

Verifying an ISOF file

Without IsoFind, without a network, without taking our word for it

What a signed file proves, what it does not, and how to check it yourself.

August 2026. Describes ISOF v1.2 as IsoFind writes it.

A standalone verifier comes with this guide: `verify_isof.py`, one dependency, no connection.

Contents

Understanding

1. Who this guide is for	3
2. What a signature attests	3
3. Anatomy of a file	4
4. What can be verified	6

Verifying

5. The serialisation rules	8
6. Level 1, integrity	9
7. Level 2, the laboratory signature	10
8. Provenance	11
9. Origin claims	13
10. An encrypted file	15

Interpreting

11. Reading a verdict	16
12. Twelve traps	17

Appendices

13. The standalone verifier	18
14. Getting a certificate	19
15. References	20

Understanding

What an ISOF file is, what its signature establishes, and what you need at hand to check it.

1. Who this guide is for

Whoever receives a file and has to decide whether they can rely on it: an auditor, a customer, a reviewer, a quality department, a forensic expert, a colleague being sent measurements.

You need no IsoFind to read this guide, nor to do what it describes. An ISOF file is a JSON file, readable in any text editor, and every check is done offline with public algorithms. That is the point of the format: a proof that required the software of whoever produced it would not be one.

What you need

To	You need
Read the content	A text editor. Nothing else.
Check integrity	Something that computes a SHA-256: Python, Node, or a command line tool.
Check a laboratory signature	A common cryptographic library. The script supplied uses only <code>cryptography</code> , in Python.
Check who issued the certificate	The IsoFind root certificate, obtained once through a channel of your choosing, and kept.
Decrypt an encrypted file	The recipient private key. Without it, nobody can.

The shortcut. If all you want is an answer, section 13 gives a one command script that produces a readable verdict. The rest of the guide explains what that verdict covers, and that is what will let you defend it in front of someone.

What this guide does not cover

The scientific interpretation of the measurements, the validity of the analytical methods, and whether the signing laboratory is competent or honest. No cryptographic device answers those questions, and a document implying otherwise would do you a disservice.

2. What a signature attests

This is the most important section of the guide, and the shortest to summarise: a signature speaks about the history of the file, never about the correctness of what it contains.

What is established

That the laboratory named in the certificate, holding a certificate issued by the IsoFind authority, sent exactly these data. Remove one measurement, add one, change a decimal, correct a date: the signature no longer holds, and the file is refused.

When the file carries a provenance block, the statement extends to the history of the changes: the sample went through exactly these changes, in this order, by these actors. One entry removed or edited breaks the chain.

What is not

That the measurements are correct. No signature can establish that. A laboratory can sign wrong values, and its signature will be perfectly valid: it will attest that it did send them, which is exactly what it is being asked.

That the laboratory is competent, accredited, or acting in good faith. A certificate attests an identity, not a quality.

That the data are complete. A signer chooses what it exports and what its signature covers. See section 12, where this is the first trap.

The distinction is not rhetorical caution. It decides what you can write in a report. "The file is intact and comes from this laboratory" is defensible; "the values are certified" is not, and nobody ever promised it.

Why it is worth having anyway

Because the question an auditor asks is almost never "are these numbers right", which they could only answer by measuring again. It is "are these the numbers the laboratory produced, or were they arranged along the way". A signed ISOF file answers that one, and it answers without asking you to trust the software that wrote it.

3. Anatomy of a file

A JSON object, a few top level blocks, and a signature block saying which of them it covers.

```
{
  "isof_version": "1.2",
  "created_at": "2026-08-26T08:00:00+00:00",
  "created_by": {
    "software": "IsoFind", "software_version": "1.0.0",
    "operator": "...", "organisation": "..."
  },
  "project": { "name": "..." },

  "samples": [ ... ], // samples, measurements included
  "methods": { ... }, // protocols used
```

```

"pipelines": { ... }, // chains of methods
"purification": { ... }, // yields
"assignments": [ ... ], // which method on which sample

"provenance": { ... }, // signed history, optional
"origin_claims": [ ... ], // origin claims, optional
"encryption": { ... }, // present if the content is encrypted

"signature": { ... } // null if the file is unsigned
}

```

A sample

Each sample carries its identity, its location where it was recorded, and up to four families of data: isotopes, elemental geochemistry, physico-chemical parameters, molecules. An exporter may send only some of them.

```

{
  "id": "1", "name": "PZ-01", "material_type": "groundwater",
  "isotope_data": [
    {
      "element": "Sr", "system": "87Sr/86Sr",
      "ratio": 0.70912, // THE MEASUREMENT, always
      "ratio_2se": 0.00002,
      "standard": "NBS 987", "instrument": "Neptune Plus",

      "derived": { // present only if a correction exists
        "value": 0.70909, "nature": "blanc",
        "formule": "correction_blanc", "motif": "Session blank",
        "superseded": [ ... ] // the corrections it replaced
      }
    }
  ]
}

```

ratio is the measurement, and stays so. When a correction has been applied it lives in **derived**, beside it, never in its place. A reader ignoring **derived** therefore gets a true but incomplete value, and never a computed value mistaken for an instrument reading. If you quote figures in a report, look for **derived**, and say which of the two you are quoting.

The signature block

```

// Level 1: a digest, recomputable by anyone
"signature": {
  "level": 1, "algorithm": "SHA-256",
  "scope": ["created_by", "samples", "methods", "purification"],
  "hash": "a3f1...", "signed_at": "...", "signed_by": "..."
}

```

```
// Level 2: an ECDSA signature, and the identity that comes with it
"signature": {
  "level": 2, "algorithm": "ECDSA-P256", "hash_algorithm": "SHA-256",
  "signed_scope": ["created_by", "samples", "methods", "purification", "provenance"],
  "signature_b64": "MEUCIQ...",
  "certificate_chain": ["<laboratory, PEM in base64>", "<issuing authority>"],
  "signed_at": "...", "signed_by": "...",
  "revocation": {
    "crl_url": "https://pki.isofind.io/crl/issuing-ca.crl",
    "crl_embedded_b64": "...", "crl_snapshot_date": "..."
  }
}
```

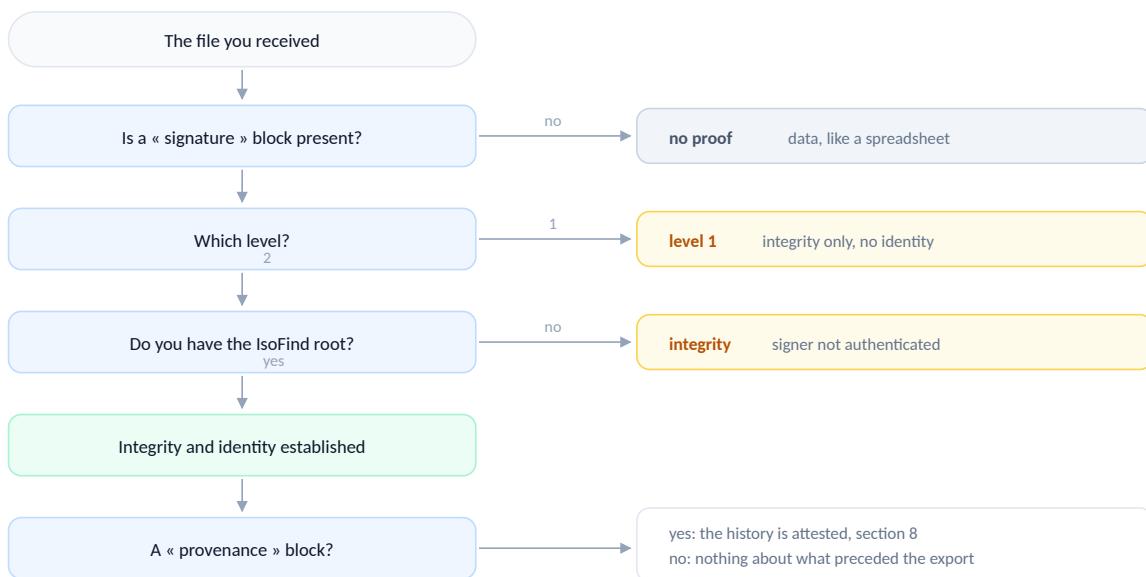
The most important field of the two is `scope`, or `signed_scope` at level 2: it lists the blocks the signature covers. What is not in it is not protected, and may have been modified afterwards with nothing to say so.

4. What can be verified

A file carries from zero to four verifiable objects, independent of one another. Start by looking at which ones are there.

Object	Present if	What it establishes
Document signature	<code>signature</code>	That the blocks in the scope have not moved. At level 2, who sent them.
Provenance	<code>provenance</code>	That the history of changes is complete and in order.
Origin claims	<code>origin_claims</code>	That the reasoning behind an origin conclusion is coherent and anchored in the data of the file.
Encryption	<code>encryption</code>	Nothing in itself: it determines what you can read, hence what you can verify.

Where to start



The branches on the right are not failures: they are different levels of proof, and what you need to know before quoting the file anywhere.

An unsigned file is not proof. It remains useful data, exchanged in a readable format, exactly like a spreadsheet. That is not a shortcoming of the file: most scientific exchanges have no need to be binding. It becomes a problem the day it is quoted as though it were.

Verifying

One procedure per verifiable object, with the exact rule and enough to redo it yourself.

5. The serialisation rules

Every check comes down to recomputing a digest over text. Two ways of writing the same JSON object give two different digests, so the writing rule is part of the format. Read this section before the ones that follow.

ISOF v1.2 uses two of them, depending on the block. It is not elegant, and the next major version will unify them on RFC 8785 as an announced break. Until then, both are reproducible from the file alone, and here they are.

Rule	Used by	Keys
A	The document signature, levels 1 and 2	In file order, not sorted
B	Provenance, origin claims	Sorted alphabetically, at every level

They share the rest: compact JSON, no space after separators, no escaping of non-ASCII characters, encoded in UTF-8.

```
# Rule A: the document signature
json.dumps(payload, separators=(",", ":"), ensure_ascii=False)

# Rule B: provenance and claims
json.dumps(payload, separators=(",", ":"), ensure_ascii=False, sort_keys=True)
```

Rule A depends on order, and is still reproducible

Not sorting looks fragile, but a JSON parser that preserves the key order of the file, which Python has done since 3.7 and JavaScript does, gives back exactly the order the signer wrote. Read the file, serialise it again, and you get its text. In Python, load with `object_pairs_hook=OrderedDict` so that this is explicit rather than assumed.

What you must not do is sort the keys yourself thinking you are being tidy: the digest will never come out right.

Numbers are the real trap. One language does not write floats like another: Python serialises `1.0`, JavaScript writes `1`. A verification written in a language other than the signer's can therefore fail on an intact file. ISOF files are written in Python; verify them in Python, or check your language's number formatting very carefully.

What goes into the computation

Never the whole file. Always the blocks the scope lists, in the order of the scope, and nothing else.

```
payload = OrderedDict((block, doc[block]) for block in scope if block in doc)
```

One exception, for encrypted files. When the scope is exactly `["encrypted_metadata"]`, the payload is not built from the file but fixed by the specification: `created_by`, `project`, `created_at`, in that order. See section 10.

6. Level 1, integrity

A SHA-256 digest over the covered blocks. It answers one question: has the file changed since that digest was computed.

The procedure

1. Read `signature.scope`, the list of covered blocks.
2. Build the payload: those blocks, in that order, the ones that exist.
3. Serialise with rule A, encode as UTF-8, compute the SHA-256.
4. Compare with `signature.hash`.

```
import json, hashlib
from collections import OrderedDict

doc = json.load(open("file.isof", encoding="utf-8"),
                object_pairs_hook=OrderedDict)
sig = doc["signature"]
scope = sig["scope"]

payload = OrderedDict((b, doc[b]) for b in scope if b in doc)
canonical = json.dumps(payload, separators=(",", ":"), ensure_ascii=False)
computed = hashlib.sha256(canonical.encode("utf-8")).hexdigest()

print("intact" if computed == sig["hash"] else "altered")
```

A digest is not a signature. Anyone can edit the file, recompute the digest, write it back into the block, and change `signed_by` to any name they like. Level 1 protects against accidental alteration, corruption in transit, a clumsy edit. It protects against nobody who is determined. For that you need level 2.

Files signed before August 2026

Until then, the signer added the `created_by` block to the payload without declaring it in the scope. Those files exist and remain verifiable: if the computation above does not come out right, and only then, start again with `created_by` appended at the end of the payload.

```
if computed != sig["hash"] and "created_by" not in scope:
    payload["created_by"] = doc["created_by"] # LAST
    # then recompute. If that comes out right, the file is intact,
    # in the earlier format.
```

That second attempt must remain the only variant you try. A verifier trying several forms until one matches would end up validating a modified file.

7. Level 2, the laboratory signature

ECDSA on the P-256 curve, with SHA-256, and a certificate that chains up to the IsoFind authority. This is the level that answers "who sent this".

The procedure, in five checks

Check	What it establishes	Without it
1. ECDSA signature	The blocks in the scope have not moved since signing.	Nothing is established.
2. Certificate chain	The certificate was issued by the IsoFind authority.	The signature holds, but by someone nothing identifies.
3. Validity dates	The certificate was not expired at signing time.	A signature made with a lapsed certificate.
4. Revocation	The certificate was not revoked at the date of the enclosed CRL.	A compromised certificate may have been used.
5. Declared identity	That the certificate subject is the laboratory you expected.	A valid certificate, from another laboratory.

The fifth is yours, and no program will do it: compare the certificate subject, printed by the script, with the laboratory whose file you were expecting.

```
import base64
from cryptography import x509
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import ec

sig = doc["signature"]
scope = sig["signed_scope"]
```

```

chain = [x509.load_pem_x509_certificate(base64.b64decode(c))
         for c in sig["certificate_chain"]]
lab    = chain[0]                                # leaf first, then the issuer

payload = OrderedDict((b, doc[b]) for b in scope if b in doc)
canonical = json.dumps(payload, separators=(",", ":"), ensure_ascii=False)

lab.public_key().verify(
    base64.b64decode(sig["signature_b64"]),
    canonical.encode("utf-8"),
    ec.ECDSA(hashes.SHA256()),
) # raises InvalidSignature if the content moved

print(lab.subject.rfc4514_string())
print(lab.fingerprint(hashes.SHA256()).hex())

```

The certificate chain

The file carries the laboratory certificate and that of the issuing authority. It does not carry the root, deliberately: a root supplied by the document it is meant to validate would prove nothing. You obtain it once, through a separate channel, and keep it.

```
python verify_isof.py file.isof --root isofind_root_ca.pem
```

Without the root, do not conclude too much. A signature whose issuer has not been checked establishes that the document has not moved since it was signed, by someone. That is useful; it is not an identity. The script says so explicitly rather than returning a green verdict.

Revocation, and what an enclosed CRL is worth

The `revocation` block holds a copy of the revocation list taken at signing time. It lets you conclude offline, which is the point, but it is dated: it establishes that the certificate was not revoked at that date, and nothing about today. For a routine check, compare the snapshot date with the signature date. For a contentious file, fetch the current list from the address in `crl_url`.

8. Provenance

The signature establishes that the file has not moved since the export. Provenance establishes what happened before: every change the sample went through, in order, with its author.

Why two chains, and why this one is verifiable

Internally, IsoFind chains its history with HMAC-SHA256, using a key that never leaves the machine. That is what stops someone with access to the database from recomputing a valid digest, and it is also what makes that chain unverifiable by you: without the key you can do nothing with it.

A second chain is therefore exported, in plain SHA-256, recomputable from the data present in the file alone. That is the one the signature seals. Signing the internal digest would have produced an attestation that looks like proof without being one: you would have read "signed by laboratory X", found the signature valid, and had no way to check what it covers.

The computation

Each event is hashed together with the digest of the previous one, starting from sixty-four zeros. The fields that enter the computation and their order are frozen by the format.

```
FIELDS = ("seq", "timestamp", "operation", "table", "row_id",
          "before", "after", "actor", "agent")

previous = "0" * 64

for e in events:
    body = {f: e.get(f) for f in FIELDS}
    # RULE B: sorted keys
    msg = previous + "|" + json.dumps(body, separators=(",", ":"),
                                     ensure_ascii=False, sort_keys=True)
    h = hashlib.sha256(msg.encode("utf-8")).hexdigest()

    assert e["seq"] == expected      # no missing link
    assert e["prev"] == previous     # no break
    assert e["hash"] == h            # no edit

    previous = h

assert previous == sample["head"]   # the head matches
```

The three checks catch three distinct frauds. The rank catches a removed event, the chaining catches an inserted one, the digest catches a modified value.

The two modes

mode	Content	What you can conclude
complet	Whole events, with the before and after values.	Everything: what the value was, what it became, and that nothing was added or removed.
resume	Metadata and a digest of the content, without the values.	That the history is complete and unedited. Not what it contains.

Summary mode is not an evasion. A supplier may have to prove to a customer that it did not touch up its measurements without revealing its internal corrections. The chain stays verifiable, and if the content is disclosed to you later, you will be able to confirm it is the one that was covered. In summary mode the hashed body is made of `seq` , `timestamp` , `operation` , `table` , `row_id` , `actor` , `agent` and `content_sha256` .

An admission not to miss. Each sample carries an `internal_chain` field saying whether the internal HMAC chain was intact at export time. A laboratory exporting a broken chain therefore declares it itself, under its own signature, and cannot keep quiet about it. If you read `valid: false` , the cryptographic verification may well pass: this is not a verification error, it is a statement, and it calls for a question to the signer.

Signing the block

Provenance is covered in one of two ways. Either it appears in the scope of the document signature, which IsoFind enforces as soon as a provenance block is present. Or it carries its own ECDSA signature, computed over the rule B serialisation of the whole block, signature excluded.

Signing the whole block rather than the head digest alone is not a detail: the head covers only the chaining, and one could change an actor name or a date without invalidating anything.

9. Origin claims

An `origin_claims` block carries conclusions of the kind "this sample is compatible with this deposit". That is no longer integrity, it is reasoning, and it is verified differently.

These claims follow the ISOF v1.3 schema. The verifier does not judge the conclusion, it checks that the reasoning holds together: that the data cited exist in the file, that they have not changed since the claim was issued, that the verdict is compatible with the claim type, and that no reference runs in a circle.

What a claim carries

Field	Content
<code>claim_type</code>	<code>origin</code> , <code>non_origin</code> , <code>compatibility</code> , <code>exclusion</code>
<code>subject</code>	The sample concerned, and its <code>sample_hash</code> at issue time
<code>target</code>	What it is compared with: deposit, supplier, geographic zone, batch, process
<code>verdict</code>	The decision, <code>match</code> , <code>no_match</code> or <code>inconclusive</code> , and its uncertainty model
<code>evidence</code>	The elements relied on, each pointing at data in the file or at another claim

Field	Content
<code>method_chain</code>	The method used, and the engine that ran it
<code>limitations</code>	Required, and non empty. The format refuses a conclusion that does not say what it fails to cover.

The refused combinations

A verdict must be compatible with the claim type, otherwise the meaning inverts without anyone noticing.

claim_type	Decisions allowed
<code>origin</code>	<code>match</code> , <code>inconclusive</code>
<code>non_origin</code>	<code>no_match</code> , <code>inconclusive</code>
<code>compatibility</code>	<code>match</code> , <code>inconclusive</code>
<code>exclusion</code>	<code>no_match</code> , <code>inconclusive</code>

The codes the verifier returns

Code	Meaning
<code>ISOE-002</code>	Malformed claim, it does not follow the schema
<code>ISOE-200</code>	Subject sample not found, and no external reference
<code>ISOE-201</code>	<code>sample_hash</code> no longer matches: the sample changed since issue
<code>ISOE-202</code>	A cited datum does not exist in the document
<code>ISOE-204</code>	Verdict incompatible with the claim type
<code>ISOE-210</code>	Cited claim missing, or of an unexpected type
<code>ISOE-211</code>	Digest of a cited claim does not match
<code>ISOE-212</code>	Cycle in the claim graph
<code>ISOE-213</code>	Depth beyond the recommended threshold, a warning
<code>ISOE-214</code>	Composite claim with no block explaining the aggregation

The `sample_hash` is recomputed like the rest: rule B, over the whole sample minus its own `sample_hash` field, prefixed with `sha256:`. A mismatch is not necessarily fraud, a claim issued before an update to the sample produces the same symptom, but either way the claim no longer applies to what it says it covers.

Read `limitations`. It is the only required field whose content no program can verify, and it is the one that tells you what the claim does not cover. A serious issuer writes something useful there.

10. An encrypted file

A file can be encrypted for one named recipient. You can then verify its origin without being able to read its content, and that is deliberate.

The `encryption` block carries `active: true`, and the data blocks hold ciphertext instead of measurements.

```
"encryption": {
  "active": true,
  "algorithm": "X25519-HKDF-SHA256 + AES-256-GCM",
  "recipient_fingerprint": "...", // for whom, without naming them
  "ephemeral_public_key": "...",
  "session_key_enc": "...", "wrap_nonce": "...", "nonce": "..."
}
```

The session key is drawn at random for this file, it encrypts the data with AES-256-GCM, and it is itself wrapped for the recipient's X25519 public key. Nobody else can recover it, the sender included once the file is written.

What remains verifiable

The signature of an encrypted file carries the scope `["encrypted_metadata"]`, and covers three fields left in clear: `created_by`, `project` and `created_at`, in that order. You can therefore establish who issued the file, for which project and when, while knowing nothing of its content.

What that signature does not cover. The encrypted blocks themselves are not part of the computation. Someone could replace the ciphertext with another without invalidating the signature; the recipient would notice on decryption, since GCM authentication would fail, but a third party checking only the signature would not. So never conclude "the encrypted content is authentic", only "the envelope comes from this issuer".

Decryption needs the recipient's private key and is not part of this guide. If the file is meant for you, IsoFind or the `isof` package will do it; once decrypted, the document is an ordinary ISO file, and sections 6 to 9 apply to what it contains.

Interpreting

What a verdict lets you write, and the reading mistakes that cost the most.

11. Reading a verdict

A verification rarely returns a plain yes or no. Here is what each outcome lets you write, and what it calls for.

Outcome	What it means	What to do
Level 2 signature valid, chain reaching the root	The covered blocks are exactly those the laboratory named in the certificate sent.	Check that this laboratory is the one whose file you were expecting, and look at the scope.
Signature valid, root not supplied	The file has not moved since it was signed, by someone whose identity is not established.	Obtain the root certificate and start again. Until then, do not put the laboratory name in your report.
Level 1 signature valid	The file has not been altered by accident. Nothing more.	If anything binding rests on it, ask for a level 2 signed file.
Earlier format reported	The file is intact; it was signed before August 2026, when <code>created_by</code> was covered without being declared.	Nothing. It is a compatibility note, not a defect of the file.
Digest or signature does not match	A covered block changed after signing. The file is worth nothing as proof.	Do not hunt for where: ask the source to send it again. A difference can also come from a tool that re-saved the JSON and reformatted it.
Provenance chain broken	The exported history was edited: an event removed, inserted or modified.	A question for the signer. The document signature may fail too, since it covers the provenance.
Internal chain declared broken	The laboratory states itself, under its own signature, that its local history was inconsistent at export.	This is not a verification error, it is an admission. Ask for an explanation before using the data.
Certificate expired	Two cases: expired before the signature, or expired since. The second is normal for an old file.	Compare <code>signed_at</code> with the certificate's end of validity.
Certificate revoked	The certificate was withdrawn by the authority, often after a compromise.	Do not rely on the file. Check the revocation date against the signature date.
No signature	Data, and nothing else.	Usable like a spreadsheet. Do not quote it as binding.

The sentence that holds

When everything passes, what you can write is: "the file received on such a date is intact, carries a valid signature from laboratory X issued by the IsoFind authority, and covers the following blocks".

What you cannot write is: "the measurements are certified", "the data are correct", or "the laboratory is accredited". None of that was checked, and none of it was ever promised.

12. Twelve traps

The reading mistakes that keep coming back, in order of what they cost.

1. **The partial scope.** A signature covers only the blocks it lists. A file can carry a perfectly valid signature and an `assignments` block modified afterwards. Read the scope before concluding, and quote it.
2. **Taking level 1 for a signature.** It is a digest, recomputable by whoever edits the file. The name in `signed_by` is declarative and binds nobody.
3. **Confusing `ratio` and `derived`.** The measurement always stays in `ratio`. When `derived` exists, a correction has been applied, and it is often the one the laboratory treats as the value in force. Quoting one for the other changes the result with nothing to say so.
4. **Believing the root is in the file.** It is not, and it never will be. A root supplied by the document it validates would prove nothing.
5. **Sorting keys when recomputing.** The document signature is not sorted. A verifier that sorts will never get the right digest, and will conclude that an intact file was altered.
6. **Verifying in another language without thinking about it.** Number formatting differs between languages. A `1.0` become `1` is enough to fail a verification.
7. **Reformatting the file before verifying it.** Opening an ISOF file in a tool that reindents and re-saves does not change the data, but may change key order or the way numbers are written. Verify the file as it arrived.
8. **Reading an enclosed CRL as today's answer.** It is a snapshot dated from the day of signing. For a dispute, fetch the current list.
9. **Ignoring `internal_chain`.** A laboratory can sign a history it itself declares was broken. The cryptographic verification passes, and yet something is wrong.
10. **Taking summary mode for complete provenance.** It proves nothing was added, removed or edited. It does not show what the values were.
11. **Concluding about the content of an encrypted file.** The signature covers only the clear metadata. The envelope is authenticated, the content is not.
12. **Confusing "no origin claim" with "origin not established".** Most files carry none, because most exchanges conclude nothing about an origin. Absence is not a negative result.

Appendices

The verifier, getting a certificate, and where to go next.

13. The standalone verifier

`verify_isof.py` comes with this guide. One dependency, no connection, no IsoFind component.

```
pip install cryptography
python verify_isof.py file.isof --root isofind_root_ca.pem
```

It runs through what sections 6 to 8 describe: the document signature at whichever level it finds, the certificate chain when a root is supplied, expiry, revocation through the enclosed CRL, then the provenance chain event by event. It exits with code 0 when everything the file allowed to be checked has passed, and 1 otherwise, so it can be called from a pipeline.

A typical output

```
ISO file      : sample.isof
format version : 1.2
created by    : HydroSciences Montpellier

Document signature
-----
PASS level 2 ECDSA signature matches
note covered blocks: created_by, samples, methods, purification, provenance
note certificate subject: O=HSM,CN=HydroSciences Montpellier
note certificate fingerprint: b4a2e1a765de1f70...
PASS certificate chains up to the supplied root
note CRL snapshot is 3 days old

Provenance
-----
note mode: complet
PASS provenance chain recomputed: 2 event(s) over 1 sample(s)

VERDICT: every check that this file allows has passed.
This says the file has not been altered since it was signed. It says
nothing about whether the measurements are correct.
```

The **WARN** and **skip** lines matter as much as the **PASS** ones. They say what could not be checked: root missing, no provenance in the file, no CRL. A green verdict obtained without them does not cover the same ground.

The script is short and readable: it is meant to be read, adapted, or reimplemented in another language from the rules in Part II. The `isof` package, published on PyPI under the MIT licence, provides a fuller reference implementation, encryption included.

14. Getting a certificate

This section is for the other side: whoever wants to issue level 2 signed files rather than verify them.

How it goes

1. The laboratory generates an ECDSA P-256 key pair on its own machine, and a certificate request carrying its identity. The private key never leaves the machine, and is never transmitted.
2. The request goes to the IsoFind authority, which checks the identity of the entity declared.
3. The signed certificate comes back and is filed in the local store, encrypted at rest.
4. From then on, every export can be signed at level 2, and each file carries the certificate along with that of the issuing authority.

What the private key never does

It does not leave the machine, it is not escrowed with a third party, and it is not recoverable if lost. A lost certificate is replaced, files already signed remain verifiable, and that is how a signature is supposed to behave.

The tiers

Signing entity	Per year
Academic laboratory or public body	80 €
Company up to 20 people	300 €
Company from 20 to 250 people	700 €
Company over 250 people	1,500 €
Regulator, defence	on request

The fee covers the identity check and running the authority, not the signed data, of which IsoFind never receives a copy. Verifying a file stays free and needs no account, permanently: a format whose verification had to be paid for would be of no use.

Level 1 remains available without a certificate. It is enough for an exchange between colleagues, an archive, a data deposit. A certificate is only of interest if your files have to be binding on someone.

15. References

The algorithms used

Use	Algorithm
Document digest, provenance chaining	SHA-256
Document and provenance signature	ECDSA P-256, SHA-256
Internal history chaining, not exported	HMAC-SHA256
Encrypting a file for a recipient	X25519, HKDF-SHA256, AES-256-GCM
Sample and claim digests	SHA-256, prefixed <code>sha256:</code>

Versions

This document describes ISOF v1.2, the version IsoFind writes into `isof_version`. The origin claim schema carries its own number, v1.3, and evolves separately. Moving the serialisation rules to RFC 8785 is planned for the next major version, as an announced break: files issued today will stay verifiable under the rules in this guide.

Vocabulary

Term	Meaning in this guide
Scope	The list of blocks a signature covers. What is not in it is not protected.
Provenance	The signed history of the changes a sample went through before export.
Internal chain	The local HMAC chaining, not exportable and not verifiable by a third party, whose state is declared in the file.
Origin claim	A reasoned conclusion about the geographic or industrial provenance of a sample.
Level 1, level 2	Digest alone, or signature with a certificate.

Going further

- The `isof` package, on PyPI, MIT licence: reference implementation of the format, reading, writing, signing and encryption.
- The root certificate of the IsoFind authority, to obtain once and keep.
- IsoFind, the software that produces these files. It is needed neither to read them nor to verify them, and that is the point.

A doubt about a file you received. The first useful reflex is not cryptographic: ask the signer to send the file again, and compare the two. A difference between two sendings shows up immediately, and it tells you more than a digest.