IMF: Immutable File Container

A Cryptographically Sealed Archive Format for
Tamper-Proof File Storage and Distribution

Technical Whitepaper — Version 1.0
Published: February 11, 2026

Benjamin Toso
benjamin.toso@gmail.com

Abstract

This paper presents IMF (Immutable File), a container format and reference implementation for creating cryptographically sealed, tamper-evident file archives. IMF combines ZIP-based storage with Ed25519 digital signatures, AES-256-GCM authenticated encryption, and a manifest-driven integrity model to provide strong guarantees of data immutability after sealing. The format supports optional features including embedded public key material for self-verifying containers, passphrase-based encryption via PBKDF2-HMAC-SHA256, and time-based expiration policies. The reference implementation is written in Go with zero external dependencies, producing a single cross-platform binary. This paper describes the problem domain, the container format specification, the cryptographic design, the lifecycle state machine, and potential applications in legal evidence packaging, regulatory compliance, software supply chain integrity, and secure document distribution.

Table of Contents

1. Introduction and Motivation

The need to ensure that digital files have not been modified after creation is a fundamental requirement across numerous domains: legal proceedings require chain-of-custody guarantees, regulatory frameworks demand audit-proof record keeping, software distribution requires integrity verification, and secure communication depends on tamper-evident packaging.

Existing solutions to this problem are either overly complex (requiring PKI infrastructure, certificate authorities, or blockchain systems), tightly coupled to specific ecosystems (such as Java JAR signing or Apple code signing), or lack critical features like integrated encryption and time-based access control.

IMF (Immutable File) addresses these limitations by providing a self-contained, portable file container format that transitions through a simple lifecycle: files are added to an open container, the container is cryptographically sealed, and once sealed, the contents become immutable. The sealed container carries everything needed for verification: content hashes, a digital signature, and optionally the public key itself, making the container self-verifiable without external infrastructure.

This paper describes version 1 of the IMF specification and its Go-based reference implementation. The design prioritizes simplicity, portability, and strong security defaults while remaining extensible for future requirements.

2. Problem Statement

Consider the following scenario: an organization needs to package a set of files (documents, images, data files) and deliver them to a third party with the guarantee that the contents have not been modified since packaging, that the package was created by a known entity, and that the contents are encrypted for confidentiality during transit and storage. Additionally, the package should expire after a defined period, preventing access to outdated or superseded information.

Current approaches require assembling multiple tools: a compression utility (ZIP, tar), a signing tool (GPG, openssl), an encryption layer (GPG, age), and an external metadata system for tracking expiration. Each tool introduces its own key management, configuration, and verification workflow. The result is a fragile, multi-step process prone to human error and difficult to automate consistently.

IMF solves this by combining all four capabilities — archival, signing, encryption, and policy enforcement — into a single container format with a unified workflow and a single binary tool.

3. Related Work and Prior Art

Several existing technologies address subsets of the problem IMF solves. Understanding their strengths and limitations motivates the IMF design.

Technology	Signing	Encryption	Self-Verifying	Expiry	Portable
PGP/GPG	Yes	Yes	No (keyring)	No	Moderate
JAR Signing	Yes	No	No (CA)	No	JVM only
Docker Content Trust	Yes	No	No (Notary)	No	Docker only
age encryption	No	Yes	N/A	No	Yes
JWT/JWS	Yes	Yes (JWE)	Partial	Yes	Data only
IMF	Yes	Yes	Yes	Yes	Yes

IMF distinguishes itself by combining all five properties in a single, portable format that requires no external infrastructure for verification. The self-verifying property is particularly novel: by optionally embedding the public key within the sealed container, recipients can verify integrity without any prior key exchange or access to a key server.

4. Container Format Specification

4.1 Physical Format

An IMF container (.imf) is a standard ZIP archive containing a defined set of entries. The choice of ZIP as the physical format provides several advantages: universal tool support for inspection, well-understood compression, and a mature specification (APPNOTE). The cryptographic layer operates above the ZIP format, treating ZIP entries as opaque byte sequences.

4.2 Container Structure

A sealed IMF container contains the following entries:

Entry Path	Required	Description
`manifest.json`	Yes	Container metadata, file list, hashes, signature
`files/*`	Yes	Stored files (plaintext or .enc encrypted)
`keyring/public.key`	No	Embedded Ed25519 public key (PEM)
`.sealed`	Yes*	Seal marker (*present only when sealed)

4.3 Manifest Schema

The manifest is a JSON document that serves as the single source of truth for the container's state, contents, and cryptographic bindings. It contains:

- **version** (integer): Schema version for forward compatibility.
- **state** (string): Either "open" or "sealed".
- **created_at** (RFC3339): Container creation timestamp.
- **sealed_at** (RFC3339, optional): Timestamp when the container was sealed.
- **expires_at** (RFC3339, optional): Expiration time after which extraction is refused.
- **public_key** (base64, optional): Embedded Ed25519 public key for self-verification.
- **encryption** (object, optional): Algorithm, KDF, salt, and iteration count.
- **files** (array): Per-file entries with path, original name, size, SHA-256 hash, and optional encrypted hash.
- **signature** (base64): Ed25519 signature over the manifest (with signature field zeroed).

5. Cryptographic Design

5.1 Integrity: SHA-256 Content Hashing

Every file added to the container is hashed with SHA-256. The hash is stored in the manifest and verified during extraction. This provides per-file tamper detection: if any byte of any file is modified, the hash will not match and extraction will fail with an explicit integrity error. When encryption is enabled, the encrypted ciphertext is also hashed, providing a second layer of integrity verification before decryption is attempted.

5.2 Authenticity: Ed25519 Digital Signatures

The manifest is signed with Ed25519, a modern elliptic curve signature scheme offering 128-bit security with compact 64-byte signatures and 32-byte keys. Ed25519 was chosen over RSA and ECDSA for its resistance to timing side-channels, deterministic signature generation (eliminating nonce-reuse vulnerabilities that have historically compromised ECDSA implementations), and excellent performance.

The signing process operates on the "signable bytes" of the manifest: the complete JSON serialization with the signature field set to an empty string. This ensures the signature covers all metadata including file hashes, timestamps, and expiration policy. Any modification to the manifest — adding files, changing hashes, altering expiry — will invalidate the signature.

5.3 Confidentiality: AES-256-GCM Encryption

When a passphrase is provided during sealing, each file is individually encrypted with AES-256-GCM (Galois/Counter Mode). GCM provides both confidentiality and authenticity in a single operation: if ciphertext is modified, decryption will fail due to the authentication tag mismatch, providing an additional tamper detection layer beyond the manifest hashes.

Each file receives a unique randomly-generated 12-byte nonce, prepended to the ciphertext. The encryption key is derived from the user's passphrase using PBKDF2-HMAC-SHA256 with 600,000 iterations and a 32-byte random salt, following OWASP 2023 recommendations for passphrase-based key derivation. The salt is stored in the manifest.

5.4 Self-Verification via Embedded Keys

A distinguishing feature of IMF is the ability to embed the signer's public key within the sealed container. This enables self-verification: the recipient can verify the container's integrity and authenticity without any prior key exchange, key server lookup, or certificate authority. The embedded key is protected by the signature itself — modifying the embedded key would invalidate the signature over the manifest that contains the key's base64 encoding.

For higher-assurance scenarios, the embedded key model does not preclude out-of-band key verification. Users can supply an external public key to the verify command, overriding the embedded key. This supports a trust-on-first-use (TOFU) model where the embedded key bootstraps verification and can be independently confirmed.

6. Lifecycle State Machine

The IMF container follows a strictly linear, irreversible lifecycle:

State	Allowed Operations	Transitions To
Open	Create, Add files, List, Info	Sealed (via Seal)
Sealed	Verify, Extract, List, Info	(terminal state)

The sealed state is terminal and irreversible. Once sealed, the container rejects all modification operations: adding files, removing files, re-sealing, or altering metadata. This is enforced at the API level (the library refuses the operation) and at the format level (the .sealed marker and cryptographic signature make tampering detectable).

The seal operation performs the following atomic sequence: (1) encrypt files if a passphrase is provided, (2) set expiration if specified, (3) embed the public key if requested, (4) transition the manifest state to sealed, (5) compute the manifest signature, (6) write the sealed marker, and (7) rewrite the container as a new ZIP archive. This ensures the container is either fully sealed or unchanged — there is no partially-sealed state.

7. Security Properties and Threat Model

7.1 Security Properties

- **Tamper detection**: Any modification to file contents is detected via SHA-256 hash mismatch. Any modification to metadata is detected via Ed25519 signature failure. Any modification to encrypted content is detected via GCM authentication tag failure.
- **Authenticity**: The Ed25519 signature binds the container contents to the signer's private key. Only the holder of the private key could have produced a valid signature.
- **Confidentiality**: When encrypted, file contents are protected with AES-256-GCM. The encryption key is derived from a passphrase using a computationally expensive KDF, resisting offline brute-force attacks.
- **Expiration enforcement**: The signed expiry timestamp in the manifest prevents extraction after the specified time. Since the expiry is covered by the signature, it cannot be modified without invalidating the container.
- **Forward compatibility**: The manifest version field allows future specification versions to be identified and handled appropriately.

7.2 Threat Model

IMF protects against the following threats: unauthorized modification of container contents after sealing (detected by hash and signature verification), unauthorized creation of containers purporting to be from a specific signer (requires the private key), unauthorized access to encrypted contents (requires the passphrase), and use of expired containers (enforced by signed timestamp). IMF does not protect against: denial of service (the container can be deleted), key compromise (standard key management practices apply), or side-channel attacks on the implementation (though Ed25519's constant-time design mitigates timing attacks).

8. Reference Implementation

The reference implementation is written in Go (golang) with zero external dependencies beyond the Go standard library. This design choice ensures the implementation is auditable, reproducible, and free from supply-chain risks introduced by third-party modules.

8.1 Architecture

Package	Responsibility
`pkg/crypto`	Ed25519 keygen/signing, AES-256-GCM encrypt/decrypt, PBKDF2 KDF, PEM encoding
`pkg/manifest`	Manifest schema, state machine, JSON serialization, signable bytes

`pkg/container`	Core API: Create, Add, Seal, Extract, Verify, List, Info
`cmd/imf`	CLI binary with 8 subcommands

8.2 CLI Interface

The CLI provides eight commands that map directly to the container lifecycle: `keygen` generates Ed25519 key pairs, `create` initializes an empty container, `add` inserts files, `seal` cryptographically locks the container, `verify` checks integrity and authenticity, `extract` retrieves files with integrity verification, `list` shows contained files, and `info` displays container metadata.

9. Applications and User Stories

The following user stories illustrate how IMF addresses real-world needs across multiple domains. Each story describes a concrete scenario, the actors involved, the workflow using IMF, and the specific properties that make IMF suitable.

9.1 Legal: Digital Evidence Preservation

Scenario: A corporate litigation team is conducting e-discovery for a fraud case. A forensic analyst collects emails, financial spreadsheets, and database exports from a server that will be decommissioned. The evidence must be preserved in a form that is independently verifiable in court and demonstrably unaltered since collection.

Workflow: The analyst creates an IMF container on-site, adds all collected files, and seals it with the firm's signing key and a 5-year expiration. The embedded public key allows opposing counsel and the court to verify integrity without requiring access to the firm's key infrastructure. The sealed container is stored alongside a hash receipt in the case management system.

Key IMF properties used: Cryptographic integrity (SHA-256 per file), digital signature (Ed25519 for chain of custody), self-verification (embedded public key for independent verification by opposing counsel), and timestamp (sealed_at proves when evidence was collected).

9.2 Medical: Patient Record Transfer

Scenario: A patient is transferring from one hospital system to another. Their records — imaging files (DICOM), lab results (HL7/FHIR exports), physician notes, and medication history — must be transferred securely. HIPAA requires that protected health information (PHI) be encrypted in transit and at rest, and the receiving institution needs assurance that records have not been altered during transfer.

Workflow: The sending hospital's EHR system exports the patient's records into an IMF container, encrypts them with a shared passphrase communicated via a secure channel, and seals the container with the hospital's institutional signing key. The container is set to expire in 30 days (after which the receiving hospital should have imported the records into their own system). The receiving hospital verifies the signature against the sending hospital's known public key, decrypts, and confirms all file hashes match before importing into their EHR.

Key IMF properties used: AES-256-GCM encryption (HIPAA compliance for PHI), integrity verification (ensures records were not altered in transit), expiration (limits window of exposure for sensitive medical data), and signature (authenticates the sending institution).

9.3 Personal: Estate Planning and Digital Legacy

Scenario: An individual wants to preserve important personal documents — their will, insurance policies, property deeds, family photographs, and instructions for digital account access — in a tamper-proof package that their executor can access. The package should be encrypted so that only the executor (who has the passphrase, stored separately with the estate attorney) can access it, and the individual wants proof that no one has modified the contents after sealing.

Workflow: The individual creates an IMF container, adds all estate documents, and seals it with encryption and an embedded public key. The passphrase is provided to their estate attorney in a sealed physical envelope. The .imf file is stored in cloud storage (Google Drive, Dropbox) and on a USB drive in a safe deposit box. Upon the individual's passing, the executor retrieves the container, obtains the passphrase from the attorney, verifies the container's integrity, and extracts the documents.

Key IMF properties used: Encryption (protects sensitive personal data), self-verification (executor can verify without technical infrastructure), portability (single .imf file works on any platform with the imf tool), and immutability (proves documents were not altered after the individual sealed them).

9.4 Financial: Regulatory Audit Compliance

Scenario: A financial services firm must retain trade records, communications, and risk calculations for 7 years under SEC Rule 17a-4, which requires that records be stored in non-rewriteable, non-erasable (WORM) format. The firm's compliance team needs to produce quarterly audit packages that regulators can independently verify.

Workflow: At the end of each quarter, an automated pipeline exports trade logs, email archives, and risk model outputs into an IMF container. The container is sealed with the firm's compliance key and a 7-year expiration matching the retention requirement. The sealed container is stored in immutable cloud storage (e.g., AWS S3 Object Lock) as a belt-and-suspenders approach. When regulators request records, the firm provides the .imf file and public key; the regulator runs `imf verify` and `imf extract` to confirm integrity and access the records.

Key IMF properties used: Immutability (satisfies WORM requirement), signature (proves the firm produced the records), expiration (aligns with retention policy), and self-verification (regulators verify independently).

9.5 Software: Supply Chain Integrity

Scenario: An open-source project maintainer releases a new version of their library. Users need assurance that the binary they download was produced by the maintainer and has not been tampered with by a compromised mirror, CDN, or man-in-the-middle attack.

Workflow: The maintainer's CI/CD pipeline builds release artifacts (binaries for multiple platforms, source tarball, SBOM, checksums file), packages them into an IMF container, and seals it with the project's signing key with the public key embedded. The .imf file is uploaded to the release page. Users download the single .imf file, run `imf verify` to confirm it was signed by the project maintainer, and extract the binary for their platform.

Key IMF properties used: Signature (authenticates the maintainer), integrity (detects tampering by mirrors/CDNs), self-verification (users verify without importing keys into GPG), and portability (single file replaces the typical release of binary + .sha256 + .asc files).

9.6 Journalism: Source Document Authentication

Scenario: A journalist receives leaked documents from a confidential source. The journalist needs to prove to their editor and legal team that the documents have not been modified since receipt, and the source wants cryptographic proof of when the documents were provided.

Workflow: Upon receiving the documents, the journalist immediately seals them into an IMF container with their personal signing key. The sealed_at timestamp in the signed manifest provides cryptographic evidence of when the documents were packaged. The journalist shares the .imf file with their editor, who verifies the signature and confirms that the files have not been altered since the journalist received them. If the story is challenged, the signed container serves as evidence of the documents' state at the time of receipt.

Key IMF properties used: Timestamp (proves when documents were sealed), signature (authenticates the journalist as the packager), immutability (proves no post-receipt modification), and optional encryption (protects source identity if container includes metadata).

9.7 Academic: Research Data Preservation

Scenario: A research team is publishing a paper in a peer-reviewed journal. The journal requires that the underlying dataset and analysis code be preserved in a verifiable form to support reproducibility. Funding agencies increasingly require data management plans that include integrity guarantees.

Workflow: At the time of submission, the research team seals their dataset, analysis scripts, and environment configuration into an IMF container with the principal investigator's signing key. The .imf file is deposited in the university's institutional repository alongside the paper. Peer reviewers and future researchers can verify that the data and code have not been modified since the paper was submitted, supporting claims of reproducibility and research integrity.

Key IMF properties used: Integrity (proves data has not been modified post-publication), timestamp (establishes when the dataset was finalized), self-verification (reviewers verify without institutional access), and portability (single file for repository deposit).

10. Future Work

- **Timestamping service integration**: Integration with RFC 3161 timestamping authorities to provide third-party attestation of seal time, strengthening non-repudiation.
- **Multi-party sealing**: Support for multiple signers (e.g., both sender and witness must sign) with threshold signature schemes.
- **Streaming extraction**: Support for extracting individual files from large containers without reading the entire archive into memory.
- **Hardware key support**: Integration with FIDO2/WebAuthn hardware security keys and HSMs for private key storage.
- **Container chaining**: Support for referencing parent containers by hash, enabling append-only audit log chains.
- **Argon2id KDF**: Migration from PBKDF2 to Argon2id for memory-hard passphrase derivation when external dependencies become acceptable.

11. Conclusion

IMF provides a practical, portable, and cryptographically rigorous solution to the problem of tamper-proof file packaging. By combining archival, signing, encryption, and policy enforcement into a single container format with a simple lifecycle model, IMF eliminates the complexity of assembling multiple tools while providing stronger guarantees than any individual component. The self-verifying container concept — where the container carries everything needed for integrity verification — is particularly valuable in distributed environments where pre-shared keys or certificate infrastructure may not be available.

The Go reference implementation demonstrates that the full IMF specification can be realized with zero external dependencies in under 1,000 lines of code, producing a single cross-platform binary suitable for integration into automated workflows. The format is intentionally simple and well-specified to encourage independent implementations in other languages.