IMF: Immutable File Container

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

Technical Whitepaper — Version 1.1

Published: February 12, 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, time-based expiration policies, and blockchain timestamping via OpenTimestamps for third-party proof of existence. 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, the blockchain anchoring mechanism, the case for .imf as a standard sealed document format with native operating system integration, and potential applications in legal evidence packaging, regulatory compliance, software supply chain integrity, intellectual property protection, 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.

Version 1.1 of IMF adds blockchain timestamping via OpenTimestamps, providing third-party proof that a container existed at a specific point in time. This strengthens non-repudiation and enables use cases where provable time-of-creation is critical, such as intellectual property protection, regulatory compliance, and legal evidence preservation.

This paper describes version 1.1 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. Finally, the organization needs provable, third-party evidence of when the package was created — not just a self-reported timestamp, but a cryptographic proof anchored to an immutable public ledger.

Current approaches require assembling multiple tools: a compression utility (zip, tar), a signing tool (GPG, sigstore), an encryption tool (age, GPG), a timestamping service, and custom scripting to tie them together. Each tool has its own key management, its own file format, and its own verification procedure. The result is fragile, error-prone, and difficult for non-technical users.

3. Related Work and Prior Art

Technology	Signing	Encryption	Self-Verifying	Expiry	Timestamp	Portable

PGP/GPG	Yes	Yes	No (keyring)	No	No	Moderate
JAR Signing	Yes	No	No (CA)	No	No	JVM only
Docker Trust	Yes	No	No (Notary)	No	No	Docker only
age	No	Yes	N/A	No	No	Yes
JWT/JWS	Yes	Yes (JWE)	Partial	Yes	No	Data only
IMF	Yes	Yes	Yes	Yes	Yes	Yes

IMF distinguishes itself by combining all six 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. The blockchain timestamping property is unique among file container formats: by anchoring the container hash to the Bitcoin blockchain via OpenTimestamps, IMF provides third-party proof of existence without requiring any accounts, fees, or infrastructure.

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

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.

5.4 Self-Verification: Embedded Public Keys

IMF optionally embeds the signer's Ed25519 public key within the sealed container, enabling self-verification without any external key infrastructure. This is stored as a PEM-encoded key in the keyring/public.key entry. 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

State	Allowed Operations	Transitions To

Open	Create, Add files, List, Info	Sealed (via Seal)
Sealed	Verify, Extract, Anchor, 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. Blockchain Timestamping

Version 1.1 of IMF introduces blockchain timestamping via OpenTimestamps, addressing a key limitation of the original design: while IMF could prove *what* was in a container and *who* sealed it, it could not independently prove *when*. The sealed_at timestamp in the manifest is self-reported and therefore only as trustworthy as the signer. Blockchain anchoring provides third-party, immutable proof of existence at a specific point in time.

7.1 Design Goals

The timestamping feature was designed with the following constraints:

- **Zero infrastructure**: No accounts, API keys, wallets, tokens, or fees required.
- **Zero dependencies**: Implementation uses only Go standard library (net/http, crypto/sha256).
- **Optional**: Timestamping is an add-on; containers are fully functional without it.
- **Independently verifiable**: The proof can be verified by anyone using the Bitcoin blockchain.
- **Minimal data on-chain**: Only a 32-byte SHA-256 hash is anchored, not the container itself.

7.2 Protocol: OpenTimestamps

OpenTimestamps (OTS) is an open-source protocol that aggregates cryptographic digests and anchors them to the Bitcoin blockchain. The protocol uses a Merkle tree structure to batch thousands of timestamps into a single Bitcoin transaction, making individual timestamps effectively free. OTS has been operational since 2016 and is widely used in legal, financial, and archival contexts.

7.3 Anchoring Process

The anchoring process for an IMF container consists of three steps:

- **Step 1: Hash**: Compute the SHA-256 digest of the entire sealed .imf file. This covers every byte of the container including all files, the manifest, the signature, and the sealed marker.
- **Step 2: Submit**: POST the raw 32-byte digest to an OpenTimestamps calendar server (e.g., a.pool.opentimestamps.org/digest). The server returns a compact binary proof file.
- **Step 3: Save**: Store the proof as <container>.ots alongside the .imf file. The proof is initially "pending" — within a few hours, the calendar server batches it into a Bitcoin transaction, upgrading the proof to a full Bitcoin attestation.

7.4 Verification

Anchor verification operates at two levels:

- **Local verification** (imf anchor -verify): Recomputes the container's SHA-256 hash and confirms it matches the hash embedded in the .ots proof file. This detects if the container has been modified after anchoring.

- **Bitcoin verification** (via opentimestamps.org or the ots CLI): Traces the Merkle path from the container hash through the proof to a specific Bitcoin block header, providing cryptographic proof of the exact time the hash was committed to the blockchain.

7.5 Security Properties of Timestamping

Blockchain timestamping adds the following security properties beyond IMF's base guarantees:

- **Non-repudiation of time**: The signer cannot claim the container was created at a different time than when it was anchored. The Bitcoin blockchain provides an independent, publicly auditable clock.
- **Proof of existence**: The timestamp proves the container (and all its contents) existed in their exact form at the time of anchoring. This is valuable for intellectual property disputes, prior art claims, and regulatory filings.
- **Proof of non-modification**: Any modification to the container after anchoring will change its SHA-256 hash, breaking the chain to the Bitcoin block. This provides a second layer of tamper detection independent of the Ed25519 signature.
- **Third-party trust**: Unlike the self-reported sealed_at timestamp, the Bitcoin timestamp does not depend on trusting the signer. It depends only on the integrity of the Bitcoin blockchain, which is secured by proof-of-work consensus.

7.6 Timestamp Use Cases

7.6.1 Intellectual Property Protection

An inventor or researcher develops a novel algorithm, design, or creative work. Before filing for a patent or publishing, they seal their documentation, source code, and design files into an IMF container and anchor it to Bitcoin. The blockchain timestamp provides independently verifiable proof that the work existed at a specific date, establishing priority in any future dispute over who created the work first. This is particularly valuable for defensive prior art strategies where public disclosure is preferred over patent filing.

7.6.2 Regulatory Filing Compliance

Financial institutions subject to SEC Rule 17a-4, FINRA, or MiFID II must demonstrate that records were created and preserved at specific times. A blockchain-anchored IMF container provides an immutable, third-party timestamp that satisfies audit requirements without depending on the institution's own timekeeping infrastructure. Auditors can independently verify when records were committed by checking the Bitcoin block timestamp.

7.6.3 Legal Evidence Chain of Custody

In litigation, the provenance and timing of evidence is critical. A forensic analyst who seals collected evidence into an IMF container and immediately anchors it gains a third-party timestamp proving when the evidence was packaged. This addresses the common challenge of "when was this evidence actually collected?" without relying on the analyst's word or the organization's system clocks, which opposing counsel could challenge.

7.6.4 Whistleblower and Journalism Protection

A journalist receiving leaked documents can seal them into an IMF container and anchor the hash immediately. The blockchain timestamp proves the documents existed in their exact form at a specific time, countering claims that documents were fabricated or modified after the fact. Combined with IMF's encryption, the container protects the contents while the timestamp provides an indelible receipt.

7.6.5 Academic Research Integrity

At the time of paper submission, a research team anchors their dataset, analysis code, and results to Bitcoin. If questions arise about when the data was collected or whether results were modified after peer review, the blockchain timestamp provides definitive proof. This is stronger than a journal submission timestamp, which depends on the journal's infrastructure.

7.6.6 Contract and Agreement Timestamps

Parties to a contract seal the signed agreement into an IMF container and anchor it. The blockchain timestamp proves when the agreement was finalized, independent of either party's claims. This is useful for international contracts spanning time zones, where "date of execution" may be disputed, and for digital-first agreements where no physical notary is involved.

8. Security Properties and Threat Model

8.1 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.
- **Temporal proof**: When anchored to the Bitcoin blockchain, the container's existence at a specific time is provable by any third party with access to the blockchain.
- **Forward compatibility**: The manifest version field allows future specification versions to be identified and handled appropriately.

8.2 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), use of expired containers (enforced by signed timestamp), and false claims about when a container was created (addressed by blockchain anchoring). 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).

9. 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.

9.1 Package Structure

Package	Description
pkg/container	Core API: Create, Add, Seal, Extract, Verify, List, Info
pkg/crypto	Ed25519 signing, AES-256-GCM encryption, PBKDF2 KDF

pkg/manifest	JSON manifest schema, serialization, state management
pkg/anchor	OpenTimestamps blockchain anchoring and verification
cmd/imf	CLI binary with 10 subcommands + web GUI

9.2 CLI Interface

The CLI provides ten 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, info displays container metadata, anchor timestamps the container on the Bitcoin blockchain, and gui launches a web-based graphical interface.

9.3 Web GUI

The reference implementation includes a built-in web GUI launched via 'imf gui'. The GUI serves a single-page application from the Go binary with a REST API backend. It provides drag-and-drop file management, a step-by-step workflow, a Finder-style file browser for extracted files, and integrated blockchain anchoring with a one-click "Anchor to Bitcoin" button. The web server listens only on 127.0.0.1 — all operations happen locally.

10. 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.

10.1 Legal: Digital Evidence Preservation

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. The analyst creates an IMF container on-site, adds all collected files, seals it with the firm's signing key and a 5-year expiration, and immediately anchors it to Bitcoin. The blockchain timestamp provides court-admissible proof of when the evidence was packaged, independent of the firm's system clocks.

Key IMF properties used: Cryptographic integrity (SHA-256), digital signature (Ed25519), self-verification (embedded public key), timestamp (sealed_at + Bitcoin anchor), expiration.

10.2 Medical: Patient Record Transfer

A patient is transferring between hospital systems. Their records — imaging files, lab results, physician notes, and medication history — must be transferred securely with HIPAA compliance. The sending hospital seals the records into an encrypted IMF container with a 30-day expiration and anchors it for audit trail purposes. The receiving hospital verifies the signature, decrypts, and confirms all file hashes match before importing into their EHR system.

Key IMF properties used: AES-256-GCM encryption (HIPAA compliance), integrity verification, expiration, signature, blockchain timestamp (audit trail).

10.3 Personal: Estate Planning and Digital Legacy

An individual preserves their will, insurance policies, property deeds, and family photographs in an encrypted IMF container. The passphrase is provided to their estate attorney in a sealed envelope. The container is anchored to Bitcoin, providing provable evidence of when the estate documents were finalized. Upon the individual's passing, the executor retrieves the container, verifies integrity, and extracts the documents.

Key IMF properties used: Encryption, self-verification, portability, immutability, blockchain timestamp (proves documents were not altered after a specific date).

10.4 Financial: Regulatory Audit Compliance

A financial services firm must retain trade records for 7 years under SEC Rule 17a-4 in WORM format. Each quarter, an automated pipeline seals records into an IMF container and anchors it to Bitcoin. The blockchain timestamp provides regulators with third-party proof that records were committed at the reported time, without relying on the firm's internal clocks.

Key IMF properties used: Immutability (WORM compliance), signature, expiration, self-verification, blockchain timestamp (third-party audit proof).

10.5 Software: Supply Chain Integrity

An open-source project maintainer packages release artifacts into an IMF container and anchors it. Users download the .imf file, verify the signature, and extract their platform binary. The blockchain timestamp proves the release was created at a specific time, detecting backdated or retroactively modified releases.

Key IMF properties used: Signature, integrity, self-verification, portability, blockchain timestamp (release provenance).

10.6 Intellectual Property: Prior Art Establishment

A startup develops a novel algorithm and wants to establish prior art before publishing. The team seals their research notes, source code, and technical documentation into an IMF container and anchors it to Bitcoin. The blockchain timestamp provides cryptographic proof that the work existed at a specific date, serving as a defensive prior art record that cannot be disputed or backdated.

Key IMF properties used: Immutability, signature, blockchain timestamp (priority proof), self-verification (anyone can verify the claim independently).

11. Toward a Standard Sealed Document Format

The preceding sections describe IMF as a technical specification and reference implementation. This section argues that .imf has the properties necessary to become a widely adopted standard file format — a "sealed document" type analogous to how .pdf became the standard for portable documents and .zip became the standard for compressed archives.

11.1 Precedent: How File Formats Become Standards

The most successful file formats share a common adoption pattern. PDF succeeded not because Adobe mandated it, but because it solved a universal problem (portable rendering), had a small free reader (Acrobat Reader), and an open specification that anyone could implement. ZIP succeeded because it was simple, useful, and had readers built into every operating system. In both cases, the format was self-contained (a .pdf or .zip carries everything needed to be useful) and the reader was trivially installable.

IMF follows this same pattern. The .imf format solves a universal problem (tamper-proof file packaging), produces self-contained files that carry everything needed for verification, and has a reference reader that compiles to a single binary with zero external dependencies. The format specification is fully documented, openly licensed (Apache 2.0), and simple enough that independent implementations can be created in any language.

11.2 Self-Contained by Design

A critical property for format adoption is self-containment. A .pdf file does not require the original authoring application to be viewed. A .zip file does not require the original compression tool to be extracted. Similarly, a .imf file contains everything needed for its own verification and extraction:

- All original files (plaintext or encrypted)
- SHA-256 integrity hashes for every file
- An Ed25519 digital signature proving authorship
- The signer's public key (embedded for self-verification)
- Expiration policy and timestamps
- A sealed marker enforcing immutability

No external key servers, certificate authorities, blockchain nodes, or internet connectivity is required to verify an .imf file. The recipient needs only the .imf file and a reader.

11.3 Native Operating System Integration

For a file format to feel native, it must integrate with the operating system's file management paradigm. Users should be able to double-click a file and have it open in the appropriate reader, just as they double-click a .pdf or .docx file today.

The IMF reference implementation includes a macOS application bundle (IMF Viewer.app) that registers the .imf file extension and the com.imf.container Uniform Type Identifier with the operating system. Once installed, .imf files display with a custom icon and open in the viewer on double-click. The viewer provides

immediate visual feedback: a green checkmark for verified containers, a red warning for tampered ones. This interaction model mirrors how users already interact with other trusted file types.

The same integration pattern is extensible to Windows (via registry-based file association and a .msi installer) and Linux (via .desktop files and MIME type registration). The goal is that receiving a .imf file should feel as natural as receiving a .pdf.

11.4 Registered Type Identity

The .imf format declares the following identifiers for cross-platform recognition:

- **File extension**: .imf
- **MIME type**: application/x-imf
- **Uniform Type Identifier** (macOS): com.imf.container
- **Conforms to**: public.data, public.archive (recognized as a data archive by the OS)

These identifiers enable operating systems, email clients, cloud storage providers, and web browsers to recognize .imf files and route them to the appropriate handler. As adoption grows, these identifiers could be registered with IANA for formal MIME type recognition.

11.5 Minimal Reader Footprint

A file format is only as accessible as its reader. The IMF reference reader compiles to a single binary of approximately 6-8 megabytes with zero runtime dependencies. It runs on macOS, Windows, and Linux without installation of frameworks, runtimes, or libraries. For comparison, Adobe Acrobat Reader is approximately 250 megabytes and requires administrative installation. The IMF reader's minimal footprint eliminates the primary adoption barrier for new file formats.

The format's reliance on well-understood cryptographic primitives (Ed25519, AES-256-GCM, SHA-256) and a standard container format (ZIP) means that independent readers can be implemented in any language with access to these primitives — which includes virtually every modern programming language's standard library.

11.6 The Trust Model for Sealed Documents

Today, when someone receives a document, they have no way to know if it has been altered since the author sent it. Email attachments can be intercepted. Cloud-shared files can be modified by anyone with access. USB drives can be tampered with in transit. The recipient must simply trust the delivery channel.

A .imf file inverts this trust model. The document carries its own proof of integrity. The recipient does not need to trust the delivery channel, the email server, the cloud provider, or the intermediary who handed them the USB drive. They verify the container directly, and the cryptographic signature either confirms or denies integrity. Optionally, the blockchain timestamp proves when the document was sealed, independent of any party's claims.

This trust model — where the document itself is the proof, not the channel — is the fundamental value proposition of .imf as a standard file format. It transforms document exchange from "trust the sender" to

"verify the seal."

11.7 Adoption Path

The proposed adoption path for .imf as a standard format follows the precedent set by PDF, ZIP, and other successful formats:

- **Phase 1 — Tool adoption**: Professionals in legal, financial, and compliance domains adopt IMF for specific high-value use cases where tamper-proof packaging is required.
- **Phase 2 — Format recognition**: Operating systems, email clients, and cloud storage providers recognize .imf files and offer native preview or verification indicators.
- **Phase 3 — Ecosystem growth**: Independent implementations emerge in multiple languages. Libraries enable applications to create and verify .imf files natively.
- **Phase 4 — Standard integration**: The .imf format is submitted for IANA MIME type registration and considered for inclusion in document exchange standards.

12. Future Work

- **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.
- **Timestamp upgrading**: Automatic upgrade of pending OpenTimestamps proofs to full Bitcoin attestations once the calendar server confirms the transaction.
- **Cross-platform native apps**: Signed and notarized application bundles for macOS (.app), Windows (.msi), and Linux (.deb/.AppImage) with native file type registration, enabling seamless double-click-to-open interaction on all major platforms.
- **WebAssembly reader**: Compilation of the Go reference implementation to WebAssembly, enabling browser-based verification and extraction as a Progressive Web App with no installation required.
- **IANA MIME type registration**: Formal registration of application/x-imf as a recognized media type with the Internet Assigned Numbers Authority.

13. Conclusion

IMF provides a practical, portable, and cryptographically rigorous solution to the problem of tamper-proof file packaging. By combining archival, signing, encryption, policy enforcement, and blockchain timestamping 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 addition of blockchain timestamping in version 1.1 addresses the critical question of "when" — providing third-party, immutable proof of existence that does not depend on trusting the signer or any centralized authority.

Beyond its technical merits, .imf has the properties necessary to become a standard file format for sealed documents: self-containment, a minimal reader, an open specification, and native operating system integration. The trust model it introduces — where the document itself is the proof of integrity, independent of the delivery channel — addresses a gap that no existing standard file format fills. Just as .pdf standardized portable document rendering and .zip standardized compressed archives, .imf has the potential to standardize tamper-proof document exchange.

The Go reference implementation demonstrates that the full IMF specification — including blockchain timestamping and native application packaging — can be realized with zero external dependencies, producing a single cross-platform binary suitable for both end-user interaction and integration into automated workflows. The format is intentionally simple and well-specified to encourage independent implementations in other languages and to lower the barrier to ecosystem adoption.

This document establishes prior art for the IMF container format, its associated cryptographic architecture, blockchain anchoring mechanism, and its proposed role as a standard sealed document format. It is published under the Apache License 2.0 with a defensive prior art intent.