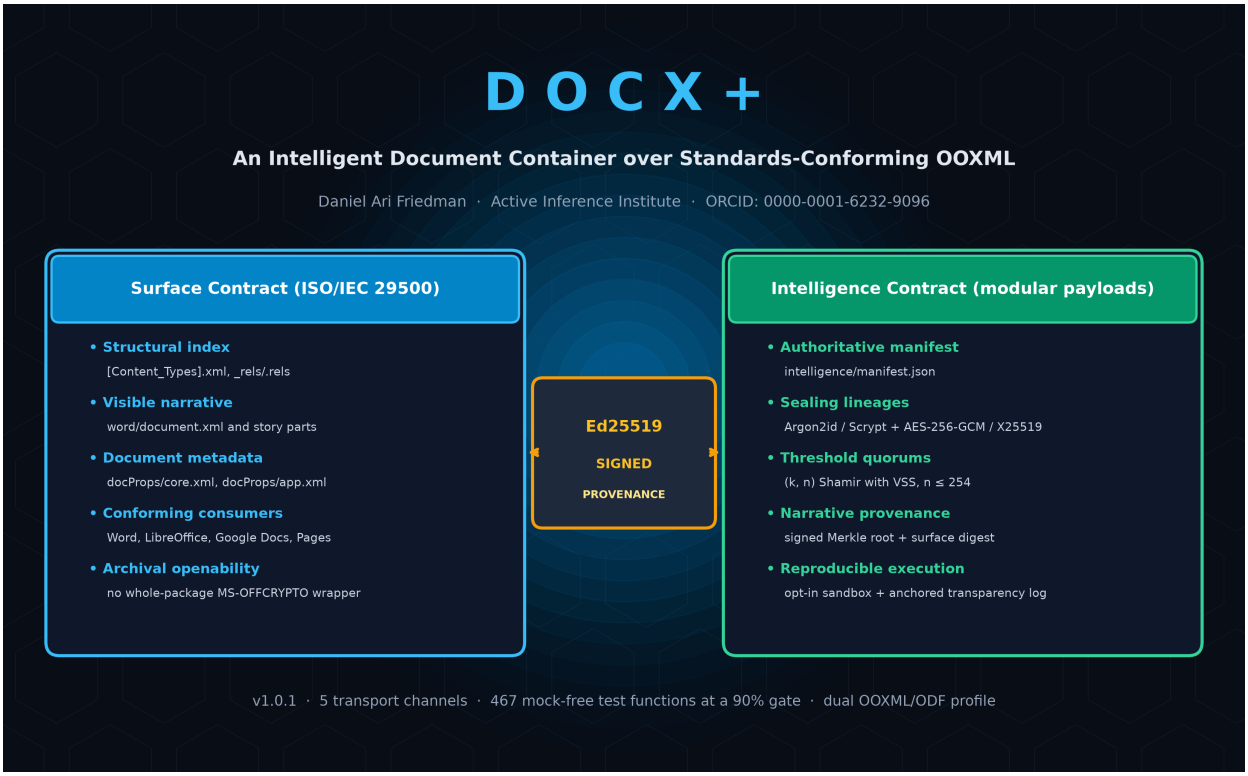


# docxplus: An Intelligent Document Container

*Carrying Signed, Encrypted, Modular, Self-Verifying Payloads Inside an Ordinary .docx*

**Daniel Ari Friedman**

Active Inference Institute

`daniel@activeinference.institute` · ORCID: 0000-0001-6232-9096

Version: 1.0.1 · Date: 2026-08-17

Standard Reference: ISO/IEC 29500-2 (OPC) · OASIS OpenDocument v1.3/v1.4

DOI: 10.5281/zenodo.21985580

# Abstract

A `.docx` file is a ZIP archive governed by the Open Packaging Conventions (OPC; ISO/IEC 29500-2 / ECMA-376-2) (Information Technology – Document Description and Processing Languages – Office Open XML File Formats (Parts 1–4) 2021), and its OpenDocument counterpart (OASIS ODF v1.3/v1.4) (Open Document Format for Office Applications (OpenDocument) Version 1.3 2021) is built the same way. Both standards document extension points — auxiliary package parts, custom XML datastores, application property sets, embedded media, markup-compatibility choice blocks — that exist precisely so a package can carry more than a conforming consumer knows how to read. **docxplus** is an open specification and reference implementation that takes those affordances seriously. A single archive satisfies two independent contracts at once: it is a conforming `.docx` or `.odt` that opens unremarkably in Microsoft Word, LibreOffice, and Google Docs, and it is an authenticated, modular carrier of typed computational payloads indexed by a signed manifest.

The design turns on one decision: seal payloads, never the package. Whole-package MS-OFFCRYPTO encryption (Microsoft Corporation 2024) buys confidentiality by destroying the artefact — the prose becomes unreadable to anyone without a credential, and unreadable to archives permanently. Sealing each module instead leaves the surface document public and openable while confidentiality applies exactly where it is wanted. Modules seal under memory-hard password derivation (Argon2id (Biryukov et al. 2021) or Scrypt (Percival and Josefsson 2016), feeding AES-256-GCM), X25519 multi-recipient key encapsulation (Langley et al. 2016), Shamir $k$-of-$n$ threshold sharing (Shamir 1979) with verifiable-share commitments (Feldman 1987), or a decoy frame structurally indistinguishable from an ordinary sealed one. Payloads are typed (`bytes`, `text`, `json`, a nested `docxplus` container, or a whole reproducible `project` tree), so a paper can carry the code and data that produced it.

Provenance rests on a signed Merkle tree (Merkle 1987) over the module roster together with a composite digest over every package part, the content-type map, and the relationship graph — excluding only the manifest that carries the digest — so editing a footnote, a style, or the officeDocument relationship breaks the signature exactly as editing a payload does. Inclusion proofs let a third party confirm one module belongs to the signed set without seeing the others, and detached co-signatures let an institution vouch alongside an author (Josefsson and Liusvaara 2017). Throughout, a verdict of *authentic* requires the caller to pin the key they trust; the key travelling inside the manifest is self-asserted, and we say so wherever the distinction bites. A `project` module may additionally carry a signed **reproduction attestation** binding its source to an output digest, which a reader checks cryptographically while executing nothing, or re-runs by explicit opt-in inside a confined, resource-capped sandbox. Attestations chain into an append-only transparency log whose authenticity comes from a signed tree head, because a hash chain on its own proves only that a log does not contradict itself.

We implement and evaluate 5 spec-sanctioned transport channels: custom XML parts, auxiliary package parts, custom document properties, Markup Compatibility and Extensibility (MCE) choice blocks, and least-significant-bit steganography in a carrier image the document visibly displays. Concealment is measured, not asserted — a chi-squared detector (Westfeld and Pfitzmann 2000) ships in-tree, with a prefix sweep that localises the partially-filled carriers whole-image statistics miss, alongside an optional compiled backend (Fridrich et al. 2001). Across 467 mock-free test functions under a 90% coverage gate we verify deterministic serialisation, round-trip integrity, openability in mainstream word processors, and the adversarial boundaries established by 14 red-team cycles closing 88 confirmed findings — a record that includes the occasions when an earlier fix proved incomplete, and one negative result withdrawn rather than shipped.


# 1 The docxplus Format

## 1.1 What a Document Cannot Currently Do

A `.docx` file is an Open Packaging Conventions (OPC; ISO/IEC 29500-2 / ECMA-376-2) container (Information Technology – Document Description and Processing Languages – Office Open XML File Formats (Parts 1–4) 2021): a ZIP archive whose mandatory index parts are `[Content_Types].xml` and `_rels/.rels`, and in which every valid part is reached by following an explicit, typed relationship. The standard provides that "a package may contain additional files", and that mapped content-control data may live in custom XML parts. These are not loopholes discovered by inspection. They are documented extension points, put there so that a package can outlive the assumptions of any one consumer.

Yet the moment a document needs to carry something more — an analysis script, a dataset, a sealed attachment

for one named reader — the available options are both bad. Whole-package encryption (the MS-OFFCRYPTO compound envelope (Microsoft Corporation 2024)) protects the payload by making the entire artefact unreadable: the non-secret prose disappears behind a credential, and the file stops being a document at all for anyone without one, archives included. The alternative, an unencrypted attachment sitting beside the file, has no authenticated provenance, no typing, and no access control worth the name. Neither option is a document that carries intelligence; one is a locked box, the other a paper clip.

The gap matters because the document is where results are actually communicated. Reproducible-research practice has converged on the principle that code, data, and the narrative describing them should travel together and be independently re-executable (Peng 2011; Sandve et al. 2013), and the FAIR principles make machine-actionable metadata a first-class requirement rather than a courtesy (Wilkinson et al. 2016). Yet the artefact a reader receives is almost always the one component that carries none of it: a rendered document, severed from its inputs, linked at best to a repository that may move or change. Supplementary-material archives and data repositories address the storage problem while leaving the binding problem open — nothing cryptographically ties the figure on the page to the code that produced it.

This paper asks what falls out of refusing the original choice. How much structured, authenticated, selectively sealed, and optionally concealed material can a document carry while remaining a strictly *conforming* Office document across mainstream word processors?

## 1.2 Two Contracts, One Archive

**Definition 1 (Surface Contract).** A package $\mathcal{P}$ satisfies the **Surface Contract** if it strictly conforms to ISO/IEC 29500-2 (OPC) and Part 1 (WordprocessingML), comprising valid `[Content_Types].xml`, root relationship graphs `_rels/.rels`, a typed main story part `word/document.xml`, and deterministic ZIP serialization with zero colliding part names. Any conforming consumer $\mathcal{C}_{\text{std}}$ parses $\mathcal{P}$ without error.

**Definition 2 (Intelligence Contract).** A package $\mathcal{P}$ satisfies the **Intelligence Contract** if it contains an authoritative manifest part $\mathcal{M}$ = `intelligence/manifest.json` that maps a finite set of module slots $\mathcal{S} = \{s_1, \ldots, s_k\}$ to their transport channel, content type, Blake2b ciphertext digest, sealing parameters, and optional reproduction attestation, signed by an Ed25519 key (Josefsson and Liusvaara 2017) over a canonical JSON serialization.

**Theorem 1 (Dual-Contract Independence).** Let $\mathcal{P}$ be a docxplus package satisfying Definition 1 and Definition 2. Then the addition, removal, encryption, or steganographic concealment of any module $s \in \mathcal{S}$ leaves the surface validity of $\mathcal{P}$ invariant under $\mathcal{C}_{\text{std}}$.

The independence in Theorem 1 is what makes the format usable rather than merely clever: an author can add, seal, or remove intelligence without ever risking the document's openability, and a reader who has never heard of docxplus is never inconvenienced. Fig. 1 shows both contracts as parts of a single archive, and where each channel writes.

## 1.3 Transport Channels

docxplus defines 5 spec-sanctioned transport channels, each implementing a uniform `embed`/`extract`/`capacity` lifecycle:

**Definition 3 (Transport Channel).** A transport channel $\Gamma = (\text{embed}, \text{extract}, \text{capacity})$ is a triple of deterministic operations mapping an OPC package $\mathcal{P}$ and payload bytes $b \in \{0,1\}^*$ to an updated package $\mathcal{P}'$ and a manifest locator record, such that $\text{extract}(\mathcal{P}', \text{embed}(\mathcal{P}, b)) = b$.

Each channel below satisfies Definition 3; the round-trip identity in that definition is what every channel's test asserts against real packages rather than mocks, so the definition is a checked obligation rather than a description. The supported channels comprise:

- `custom_xml` — Base64 payloads in `customXml/itemN.xml` parts registered via `customXml/itemPropsN.xml`. Layout engines ignore these parts while they remain fully compliant with the OOXML custom XML mapping schemas (Information Technology – Document Description and Processing Languages – Office Open XML File Formats (Parts 1–4) 2021).
- `package_part` — Raw binary payloads in `intelligence/payloadN.dxp` under a declared content type. The high-throughput channel: large datasets, code archives, and ciphertext envelopes.

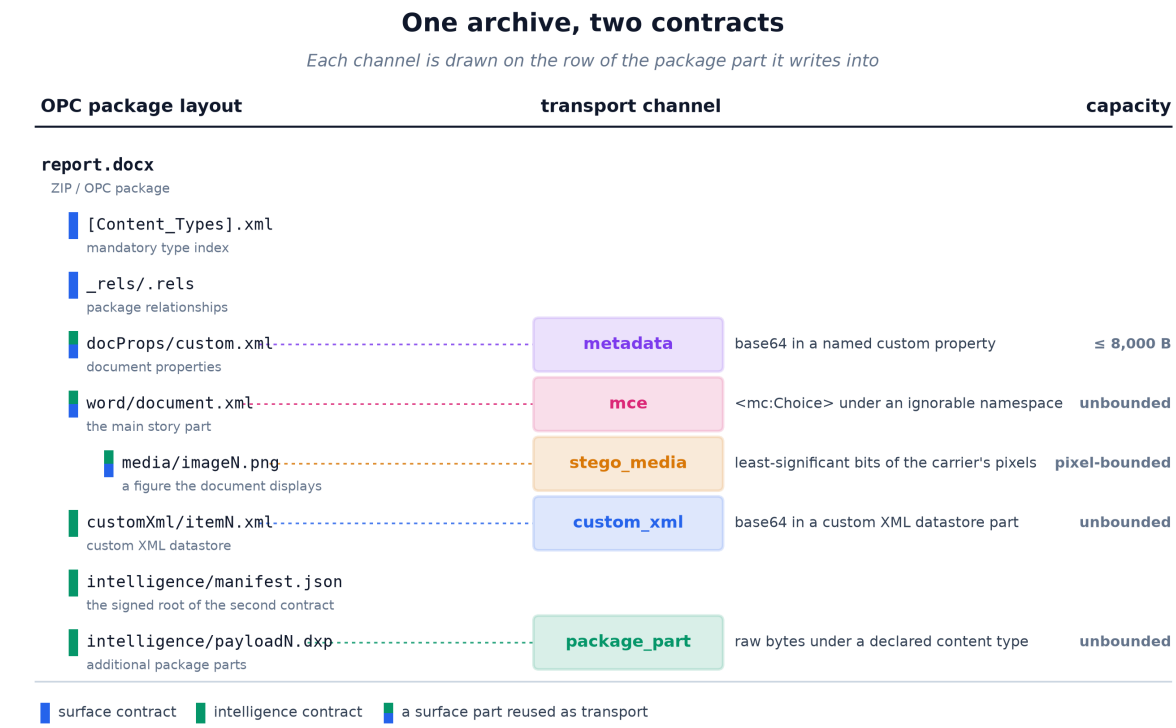


**Figure 1.** The package tree with every transport channel drawn on the row of the part it writes into. A blue marker is a part the surface contract requires, a green one a part only docxplus reads, and a split marker a required part reused as transport — which is how three of the 5 channels carry a payload without adding any part a conforming reader would find surprising. Rows with no badge carry nothing. Capacities are read from the live code constants.

- **`metadata`** — String-encoded payloads in custom document properties within `docProps/custom.xml`. Bounded at 8000 bytes, which suits routing tags, identifiers, and key shares rather than content.
- **`stego_media`** — Payloads in the least-significant bits of RGB image parts such as `word/media/imageN.png`. The carrier is a figure the document actually displays, so the channel offers concealment on top of confidentiality — with the caveat, quantified in sec. 2.4, that concealment is detectable.
- **`mce`** — Markup Compatibility and Extensibility (ISO/IEC 29500-3) `<mc:AlternateContent>` blocks inside `word/document.xml`. Payloads sit in `<mc:Choice>` guarded by an ignorable namespace, above an *empty* `<mc:Fallback/>`: a fallback carrying visible markup would add a paragraph per concealed module, which is precisely the surface change Theorem 1 forbids.

One rule governs discovery: the manifest is authoritative. Readers resolve modules through manifest declarations, never by walking ZIP entries and guessing, which is what keeps an attacker from introducing a part the reader will treat as intelligence.

## 1.4 The OpenDocument Sibling

Nothing in the design depends on OOXML specifically, and the ODT profile (OASIS ODF v1.3/v1.4 Part 2) (Open Document Format for Office Applications (OpenDocument) Version 1.3 2021) demonstrates that. An ODT package places the `mimetype` entry first and stored uncompressed so consumers can identify the format positionally, registers its parts in `META-INF/manifest.xml`, and carries the same signed intelligence layer as the OOXML profile: the manifest of Definition 2, every sealing lineage, the Merkle root over the module set, a surface digest over the visible ODF content, and the same co-signature policy.

That parity is a property of the *code*, not a parallel implementation that resembles it. Sealing and unsealing are shared between the two profiles rather than written twice, because a second implementation is free to drift on exactly the details that carry the security: the chaff frame that makes a decoy indistinguishable, the slot bound as

AAD, the verifiable-share requirement recorded in the signed manifest. Two profiles that agree only by inspection eventually disagree under attack.

Untrusted intake is scanned on both sides too, though not by the same code: ODF's threat surface is Basic and Scripts containers and off-package `xlink:href` targets, not VBA and `altChunk`. Running the OOXML scan against an ODF package would pass vacuously, which is worse than not scanning, because it would report clean.

Two channels do not cross over, and saying which is part of the claim. ODF has no custom XML datastore part and no Markup Compatibility element, so `custom_xml` and `mce` are OOXML-only; ODT payloads ride as ODF package entries, the unbounded channel. Analogues for the metadata and media channels are plausible and not yet built.

The sibling profile is a second front door into the same container, which is why it enforces the same intake ceilings as the OPC reader — entry count, decompression ratio, and rejection of traversal or absolute entry names. A second entrance that is easier to force is not a portability feature; it is the weakest link, and sec. 3.6 records the cycles in which this one was found wanting, twice.

# 2 Implementation

## 2.1 The Package Layer and Its Determinism

The container engine (`opc.py`) implements the Open Packaging Conventions directly over the standard library's ZIP facilities: part indexing, extension-based `Default` and part-specific `Override` content types, and root-anchored relationship graphs. Serialisation is deterministic so that a build is a function of its inputs rather than of the machine that ran it. Every archive entry carries the fixed DOS timestamp 1980-01-01, `[Content_Types].xml` is written first, and the remaining parts follow in lexicographic order. Packaging invariants are enforced in both directions: duplicate ZIP entries and colliding part paths are rejected on read and on write, content types are validated, and XML is parsed with entity expansion disabled.

`wordml.py` synthesises the minimal conforming WordprocessingML surface document. `odt.py` builds the OASIS OpenDocument sibling, placing `mimetype` uncompressed as the first entry, generating `META-INF/manifest.xml`, and applying the same entry-count, decompression-ratio, and path-traversal guards the OPC reader applies.

## 2.2 Sealing and Key Derivation

**Definition 4 (DXE1 Symmetric Sealing Envelope).** A `DXE1` envelope encrypts payload $P$ under symmetric key $K = \mathrm{KDF}(\mathrm{passphrase}, \mathrm{salt})$ using AES-256-GCM. The module slot name $s$ is bound as Additional Authenticated Data (AAD):

$$C, T = \text{AES-GCM-Encrypt}_K(\mathrm{IV}, P, \mathrm{AAD} = s)$$

The manifest stores $\mathrm{Blake2b}(C \parallel T)$ rather than a plaintext digest.

**Proposition 1 (Slot-Splicing Invariance).** By Definition 4, a ciphertext $C$ generated for slot $s_i$ cannot be spliced into slot $s_j$ $(i \neq j)$ without causing the GCM authentication tag verification to fail closed.

Two choices in Definition 4 are worth drawing out. Binding the slot as AAD gives Proposition 1 for free: a ciphertext is cryptographically fixed to the position it was sealed for. Digesting the *ciphertext* rather than the plaintext removes an offline confirmation oracle — a plaintext digest would let anyone holding the file test guesses against a short secret without ever attempting decryption. The digest is checkable without any credential, and is checked before decryption is attempted.

Key derivation favours memory-hard functions, which is what raises the cost of offline dictionary attacks on GPU and ASIC hardware:

- **Argon2id** (RFC 9106 recommended profile) (Biryukov et al. 2021) — 64 MiB memory cost, time cost $t = 3$, parallelism $p = 4$.
- **Scrypt** (the default) (Percival and Josefsson 2016) — $N = 32768$ $(2^{15})$, $r = 8$, $p = 1$: memory-hard at negligible interactive cost.
- **PBKDF2-HMAC-SHA512** (compatibility) — 600000 iterations, per OWASP 2023 guidance, for constrained or FIPS-bound environments.

Because the envelope declares its own work factors, those factors are attacker-controlled input, and readers cap them. The ceilings bound *memory*, not merely the iteration parameter: Scrypt's footprint is $128 \cdot N \cdot r$, so a cap on $N$ that leaves $r$ free bounds nothing at all. The reader admits $128 \cdot N \cdot r \leq 256$ MiB, the same ceiling Argon2id already enforced. sec. 3.6 returns to how that gap survived a first hardening pass.

Fig. 2 traces a payload through the whole path. It branches only at the sealing mode, and every branch reconverges on the same stored bytes — which is what the manifest records a digest of. The alternative, digesting the plaintext, would ship an offline oracle against the passphrase inside the file the attacker already has. Every parameter in the figure is read from the live constants rather than transcribed.

The recipients lineage keeps identities out of the envelope and, by default, does not keep their number out: one wrapped content key per recipient, plus an explicit count field, so the total is readable from the envelope's length. The manifest records neither identities nor number, which makes the omission an inconsistency rather than a policy — for the blind-review packet this lineage is meant to serve, "sealed to three people" is itself information about the review. Padding raises the slot count to a fixed bucket by wrapping to freshly generated public keys whose private halves are discarded before the call returns, so a padded slot is a real wrap that nobody can open, exactly as the chaff frame is. It is opt-in, and the unpadded default therefore still leaks the count.

Beyond symmetric sealing, `crypto.py` provides X25519 multi-recipient key encapsulation (`DXE2`) (Langley et al. 2016), wrapping one content key separately for each recipient, alongside Ed25519 signing (Josefsson and Liusvaara 2017). `shamir.py` implements $(k, n)$ threshold sharing over GF(256) (Shamir 1979) with verifiable-share commitment tags (Feldman 1987). A commitment binds only if the verifier insists on it: an adversary can strip the tag and present the same bytes as a legacy-format share, so reconstruction must be told to require the verifiable form. Threshold modules therefore record that requirement in the signed manifest, where it cannot be downgraded without breaking the signature. Decoy modules carry two independent frames, the second indistinguishable from chaff. Indistinguishability has a dynamic half that the static one does not imply: frames were once tried in order and the first success returned, so opening with the real password cost one key derivation and opening with the cover story cost two — a factor of two on a deliberately expensive function, and therefore an answer, on the wall clock, to the one question the lineage exists to refuse. Every frame is now attempted whichever one matches, which costs a derivation per frame and buys the property the format claims. Signatures are computed over a canonical, whitespace-invariant JSON body.

## 2.3 Binding the Payloads to the Prose

**Definition 5 (Composite Surface Digest).** Let $\mathcal{P}$ be a package and let $\mathcal{M}$ be its intelligence manifest part. The **composite surface digest** $D_{\text{surface}}$ is the Blake2b digest of three sorted families, each tagged so a part, a content type, and a relationship edge cannot collide: every part and its bytes except $\mathcal{M}$; the content-type map (defaults and overrides) except the override for $\mathcal{M}$; and every relationship edge except the one that points at $\mathcal{M}$. The manifest is the sole exclusion because it carries this digest and cannot contain itself.

**Theorem 2 (Package-Graph Integrity).** By Definition 5, any alteration to a rendered part, a content type, or a relationship edge — including a swap of the officeDocument target — modifies $D_{\text{surface}}$ and invalidates the Ed25519 manifest signature.

`payloads.py` keeps an extensible registry of payload encoders (`bytes`, `text`, `json`, `project`, `docxplus`). The `project` handler packs an entire directory tree into a deterministic, traversal-guarded, size-capped tarball, which is what lets a document carry the software that produced it.

`provenance.py` builds a Merkle tree (Merkle 1987) over the module digests, giving logarithmic inclusion proofs: a third party can confirm that one module belongs to the signed set without being shown the others. The signature covers that root together with the composite surface digest of Definition 5, so tampering with a header, a style, a font table, or the officeDocument relationship breaks provenance exactly as tampering with a payload does. Theorem 2 states the property. An earlier version hashed a list of story-part *names*; sec. 3.6 records why a naming convention cannot carry it. Authenticity, throughout, requires the caller to pin `expected_public_key`; a key that travels inside the manifest is an identity claim, not a verified one.

`transparency.py` maintains an append-only log of reproduction attestations, following the tamper-evident-logging construction of Crosby and Wallach (Crosby and Wallach 2009) and the signed-tree-head discipline that Certificate Transparency established for the same problem in the WebPKI (Laurie et al. 2013). Hash chaining as a tamper-evidence primitive dates to Lamport's one-way password chains (Lamport 1981); the lesson the later

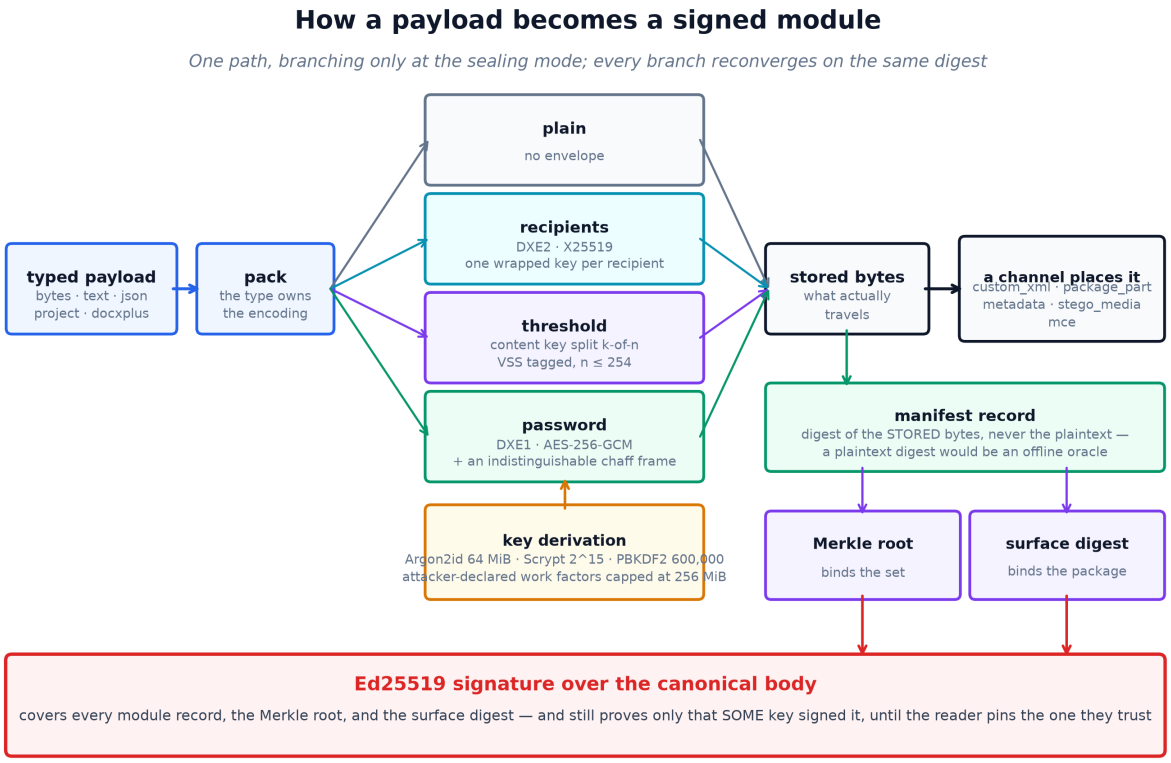


**Figure 2.** How a typed payload becomes a signed module. The path branches only at the sealing mode, and all 4 branches reconverge on the same stored bytes — which is what the manifest digests. Digesting the plaintext instead would leave an offline oracle against the passphrase in a file the attacker already holds. Key-derivation profiles and the ceilings imposed on attacker-declared work factors are read from the live constants rather than transcribed.

work adds is that a chain must be *anchored* to mean anything. Chain verification establishes only self-consistency, which is weaker than it sounds: an adversary who rewrites the log from its first entry produces an equally self-consistent chain, and because no entry references the tip, the final entry's body can be edited in place without breaking any linkage. Authenticity therefore rests on a **signed tree head** — an Ed25519 signature, domain-separated from the manifest signature so it cannot be replayed as one, over the pair (log length, Merkle root). Committing to the length as well as the root defeats truncation replay, since an earlier head cannot describe a longer log. The `verify-transparency` command checks the chain, the tree head under a caller-pinned signer, a pinned root, and per-entry inclusion proofs, failing closed on each and reporting a log offered without a tree head as explicitly unauthenticated.

Determinism is a precondition rather than a nicety here: an attestation over a build that varies between runs attests nothing, which is the argument the Reproducible Builds project has made for toolchains generally (Reproducible Builds Project 2024). `reproduce.py` is the only path that executes carried code, and it does so solely on an explicit opt-in, inside a scrubbed sandbox: dynamic-linker injection variables purged, resource and file-size limits clamped, wall-clock timeout enforced, network denied, and writes confined to the project and temporary directories where the platform supports it.

## 2.4 Concealment, and Measuring It

`lsb.py` provides a pure-Python least-significant-bit codec. Payloads are framed under a `DXL1` magic header and written across the RGB channels of PNG image parts. `steg_bridge.py` integrates the optional `docxology/steganographer` Rust backend, which adds BLAKE3 hashing, Ed25519 payload signatures, Reed-Solomon error correction, and its own `analyze` command.

That backend is optional, and a security property that depends on an optional dependency is not a property.

`steg_bridge.py` therefore implements the chi-squared attack on LSB replacement (Westfeld and Pfitzmann 2000) directly, requiring nothing beyond the imaging library. Embedding equalises the frequencies within each pairs-of-values bin, so a *low* statistic is the evidence of embedding rather than the usual reverse; the upper-tail probability comes from a regularised incomplete gamma function evaluated in-tree rather than through a numerical dependency.

That equalisation requires the embedded bits to be *uniform*, which is the attack's necessary condition and therefore its boundary. Sealed modules are ciphertext, so the default path is the maximally detectable one: a fully embedded carrier reports $p \approx 1$. An **unsealed low-entropy payload is not detected at any fill rate** — plaintext, a constant fill, or a structured record leaves the pairs-of-values asymmetry intact and reports $p \approx 0$, indistinguishable from a clean carrier. A clean verdict is evidence about this attack, never evidence that a carrier is unmodified. Two further wrinkles matter in practice: because the codec fills carriers sequentially, a partially-filled carrier is invisible to whole-image analysis, since the untouched remainder dominates the histogram. The detector answers this by sweeping increasing prefixes of the sample stream, and the prefix at which the statistic collapses both localises the payload and estimates its extent, though only coarsely: the estimate is quantised to the sweep's step size and biased upward. A `redundancy=N` mode replicates a payload across $N$ carriers so the document survives losing all but one.

`mce.py` implements the Markup Compatibility and Extensibility channel (ISO/IEC 29500-3) (Information Technology – Document Description and Processing Languages – Office Open XML File Formats (Parts 1–4) 2021), wrapping payloads in `<mc:Choice>` elements under an ignorable namespace in `word/document.xml`. A compliant application that does not recognise the namespace discards the Choice branch and renders the fallback, with no warning and no error. That fallback is deliberately empty, so a concealed module leaves the paragraph count unchanged; an earlier version emitted a blank `<w:p>` and thereby falsified the independence it was supposed to demonstrate. Placement is equally load-bearing: `CT_Body` is `(EG_BlockLevelElts*, sectPr?)`, so the block is inserted *before* the body-level `<w:sectPr>` rather than appended at `</w:body>`.

## 2.5 Composition, Validation, and Untrusted Input

`container.py` runs the document lifecycle from both ends. `DocxPlusBuilder` queues typed modules, applies per-module sealing, computes the surface digest and Merkle root, and serialises deterministically; `DocxPlusReader` recovers the manifest, extracts payloads, and verifies provenance, co-signers, and attestations.

`odt_container.py` provides `OdtPlusBuilder` and `OdtPlusReader`, the same lifecycle over an ODF package. It reuses `container`'s sealing step and unsealing path verbatim; only placement differs, since ODF locates parts through `META-INF/manifest.xml` rather than a relationship graph.

`validate.py` audits both contracts for both containers. For OOXML: OPC structure (index parts, relationship reachability, absence of ZIP collisions) and intelligence structure (Merkle root consistency, per-module ciphertext digests, signature verification, and recomputation of the composite surface digest). That last check is recent and its absence was a defect rather than an omission: the signature covers the digest as a *stored field*, so a package whose visible prose had been rewritten still had a self-consistent signature and passed validation with no findings at all. Only `verify` caught it, and `validate` is the command a release process runs. For ODF: the positional `mimetype` rule, manifest completeness — an undeclared entry is unreachable to a conforming consumer, the ODF analogue of OPC reachability — and the same intelligence checks.

One validator rule exists for a feature that does not: a package carrying whole-package OPC signatures is rejected unless their combined reference set covers the intelligence manifest and every part it names. A signature enumerating only the conventional Word parts would render as valid in a desktop office suite over a document whose intelligence layer had been stripped, so the trust indicator would be attesting the absence of what a reader assumes it covers. Writing the rule before the signing code is deliberate; it is the invariant that keeps the feature from becoming a way to launder missing payloads (sec. 4.3).

`intake.py` is the hardened gateway for files of unknown origin, reporting external relationship targets, macro parts, and foreign `altChunk` imports statically, executing nothing. The `docxplus` CLI is a thin orchestration layer over these modules and implements no logic of its own; it currently exposes 19 commands: analyze-carrier, build, extract, graph, inspect, keygen, odt-build, odt-extract, odt-inspect, odt-scan, odt-validate, reproduce, scan, transparency-append, unpack-project, validate, verify, verify-reproduction, verify-transparency.

# 3 Evaluation

Every quantity in this section is substituted at render time from live code constants via `scripts/render_manuscript.py` and `src/docxplus/manuscript_vars.py`. The manuscript sources contain no hardcoded metrics, so a claim here cannot drift from the implementation without the render failing. The figures draw from the same source.

## 3.1 Round-Trip Integrity Across Every Sealing Mode

The reference dossier (`src/docxplus/reference_docs.py`, `scripts/04_dossier.py`) carries 5 heterogeneous modules in a single archive, covering all 4 sealing lineages (`password`, `plain`, `recipients`, `threshold`) across 2 of the 5 transport channels (`custom_xml`, `package_part`). Sealing is what the dossier is for; the remaining channels are exercised by the round-trip and channel test suites rather than here, and the table below is generated from the document the code actually produces:

| Slot | Channel | Sealing |
|---|---|---|
| `annex` | package_part | password |
| `brief` | custom_xml | plain |
| `notes` | package_part | password |
| `review` | package_part | recipients |
| `vault` | package_part | threshold |

All 5 modules extract and verify:

- **Structure.** The container passes OPC conformance (`opc_valid = True`): no part collisions, no broken relationship pointers.
- **Provenance.** The Ed25519 manifest signature, the module Merkle tree, and the composite surface digest over the package graph all validate.
- **Sealing.** Argon2id and Scrypt password modules decrypt; X25519 multi-recipient envelopes open for each authorised key; Shamir $(k, n)$ shares reconstruct on reaching quorum and are refused when a share's verifiable tag is absent or wrong; dual-frame decoy modules return their respective plaintexts under their respective passphrases.
- **Reproducibility.** Carried `project` tarballs unpack without path traversal, and reproduction attestations verify cryptographically.

The harness comprises 467 test functions under a 90% coverage gate, with no mocks anywhere: tests run against real cryptographic primitives, real ZIP archives, real subprocess CLI invocations, and the compiled Rust steganography engine when it is present.

## 3.2 A Manuscript That Carries Its Own Repository

`scripts/05_living_manuscript.py` packs the complete, runnable docxplus repository — source and test suite — into a `.docx` carrying a signed reproduction attestation. That `.docx` is a sibling of the PDF you are reading, not this file; the evaluation below describes the carried archive, not the rendered paper.

A recipient can engage at three levels of increasing commitment. The last two are the branches that fork in fig. 3:

1. **Read it.** Open the `.docx` in Word, LibreOffice Writer, or Google Docs. Nothing about the intelligence layer intrudes.
2. **Verify it, executing nothing.** `docxplus verify-reproduction` checks the Ed25519 signature and the digest chain over the attested recipe. This is the default trust path, and it treats the document as inert data throughout.
3. **Re-run it.** `docxplus reproduce` opts in to instantiating the project in a confined sandbox, executing the attested command, and comparing the resulting digests against the author's.

The third tier is the interesting one precisely because it is optional. A document that reproduced itself on open would be malware; the value lies in the reader choosing, on their own hardware, when to spend that trust. What

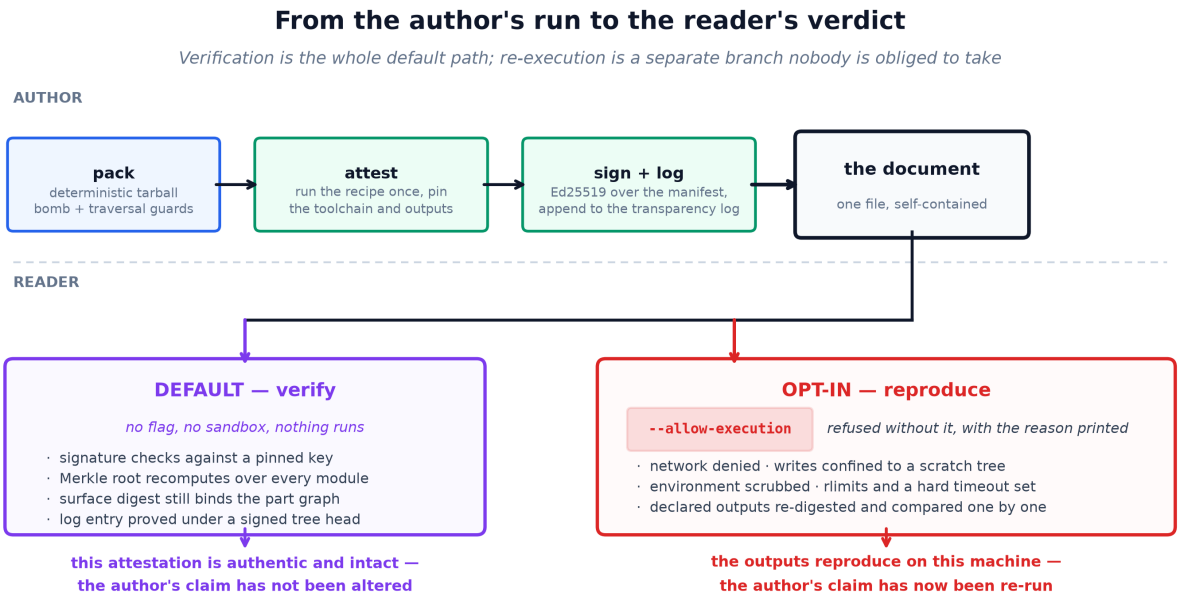


**Figure 3.** Authoring runs left to right and ends at one self-contained file; what a reader does with that file forks. The default branch is entirely cryptographic and executes nothing. The other requires an explicit flag and re-runs the attested command under confinement. The fork exists because the two verdicts are not interchangeable: one says the author's claim has not been altered, the other says it reproduces on this machine, and neither substitutes for the other.

a match proves is bounded and worth stating: the declared command produced the sealed outputs on a matching toolchain. It says nothing about whether the method was sound.

## 3.3 Carrying a Project Out and Back

The claim that a document can carry the software that produced it is only worth as much as the fidelity of the round trip, so the fidelity is measured rather than asserted. `scripts/06_project_roundtrip.py` builds a project tree chosen to contain exactly the cases naive packing loses — an executable entrypoint, an empty directory, a zero-byte file, filenames with spaces and non-ASCII characters, two files with identical content, and build junk that must *not* travel — carries it into both containers, and compares what comes back file by file, byte by byte, and mode by mode.

Over a 14-file, 9-directory tree, all 18 of 18 invariants hold. Both profiles validate against their own conformance rules; both verify provenance under a pinned key and refuse a wrong one; both extract 12 files identical to the originals; both carry and cryptographically verify a reproduction attestation. The sealed `.docx` is 4,470 bytes and the `.odt` 3,553 bytes, and the packed project payload is *byte-identical between them* — parity that holds because the two profiles share the packing code rather than agreeing by inspection.

Fig. 4 states the contract in full: what is preserved, what is normalised away on purpose, what is refused outright, and what never travels at all.

Two results are worth separating from the pass count. The first is what the harness found on its way to passing: packing previously forced every file to mode `0644`, so a carried `run.sh` came back non-executable, and empty directories vanished under a files-only walk. Both are now preserved. The second is a security finding rather than a fidelity one. `Path.is_file()` follows symbolic links, so a tree containing `creds -> ~/.ssh/id_rsa` had the *key's contents* packed into the document under the link's name, with nothing in the manifest to suggest it. The unpack side had always rejected link members; the pack side was the open door. Symlinks are now refused unless a caller passes `follow_symlinks=True` and thereby says so in a way that survives review.

Fig. 5 sets the two profiles side by side. What matters there is not the count of ticks but the dividing line: above

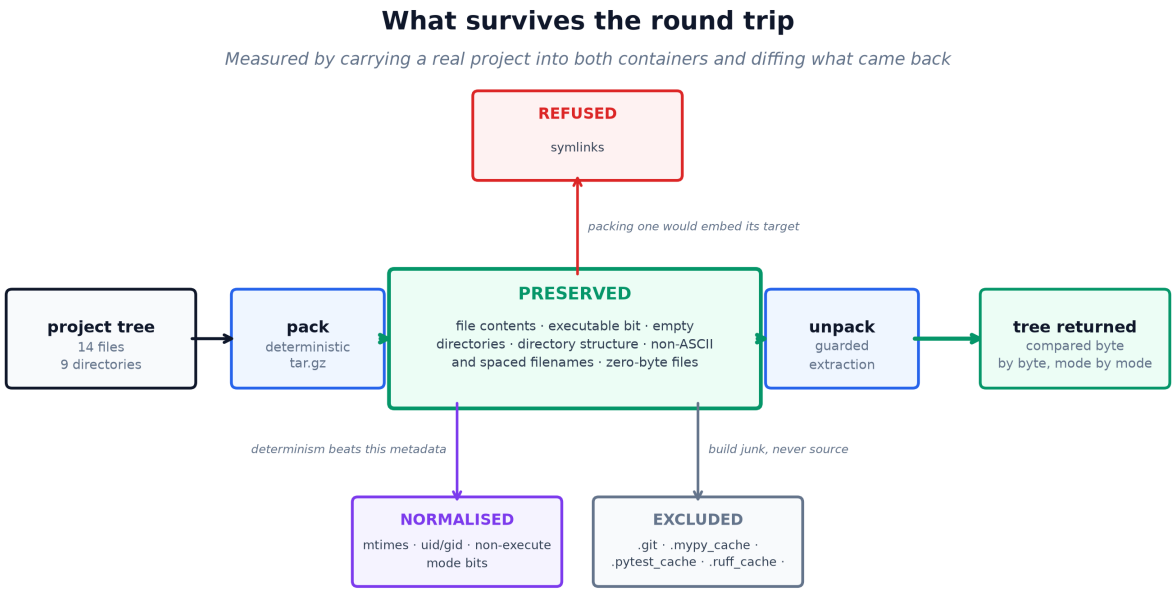


**Figure 4.** A real project tree carried into a container and diffed against what came back. The through-line is what survives byte for byte, including the executable bit — a carried entrypoint that returns non-executable is not carried software. Three diversions leave that line at the point they occur: metadata normalised because determinism is worth more than an mtime, symlinks refused outright because packing one would embed its target, and build directories never packed at all. Counts are read from the harness output at render time, so a fidelity regression changes this figure rather than merely failing a test.

it the profiles execute the *same code*, so parity is structural rather than a coincidence that survives until someone patches one side.

The harness also exercises what a single-container round trip cannot reach: a signed `.docx` carried as a nested module *inside* a signed `.odt`, opened through a dispatcher that reads the container's own magic rather than trusting the caller to declare it, with the inner document's provenance and project both verifying after extraction.

## 3.4 Channel Capacity

Capacity spans four orders of magnitude, and the spread is the design's point rather than an artefact: different channels answer different needs.

Fig. 6 plots the spread on a log scale, distinguishing the channels with a real format ceiling from those that simply have none:

- **`metadata`** is deliberately small, bounded at 8000 bytes in `docProps/custom.xml`: routing headers, identifiers, status flags, key shares.
- **`custom_xml`, `package_part`, and `mce`** carry no format-imposed ceiling, scaling until host storage or the reader's own decompression caps intervene.
- **`stego_media`** scales with carrier resolution under 1-bit LSB encoding: a $256 \times 256$ RGB PNG holds 24568 bytes, a $512 \times 512$ PNG holds 98296. Under `redundancy=N` the payload is replicated across $N$ carriers, trading capacity for survival of carrier loss.

## 3.5 Conformance, Openability, and Determinism

Generated packages comply with ISO/IEC 29500-2 (Information Technology – Document Description and Processing Languages – Office Open XML File Formats (Parts 1–4) 2021). Because protection is applied per payload

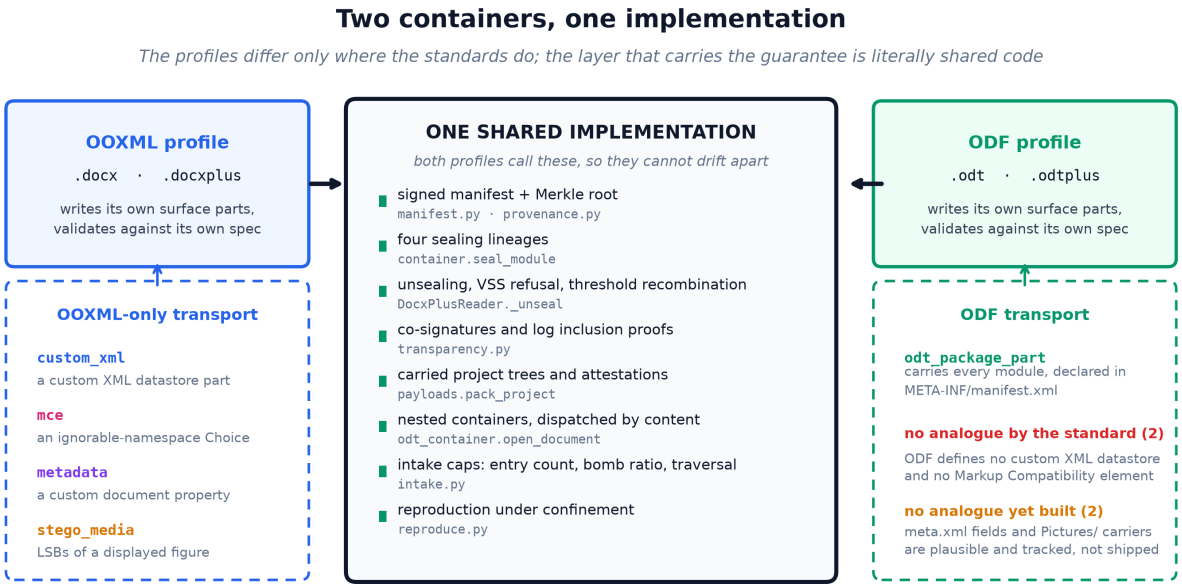


**Figure 5.** The two profiles drawn as shells around a shared core. Every capability in the centre block is one implementation both profiles call, which makes parity a structural property rather than a coincidence that survives until someone patches one side. Divergence is confined to the dashed boxes at the transport edge — two constructs ODF does not define, and two analogues that are plausible but not built — and none of it touches how a payload is sealed, digested, signed, or refused. That is what stops the weaker profile becoming the one an attacker chooses to present.

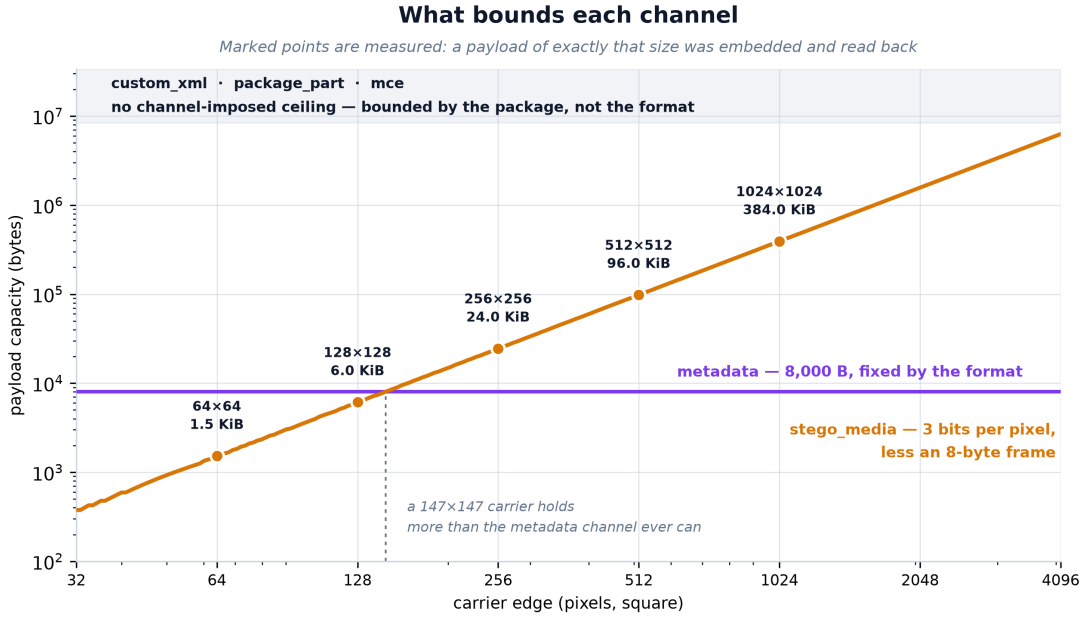


**Figure 6.** What bounds each channel, plotted against the axis on which they actually differ. `metadata` has a hard ceiling fixed by the format and flat in carrier size; `stego_media` scales with carrier area and overtakes that ceiling at roughly 147 pixels a side; the remaining three have no channel-imposed limit at all and are drawn as an open region rather than given an invented number. The marked points are measured rather than computed: a payload of exactly that size was embedded into a real carrier and read back before the point was plotted.

rather than by wrapping the archive in an MS-OFFCRYPTO compound file (Microsoft Corporation 2024), the document stays readable by ordinary tools — the property sec. 1.1 set out to preserve.

Determinism is asserted, not assumed: `test_container.py::test_build_is_deterministic` requires byte-identical digests across repeated unencrypted builds. Encrypted and signed builds introduce fresh salts, IVs, and nonces by design, while part ordering stays fixed. Headless LibreOffice conversions confirm that the resulting files open and render.

## 3.6 Adversarial Verification

Security claims are worth what their falsification attempts are worth, so the format has been through 14 cycles of adversarial review closing 88 confirmed findings. Each cycle decomposed the design into atomic claims, attacked them from independent perspectives, and required a second reviewer to *reproduce* a finding against the running system before it was accepted. Every accepted finding produced both a fix and a regression test that fails without it. The complete record — severities, reproduction steps, and results withdrawn rather than shipped — is maintained in `docs/redteam-audit.md` and is not reproduced here.


**Figure 7.** Threat classes grouped by the layer each one attacks, with the boundary it fails against drawn between the attack and the invariant. Every arrow stops on the wall. None of the classes is hypothetical — each was reproduced against a running build before its control was written — and the grouping shows what a flat list obscured: the coverage spans the surface contract, the manifest, the sealing layer, and the sandbox, rather than concentrating where the code was easiest to harden.

Fig. 7 maps the classes to their controls. Four findings generalise past this implementation, and they are the reason the review is worth reporting at all.

**A naming convention cannot carry a security property.** The manifest signature originally bound a list of part *names*, while OPC resolves the rendered document through the officeDocument *relationship* (Information Technology – Document Description and Processing Languages – Office Open XML File Formats (Parts 1–4) 2021). Every test passed, because each asserted what the code said it did; the defect lay in the space between two individually correct components, the position Dolev and Yao formalised for protocols (Dolev and Yao 1983). What closed it was a change of question — asking of each constraint whether it was a law of the format or a convention this project had chosen — rather than a better test.

**A control is only as strong as the verifier's obligation to invoke it.** Verifiable secret shares (Feldman 1987) carried integrity tags defeatable by stripping the header rather than by cryptanalysis, because nothing obliged the reconstructing party to demand the tagged form. Recording the requirement in the *signed* manifest moves it from the verifier's discretion to the attacker's impossibility. Downgrade resistance has to be structural; TLS reached the same conclusion about version negotiation only after repeated failures (Rescorla 2018).

**Absence of a failing test is not evidence of a working control.** Two findings were silent: a determinism guarantee implemented by assignment to a non-existent attribute, and a capability documented in prose with no implementing code path. Both produced artefacts that looked correct under inspection. Tests defend against the failure modes they were shaped to anticipate, and neither of these was shaped like a failure — the selection problem Goodenough and Gerhart set out at the foundation of testing theory (Goodenough and Gerhart 1975).

**A guarantee delegated to a neighbour is not a guarantee.** The Merkle construction padded odd levels by duplicating the trailing node, which makes the tree over three leaves hash identically to a four-leaf tree whose last two leaves are equal — the second-preimage ambiguity found in Bitcoin as CVE-2012-2459. The documented promise that adding a module always changes the root was therefore false in that case. It was unreachable through a well-formed manifest only because slots are unique, and slots were unique only because a *different* module checked them on the write path; the read path did not. In the same round the validator was found to recompute the Merkle root but not the surface digest, so a package whose visible prose had been rewritten passed `docxplus validate` with no findings at all, protected only by whether the reader happened to run `verify` instead. Both fixes move the check into the component that owns the property: RFC 6962 tree splitting makes the root unambiguous by construction rather than by a caller's diligence, and the validator recomputes what it had been trusting. A property that holds because something else is careful is a property with an undocumented dependency, and undocumented dependencies are what change when code does.

The record also carries a negative result. A sample-pair analysis estimator (Dumitrescu et al. 2003) was implemented and measured against carriers embedded at known rates. It proved monotonic in the true rate but mis-scaled by roughly an order of magnitude and dependent on carrier statistics, so it was withdrawn rather than shipped under a name implying a calibration it did not possess.

# 4 Conclusion

## 4.1 What This Demonstrates

An ordinary Office container can be an authenticated, selectively sealed computational carrier without giving up conformance to the standards that make it ordinary. The mechanism is the separation this paper began with: hold the **surface contract** (OPC/OOXML and ODF conformance) apart from the **intelligence contract** (a signed, modular payload manifest), and the two stop competing. Universal readability and fine-grained cryptographic access control turn out not to be a trade-off, only a design that had not been drawn that way.

Concretely:

1. **Payload-level rather than package-level sealing.** Argon2id (Biryukov et al. 2021), Scrypt (Percival and Josefsson 2016), and AES-256-GCM; X25519 multi-recipient encapsulation (Langley et al. 2016); verifiable threshold sharing (Feldman 1987); decoy chaffing — all applied to modules, leaving the document open.
2. **Provenance over the package graph.** Merkle trees (Merkle 1987) over the module set bound together with every part, content type, and relationship — not merely the story parts — under one signature (Josefsson and Liusvaara 2017).
3. **Reproducibility a reader can check without trusting the author.** Carried project archives, signed reproduction attestations verifiable with zero execution, an anchored transparency log, and re-execution

confined behind an explicit opt-in.

4. **5 spec-sanctioned transport channels**, spanning custom XML, package parts, document properties, MCE choice blocks, and LSB steganography whose detectability is measured in-tree (Westfeld and Pfitzmann 2000) rather than asserted.
5. **Standards parity** across Office Open XML (Information Technology – Document Description and Processing Languages – Office Open XML File Formats (Parts 1–4) 2021) and OASIS OpenDocument Text (Open Document Format for Office Applications (OpenDocument) Version 1.3 2021): the same signed manifest, sealing lineages, and provenance in both containers, implemented by shared code rather than by two implementations that agree today. Parity holds at the intake boundary too, where the sibling profile enforces the same ceilings. Two OOXML channels have no ODF analogue and are named as such.

## 4.2 What This Does Not Do

The boundaries below are load-bearing. Stating them is part of the contribution, since a security property a reader misunderstands is worse than one absent (`docs/security-model.md`).

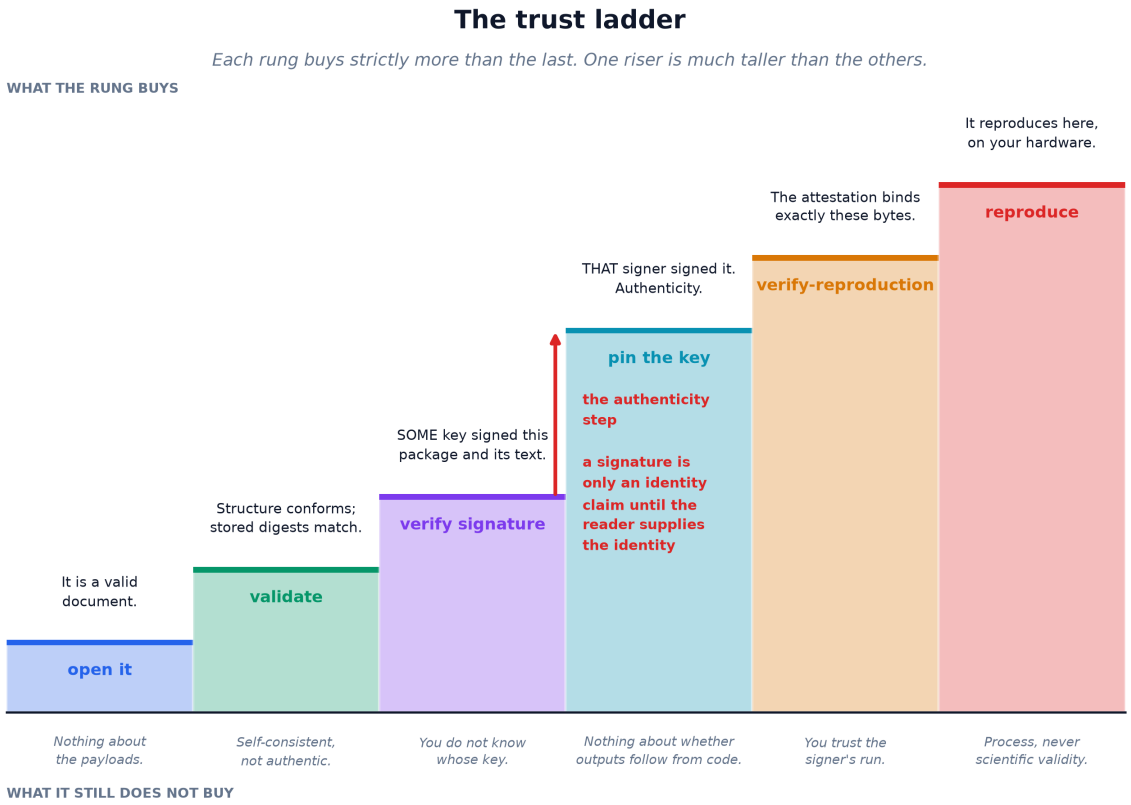


**Figure 8.** What a reader may conclude at each rung of verification, what it costs them, and — the column that usually goes missing — what it still does not buy. The rungs are cumulative and none is skippable. The step from *verify signature* to *pin the key* is the one that carries the weight: until the reader supplies the identity they trust, a valid signature establishes only that the document is internally consistent with itself, which is a property a forger can supply just as easily as an author.

The boundaries below correspond to the rungs in fig. 8; each one names a conclusion the format does *not* license. Stating them precisely is itself a design obligation: cryptographic software fails far more often through misunderstood interfaces and unstated assumptions than through broken primitives (Lazar et al. 2014), and a system whose guarantees a reader over-reads has failed even when every algorithm in it is sound.

- **The surface is public on purpose.** docxplus is not digital rights management. The prose is meant to be readable, and confidentiality reaches only the modules marked for it.
- **Concealment is obfuscation, not secrecy.** LSB embedding is detectable by statistical steganalysis,

which is why the chi-squared detector ships with the tool rather than being left to an adversary. Its own boundary is stated with it: the attack keys on the uniformity a random payload imposes, so it finds sealed modules and misses unsealed low-entropy ones entirely. A clean verdict bounds one attack, not the space of them. Treat a concealed module as hidden from casual inspection, never as undiscoverable.

- **Deniability holds against inspection, not against proof.** A decoy is structurally indistinguishable from an ordinary sealed module — same manifest record, same two frames — and randomly sized chaff keeps size from implying a payload length, so nothing in the file reveals a second payload. But an adversary who knows the scheme knows a second frame is *always* present, and cannot be compelled to accept it as chaff. The property claimed is inspection resistance for a single document, not indistinguishability under a distributional attack.
- **Integrity is not authenticity.** A valid signature proves the manifest was signed by whoever holds the embedded key. Only comparison against a caller-pinned key makes that signer anyone in particular.
- **A reproduction proves process, not science.** A digest match shows the declared command produced the attested output on a matching toolchain. Whether the method was correct is outside what any container can attest.
- **Desktop suites do not verify the intelligence signature.** The signed body now binds the module set and the whole package graph, but Word and LibreOffice will not check it. Native validation is a different signature (OPC XML-DSig), and that is the first item below.

### 4.3 Future Work

The nearest open item is producing a whole-package OPC XML-DSig signature alongside the manifest signature, which would let desktop office suites validate a docxplus document natively. The assessment in `docs/opc-signatures.md` sets out why it is not yet implemented: it needs canonical XML and an X.509 trust story, and Office will not verify Ed25519. The invariant it must satisfy is already enforced. An OPC signature enumerating only the conventional Word parts would display as valid over a package whose intelligence layer had been stripped, which is worse than no signature at all — it would launder the absence of the thing it appeared to attest. The validator therefore already rejects any package whose OPC signature reference set omits a manifest-named part, written before the signing code precisely so the feature cannot ship without it.

Beyond that: ODF analogues of the metadata and media channels, so the two profiles differ only where the standards genuinely do; calibrating sample-pair analysis against natural-image carriers, so the withdrawn estimator of sec. 3.6 can return with the accuracy its name implies; validating produced documents against Office-o-tron and the ODF Toolkit in continuous integration; extending formal verification to the container parsing logic; and selective attribute disclosure via zero-knowledge proofs, for workflows where a reader must confirm a property of a payload without opening it.


Biryukov, Alex, Daniel Dinu, Dmitry Khovratovich, and Simon Josefsson. 2021. *Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications*. No. 9106. Request for Comments. RFC 9106; IETF. https://doi.org/10.17487/RFC9106.

Crosby, Scott A., and Dan S. Wallach. 2009. "Efficient Data Structures for Tamper-Evident Logging." *Proceedings of the 18th USENIX Security Symposium*, 317–34.

Dolev, Danny, and Andrew C. Yao. 1983. "On the Security of Public Key Protocols." *IEEE Transactions on Information Theory* 29 (2): 198–208. https://doi.org/10.1109/TIT.1983.1056650.

Dumitrescu, Sorina, Xiaolin Wu, and Zhe Wang. 2003. "Detection of LSB Steganography via Sample Pair Analysis." *IEEE Transactions on Signal Processing* 51 (7): 1995–2007. https://doi.org/10.1109/TSP.2003.812753.

Feldman, Paul. 1987. "A Practical Scheme for Non-Interactive Verifiable Secret Sharing." *28th Annual Symposium on Foundations of Computer Science (FOCS)*, 427–37. https://doi.org/10.1109/SFCS.1987.4.

Fridrich, Jessica, Miroslav Goljan, and Rui Du. 2001. "Detecting LSB Steganography in Color, and Gray-Scale Images." *IEEE Multimedia* 8 (4): 22–28. https://doi.org/10.1109/93.959097.

Goodenough, John B., and Susan L. Gerhart. 1975. "Toward a Theory of Test Data Selection." *IEEE Transactions on Software Engineering* SE-1 (2): 156–73. https://doi.org/10.1109/TSE.1975.6312836.

Information Technology – Document Description and Processing Languages – Office Open XML File Formats (Parts 1–4), Pub. L. Nos. ISO/IEC 29500:2021 (2021).

Josefsson, Simon, and Ilari Liusvaara. 2017. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. No. 8032. Request for Comments. RFC 8032; IETF. https://doi.org/10.17487/RFC8032.

Lamport, Leslie. 1981. "Password Authentication with Insecure Communication." *Communications of the ACM* 24: 770–72. https://doi.org/10.1145/358790.358797.

Langley, Adam, Mike Hamburg, and Sean Turner. 2016. *Elliptic Curves for Security*. No. 7748. Request for Comments. RFC 7748; IETF. https://doi.org/10.17487/RFC7748.

Laurie, Ben, Adam Langley, and Emilia Kasper. 2013. *Certificate Transparency*. RFC No. 6962. IETF. https://doi.org/10.17487/RFC6962.

Lazar, David, Haogang Chen, Xi Wang, and Nickolai Zeldovich. 2014. "Why Does Cryptographic Software Fail? A Case Study and Open Problems." *Proceedings of the 5th Asia-Pacific Workshop on Systems (APSys)*. https://doi.org/10.1145/2637166.2637237.

Merkle, Ralph C. 1987. "A Digital Signature Based on a Conventional Encryption Function." *Advances in Cryptology – CRYPTO '87*, Lecture notes in computer science, vol. 293: 369–78. https://doi.org/10.1007/3-540-48184-2_32.

Microsoft Corporation. 2024. *[MS-OFFCRYPTO]: Office Document Cryptography Structure*. No. v20240521. Microsoft Corporation.

Open Document Format for Office Applications (OpenDocument) Version 1.3 (2021).

Peng, Roger D. 2011. "Reproducible Research in Computational Science." *Science* 334 (6060): 1226–27. https://doi.org/10.1126/science.1213847.

Percival, Colin, and Simon Josefsson. 2016. *The Scrypt Password-Based Key Derivation Function*. No. 7914. Request for Comments. RFC 7914; IETF. https://doi.org/10.17487/RFC7914.

Reproducible Builds Project. 2024. *Reproducible Builds: A Set of Software Development Practices for Independently Verifiable Paths from Source to Binary Code*. Https://reproducible-builds.org/.

Rescorla, Eric. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC No. 8446. IETF. https://doi.org/10.17487/RFC8446.

Sandve, Geir Kjetil, Anton Nekrutenko, James Taylor, and Eivind Hovig. 2013. "Ten Simple Rules for Reproducible Computational Research." *PLOS Computational Biology* 9 (10): e1003285. https://doi.org/10.1371/journal.pcbi.1003285.

Shamir, Adi. 1979. "How to Share a Secret." *Communications of the ACM* 22 (11): 612–13. https://doi.org/10.1145/359168.359176.

Westfeld, Andreas, and Andreas Pfitzmann. 2000. "Attacks on Steganographic Systems: Breaking the Stegovan Camouflage." *Information Hiding*, Lecture notes in computer science, vol. 1768: 61–76. https://doi.org/10.1007/10719724_5.

Wilkinson, Mark D. et al. 2016. "The FAIR Guiding Principles for Scientific Data Management and Stewardship." *Scientific Data* 3: 160018. https://doi.org/10.1038/sdata.2016.18.