colon \mathcal{C}^{\mathrm{op}} \longrightarrow \mathbf{FinVect} \tag{37}$$

describes how local sections encode authenticated metadata fields, including policy fragments, identity fingerprints, KEM transcript hashes, and Zelfire object context. The structure is finite and bounded; covers are pre-declared; gluing rules are total.

18.3 Private Datum

The private datum is not described as a generic geometric morphism to **Set**, because such a description is too loose to support a clear security claim. Instead:

$$\tau = (\text{cover\_selector}, \text{decoding\_seed}, \text{local\_section\_map}, \text{reconstruction\_policy}). \tag{38}$$

Only the holder of τ, together with ML-KEM-derived secret material from the ZelEn Core flow, can reconstruct the canonical global section used in the ZelEn object transcript.

18.4 Composition with ZelEn Core

$$\text{ZelEn-MTE} = \text{ZelEn Core} + \text{finite-site encoding} + \text{authenticated Zelfire object semantics}. \tag{39}$$

Even if the entire MTE design is published, ZelEn confidentiality must hold because confidentiality reduces to ML-KEM, the KMAC256 KDF, and AES−256−GCM, all of which retain their security under public design disclosure. MTE provides operational and structural value; it does not provide cryptographic hardness.

18.5 Why Include MTE at All

MTE encodes Zelfire-native object semantics—policy graphs, scope hierarchies, contextual descriptors—in a form that is bounded, total, and authenticable. It is useful for object-level governance and for compact representation of complex policy. It is documented as an extension to avoid any suggestion that ZelEn confidentiality depends on it.

19 Algorithm Agility and Reserved Future Suites

ZelEn registers suite identifiers in a central registry. Current and reserved entries appear in table 3.

Table 3: ZelEn Suite Registry

Suite ID	Name	KEM	Signature	AEAD
0x0101	ZELEN-PQ5	ML-KEM-1024	ML-DSA-87	AES-256-GCM
0x0102	ZELEN-PQ5-SLH	ML-KEM-1024	ML-DSA-87 + SLH-DSA	AES-256-GCM
0x0201	ZELEN-HYBRID-PQ5	ML-KEM-1024 + X25519/P-384	ML-DSA-87	AES-256-GCM
0x0301	ZELEN-HQC-RESERVED	HQC (TBD)	TBD	AES-256-GCM

19.1 Suite Agility Rules

- The suite identifier is authenticated as part of the canonical header and signed in the transcript. Downgrade attacks are detected because the suite identifier in the encryptor's record cannot be silently altered.
- A ZelEn implementation that recognizes only ZELEN-PQ5 must reject objects of other suites cleanly with the uniform error code.
- A ZelEn implementation that recognizes multiple suites must apply local policy to determine which are accepted.

19.2 Hybrid Composition (ZELEN-HYBRID-PQ5)

In ZELEN-HYBRID-PQ5, the hybrid shared secret is derived using a KMAC256-based combiner whose input is a length-prefixed canonical encoding of *both* component shared secrets, *both* component ciphertexts, *both* component encapsulation keys, the suite identifier, and per-component parameter-set identifiers. This conservative input set follows NIST SP 800-227 composite-KEM guidance, which treats the combiner as taking shared secrets together with ciphertexts, encapsulation keys, and parameter-set context, and stresses domain separation for the composite scheme.

$$K_{\text{hybrid}} \;=\; \mathsf{KMAC256}\big(\varepsilon,\; X_{\text{hybrid}},\; L{=}256,\; S{=}\texttt{"ZELEN-HYBRID v1 combiner"}\big), \tag{40}$$

where the combiner input X_{hybrid} is the canonical, length-prefixed concatenation

$$X_{\text{hybrid}} \;=\; \mathsf{CE}\big(\mathsf{suite\_id},\, \text{paramset\_pq},\, \text{paramset\_cl},\, K_{\text{pq}},\, K_{\text{cl}},\, c_{\text{pq}},\, c_{\text{cl}},\, ek_{R,\text{pq}},\, ek_{R,\text{cl}}\big). \tag{41}$$

Here:

- K_{pq} is the post-quantum (ML-KEM) shared secret and K_{cl} is the classical KEM shared secret;
- c_{pq} and c_{cl} are the corresponding KEM ciphertexts produced by the encryptor;
- $ek_{R,\text{pq}}$ and $ek_{R,\text{cl}}$ are the recipient's encapsulation keys for each component;
- paramset_pq and paramset_cl are unique algorithm-and-parameter-set identifiers (e.g., `"ML-KEM-1024"` and `"X25519"` or `"P-384"`), recorded canonically;
- CE is the deterministic, injective canonical encoding from assumption 23.6, which provides length-prefixed framing of every component so that distinct component widths cannot encode to the same byte string;
- the customization string S provides domain separation for the composite scheme.

The ε key-position argument is the empty key; combiner secrecy is derived entirely from the shared-secret components inside X_{hybrid}. Including ciphertexts and encapsulation keys binds the hybrid key to the exact KEM transcripts of the encryptor's run, preventing certain mix-and-match composite-KEM attacks documented in the literature on hybrid combiners.

Proposition 19.1 (Hybrid Combiner Robustness)**.** *Modeling* KMAC256 *as a random oracle with the customization string S providing domain separation, if at least one of* ML-KEM *or the classical KEM is IND-CCA-secure (and the other is at minimum correct), then the hybrid construction ZELEN-HYBRID-PQ5 produces a key K_{hybrid} that is computationally indistinguishable from uniform under chosen-ciphertext attack against the hybrid construction. The advantage is bounded by*

$$\mathsf{Adv}^{\text{IND-CCA}}_{\text{hybrid}}(\mathcal{A}) \;\le\; \min\big(\mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{ML\text{-}KEM}}(\mathcal{B}_1),\; \mathsf{Adv}^{\text{IND-CCA}}_{\text{classical}}(\mathcal{B}_2)\big) + q_{\text{RO}}/2^{256},$$

where q_{RO} is the number of random-oracle queries. The construction is therefore robust in the migration sense: confidentiality holds as long as either underlying KEM holds, and the inclusion of $c_{\text{pq}}, c_{\text{cl}}, ek_{R,\text{pq}}, ek_{R,\text{cl}}$ inside X_{hybrid} binds the combiner output to the encryptor's KEM transcripts.

Remark 19.2. The $\min(\cdot,\cdot)$ form (rather than a sum) reflects that an adversary need only break one of the two component KEMs to attack the hybrid. The standard-model variant of proposition 19.1 is provable under the assumption that KMAC256 is a dual-PRF in both its key and message arguments; this is a conservative strengthening of the standard PRF assumption and is consistent with current SHA-3 sponge-PRF analyses.

20 Performance Analysis

20.1 Key and Signature Sizes

20.2 Per-Object Overhead

For a ZELEN-PQ5 object:

$$\text{overhead} \;\approx\; \underbrace{145}_{\text{header}} + \underbrace{1568}_{c_{\text{kem}}} + \underbrace{|Z_{\text{cert},S}|}_{\sim 8-16\,\text{KiB}} + \underbrace{16}_{\text{tag}} + \underbrace{4627}_{\Sigma} + \underbrace{16}_{\text{length prefixes}}. \tag{42}$$

Table 4: ZELEN-PQ5 Key and Signature Sizes (approximate)

Object	**Component**	**Size (bytes)**
ML-KEM-1024	Encapsulation key ek	1,568
ML-KEM-1024	Decapsulation key dk	3,168
ML-KEM-1024	Ciphertext c_{kem}	1,568
ML-KEM-1024	Shared secret K	32
ML-DSA-87	Verification key vk	2,592
ML-DSA-87	Signing key sk^{sig}	4,896
ML-DSA-87	Signature	4,627
AES−256−GCM	Authentication tag	16
AES−256−GCM	Nonce	12
ZelEn fixed header	v1 layout	145

For payloads of moderate size (one megabyte and above), the constant overhead is on the order of 10–20 KiB, dominated by certificate and signature. For very small payloads the overhead is significant and is acceptable: ZelEn is an object format for governed enterprise data, not a stream cipher for bytewise messaging.

20.3 Throughput Model

Let:

$$\begin{aligned} T_{\mathsf{Encaps}}, T_{\mathsf{Decaps}} &= \text{ML-KEM encapsulation/decapsulation latency},\\ T_{\mathsf{Sign}}, T_{\mathsf{Verify}} &= \text{ML-DSA sign/verify latency},\\ B_{\mathsf{AES-256-GCM}} &= \text{symmetric throughput (bytes/sec)},\\ |M| &= \text{plaintext size (bytes)}. \end{aligned}$$

Per-object encryption latency is approximately

$$T_{\text{enc}}(|M|) \;\approx\; T_{\mathsf{Encaps}} + T_{\mathsf{Sign}} + |M|/B_{\mathsf{AES-256-GCM}} + O(\log |M|), \tag{43}$$

and per-object decryption latency is approximately

$$T_{\text{dec}}(|M|) \;\approx\; T_{\mathsf{Decaps}} + 2T_{\mathsf{Verify}} + |M|/B_{\mathsf{AES-256-GCM}}, \tag{44}$$

where $2T_{\mathsf{Verify}}$ accounts for certificate verification and object signature verification.

On modern server hardware with AES-NI / VAES, $B_{\mathsf{AES-256-GCM}}$ exceeds several gigabytes per second per core. ML-KEM-1024 encapsulation and decapsulation are typically well under one millisecond on commodity hardware. ML-DSA-87 signing is on the order of a few milliseconds; verification is similar. For typical sizes, per-object cryptographic latency is dominated by signature operations, not by symmetric throughput.

20.4 Recommendations

- For large bulk transfers, use chunked mode to enable streaming verification and bounded memory consumption.
- For high-volume archival workloads, batch certificate verification and reuse certificate trust caches across objects.
- For latency-critical workloads, consider keeping warm ML-KEM and ML-DSA contexts to avoid initialization overhead.

21 Compliance and Validation Roadmap

21.1 Standards Used

- FIPS 197 (AES).
- FIPS 203 (ML-KEM).
- FIPS 204 (ML-DSA).
- FIPS 205 (SLH-DSA, optional).
- NIST SP 800-38D (AES−256−GCM).
- NIST SP 800-108 Rev. 1 (key derivation using pseudorandom functions, including KMAC-based KDF profiles).
- NIST SP 800-185 (cSHAKE256, KMAC256, TupleHash, ParallelHash).
- NIST SP 800-227, September 2025 (recommendations for KEM use).
- NIST SP 800-90A/B/C (random bit generation; SP 800-90C finalized September 2025).

21.2 Validation Distinction

Using FIPS-standardized algorithms is *not* the same as having a FIPS-validated cryptographic module. Production deployments targeting regulated environments (FedRAMP, FISMA, defense, regulated finance, healthcare) should use or pursue FIPS 140-3 validated modules where required. ZelEn permits and encourages such modules; ZelEn does not, by virtue of using FIPS-standardized algorithms, automatically inherit FIPS module validation.

21.3 Roadmap

- **Stage 1: Reference implementation conformance.** The reference ZelEn implementation passes a documented test-vector battery for all suites and conformance-tests the parser against a fuzz corpus.
- **Stage 2: Independent cryptographic review.** ZelEn submits its specification and a forthcoming conformant reference implementation (distinct from the illustrative prototype in appendix C) to independent cryptographic review, with public response to findings.
- **Stage 3: FIPS 140-3 module validation.** Vendors deploying ZelEn into regulated environments validate their cryptographic modules against FIPS 140-3.
- **Stage 4: Common Criteria evaluation.** For high-assurance environments, ZelEn-bearing products may pursue Common Criteria evaluation against an applicable Protection Profile.
- **Stage 5: Continuous monitoring.** ZelEn participates in ongoing monitoring of standards updates, parameter recommendations, and cryptanalytic results, with suite-registry updates and migration guidance issued accordingly.

22 Testing, Test Vectors, and Conformance

22.1 Test Vector Corpus

For each registered suite, the corpus includes:

- deterministic test vectors for KEM with fixed seeds;
- deterministic test vectors for signatures with fixed seeds where the algorithm permits;
- full encryption/decryption round-trip vectors across plaintext sizes (empty, 1 byte, 16 bytes, 1 KiB, 1 MiB, 1 GiB);

- multi-recipient vectors with $N \in \{1, 2, 8, 64\}$;
- chunked vectors with various chunk-size configurations;
- explicit malformed-object rejection vectors.

22.2 Negative Testing

Negative tests include:

- missing magic at `0x00`;
- missing `ZELC` marker at `0x7D`;
- altered header bytes (each byte position swept);
- oversized length prefixes;
- truncated KEM block;
- truncated payload;
- swapped certificate;
- wrong recipient fingerprint;
- wrong sender fingerprint;
- downgraded suite identifier;
- replayed nonce;
- altered signature.

Each negative test must result in a single uniform error code.

22.3 Fuzzing

Coverage-guided fuzzing of the parser against random and structurally aware corpora is part of conformance. The fuzzing must cover at least: header parsing, length-prefix parsing, certificate parsing, recipient-list parsing, and chunk-manifest parsing.

22.4 Interoperability Requirement

Two independent implementations passing the conformance suite are required to release any new suite identifier as Default.

23 Formal Security Model

This section develops the formal security model for ZelEn Core. We give game-based definitions, state the security goal, and prove an advantage bound by reduction to the underlying primitives.

23.1 Cryptographic Building Blocks

We assume the existence of the following building blocks, each modeled as a tuple of probabilistic polynomial-time algorithms.

Definition 23.1 (KEM)**.** A key encapsulation mechanism $\mathsf{KEM} = (\mathsf{KEM.KeyGen}, \mathsf{KEM.Encaps}, \mathsf{KEM.Decaps})$ comprises:

- $(ek, dk) \leftarrow \mathsf{KEM.KeyGen}()$;
- $(K, c) \leftarrow \mathsf{KEM.Encaps}(ek)$ with $K \in \{0,1\}^{\kappa}$ and c a ciphertext;
- $K \leftarrow \mathsf{KEM.Decaps}(dk, c)$.

We require correctness: for honestly generated (ek, dk) and $(K, c) \leftarrow \mathsf{KEM.Encaps}(ek)$, we have $\Pr[\mathsf{KEM.Decaps}(dk, c) = K] = 1 - \mathsf{negl}(\kappa)$.

Definition 23.2 (IND-CCA security of KEM)**.** For an adversary $\mathcal{B}$ in the IND-CCA game $\mathsf{Game}_{\mathsf{KEM}}^{\text{IND-CCA}}$:

1. Challenger samples $(ek, dk) \leftarrow \mathsf{KEM.KeyGen}()$, $(K_0, c^*) \leftarrow \mathsf{KEM.Encaps}(ek)$, $K_1 \xleftarrow{\$} \{0,1\}^\kappa$, $b \xleftarrow{\$} \{0,1\}$.
2. $\mathcal{B}$ receives (ek, c^*, K_b).
3. $\mathcal{B}$ has oracle access to $\mathsf{KEM.Decaps}(dk, \cdot)$ on any input $c \neq c^*$.
4. $\mathcal{B}$ outputs b' and wins if $b' = b$.

The advantage is

$$\mathsf{Adv}_{\mathsf{KEM}}^{\text{IND-CCA}}(\mathcal{B}) = \left|\Pr[b' = b] - \tfrac{1}{2}\right|. \tag{45}$$

Definition 23.3 (AEAD security)**.** An AEAD scheme $\mathsf{AEAD} = (\mathsf{AEAD.Enc}, \mathsf{AEAD.Dec})$ over key space $\{0,1\}^k$, nonce space $\{0,1\}^\ell$, plaintext space $\{0,1\}^*$, and AAD space $\{0,1\}^*$ is secure if no PPT adversary $\mathcal{C}$ can distinguish, with non-negligible advantage, between (i) a real oracle that responds to $\mathsf{Enc}(k, n, m, a)$ queries with the true ciphertext-tag pair, and (ii) an ideal oracle that responds with random strings of the appropriate length, subject to nonce uniqueness for a given key. The corresponding INT-CTXT advantage is the probability that $\mathcal{C}$ produces a forgery (n, c, t, a) that the real Dec accepts but that was not the output of any prior Enc query.

Definition 23.4 (PRF security)**.** A function $F\colon \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^n$ is a (q, t, ε)-secure PRF if no adversary $\mathcal{D}$ running in time t and making q queries can distinguish $F(k, \cdot)$ for $k \xleftarrow{\$} \{0,1\}^k$ from a uniformly random function with advantage exceeding ε.

Definition 23.5 (EUF-CMA security of Sig)**.** A signature scheme $\mathsf{Sig} = (\mathsf{Sig.KeyGen}, \mathsf{Sig.Sign}, \mathsf{Sig.Verify})$ is existentially unforgeable under chosen-message attack if no PPT adversary $\mathcal{E}$, given the public verification key and oracle access to $\mathsf{Sig.Sign}$, can produce a valid signature on a message not previously queried, with non-negligible advantage.

23.2 Canonical Encoding Assumption

Assumption 23.6 (Canonical Encoding)**.** The canonical encoding CE is deterministic, injective on its domain, and total. Specifically, for any two distinct structured inputs $x \neq y$ in the domain of CE, $\mathsf{CE}(x) \neq \mathsf{CE}(y)$. The probability of an implementation accepting two distinct structured inputs as encoding to the same byte string is denoted $\varepsilon_{\mathsf{CE}}$ and is assumed to be cryptographically negligible.

23.3 Parser Failure Probability

Assumption 23.7 (Parser Failure)**.** The probability that a conformant parser accepts a non-canonical or malformed object as well-formed is denoted $\varepsilon_{\text{parse}}$. This is an *implementation-assurance assumption*: it is not automatically cryptographically negligible by virtue of the specification, and it is not bounded by any of the cryptographic primitives. Its value depends on the engineering-assurance level of a specific implementation, including coverage-guided fuzzing of the parser, formal verification of length-prefix handling, memory-safety properties of the implementation language, and the depth of negative test coverage in the conformance suite. This document treats $\varepsilon_{\text{parse}}$ as a concrete term that appears in the security bound (eq. (47)) and that an evaluating party must independently assess for any specific implementation. A well-engineered, fuzz-hardened, memory-safe implementation can drive $\varepsilon_{\text{parse}}$ to operationally negligible levels, but ZelEn does not claim a primitive-level reduction for parser correctness.

23.4 ZelEn Object Security Game

Definition 23.8 (ZelEn Object Security)**.** The ZelEn object security game $\mathsf{Game}_{\text{ZelEn}}^{\text{obj-sec}}$ between challenger and adversary $\mathcal{A}$ proceeds as follows:

1. Challenger generates honest sender S and recipient R identities, with public bundles $Z_{\text{pub},S}, Z_{\text{pub},R}$ and certificates $Z_{\text{cert},S}, Z_{\text{cert},R}$. The trust-anchor verification key vk_I is published. $\mathcal{A}$ receives $(vk_I, Z_{\text{cert},S}, Z_{\text{cert},R}, Z_{\text{pub},S}, Z_{\text{pub},R})$.
2. $\mathcal{A}$ has oracle access to:
   - $\mathsf{ZelEn.Encrypt}(\cdot, Z_{\text{pub},R}, Z_{\text{key},S}, Z_{\text{cert},S})$ on any plaintext;
   - $\mathsf{ZelEn.Decrypt}(\cdot, Z_{\text{key},R})$ on any `.zelen` object except a designated challenge object.
3. $\mathcal{A}$ submits two equal-length plaintexts M_0, M_1. Challenger samples $b \xleftarrow{\$} \{0,1\}$ and returns $Z^* \leftarrow \mathsf{ZelEn.Encrypt}(M_b, Z_{\text{pub},R}, Z_{\text{key},S}, Z_{\text{cert},S})$.
4. $\mathcal{A}$ continues to query $\mathsf{ZelEn.Decrypt}$ on any object $\neq Z^*$, and outputs b'.
5. $\mathcal{A}$ wins if $b' = b$.

The advantage is

$$\mathsf{Adv}^{\text{obj-sec}}_{\text{ZelEn}}(\mathcal{A}) = \left|\Pr[b' = b] - \tfrac{1}{2}\right|. \tag{46}$$

23.5 Main Theorem

Theorem 23.9 (ZelEn Core Object Security)**.** *Let $\mathcal{A}$ be a probabilistic polynomial-time adversary against the ZelEn object-security game. Then there exist PPT reductions $\mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}$ such that*

$$\boxed{\mathsf{Adv}^{\text{obj-sec}}_{\text{ZelEn}}(\mathcal{A}) \;\le\; \mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{ML\text{-}KEM}}(\mathcal{B}) + \mathsf{Adv}^{\text{AEAD}}_{\mathsf{AES-256-GCM}}(\mathcal{C}) + \mathsf{Adv}^{\text{PRF}}_{\mathsf{KMAC256}}(\mathcal{D}) + \mathsf{Adv}^{\text{EUF-CMA}}_{\mathsf{ML\text{-}DSA}}(\mathcal{E}) + \varepsilon_{\mathsf{CE}} + \varepsilon_{\text{parse}}.} \tag{47}$$

The reductions $\mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}$ run in time approximately equal to that of $\mathcal{A}$ plus a polynomial overhead, and make at most as many oracle queries as $\mathcal{A}$.

Proof Sketch. We proceed by a sequence of games, each modifying the previous by a hybrid step whose distinguishing probability is bounded by an advantage against one of the underlying primitives.

Game G_0: the original $\mathsf{Game}^{\text{obj-sec}}_{\text{ZelEn}}$. By definition, $\Pr[\mathcal{A} \text{ wins } G_0] = \frac{1}{2} + \mathsf{Adv}^{\text{obj-sec}}_{\text{ZelEn}}(\mathcal{A})$.

Game G_1 (Replace KEM secret with random). In the challenge encryption, replace the shared secret K output by $\mathsf{ML\text{-}KEM.Encaps}$ with a uniformly random $K' \xleftarrow{\$} \{0,1\}^{256}$. Any distinguishing advantage between G_0 and G_1 yields a reduction $\mathcal{B}$ against $\mathsf{ML\text{-}KEM}$ IND-CCA: $\mathcal{B}$ embeds its IND-CCA challenge (c^*, K_b) into the ZelEn challenge encryption, simulates ZelEn decryption oracles by relaying KEM ciphertexts (other than c^*) to its own KEM-decapsulation oracle, and returns $\mathcal{A}$'s guess. Hence

$$|\Pr[\mathcal{A} \text{ wins } G_0] - \Pr[\mathcal{A} \text{ wins } G_1]| \le \mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{ML\text{-}KEM}}(\mathcal{B}).$$

Game G_2 (Replace $\mathsf{KMAC256}$ outputs with random under fresh key). Since $K' \xleftarrow{\$} \{0,1\}^{256}$ is now uniformly random, by the PRF security of $\mathsf{KMAC256}$ we may replace $\text{PRK}, k_{\text{enc}}, k_{\text{hdr}}, k_{\text{sigctx}}$ with uniformly random values of the appropriate lengths. Any distinguishing advantage yields a reduction $\mathcal{D}$ against $\mathsf{KMAC256}$ PRF security:

$$|\Pr[\mathcal{A} \text{ wins } G_1] - \Pr[\mathcal{A} \text{ wins } G_2]| \le \mathsf{Adv}^{\text{PRF}}_{\mathsf{KMAC256}}(\mathcal{D}).$$

Game G_3 (Replace AEAD output with random). With k_{enc} now uniformly random and the nonce n fresh, the AEAD output (C, T) on the challenge plaintext M_b is indistinguishable from a uniformly random string of equal length under AEAD security. Any distinguishing advantage yields a reduction $\mathcal{C}$:

$$|\Pr[\mathcal{A} \text{ wins } G_2] - \Pr[\mathcal{A} \text{ wins } G_3]| \le \mathsf{Adv}^{\text{AEAD}}_{\mathsf{AES-256-GCM}}(\mathcal{C}).$$

In G_3, the challenge ciphertext is independent of b, so $\Pr[\mathcal{A} \text{ wins } G_3] = \frac{1}{2}$, modulo the analysis of the signature and parser components.

Signature unforgeability. For an adversary that attempts to win the game by submitting a forged `.zelen` object verifying under $Z_{\text{cert},S}$, the canonical encoding of SIG_INPUT is uniquely determined by

the components $(H, c_{\text{kem}}, C, T, \mathsf{fp}_S, \mathsf{fp}_R, \mathsf{suite\_id}, \text{policy\_hash})$ (by injectivity of CE, assumption 23.6). A successful forgery yields an EUF-CMA forgery against ML-DSA via reduction $\mathcal{E}$, contributing $\mathsf{Adv}^{\text{EUF-CMA}}_{\text{ML-DSA}}(\mathcal{E})$ to the bound.

Encoding and parser terms. The encoding ambiguity probability $\varepsilon_{\mathsf{CE}}$ accounts for any case where two distinct structured inputs encode to the same byte string under CE; injectivity of CE makes this term cryptographically negligible. The parser failure probability $\varepsilon_{\text{parse}}$ accounts for any case where a non-canonical or malformed object is incorrectly accepted as well-formed; per assumption 23.7, this is an implementation-assurance term, not a cryptographic-assumption term, and must be assessed per implementation.

Summing the four primitive advantages and the two structural terms yields eq. (47). The reductions are tight up to standard hybrid losses and are PPT. □

23.6 Authenticity Sub-Theorem

Theorem 23.10 (ZelEn Sender Authenticity)**.** *Let $\mathcal{A}$ be a PPT adversary attempting to produce a `.zelen` object Z that successfully decrypts under $Z_{\text{key},R}$ and verifies under $Z_{\text{cert},S}$, where $\mathcal{A}$ does not hold sk_S^{sig} and where Z was not output by any honest encryption oracle. Then*

$$\mathsf{Adv}^{\text{auth}}_{\text{ZelEn}}(\mathcal{A}) \;\le\; \mathsf{Adv}^{\text{EUF-CMA}}_{\text{ML-DSA}}(\mathcal{E}) + \mathsf{Adv}^{\text{AEAD}}_{\mathsf{AES-256-GCM}}(\mathcal{C}) + \varepsilon_{\mathsf{CE}}. \tag{48}$$

Proof Sketch. A successful forgery requires producing a valid Σ on a fresh SIG_INPUT, which yields an EUF-CMA forgery against ML-DSA via reduction $\mathcal{E}$, or producing a valid AEAD ciphertext-tag pair under a key derived from a K that the adversary did not produce (reducing to AEAD INT-CTXT and ultimately to AES−256−GCM AEAD security and the KMAC256 PRF). Injectivity of CE ensures distinct encryption queries yield distinct SIG_INPUT values. □

23.7 Limitations of the Proof

The proof assumes a faithful implementation. The following are explicitly not modeled:

- Side-channel attacks (timing, power, electromagnetic) on host hardware.
- RNG failure or compromise.
- Compromise of sk_R^{kem} or sk_S^{sig}.
- Parser deviations that violate assumption 23.7.
- The MTE extension is not part of this proof; ZelEn-MTE is documented separately and any structural-encoding security claims are scoped to that extension's own analysis.

23.8 Concrete Bounds Discussion

ZELEN-PQ5 targets NIST security-strength Category 5 for its public-key components and AES-256-class strength for its symmetric components. NIST's security-strength model in FIPS 203 and FIPS 204 expresses post-quantum hardness through relative computational-resource comparisons (e.g., "at least as hard as key search on AES-256"), *not* as a single advantage figure such as 2^{-256}. We therefore state the per-primitive contributions to the bound qualitatively, with the implementation-specific terms made explicit.

- $\mathsf{Adv}^{\text{IND-CCA}}_{\text{ML-KEM-1024}}(\mathcal{B})$: bounded by NIST Category 5 hardness assumptions on the underlying Module-LWE / Module-LWR problems for the FIPS 203 parameter set, against any quantum adversary respecting Category-5 work-factor bounds.
- $\mathsf{Adv}^{\text{EUF-CMA}}_{\text{ML-DSA-87}}(\mathcal{E})$: bounded by NIST Category 5 hardness assumptions on the underlying Module-SIS / Module-LWE problems for the FIPS 204 parameter set.
- $\mathsf{Adv}^{\text{PRF}}_{\text{KMAC256}}(\mathcal{D})$: bounded by the cryptanalytic resistance of the underlying Keccak-f[1600] permutation, with the standard sponge-PRF analysis applying.

- $\mathsf{Adv}^{\mathrm{AEAD}}_{\mathsf{AES-256-GCM}}(\mathcal{C})$: dominated, in practical settings, by the GCM data-volume and invocation-count limits of NIST SP 800-38D. With 128-bit authentication tags, the per-tag forgery resistance is bounded by approximately $q_{\mathrm{dec}}/2^{128}$ for q_{dec} verification queries; with 96-bit nonces sampled uniformly, the nonce-collision contribution per encryption key is approximately $q_{\mathrm{enc}}^2/2^{96}$ for q_{enc} encryptions. ZelEn's per-object KDF derivation makes $q_{\mathrm{enc}} \approx 1$ per k_{enc}, removing nonce-collision pressure operationally.
- $\varepsilon_{\mathsf{CE}}$: cryptographically negligible under assumption 23.6.
- $\varepsilon_{\mathrm{parse}}$: an implementation-assurance term per assumption 23.7; not a cryptographic-assumption term, and not automatically cryptographically negligible. Its operational value is reduced by coverage-guided fuzzing, formal verification of length handling, and memory-safe implementation, and must be assessed per implementation by an evaluator.

The dominant practical term in the bound is the GCM tag-forgery term scaled by verification queries, which is 128-bit-bounded and is therefore the asymmetric component to the otherwise Category-5 public-key strength. This asymmetry is intrinsic to AES-GCM with a 128-bit tag, not a defect of ZelEn; it is shared by every AES-GCM-based protocol. Operational guidance is therefore to limit q_{dec} per key per the SP 800-38D usage bounds; in ZelEn, this is automatic because k_{enc} is freshly derived per object.

24 Hostile Q&A

This section anticipates criticism from cryptographers, security engineers, and enterprise CISOs.

Q1. Are you inventing new cryptography? No. ZelEn employs standardized post-quantum primitives for its security foundation. The invention is the governed object architecture: file semantics, transcript binding, certificate model, lifecycle controls, ZelC enforcement, and the optional MTE structural layer. The formal security argument (theorem 23.9) reduces ZelEn confidentiality and authenticity to ML-KEM, AES−256−GCM, KMAC256, and ML-DSA.

Q2. Why not just use OpenSSL or a generic PQC library? Generic libraries expose primitives. ZelEn defines a governed object format, an identity model, an authenticated transcript, a certificate system, parser rules, lifecycle policy, and compiler/runtime enforcement. These are exactly the layers that determine whether a deployment is safe. A generic library is a necessary ingredient; it is not sufficient for an enterprise post-quantum deployment.

Q3. Is MTE security through obscurity? No. ZelEn confidentiality must survive disclosure of the MTE design. Secrets are key material and policy authority, not hidden algorithms. MTE is an extension, not a load-bearing layer. theorem 23.9 does not include any MTE-derived advantage term; the bound holds whether or not MTE is in use.

Q4. Does `ZELC` stop forgery? No. `ZELC` is a forensic and parser sanity marker. Forgery resistance comes from AES−256−GCM AEAD authentication on the header, the KMAC256-derived header tag, and the ML-DSA object signature over the canonical transcript.

Q5. Does ZelEn provide forward secrecy? Not automatically for static stored files encrypted to long-term recipient public keys. ZelEn supports rotation, re-encryption, and recovery-policy mitigation, but at-rest encryption to a static key is not the same as a ratcheting messaging protocol. Organizations that require forward secrecy for archival encryption can layer ephemeral-key envelopes over ZelEn or use ephemeral recipient keys with a recipient-key management service.

Q6. What if ML-KEM is weakened? ZelEn supports suite agility. Migration mode (ZELEN-HYBRID-PQ5) hedges by combining ML-KEM with a classical KEM. Reserved profiles such as ZELEN-HQC-RESERVED can be introduced if the cryptographic community concludes it is prudent. Optional SLH-DSA counter-signatures provide diversity for the signature side.

Q7. Is this proprietary format lock-in? ZelEn is governed portability, not hidden lock-in. The object transcript and validation rules are documented sufficiently to support audit, test vectors, conformance suites, and forensic validation. Decryption remains controlled by keys, certificates, and policy, exactly as it should be. Two independent implementations passing the conformance suite are required for any new suite ID to be released as Default.

Q8. Why mandate AES−256−GCM rather than offering AEAD agility? Choice surface is risk surface. ZELEN-PQ5 fixes AES−256−GCM with a strict 96-bit nonce because the cost of optionality at this layer exceeds its benefit, and because AES−256−GCM aligns with the AES-256-class symmetric strength implied by Category 5 / PQ5. ChaCha20-Poly1305 may be added in a future profile if a compelling motivation emerges (e.g., constrained environments without AES-NI).

Q9. Why a single-byte header-length field? The v1 fixed-header profile uses a single byte for compactness in the demonstration profile. Future profiles can extend with a varint or 32-bit field; the offset semantics of `ZELEN` at `0x00` and `ZELC` at `0x7D` are invariant. Production specifications targeting unbounded header growth should adopt the extended length form from the start.

Q10. How does ZelEn interact with TLS? Orthogonally. TLS protects transport. ZelEn protects objects. A `.zelen` file traveling over TLS is doubly protected; a `.zelen` file at rest is protected even when TLS does not apply. ZelEn does not depend on TLS, and TLS does not depend on ZelEn.

Q11. Why have certificates at all? Why not just public keys? Bare public keys make small-scale prototypes work but make large-scale governance hard. Certificates carry policy, validity, key usage, issuer attestation, and a verifiable chain. ZelEn's certificate model is what enables enterprise-scale identity, audit, and revocation.

Q12. Why both an KMAC256-derived header tag and AES-GCM AAD authentication of the same header? Defense in depth. The KMAC256 header tag is verifiable independent of AES−256−GCM decapsulation, allowing a parser to authenticate the header before attempting to release any plaintext. AES-GCM AAD authentication binds the header to the payload. Either alone would suffice in the formal model; both together provide implementation latitude (early header rejection without AES initialization) and resistance to bugs in one component compromising the other.

Q13. Why fixed offsets rather than self-describing structures? Speed of parser rejection and forensic identifiability. A fixed-offset header can be validated in microseconds by tools that do not understand the cryptographic layer. Self-describing formats (CBOR, ASN.1) are admitted for variable-length structured fields where flexibility outweighs the parser-complexity cost.

Q14. Does ZelEn protect against a malicious certificate authority? A malicious issuer is a fundamental trust-anchor failure that no certificate-bearing system survives. ZelEn supports transparency-anchored profiles, where issued certificates are logged to an append-only structure (e.g., Rosecoin or a Certificate-Transparency-style log), to detect malicious issuance after the fact. ZelEn does not claim to prevent malicious issuance; it claims to make it detectable and revocable.

Q15. What is the upgrade path if a primitive is broken? Suite-ID-based negotiation. If ML-KEM were to be weakened, ZelEn introduces a new suite ID for a successor KEM, and recipients gradually re-issue `.zpub` bundles under the new suite. Senders and recipients negotiate the strongest mutually accepted suite. Old objects remain decryptable under the original suite if local policy permits.

Q16. How does ZelEn handle very large files? Chunked mode (section 16) supports streaming encryption and decryption with bounded memory. Per-chunk authentication surfaces tampering early. Final delivery may be deferred until the object signature has verified.

Q17. What about quantum-safe handshakes within the encryption flow? ML-KEM-1024 is the quantum-safe handshake. The KEM produces a shared secret that is then expanded by KMAC256 into per-purpose keys. There is no separate handshake; the KEM is the handshake.

Q18. Why not authenticate the plaintext directly with the signature? The signature transcript covers $\mathsf{H}(C)$, where C is the AES-GCM ciphertext. Because AES−256−GCM is an AEAD, C uniquely determines M given k_{enc} and n. Signing the ciphertext rather than the plaintext yields the same authenticity guarantee while avoiding the need for the signing context to access plaintext. This also enables hybrid signing-then-encrypting deployments where the signer does not see plaintext.

25 Comparison with Related Standards

ZelEn occupies a niche distinct from existing cryptographic envelope formats. We compare it briefly with three relevant designs.

25.1 Cryptographic Message Syntax (CMS, RFC 5652)

CMS is the X.509-aligned envelope format used by S/MIME and timestamping. CMS supports KEM-based encryption (RFC 9629) and signed-data envelopes. Differences from ZelEn:

- CMS uses ASN.1 DER encoding; ZelEn uses a fixed binary header for the security-critical portion and canonical CBOR (or equivalent) for variable-length structured fields. Fixed offsets enable microsecond parser rejection.
- CMS does not mandate a specific cryptographic profile; ZelEn fixes ZELEN-PQ5 as the default to remove algorithm-selection foot-guns.
- CMS does not provide a forensic file-level brand marker; ZelEn provides `ZELEN` at `0x00` and `ZELC` at `0x7D`.
- CMS does not enforce compiler-level type separation; ZelEn does, via ZelC.

25.2 JOSE (JWE / JWS, RFC 7515–7520)

JOSE uses JSON encoding for cryptographic envelopes and supports both AEAD encryption (JWE) and signing (JWS). Differences from ZelEn:

- JOSE uses Base64URL-encoded JSON; ZelEn uses binary. The size and parser cost of JOSE are higher.
- JOSE has historically suffered from algorithm-confusion attacks (e.g., `alg=none`); ZelEn's authenticated suite ID and constant-error policy mitigate this class of attack.
- JOSE does not provide a forensic file-level marker; ZelEn does.

25.3 age (`age-encryption.org`)

age is a modern file encryption format optimized for simplicity. Differences from ZelEn:

- age uses X25519 and ChaCha20-Poly1305; ZelEn uses ML-KEM-1024 and AES−256−GCM. age is not post-quantum.
- age does not include a certificate model; ZelEn does.
- age does not include a signed transcript; ZelEn does.
- age does not include policy fields, multi-recipient authentication, or compiler-level enforcement; ZelEn does.
- age is deliberately minimalist; ZelEn is deliberately enterprise-governed.

ZelEn does not aim to replace age for personal-use file encryption. age is excellent in its niche; ZelEn occupies a different one.

25.4 Hybrid Public Key Encryption (HPKE, RFC 9180)

HPKE is a modern KEM-based encryption framework that has been adopted by TLS Encrypted Client Hello, MLS, and other protocols. Differences from ZelEn:

- HPKE is a general-purpose encryption framework; ZelEn is a complete object-security architecture including certificates, lifecycle, parser rules, and compiler enforcement.
- HPKE supports several modes (Base, PSK, Auth, AuthPSK); ZelEn fixes a single object model with a signed transcript, providing AES−256−GCM-bound and ML-DSA-bound authentication simultaneously.
- HPKE was originally specified with classical KEMs; post-quantum HPKE profiles are under active standardization. ZelEn-style envelopes can be built on top of HPKE in principle, but ZelEn defines the envelope, certificate, lifecycle, and governance layers that HPKE does not.

26 Implementation Considerations

26.1 Memory-Safe Languages

Reference and production implementations should be written in memory-safe languages. Rust is strongly preferred. Go and modern C# are acceptable. C and C++ are permitted only with rigorous static analysis, sanitizer-driven CI, and bounded use of `unsafe`.

26.2 Constant-Time Discipline

All operations on secret material must be constant-time with respect to the secret. This includes:

- ML-KEM and ML-DSA implementations.
- AES−256−GCM implementations (use AES-NI / VAES where available; fall back to bitsliced constant-time implementations).
- Comparison of MAC tags, header tags, and fingerprints (use constant-time comparison routines).
- Conditional branches over secret-dependent values (avoid).

26.3 Hardware-Backed Key Storage

Production `.zkey` material should reside in:

- Hardware Security Modules (HSMs) supporting ML-KEM and ML-DSA (vendor adoption is in progress; ZelEn implementations should be modular to swap HSM providers as they qualify).
- Trusted Execution Environments (Intel SGX, ARM TrustZone, AMD SEV) where permitted by threat model.
- Operating-system-managed sealed key stores (Apple Secure Enclave, Windows TPM-backed key storage).

26.4 CSPRNG Discipline

The host CSPRNG must satisfy NIST SP 800-90A/B/C. On Linux, prefer `getrandom(2)` with the appropriate flags. On Windows, prefer `BCryptGenRandom`. On macOS/iOS, prefer `SecRandomCopyBytes`. Implementations must not roll their own RNG.

26.5 Logging Discipline

ZelEn audit logs must:

- never record key material or plaintext;
- record fingerprints, suite IDs, object sizes, and constant-error result codes;
- be append-only and integrity-protected (cryptographic hash chain or signed log entries);
- respect data-protection regulation (PII handling, retention limits).

26.6 Hardware Acceleration

ZelEn implementations should detect and use:

- AES-NI / VAES for AES−256−GCM;
- SHA extensions where available;
- SIMD instructions for KMAC256 / cSHAKE256 (e.g., AVX-512 Keccak permutation);
- future ML-KEM / ML-DSA accelerators as they become available.

27 Conclusion

Final Assessment.
ZelEn is a quantum-safe governed object-security architecture that turns NIST post-quantum primitives into deployable Zelfire-native keys, certificates, encrypted containers, policy enforcement, and audit-ready enterprise cryptographic workflows.

ZelEn does not replace NIST cryptography. ZelEn does not invent new public-key hardness. ZelEn does not promise unbreakability. ZelEn does not protect endpoints, plaintext after decryption, or systems with broken randomness or compromised keys. These boundaries are deliberate. They reflect the discipline of an architecture that knows what it is, and what it is not.

ZelEn provides a standards-anchored cryptographic suite (ZELEN-PQ5) using ML-KEM-1024, ML-DSA-87, AES−256−GCM with strict 96-bit nonces, and a domain-separated KMAC256 key schedule. ZelEn provides four native object classes (`.zkey`, `.zpub`, `.zcert`, `.zelen`) with type-level role separation. ZelEn provides an authenticated canonical header carrying `ZELEN` at offset `0x00` and a fixed ZelC brand marker `ZELC` at offset `0x7D`, bound into the cryptographic transcript. ZelEn provides certificate-based identity, signed transcripts that cover header, KEM ciphertext, payload, GCM tag, fingerprints, and policy. ZelEn provides multi-recipient and chunked profiles, parser-safety rules, ZelC compiler enforcement, key lifecycle controls, suite agility, and a clear compliance and validation roadmap.

The core thesis stands. NIST builds the cryptographic engines. ZelEn builds the armored vehicle around them. The result is a deployable, audit-ready, post-quantum native object format for the Zelfire / ZelC ecosystem.

ZelEn is not new magic encryption. It is a quantum-safe governed object-security architecture that turns NIST post-quantum primitives into deployable Zelfire-native keys, certificates, encrypted containers, policy enforcement, and audit-ready enterprise cryptographic workflows.

A Mathematical Notation Summary

Table 5: Notation Summary

Symbol	Meaning
$\{0,1\}^n$	set of n-bit strings
$x \xleftarrow{\$} S$	uniform random sampling of x from S
$a \,\|\, b$	concatenation of byte strings a and b
$\mathsf{Trunc}_n(x)$	truncation of x to first n bits
$\mathsf{CE}(\cdot)$	canonical encoding (deterministic, injective, total)
$\mathsf{H}(\cdot)$	cryptographic hash (default SHA−256)
$\mathsf{KMAC256}(K, X, L, S)$	KMAC256 keyed PRF/XOF (NIST SP 800-185), with key K, message X, output length L *in bits*, and customization string S
ML-KEM.Encaps	ML-KEM encapsulation (FIPS 203)
ML-KEM.Decaps	ML-KEM decapsulation (FIPS 203)
ML-DSA.Sign	ML-DSA signing (FIPS 204)
ML-DSA.Verify	ML-DSA verification (FIPS 204)
AES−256−GCM.Enc	AES-256-GCM authenticated encryption (SP 800-38D)
AES−256−GCM.Dec	AES-256-GCM authenticated decryption (SP 800-38D)
$\mathsf{fp}_S, \mathsf{fp}_R$	sender, recipient 32-byte fingerprints
suite_id	16-bit big-endian suite identifier
K	ML-KEM shared secret (256 bits)
c_{kem}	ML-KEM ciphertext
n	AES−256−GCM nonce (96 bits)
H_0	preliminary header (header-tag region zeroed)
H	finalized header (header-tag region populated)
C, T	AES−256−GCM ciphertext, authentication tag
Σ	ML-DSA object signature
$\mathsf{Adv}^X_Y(\mathcal{Z})$	advantage of adversary $\mathcal{Z}$ in security game X for primitive Y
$\varepsilon_{\mathsf{CE}}$	canonical encoding ambiguity probability
$\varepsilon_{\text{parse}}$	parser failure probability

B Detailed Header Layout (Reference)

The fixed header for ZELEN-PQ5 v1 is exactly 145 (`0x91`) bytes. The complete byte-by-byte layout is:

Offset (hex)	Length	Field	Encoding
`0x00`	5	Magic	ASCII `Z E L E N` = `0x5A 0x45 0x4C 0x45 0x4E`
`0x05`	1	Format version	u_8 = `0x01`
`0x06`	1	Flags bitfield	u_8
`0x07`	1	Security tier	u_8 = `0x05`
`0x08`	1	Header length	u_8 = `0x91` (v1)
`0x09`	2	Suite ID	u_{16} BE = `0x01 0x01`
`0x0B`	2	Certificate type	u_{16} BE
`0x0D`	4	Payload length	u_{32} BE
`0x11`	32	Sender fingerprint	raw bytes
`0x31`	32	Recipient identifier	raw bytes; meaning depends on profile (see header table notes)
`0x51`	32	KEM ciphertext digest	raw bytes
`0x71`	12	AEAD nonce	raw bytes
`0x7D`	4	ZelEn brand mark	ASCII `Z E L C` = `0x5A 0x45 0x4C 0x43`
`0x81`	16	Header authentication tag	raw bytes

Flag bitfield (offset `0x06`).

Bit	Meaning	When set
0	Signature present	object signature Σ included
1	Multi-recipient mode	recipient list block present
2	Chunked payload	chunk manifest present, payload split into chunks
3	Compression applied	payload compressed before AEAD encryption
4	MTE extension present	MTE-encoded metadata block present
5	Counter-signature (SLH-DSA)	second signature block present
6	Reserved	must be zero in v1
7	Reserved	must be zero in v1

C Python Illustrative Prototype

The following Python code is a *non-conformant illustrative prototype*, intended for pedagogical purposes only. It is *not* a reference implementation in the conformance sense. It is not production key storage, it is not a validated cryptographic module, it does not enforce the full ZelEn governance model, and it deliberately violates several of the normative requirements of this specification, as detailed below. Production deployments must use validated cryptographic modules where required, must protect `.zkey` material with hardware-backed storage, and must implement the full parser-safety, policy, and lifecycle requirements of the specification.

Known deviations from the ZelEn specification. This prototype deviates from the specification in the following ways:

- **Constant-error policy violation.** The code raises distinct `ValueError` messages ("missing ZELEN magic", "missing ZELC brand mark", "unsupported suite", "KEM digest mismatch", "wrong recipient", "header tag failed", "sender fp mismatch", "object signature failed", "zcert signature failed"). A conformant implementation must collapse all of these to a single uniform error code with no observable distinction in timing, content, or side channel.
- **KDF construction.** The prototype uses a SHAKE-256-based XOF instead of KMAC256. A conformant implementation must use KMAC256 per SP 800-185 with explicit output length in bits.
- **Canonical encoding.** The prototype uses sorted-key JSON; a conformant implementation must use a strict binary canonical encoding with deterministic, injective encoding rules.
- **Key storage.** The prototype keeps raw private material in memory and on disk in unprotected form; a conformant implementation must use hardware-backed or sealed key storage.
- **Certificate model.** The prototype uses only self-signed certificates; a conformant implementation must support a proper trust hierarchy.
- **Profile coverage.** The prototype implements the single-recipient non-chunked profile only; multi-recipient and chunked profiles are out of scope.

The prototype is included to make the specification concrete and to support "does this round-trip work end-to-end" testing on a developer's laptop. Anyone reading this prototype must treat its non-conformances as deliberate teaching simplifications, not as reference behavior.

Dependencies. `liboqs-python` (Open Quantum Safe) for ML-KEM-1024 and ML-DSA-87. The `cryptography` library for AES−256−GCM. Standard library for hashing, serialization, and structure.

```
"""
ZelEn illustrative prototype.

NON-CONFORMANT. PEDAGOGICAL ONLY. NOT A REFERENCE IMPLEMENTATION.

Requires:
    pip install cryptography
    plus liboqs-python installed from Open Quantum Safe.

This is not production key storage and deliberately violates the
constant-error policy of the ZelEn specification for teaching clarity.
"""
from __future__ import annotations
import base64, hashlib, hmac, json, os, struct, time
from typing import Any, Dict, Tuple
import oqs
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

KEM_ALG = "ML-KEM-1024"
SIG_ALG = "ML-DSA-87"
```

```
SUITE_ID = 0x0101
SUITE_NAME = "ZELEN-PQ5"

HEADER_LEN      = 0x91
OFF_MAGIC       = 0x00
OFF_VERSION     = 0x05
OFF_FLAGS       = 0x06
OFF_TIER        = 0x07
OFF_HEADER_LEN  = 0x08
OFF_SUITE_ID    = 0x09
OFF_CERT_TYPE   = 0x0B
OFF_PAYLOAD_LEN = 0x0D
OFF_SENDER_FP   = 0x11
OFF_RECIPIENT_FP = 0x31
OFF_KEM_DIGEST = 0x51
OFF_NONCE       = 0x71
OFF_ZELC        = 0x7D
OFF_HEADER_TAG = 0x81

MAGIC = b"ZELEN"
BRAND = b"ZELC"

def b64e(raw): return base64.b64encode(raw).decode("ascii")
def b64d(text): return base64.b64decode(text.encode("ascii"))
def sha256(data): return hashlib.sha256(data).digest()
def lp(data): return len(data).to_bytes(4, "big") + data
def canonical(obj):
    return json.dumps(obj, sort_keys=True, separators=(",",
        ":")).encode("utf-8")

def xof(label, *parts, out_len):
    """Prototype domain-separated XOF.
    Production ZelEn must use SP 800-185 KMAC256/cSHAKE256."""
    h = hashlib.shake_256()
    h.update(lp(b"ZELEN-XOF-v1"))
    h.update(lp(label.encode("utf-8")))
    for part in parts:
        h.update(lp(part))
    return h.digest(out_len)

def zpub_fingerprint_from_body(zpub_without_fp):
    return sha256(canonical(zpub_without_fp)).hex()

def verify_zpub_fingerprint(zpub):
    fp = zpub["fingerprint"]
    body = dict(zpub); del body["fingerprint"]
    expected = zpub_fingerprint_from_body(body)
    if not hmac.compare_digest(fp, expected):
        raise ValueError("zpub fingerprint mismatch")

def generate_zelen_identity(subject):
    with oqs.KeyEncapsulation(KEM_ALG) as kem:
        pk_kem = kem.generate_keypair()
        sk_kem = kem.export_secret_key()
    with oqs.Signature(SIG_ALG) as signer:
        pk_sig = signer.generate_keypair()
        sk_sig = signer.export_secret_key()
        zpub_body = {
            "type": "zpub", "version": 1, "suite": SUITE_NAME,
            "suite_id": SUITE_ID, "subject": subject,
            "kem_alg": KEM_ALG, "sig_alg": SIG_ALG,
            "pk_kem": b64e(pk_kem), "pk_sig": b64e(pk_sig),
```

```
            "key_usage": ["encrypt", "verify", "cert-subject"],
            "created_unix": int(time.time()),
        }
        fingerprint = zpub_fingerprint_from_body(zpub_body)
        zpub = dict(zpub_body); zpub["fingerprint"] = fingerprint
        cert_tbs = {
            "type": "zcert-tbs", "version": 1,
            "issuer": subject, "subject": subject,
            "subject_public": zpub,
            "not_before_unix": int(time.time()),
            "not_after_unix": int(time.time()) + 365 * 24 * 3600,
            "policy": {"export": "no-export-preferred",
                       "zelen_required": True,
                       "allowed_suites": [SUITE_NAME]},
        }
        cert_sig = signer.sign(canonical(cert_tbs))
    zcert = {"type": "zcert", "version": 1, "signature_alg": SIG_ALG,
             "tbs": cert_tbs, "signature": b64e(cert_sig)}
    zkey = {"type": "zkey", "version": 1, "suite": SUITE_NAME,
            "subject": subject, "fingerprint": fingerprint,
            "public": zpub, "sk_kem": b64e(sk_kem), "sk_sig": b64e(sk_sig),
            "local_policy": {"export": "demo-only"}}
    return zkey, zpub, zcert
```

Listing 1: ZelEn illustrative prototype (single-recipient, non-chunked, distinct-error)

```
def build_header_without_tag(*, sender_fp, recipient_fp,
                             kem_digest, nonce, payload_len, flags=0x01):
    if len(sender_fp) != 32: raise ValueError("sender fp must be 32 bytes")
    if len(recipient_fp) != 32: raise ValueError("recipient fp must be 32
        bytes")
    if len(kem_digest) != 32: raise ValueError("KEM digest must be 32 bytes")
    if len(nonce) != 12: raise ValueError("nonce must be 12 bytes")
    h = bytearray(HEADER_LEN)
    h[OFF_MAGIC:OFF_MAGIC + 5] = MAGIC
    h[OFF_VERSION] = 1
    h[OFF_FLAGS] = flags
    h[OFF_TIER] = 5
    h[OFF_HEADER_LEN] = HEADER_LEN
    struct.pack_into(">H", h, OFF_SUITE_ID, SUITE_ID)
    struct.pack_into(">H", h, OFF_CERT_TYPE, 0x0001)
    payload_len32 = 0xFFFFFFFF if payload_len >= 2**32 else payload_len
    struct.pack_into(">I", h, OFF_PAYLOAD_LEN, payload_len32)
    h[OFF_SENDER_FP:OFF_SENDER_FP + 32] = sender_fp
    h[OFF_RECIPIENT_FP:OFF_RECIPIENT_FP + 32] = recipient_fp
    h[OFF_KEM_DIGEST:OFF_KEM_DIGEST + 32] = kem_digest
    h[OFF_NONCE:OFF_NONCE + 12] = nonce
    h[OFF_ZELC:OFF_ZELC + 4] = BRAND
    h[OFF_HEADER_TAG:OFF_HEADER_TAG + 16] = b"\x00" * 16
    return bytes(h)

def header_with_tag(header_without_tag, tag):
    h = bytearray(header_without_tag)
    h[OFF_HEADER_TAG:OFF_HEADER_TAG + 16] = tag
    return bytes(h)

def zero_header_tag(header):
    h = bytearray(header)
    h[OFF_HEADER_TAG:OFF_HEADER_TAG + 16] = b"\x00" * 16
    return bytes(h)

def derive_keys(shared_secret, header_without_tag):
```

```
    prk = xof("ZELEN-PQ5 key schedule", shared_secret,
              header_without_tag, out_len=64)
    k_enc = xof("AES-256-GCM payload key", prk, out_len=32)
    k_hdr = xof("ZelEn header authentication key", prk, out_len=32)
    return k_enc, k_hdr

def compute_header_tag(k_hdr, header_without_tag):
    return xof("ZelEn fixed header tag", k_hdr,
               header_without_tag, out_len=16)

def signature_input(header, kem_ct, sender_cert_bytes, ciphertext_and_tag):
    transcript = {
        "domain": "ZELEN-SIG-v1", "suite": SUITE_NAME,
        "header_hash": sha256(header).hex(),
        "kem_ciphertext_hash": sha256(kem_ct).hex(),
        "sender_cert_hash": sha256(sender_cert_bytes).hex(),
        "ciphertext_tag_hash": sha256(ciphertext_and_tag).hex(),
    }
    return canonical(transcript)
```

Listing 2: ZelEn header construction and key derivation

```
def pack_block(data): return struct.pack(">I", len(data)) + data

def read_block(blob, pos):
    if pos + 4 > len(blob): raise ValueError("truncated block length")
    n = struct.unpack(">I", blob[pos:pos + 4])[0]
    pos += 4
    if pos + n > len(blob): raise ValueError("truncated block body")
    return blob[pos:pos + n], pos + n

def zelen_encrypt(plaintext, recipient_zpub, sender_zkey, sender_zcert):
    verify_zpub_fingerprint(recipient_zpub)
    recipient_fp = bytes.fromhex(recipient_zpub["fingerprint"])
    sender_fp = bytes.fromhex(sender_zkey["fingerprint"])
    pk_kem_R = b64d(recipient_zpub["pk_kem"])
    with oqs.KeyEncapsulation(KEM_ALG) as kem_sender:
        kem_ct, shared_secret = kem_sender.encap_secret(pk_kem_R)
    kem_digest = sha256(kem_ct)
    nonce = os.urandom(12)
    header0 = build_header_without_tag(
        sender_fp=sender_fp, recipient_fp=recipient_fp,
        kem_digest=kem_digest, nonce=nonce, payload_len=len(plaintext))
    k_enc, k_hdr = derive_keys(shared_secret, header0)
    htag = compute_header_tag(k_hdr, header0)
    header = header_with_tag(header0, htag)
    aesgcm = AESGCM(k_enc)
    ciphertext_and_tag = aesgcm.encrypt(nonce, plaintext,
                                        associated_data=header)
    sender_cert_bytes = canonical(sender_zcert)
    sig_msg = signature_input(header, kem_ct, sender_cert_bytes,
                              ciphertext_and_tag)
    sk_sig_S = b64d(sender_zkey["sk_sig"])
    with oqs.Signature(SIG_ALG, sk_sig_S) as signer:
        sig = signer.sign(sig_msg)
    return (header + pack_block(kem_ct) + pack_block(sender_cert_bytes)
            + pack_block(ciphertext_and_tag) + pack_block(sig))

def zelen_decrypt(zelen_blob, recipient_zkey):
    if len(zelen_blob) < HEADER_LEN:
        raise ValueError("truncated .zelen object")
    header = zelen_blob[:HEADER_LEN]
```

```
    if header[OFF_MAGIC:OFF_MAGIC + 5] != MAGIC:
        raise ValueError("missing ZELEN magic")
    if header[OFF_ZELC:OFF_ZELC + 4] != BRAND:
        raise ValueError("missing fixed ZELC brand mark at 0x7D")
    suite_id = struct.unpack(">H", header[OFF_SUITE_ID:OFF_SUITE_ID + 2])[0]
    if suite_id != SUITE_ID:
        raise ValueError("unsupported suite")
    nonce = header[OFF_NONCE:OFF_NONCE + 12]
    header0 = zero_header_tag(header)
    header_tag = header[OFF_HEADER_TAG:OFF_HEADER_TAG + 16]
    pos = HEADER_LEN
    kem_ct, pos = read_block(zelen_blob, pos)
    sender_cert_bytes, pos = read_block(zelen_blob, pos)
    ciphertext_and_tag, pos = read_block(zelen_blob, pos)
    sig, pos = read_block(zelen_blob, pos)
    if pos != len(zelen_blob):
        raise ValueError("trailing bytes")
    if not hmac.compare_digest(sha256(kem_ct),
                               header[OFF_KEM_DIGEST:OFF_KEM_DIGEST + 32]):
        raise ValueError("KEM digest mismatch")
    recipient_fp = bytes.fromhex(recipient_zkey["fingerprint"])
    if not hmac.compare_digest(recipient_fp,
                               header[OFF_RECIPIENT_FP:OFF_RECIPIENT_FP +
                                   32]):
        raise ValueError("wrong recipient")
    sk_kem_R = b64d(recipient_zkey["sk_kem"])
    with oqs.KeyEncapsulation(KEM_ALG, sk_kem_R) as kem_recipient:
        shared_secret = kem_recipient.decap_secret(kem_ct)
    k_enc, k_hdr = derive_keys(shared_secret, header0)
    expected_htag = compute_header_tag(k_hdr, header0)
    if not hmac.compare_digest(expected_htag, header_tag):
        raise ValueError("header tag failed")
    sender_zcert = json.loads(sender_cert_bytes.decode("utf-8"))
    sender_zpub = verify_self_signed_zcert(sender_zcert)
    sender_fp = bytes.fromhex(sender_zpub["fingerprint"])
    if not hmac.compare_digest(sender_fp,
                               header[OFF_SENDER_FP:OFF_SENDER_FP + 32]):
        raise ValueError("sender fp mismatch")
    sig_msg = signature_input(header, kem_ct, sender_cert_bytes,
                              ciphertext_and_tag)
    pk_sig_S = b64d(sender_zpub["pk_sig"])
    with oqs.Signature(SIG_ALG) as verifier:
        ok = verifier.verify(sig_msg, sig, pk_sig_S)
    if not ok:
        raise ValueError("object signature failed")
    aesgcm = AESGCM(k_enc)
    return aesgcm.decrypt(nonce, ciphertext_and_tag, associated_data=header)

def verify_self_signed_zcert(zcert):
    tbs = zcert["tbs"]
    zpub = tbs["subject_public"]
    verify_zpub_fingerprint(zpub)
    pk_sig = b64d(zpub["pk_sig"])
    sig = b64d(zcert["signature"])
    with oqs.Signature(zcert["signature_alg"]) as verifier:
        ok = verifier.verify(canonical(tbs), sig, pk_sig)
    if not ok: raise ValueError("zcert signature failed")
    now = int(time.time())
    if now < tbs["not_before_unix"] or now > tbs["not_after_unix"]:
        raise ValueError("zcert validity failed")
    return zpub
```

Listing 3: ZelEn encryption and decryption

Demonstration.

```
if __name__ == "__main__":
    alice_zkey, alice_zpub, alice_zcert = generate_zelen_identity(
        "alice@zelfire.local")
    bob_zkey, bob_zpub, bob_zcert = generate_zelen_identity(
        "bob@zelfire.local")
    message = b"Zelfire confidential payload: PQ object encryption demo."
    blob = zelen_encrypt(plaintext=message, recipient_zpub=bob_zpub,
                         sender_zkey=alice_zkey, sender_zcert=alice_zcert)
    recovered = zelen_decrypt(blob, recipient_zkey=bob_zkey)
    assert recovered == message
    print("Recovered:", recovered)
    print("ZELEN magic:", blob[0:5])
    print("ZELC marker at 0x7D:", blob[0x7D:0x81])
    print("Object size:", len(blob), "bytes")
```

Listing 4: Demonstration round-trip

What this demonstrates. The prototype shows the architecture concretely. `.zkey` supplies private KEM and signature material. `.zpub` supplies public KEM and verification material. `.zcert` supplies a signed identity binding over a `.zpub`. `.zelen` supplies a fixed header, KEM ciphertext, AEAD payload, AEAD tag, sender certificate, and object signature. ML-KEM produces a shared secret; ZelEn produces a shared secret *plus* a file format, an identity binding, a signature transcript, a policy frame, a forensic marker, and a parser discipline.

What this prototype is not.

- Not a reference implementation; it is an illustrative prototype.
- Not production key storage; raw private material is visible.
- Not a validated cryptographic module.
- Not constant-error; raises distinct exceptions per failure type.
- Uses SHAKE-256-based XOF for prototype simplicity; conformant implementations must use KMAC256 per SP 800-185.
- Uses JSON canonical encoding for readability; conformant implementations must use a strict binary canonical encoding.
- Single-recipient and non-chunked; multi-recipient and chunked extensions follow the specifications in sections 15 and 16.

D Test Vector Format (Reference)

A conformance test vector for ZELEN-PQ5 round-trip is a JSON record with the following fields:

```
{
  "vector_id": "zelen-pq5-roundtrip-001",
  "suite": "ZELEN-PQ5",
  "suite_id": "0x0101",
  "kem_seed_hex": "<deterministic KEM keygen seed, hex>",
  "sig_seed_hex": "<deterministic ML-DSA keygen seed, hex>",
  "recipient_subject": "test-recipient@zelfire.test",
  "sender_subject": "test-sender@zelfire.test",
  "plaintext_hex": "<plaintext bytes, hex>",
  "nonce_hex": "<96-bit nonce, hex>",
  "expected_zelen_hex": "<full .zelen object, hex>",
  "expected_header_hex": "<145-byte fixed header, hex>",
  "expected_kem_ciphertext_hex": "<KEM ciphertext, hex>",
  "expected_aead_ciphertext_hex": "<AES-GCM ciphertext, hex>",
  "expected_aead_tag_hex": "<16-byte tag, hex>",
  "expected_signature_hex": "<ML-DSA-87 signature, hex>",
  "expected_recovered_plaintext_hex": "<must equal plaintext_hex>",
  "negative_tests": [
    {"name": "missing_zelen_magic",
     "mutation": "zero offset 0x00..0x04",
     "expected_result": "uniform error"},
    {"name": "missing_zelc_marker",
     "mutation": "zero offset 0x7D..0x80",
     "expected_result": "uniform error"},
    {"name": "downgrade_suite",
     "mutation": "rewrite suite_id to 0x0099",
     "expected_result": "uniform error"},
    {"name": "altered_kem_digest",
     "mutation": "flip last byte of offset 0x51..0x70",
     "expected_result": "uniform error"},
    {"name": "altered_payload",
     "mutation": "flip first byte of AES-GCM ciphertext",
     "expected_result": "uniform error"},
    {"name": "altered_signature",
     "mutation": "flip last byte of object signature",
     "expected_result": "uniform error"}
  ]
}
```

Listing 5: Test vector record schema

A conformant ZelEn implementation must produce identical output bytes given identical seeds (modulo the AEAD nonce, which is sampled fresh per encryption and is one of the test vector inputs in deterministic-test mode). All listed negative tests must fail with the uniform error code.

References

[1] National Institute of Standards and Technology. *Module-Lattice-Based Key-Encapsulation Mechanism Standard.* FIPS PUB 203, August 2024.

[2] National Institute of Standards and Technology. *Module-Lattice-Based Digital Signature Standard.* FIPS PUB 204, August 2024.

[3] National Institute of Standards and Technology. *Stateless Hash-Based Digital Signature Standard.* FIPS PUB 205, August 2024.

[4] National Institute of Standards and Technology. *Advanced Encryption Standard (AES).* FIPS PUB 197, May 2023 (revised).

[5] M. Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.* NIST SP 800-38D, November 2007.

[6] J. Kelsey, S. Chang, R. Perlner. *SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash.* NIST SP 800-185, December 2016.

[7] National Institute of Standards and Technology. *Recommendations for Key-Encapsulation Mechanisms.* NIST SP 800-227, September 2025.

[8] E. Barker, J. Kelsey. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators.* NIST SP 800-90A Rev. 1, June 2015.

[9] E. Barker, J. Kelsey. *Recommendation for the Entropy Sources Used for Random Bit Generation.* NIST SP 800-90B, January 2018.

[10] National Institute of Standards and Technology. *Recommendation for Random Bit Generator (RBG) Constructions.* NIST SP 800-90C, September 2025.

[11] L. Chen. *Recommendation for Key Derivation Using Pseudorandom Functions (Revised).* NIST SP 800-108 Rev. 1 (with update 1), February 2024. Defines KDF in Counter, Feedback, and Double-Pipeline modes including KMAC-based key derivation.

[12] Open Quantum Safe Project. *liboqs-python: Python 3 bindings for liboqs.* https://github.com/open-quantum-safe/liboqs-python.

[13] R. Barnes, K. Bhargavan, B. Lipp, C. Wood. *Hybrid Public Key Encryption.* RFC 9180, February 2022.

[14] R. Housley. *Cryptographic Message Syntax (CMS).* RFC 5652, September 2009.

[15] M. Jones, J. Hildebrand. *JSON Web Encryption (JWE).* RFC 7516, May 2015.

End of document. Built with love by Haja Mo.