Plectis: What a Stranger Can Check

What public evidence can and cannot show about a private system

Will Cook

July 2026

Abstract

This paper treats one named version of Plectis as a single case in a bounded descriptive analysis of testing public claims about software that cannot itself be published. The author releases selected code, fixed inputs, expected results, decision rules, and run records while the larger system remains private. What can a stranger conclude after rerunning those tests? The answer is narrow. The repository lets a reader inspect and rerun its published procedures. A matching result shows repeatability for a stated public case; it does not establish where the code came from, whether the expected answer is correct, whether the published selection resembles the private whole, or whether the private system works reliably.

The contribution is a method for making one selected disagreement local: the registered components (separately testable parts) place a prose claim beside a fixed test case, runnable procedure, expected result, decision rule, run record, and explicit inference limit. The repository calls this package a claim–evidence–limit contract. It is an executable-disclosure case study, not an evaluation of reviewer performance. A worked component calculates an audio level from five small sample arrays. A reader can repeat the calculation and, for one row, derive the expected number independently of the implementation. Even there, the run cannot show whether the pass rule tests the claim as written, how the code behaves on untested inputs, or whether the chosen formula is the right account of audio level. I chose the published fragments, cases, answers, rules, and limits. The paper therefore treats the repository as an author-curated test collection, not as independent validation of the private system. I report that its published material was copied or adapted from that system and that I developed it by directing coding tools based on artificial intelligence (AI). The argument does not depend on that account or on the claimed origin of the published material.

1 The problem

I have a larger private software system that I cannot responsibly publish: it contains personal records, credentials, live browser and account information, my working notes, and unfinished tools. I would still like strangers to be able to test some claims about it instead of accepting my say-so. I use *answerable* in a deliberately local sense: a claim is

Authorship and generative-system use. The prose in this version was generated by large-language-model agents under Will Cook's direction. Cook set the objectives and acceptance criteria, reviewed and authorised the manuscript, and is responsible for its contents. The agents are production tools, not authors. See *Statements and declarations*.

answerable when a stranger can run a public test that bears on it, see how the result is judged, and locate a disagreement in a named input, rule, or output. This is not a new synonym for truth, reproducibility, verification, or independent review. For the rest of the paper I use the plainer phrase *publicly testable claim*. The question is what conclusions such a test can support when the larger system and the unpublished pool from which the test was selected remain invisible.

Plectis is my working answer: a public software repository, stored at `github.com/wcook04/plectis`. A *repository* is an organised project folder, usually kept with its revision history; to *clone* a repository is to copy that folder and history to another computer. Everything this paper says a stranger can check is in that repository. This paper describes how its claims are exposed and where the evidence stops.

The main result is a boundary, not a score. Public code, inputs, rules, and run records can make a selected claim easier to challenge. They cannot by themselves establish three missing premises: that the public code came from the private system, that the project's expected answers and pass rules are correct, or that the selected cases represent the unseen whole. The audio example in Section 4 shows all three limits in one elementary calculation. The later inventory records what was published; it is not evidence of breadth, quality, or independence. Runeson and Höst likewise treat configuration-managed project documents as archival data while warning that records created for another purpose may omit needed information and be difficult to assess without complementary sources [1, §4.4, p. 148]. The repository therefore lets a reader test claims about the public objects it actually exposes, within the limits stated for each test.

Who made the decisions. I built the private system by directing AI coding agents: I set the objectives and review criteria, accepted or rejected their output, and remained responsible for the cases, expected answers, decision rules, and final public wording. No other person worked on the system. This account explains the concentration of control, but it remains testimony about origin: Plectis cannot independently prove how the private work was produced, and the public repository does not prove the story. The same evidential limit would hold if every line had been written by hand. One process can give an implementation and its check the same mistake; this paper compares no development methods and estimates no error rates.

The boundary of the evidence can be stated before any command is run:

Status	What belongs in it
Publicly checkable	The contents of a named public version, the commands listed for its components, the files those commands write, and the results of its public tests.
Reported here	The private system's history, the role of AI agents in producing it, and my claim that the published material came from the private system.
Not established here	Whether the public selection represents the private whole, whether the published code works beyond its fixed inputs, whether the project's own pass rules are correct or even test the stated claims, and whether the private system works reliably at all.

What I examined. The paper examines one author-curated software collection. At the named repository version, I read the list and plain descriptions of all 88 components, counted them by project group and by the kind of public evidence offered, traced the worked component from input to limit, and reran the appendix's public commands. I did not compare every pass rule with its prose claim or derive fresh expected answers; Section 8 reserves those checks for an outside evaluator. The paper defines the component contract, then follows one ordinary component from input to limit. Five distinctions separate what a passing run shows from conclusions it may appear to support. The final sections identify the choices an evaluator would need to make independently of me.

This is a descriptive analysis of one public repository, not a statistical study. Runeson and Höst define the case as the object of study and its units of analysis as items contained within it [1, §2.5, p. 138]. They also note that this boundary depends on the study context and research goals [1, §3.1, p. 139]. I borrow only those two terms: the named repository version is the single case, and each component is a *unit of analysis* (one item examined inside that case). This paper does not otherwise claim a full case-study design. No subset of the components is used to estimate performance or represent the private system.

Reading this paper requires no programming. Reproducing its runs requires basic command-line use, a terminal (a text window for commands), Git (for copying and versioning repositories), Python 3.11 or later, and `pip` (for installing Python software). The appendix gives the exact commands, environment details, and installation notes. It is a dated record of one verification pass at the version named there.

2 One public test before the general design

Consider four decimal audio samples:

$$0, \quad 0.05, \quad -0.05, \quad 0.1.$$

The published procedure squares them, averages the squares, takes the square root, and multiplies by eight. Their squared average is 0.00375, so the stated rule gives $8\sqrt{0.00375} = 0.489897\ldots$. The public program produces 0.489897949. Anyone with a calculator can reproduce the expected number without trusting the program.

That agreement establishes something real and small: this public program agrees with the stated formula on this test case. It does not establish that the formula is the right account of audio level, that the public code came from the private system, that other inputs behave similarly, or that important risks were selected for testing. Those conclusions require additional premises that the observed match does not contain.

The paper asks the same six questions of every published item:

1. What sentence is offered for belief?
2. What public object or procedure was checked?
3. Who supplied the interpretation or expected result?
4. What does the observed match establish?
5. What remains unestablished?

6. Which independently controlled evidence would strengthen it?

Section 4 gives the full command, all five audio cases, and the saved run record. We now generalise from this example to the project's common record format.

3 The component contract

A *component* in Plectis is one registered, testable part of the project. Its *fixture* is a fixed sample input: in ordinary testing language, a test case. A *receipt* is the saved record of a run. The repository supplies one from an earlier run; rerunning can write another. A *validator* is the program that applies a component's pass and refusal rules. In testing terminology, the rule and expected result together act as the test oracle: they say what should count as correct for that case. A *commit* is a named, saved version of the repository. These are Plectis's internal names for familiar objects—test case, test oracle, run record, and saved software version—and the paper uses the familiar terms whenever the internal filename is irrelevant.

Here *evidence* means the public input, procedure, output, and judgement rule offered in support of a claim. Calling them evidence does not establish their adequacy. *Contract* means only that the claim, those materials, and a limit are stated together so readers can check their fit. Nothing in these labels grades the importance of what a component does.

At commit `57ffb7f5830c` the registry, the component list read by the programs, contains 88 entries. Every entry is required to connect the same things: a claim stated in prose; a fixture; a command; the output files the command writes; a stored receipt from a prior run; an explicit rule for what counts as a pass and what must be refused; and a stated limit on what a pass establishes. Figure 1 lays the required parts out, together with the fact about them that the rest of this paper keeps returning to: every part is supplied by me, and every part is exposed to a stranger.

The repository cannot by itself support my assertions about the unseen private system. A component does something narrower: it gives a related public claim a procedure that can fail in front of a stranger. The procedure does not make the claim true or establish the published fragment's origin. It changes the form disagreement can take: a reader who doubts a component's claim can point at a particular input, a particular line of the validator, or a particular number in the output, instead of arguing with my description of software they cannot see. A component that fails is still answerable in this sense, and a component can be answerable and trivial; answerability concerns the form a claim takes, not its truth and not its importance.

A component may also end in a declared refusal rather than an answer. The worked example below refuses an input format it does not support. Components that depend on an external tool, such as the Lean proof checker (a program that verifies mathematical proofs), must record a refusal with a stated limit when the tool is absent, rather than reporting success. *Runnable* therefore means that the operation and its declared failure boundary are present in public form; it does not mean that every optional dependency exists on every machine. A refusal is informative only when it is declared in advance and distinguishable from a breakdown: a predeclared refusal marks the edge of what the component claims, while an unexpected crash is a defect. The two must not be read

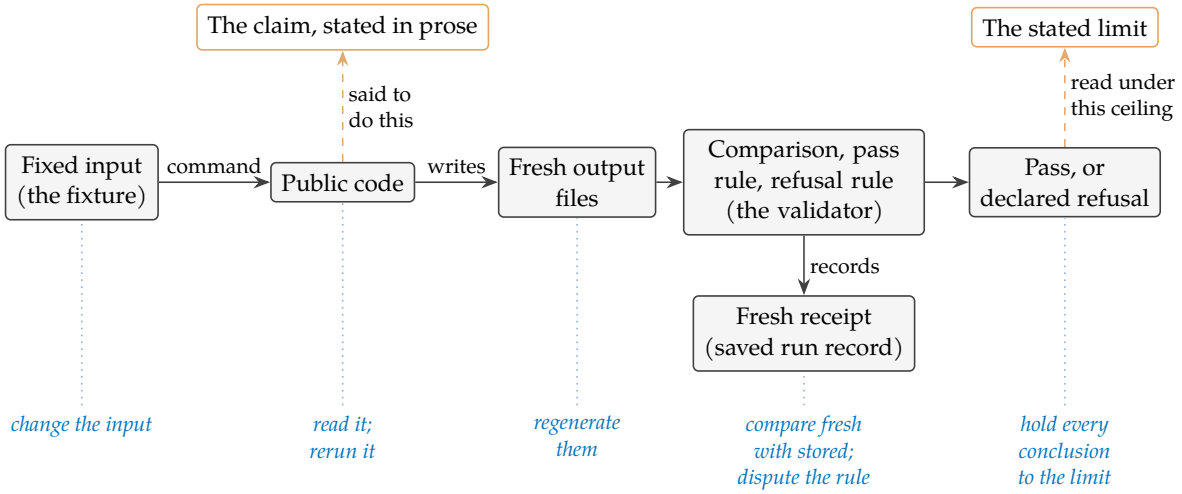

Figure 1: How to read the component contract, at commit `57ffb7f5830c`. Read the middle row from left to right: a command applies public code to a fixed input and the code writes output. The validator judges it under the published pass and refusal rules, then writes a fresh receipt recording the run. The stored receipt from the earlier run is available for comparison; it does not determine the new verdict. The two boxes above are words rather than mechanisms: the project states the claim and the limit on what a pass may support. Every part is public, and the project supplied every part. The italic blue row names what a stranger can do without asking anyone; amber marks the project's words and choices. Across the figures, a dashed outline marks something outside what the public run establishes. This figure has no dashed boxes because none of its parts is hidden. In Figures 2 and 3, dashes mark private material or conclusions not established by the observed run. Section 4 follows one component through this chain.

as the same kind of event. One honesty condition on refusals cannot be discharged from inside the repository at all: the public record shows each boundary as declared at the named commit, and it cannot show whether the boundary was drawn before or after an awkward case was first met. A boundary drawn afterwards turns a failure into a designed refusal in the telling. Saving the version before an outside evaluation begins, as Section 8 proposes, is what would make that ordering checkable.

The idea does not depend on software. The contract needs a claim in words; a determinate public case; a procedure a stranger can cause to run; an observable result; a rule connecting result to claim; a declared boundary of application; and a stated ceiling on inference. That the case here is a file of numbers, the procedure a command, and the record a text file is a convenience of this example. The same contract could be satisfied in another field by a laboratory protocol with declared rejection conditions, or by a sealed evaluation service with published scoring rules. I make no claim about those settings; the point is only that the contract is a shape for exposing claims to strangers, and this repository is one example. Read simply as software, it is a test suite with a registry and unusually complete paperwork. That is a fair description. The paper asks what such a test suite lets a stranger challenge, and which conclusions still lie beyond it.

This local notion overlaps with reproducibility and transparency but is not a synonym for either. A run can repeat perfectly with no stated claim attached, and a claim can be answerable while still awaiting a good test. The design also begins from partial disclosure: it offers one limited point of contact, not a view of the hidden system. A component exposes the published fragment to challenge. It does not make the private whole answerable.

Answerability is not the same property as computational reproducibility, although the component contract supplies materials needed to test the latter. Under the convention used by the National Academies, computational reproducibility means obtaining consistent results using the same input data, computational steps, methods, code, and conditions of analysis [2, conclusion 3-1]. The report separately cautions that exact reproducibility does not guarantee computational correctness: undetected erroneous code can reproduce the same erroneous result [2, p. 9]. In software testing, a *test oracle* is a procedure used to distinguish correct from incorrect behaviour. The challenge of making that distinction is called the test oracle problem [3, p. 507]. Barr et al. distinguish a partial test oracle D from a conceptual ground-truth oracle that always gives the right answer, modelled as a total oracle G, while observing that such ground truth is unknowable except in trivial cases [3, Defs. 2.4 and 2.6–2.8, pp. 509–510]. Soundness requires $D(a) \Rightarrow G(a)$, and completeness requires $G(a) \Rightarrow D(a)$. A passing validator therefore applies its published rule; it does not by itself establish that the rule is sound. The risk that a curated collection disproportionately exposes favourable cases is analogous to selective publication and the file-drawer problem [4, p. 638]. An *assurance case* links system claims to arguments and evidence. The Object Management Group's Structured Assurance Case Metamodel (SACM), version 2.3, defines such a case as a collection of auditable claims, arguments, and evidence. It separately represents an artifact citation and the *AssertedEvidence* relationship said to connect that artifact to a claim; the relationship is itself an assertion by the case's author. Representing an argument is therefore not the same as establishing its validity [5, secs. 1.1, 4, 7.3, 11.9–11.15]. The GSN Community Standard states the boundary more directly: GSN documents an asserted chain of reasoning and its evidential references, but the notation itself establishes neither the truth of the argument nor the sufficiency of its support [6, §0:3.2, p. 11; §§0:4.11–0:4.13, p. 15]. Plectis is not an implementation of that standard or of GSN, and makes no claim that the private system is safe. Its validator applies a published rule and records a result; it neither models nor independently justifies the evidential link from that result to the prose claim. The Association for Computing Machinery (ACM) calls digital research materials such as code, scripts, and data *artifacts*. Its *Artifacts Evaluated—Functional* badge means those materials passed an independent audit for documentation, consistency, completeness, ability to run, and evidence of verification and validation. The NISO recommended practice separates object review, reproduction with the author's objects, and replication by a new study [7, secs. 2.2–2.4]. ACM's harmonised current names are *Results Reproduced* and *Results Replicated*; its badges are independent, and results validation requires a peer-reviewed report [8]. Plectis claims neither status: its receipts are author-generated, not independent review or validation. These standards are comparison points, not statuses awarded here. This paper claims no new theory of assurance,

testing, or reproducibility; it reads each public procedure against its claim and stated limit.

4 One run, examined

The component `batch8_audio_level_rms_port` computes a loudness level from short arrays of audio samples. I chose it because it is ordinary. It needs only a square root and no knowledge of the private system; its claim is small enough that every part of the evidence can be seen at once.

The reported origin is this. The private system contains a small macOS recording application, and one of that application's functions reduces a short stretch of audio samples held in memory to a level between zero and one. The public component re-expresses that calculation in Python. The component's code and its stored receipts name the private file it claims to re-express (`AudioLevelMonitor.swift`); a stranger cannot open that file, so the name is a report, not evidence. What the stranger can check is everything else.

The declared calculation begins with root-mean-square (the `rms` in the component's name): square each sample, average the squares, and take the square root. It then applies the project's gain of eight and caps the result at one. Signed sixteen-bit integer samples are first divided by 32,767, the largest positive value in that representation, to bring them onto the same scale. An empty input must produce zero. An unrecognised format name must be refused rather than guessed at.

The component's public command runs it against its fixture and writes fresh output outside the repository (the backslashes split one long command across lines):

```
PYTHONPATH=src python3 -m plectis \
  batch8-audio-level-rms-port run \
  --input fixtures/first_wave/batch8_audio_level_rms_port/input \
  --out /tmp/plectis-audio-rms \
  --acceptance-out /tmp/plectis-audio-rms-acceptance.json
```

At commit `57ffb7f5830c`, the run produced:

Case	Input	Expected	Observed	Result
Decimal samples	0.0, 0.05, −0.05, 0.1	0.489897949	0.489897949	pass
Sixteen-bit integer samples	0, 1638, −1638, 3277	0.489892970	0.489892970	pass
Values above the cap	1.0, −1.0, 0.5	1.000000000	1.000000000	pass
Empty input	(none)	0	0	pass
Unknown format	`pcm24`	refusal	refusal	pass

Here *pass* means "refused as expected"; `pcm24` was not accepted. Because the inputs and the rule are both printed above, the first row can be checked without the repository or a computer: the squares of the four samples average to 0.00375, whose square root

is 0.061237 …, and eight times that is 0.489897 …. A reader with a calculator can recompute this expected value from the stated formula alone. For this one case the reader can supply the test oracle: elementary arithmetic gives an expected number derived independently of the program's output, but not of the project's formula.

Almost nothing else in the collection is like that, which is why the example matters. Elsewhere the project usually supplies the implementation, the input, the expected values, and the validator through the same author-controlled process. A test assembled that way can pass for four different reasons: the implementation and the expectation are both correct; the two share a mistake; the chosen cases happen to sit where the implementation behaves; or the validator checks a weaker property than the prose claim suggests. A matching result therefore establishes repeatability under that public test. Independent evidence bearing on correctness needs an expected result justified apart from the implementation. Arithmetic supplies that for the first row only after accepting the project's formula. Fresh inputs alone do not supply one.

The example suggests a question for any software test: where did the expected value come from, and who supplied it? The answer changes what a match can support:

Who or what supplies the expected answer	**What a matching result then supports**
The project itself: implementation, cases, expectations, and rule from one author-controlled process.	Repeatability under the project's own test; the four readings above all stay open.
The reader derives a result from stated premises, as in the decimal-samples row.	Agreement with the stated formula on that case, not proof that the formula is right.
An established external tool or reference.	Whatever the reference itself warrants, on the cases run, within the reference's own limits.
An evaluator who chooses inputs and independently justifies expected results after the version is saved.	Independent evidence bearing on correctness for those cases, within the oracle's own justification and limits.
The world: an outcome observed apart from any program.	Evidence about a real-world outcome, where one exists.

Most components in the collection sit in the first position, except where a reader promotes a case to the second by recomputing it, as the worked example's decimal-samples row invites. Components that invoke a named external tool occupy a hybrid of the first and third positions: the project still chooses the case and the rule, while the tool supplies a result that inherits its own limits. The evaluations in Section 8 describe ways to move consequential choices out of the project's control.

`batch8_audio_level_rms_port_validation_receipt.json` is the stored reference for the table. It sits under `receipts/first_wave/batch8_audio_level_rms_port/`, so a fresh run is compared against a fixed record rather than against memory. A stored record by itself does not prove that the stated command produced it; its use is that anyone can generate a fresh one and compare the two.

The limit is part of the component. This run checks one public Python calculation over synthetic sample arrays (arrays composed for the test, not captured from a device). It does not run the original application, start an audio session, request microphone permission, or capture sound, and it cannot establish that the named private file is where the calculation came from. Nor does the table show that the task was difficult (it was not), that the component is useful elsewhere, or that the remaining entries resemble it. What it shows is that one assertion now has a form in which a stranger can rerun it, dispute its rule, perturb its input, or replace my expected value with a better-derived one.

The calculator check derives one expected number without the implementation. It shows agreement with the project-supplied formula on that case, not that the formula is the right audio rule. It does not establish origin, reach, or risks beyond the declared scans. Its correctness evidence is limited to that calculation. The next section treats these five gaps separately.

5 Five distinctions

Five labels organise the gaps in Figure 3: *origin*, where the material came from; *correctness*, whether an answer is right; *meaning*, whether a rule tests its claim; *reach*, whether shown cases stand for unshown ones; and *risk*, what the declared checks may miss. These short labels correspond to familiar questions about provenance, test-oracle validity, claim–test fidelity, representativeness, and residual risk. They are not five new kinds of evidence. Each subsection separates what the public run directly shows from the additional premise needed for a stronger conclusion.

Public execution versus where the code came from

A reader can establish what a named public commit contains and what its commands do. No public run establishes where the material came from. Plectis records that boundary. Each copied file has a claimed source and a SHA-256 message digest, used here as a fingerprint: a 256-bit value calculated from its contents and used to detect change [9, abstract, p. iii; §1 and Fig. 1, p. 3; §6.2, p. 21]. Two different files can in principle have the same digest: FIPS 180-4 treats finding such a pair as computationally infeasible, not mathematically impossible [9, p. iv]. For this comparison, the relevant question is whether one can start with a specified file and find a different file with the same fingerprint. The National Institute of Standards and Technology (NIST) calls this second-preimage resistance and expects an approved hash function to make such a search computationally infeasible. That differs from collision resistance, which asks only for any pair of different inputs that match [10, security-strength table]. A match therefore supports a narrow claim: the public file's contents agree with the version represented in its public record. But I control both sides, so that agreement is evidence of internal consistency, not of origin. The same holds for the audio component's named source file: the name is an entry I wrote. Origin claims could only be strengthened by something a public repository cannot supply about itself, such as commitments published before the fact, or a trusted person permitted to examine both sides. The term *provenance* means an object's origin and history. Here, that origin remains reported rather than publicly

established. FIPS 180-4 itself similarly cautions that conformance to the standard does not assure that a particular implementation is secure [9, qualification 12, p. v].

Repeatability versus correctness

A run that yields the same checked values and verdict as the stored receipt shows that the public code, on the supplied input, in a sufficiently similar environment, produces the same result again. Correctness is a different property: it needs a standard for what the result should have been, and a standard is only worth something if it does not merely restate the implementation. The worked example makes the difference concrete. The first row's expected number comes from arithmetic applied to the project's formula, not from the implementation. That supports agreement with the formula on one case; it does not validate the formula. Most components have only the project's own expected values and validator. Passing tests anywhere in Plectis should be read at that strength and no more.

A validator's rule versus the stated claim

A pass is a verdict from the validator, and a validator checks exactly the property written into it, which can be narrower than the claim as worded. A component whose prose speaks of handling a class of inputs may in fact verify that one output file exists and matches a stored shape; a rule can hold while the sentence it stands under goes untested. The two previous distinctions do not cover this gap: a result can be repeatable, and correct as far as the checked property goes, while the checked property is beside the stated point. Plectis makes the gap inspectable rather than closed. Every validator is public, so a reader can put the rule beside the prose and object where the rule is weaker, and the review in Section 9 makes exactly that comparison its central step. Since the same author-controlled project produced both the claims and the rules, agreement between them records one judgement twice, and no count of passing components can substitute for reading one rule against one sentence.

Selected cases versus general behaviour

Selection acts at two levels. Within a component, the fixture covers only a few possible inputs. Across the collection, the published components are my selection from a private whole.

I chose which mechanisms could be given public form, which inputs to include, which expected values to store, and which pass rules to enforce. Nothing in the public record shows how many candidates there were, what was excluded beyond the privacy rules, or whether the collection resembles ordinary work in the private system. A collection can consist entirely of genuine items and still give an unrepresentative picture of what it was drawn from. This is analogous to selective publication and the file-drawer problem, not the same process [4, p. 638]. Figure 2 draws the situation: public checking reaches everything to the right of the boundary, and the pool the selection was drawn from stays on the left.

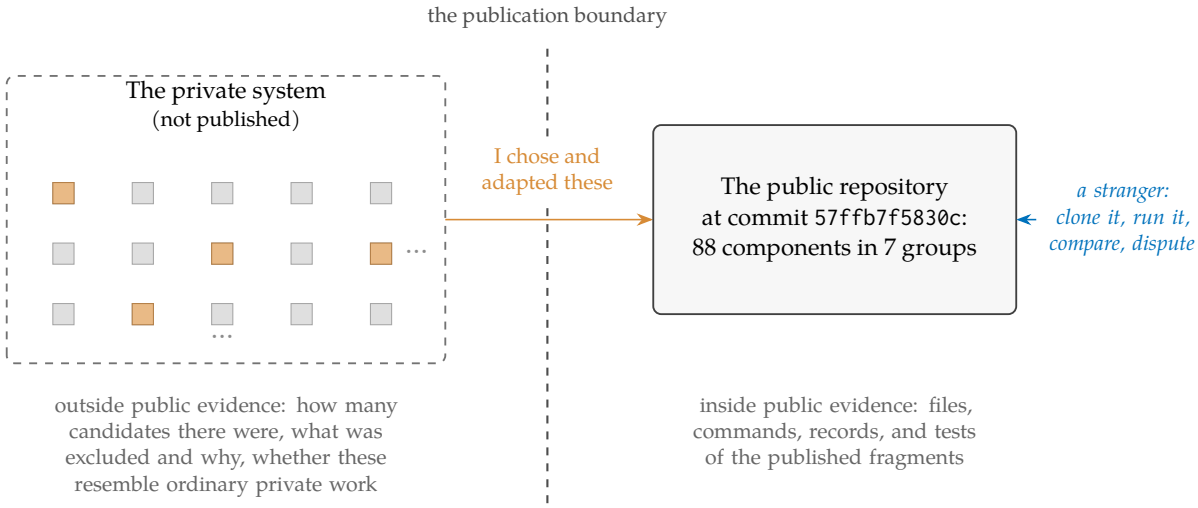

Figure 2: The selection problem. Each square stands for a private mechanism; the darker ones were given public form. Each public component has the parts shown in Figure 1. The grid trails off because its true extent is exactly what the public side cannot know. The count of squares on the left, the rule that picked the darker ones, and the resemblance between the two sides are all invisible from the right. The collection supplies directly checkable evidence about the published files and their behaviour under named tests, but no basis by itself for generalising to the hidden whole.

The registry's count should be read in that light. A registry of 88 entries defines exactly what is published. A later evaluation would need that list to draw a sample and report failure rates against a fixed total. The count is an inventory, not a score, and even the unit of counting is a choice I control, since one mechanism could have been registered as several components, or several as one.

Risk reduction versus guarantee

The publication process runs checks whose honest description is that they reduce specific risks. Automated scans search copied material for specified patterns of credentials, personal information, and private file paths, and record what was omitted or rejected; a scan establishes that the scanner did not find the patterns it was designed to find, and nothing stronger. Pages such as the component map are generated from the registry, and tests compare page and registry, which prevents the particular failure of prose drifting away from records; a generator and its records can still be wrong together. The documented `tour` run takes the public checkout as its explicit input and does not automatically discover a neighbouring private repository. The release-export contamination check is an explicit exception: when a copy of the private project folder is available, it may compare the contents of public source files with private files to detect leakage. None of this amounts to a guarantee that no secret, no identifying detail, and no inconsistency survives. Publication remains a judgement about acceptable risk, and I made it.

Origin evidence and privacy pull in opposite directions. Establishing provenance would require someone to examine private material, and the privacy constraint that

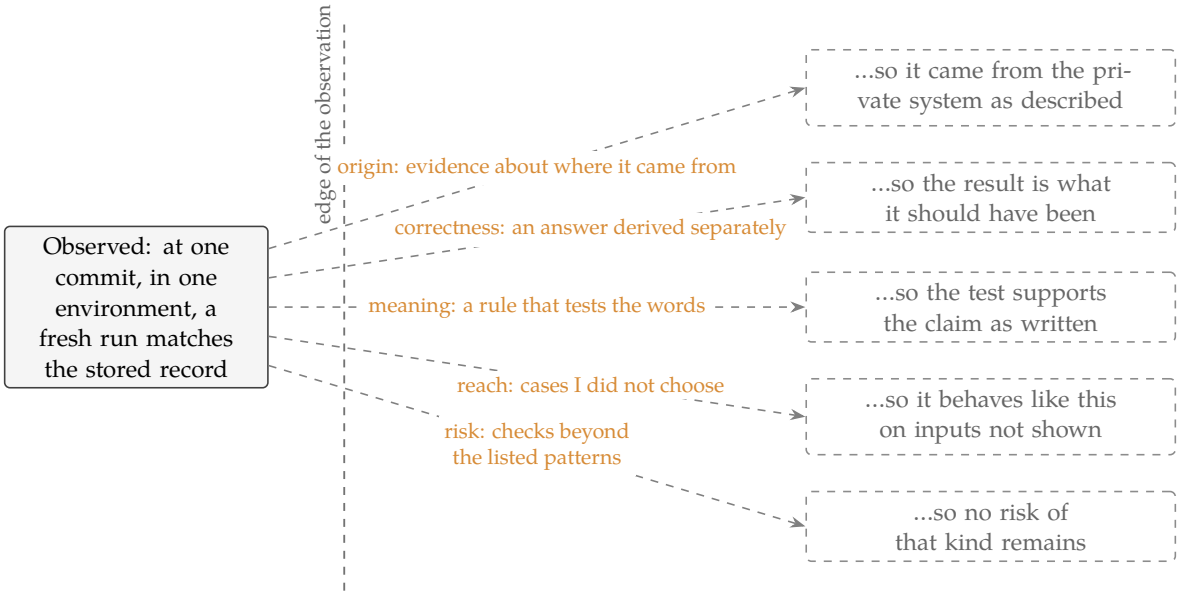

Figure 3: Five conclusions a matching run invites and does not license. Each dashed arrow needs a further assumption that the observation itself cannot supply. From top to bottom, the tags follow the order discussed above: origin (provenance), correctness, meaning, reach, and risk. The contract's contribution is not to cross these gaps but to make them nameable, one component at a time.

motivates the whole design forbids exactly that examination in public. Some claims are therefore permanently in the reported category, short of a confidential review. The honest treatment is to label them, not to dress them up as public proof through bookkeeping that one maintainer controls.

The same mistake lies behind all five: treating an observation as if it proved something never tested. A broader conclusion needs an added assumption. For origin, one must assume that the public object came from the private one. For correctness, one must know that the expected answer was right; for meaning, that the rule tested what the claim said. For reach, one must assume that the chosen cases stand for the unchosen; for risk, that the scan covered the relevant risks. The observation itself supplies none of these assumptions. The contract makes each one explicit. Once named, a gap can be disputed, assigned a cost to address, or deliberately left open. An unnamed gap is easy to cross without noticing. These five are not exhaustive. Two more gaps appeared earlier: observing a refusal today does not show when it was drawn (Section 3), and observing a number at one commit does not describe the next (the appendix). These are simply the five gaps this collection most needs a reader to hold.

6 What the collection contains

This section is a dated inventory, not a result. The categories are useful for finding a test and for seeing which kind of limitation accompanies it. Their names and counts do not measure importance, correctness, independence, or coverage of the private system.

At commit `57ffb7f5830c` the 88 components fall into 7 groups:

Group	Count	What its components do
Getting started	2	Show a new reader where to begin and guide a first inspection.
Project mapping	12	Map files and written rules, find relevant material, and route a question to the part of the project that owns it.
Computer-checked mathematics	20	Check records and fixed examples from formal mathematics, including small Lean statements and one exact finite calculation.
AI reliability and safety tests	20	Replay fixed cases involving hostile instructions, software boundaries, stored information, tool use, and benchmark rules.
Research-method tests	9	Exercise fixed cases involving replication, conflicting forecasts, explaining model behaviour, simulation, and laboratory safety.
Public-copy maintenance	20	Check copied source, generated pages, publication boundaries, and disagreement between records and derived files.
Work recovery	5	Record unfinished changes and test how work resumes after interruption.

The groups do not share a subject. Their only unity is the contract of Section 3: each item became a small public test. The collection is organised by how claims are checked, not by a common scientific question.

The registry also classifies what kind of evidence each component supplies. It calls this classification a *route*. A route says whether the public component preserves source text, re-creates behaviour on fixed cases, records a run, or checks a declared rule. The four routes below cover all 88 entries. Each leaves a different question open:

Route	Count	What it preserves	What it cannot show
Record-checked copied source	21	The exact text of non-secret source files, checked against a public import record.	That the recorded origin is real; record and file share one maintainer.
Public reimplementation	27	One calculation's behaviour on fixed cases.	That it came from the private system; the private text is absent.
Direct or external-tool run	22	A local computation, or a named external tool's result, captured with its context.	Behaviour where the tool or environment differs; the result is as strong as the tool and the case.
Contract checker	18	Whether a public input satisfies a declared structural rule.	Whether the declared rule is the right rule to require.

Copied source preserves text but demonstrates little behaviour; a reimplementation does the converse; an external-tool run inherits the tool's limits; and a contract checker is worth exactly the rule its author chose. To inspect one item, use the public component map, `ORGANS.md` (an inherited filename). Follow it rather than guessing, because some components share a page. The audio row points to `paper_modules/batch8_audio_level_rms_port.md`, which in turn names the fixture, receipts, route, and limit.

Two cautions govern the tables. First, components are not independent witnesses: they share code, fixture style, authorship, and judgement. 88 entries are registered claims, not 88 confirmations. Pages, entries, and receipts generated from the same underlying records repeat one witness rather than supply several. Machine checks can catch their accidental disagreement. They cannot create an independent witness. The four routes impose four different limits on what may be concluded; they are not four independent methods confirming one claim.

Second, names such as computer-checked mathematics and AI reliability and safety tests borrow gravity from serious fields. Read each row as the weakest property its tests actually check—often that a fixed case replays or a record has a stated shape. The names help find components; they do not weigh them.

7 The author's hand

All five distinctions return to control. At the commit this paper describes, I hold every consequential choice on both sides of every test. I chose which private mechanisms were candidates, which were selected, and how finely each was divided into components, and so what the count counts. I chose which of the four routes each took, which inputs went into the fixtures, what the expected values are, what each validator checks, and where each refusal boundary sits. And I chose which group each entry is filed under, which failures this paper explains and in what tone, which single component the worked example honours, and where the publication boundary lies. Somebody had to make each of these choices, and the privacy constraint meant it was me. None of that is misconduct.

But that concentration of choice permits a failure mode, and it involves no false statement anywhere. Select the mechanisms that show well. Divide the favourable ones finely. Narrow each claim until its fixed cases satisfy it. Give awkward material the least demanding check. Declare as refusals whatever would otherwise fail. File small replays under weighty names. Then let the counts, the categories, and the machinery of records accumulate into an impression of scale and maturity that no individual sentence asserts. A document can be cautious sentence by sentence while its arrangement still overstates, because a reader's impression forms on what is counted, named, worked through, and left out, as much as on what is claimed. This paper is inside its own arrangement, so declaring the risk does not discharge it; nothing in the text can establish that the text is not doing what this paragraph describes.

Making disagreement local can also displace the larger question. A reader can settle an argument about one input or one rule while leaving untouched whether the selection is fair, whether the whole is sound, or what any of it is for. An objection about selection is not answered by the successful execution of any number of selected items. The registry hosts the small arguments; it does not settle the large ones.

Two corrections need no cooperation from me. First, read any component at the weakest property its validator checks, as Section 6 proposes. Second, treat every count as an inventory, and every agreement among my records as one witness. A third point concerns what no reader can inspect: a private mechanism is easiest to publish when it can be separated into a small task, gives the same output from the same input, and runs in isolation. A collection like this therefore over-represents pure calculation, replay, and record-checking, and under-represents stateful, entangled, long-running behaviour. The published picture tilts towards what publishes well before any question of honesty arises.

The National Aeronautics and Space Administration (NASA) defines independence for independent verification and validation (IV&V), a formal software check, through three separations: technical (the evaluator did not develop the system); managerial (a separate organisation chooses scope, methods, and schedule); and financial (an independent group controls the budget without adverse financial pressure) [11, §4.4.1.2, p. 48]. These are criteria for NASA IV&V, not a universal test. Plectis has undergone no IV&V. No other outside evaluation report is cited or included, and I know of none; an unreported private rerun would remain invisible. I use the NASA criteria only for the paper's five gaps: evidence about a choice depends less on me once the choice is no longer mine. Section 8 lists evaluations that address those dependencies.

8 What stronger evidence would look like

A passing run shows that named public code produced a stated result from a supplied input in a stated environment. It does not establish private provenance, representativeness, correctness beyond self-supplied criteria, usefulness, security, originality, or the private system's reliability. The computer-checked mathematics label covers different checks: proof-work records, small public statements checked by Lean, and one exact finite calculation. The label itself supplies no theorem; each entry supports only its own public claim and stated limit. At the named commit, the repository's evidence still came

from the project. It includes no outside evaluator's report; I know of none, although a private rerun could be unreported.

Five forms of independent evaluation could add evidence that the public repository cannot supply by itself. These are not the four publication routes in Section 6. A publication route classifies evidence already in the registry; the evaluations below describe new work by someone outside the project. The order is only for exposition. It ranks nothing, and no evaluation is a prerequisite for another. Figure 4 shows which choice each evaluation transfers and which claim it still leaves open. The common question is who makes the choice that matters for that claim.

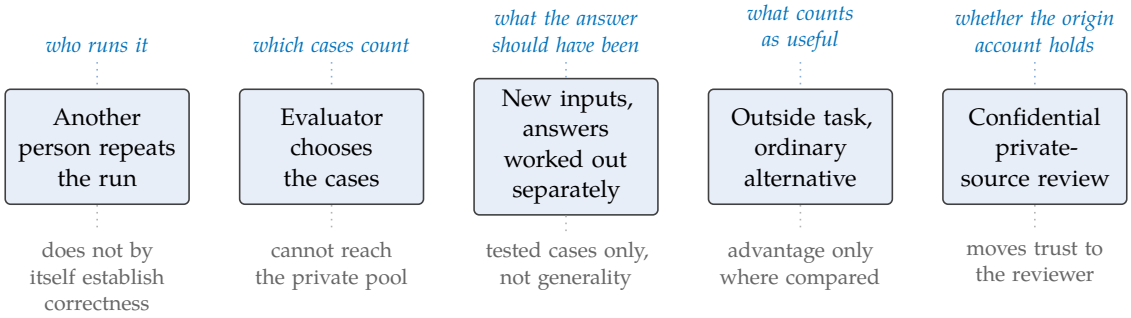

Figure 4: Five forms of independent evaluation. Above each evaluation, in italic blue, the choice that leaves my hands; below it, in grey, what that evaluation still cannot show. The evaluations are not cumulative stages. At the commit this paper describes, the repository records none, and I know of none.

Independent repetition means a person with no involvement in the project reruns the supplied commands in their own environment and publishes what happened, including failures. That would reduce dependence on my machine and test whether the published instructions suffice elsewhere. It would not by itself establish correctness.

Evaluator-controlled selection means the evaluator chooses components from the full registry, without substitution when one proves awkward, and original failures stay on the record even if I later repair them. The units themselves are open to the same challenge: an evaluator who finds that two entries are one dependent mechanism, or that a boundary excludes the difficult part of a task, is disputing the registry, not misusing it. That addresses curation within the public collection. It cannot address whether the collection represents the private system, because the private pool stays invisible.

Fresh inputs with independently derived expectations can move repeatability towards evidence bearing on correctness: inputs chosen after the version is saved, and expected results worked out by a method that does not pass through my code, whether independent calculation, a second implementation, or an external reference. The result would support claims about those cases only to the extent that the derivation is credible and independent and the cases are adequate; general correctness still would not follow.

Comparison on outside tasks is what a claim of advantage over ordinary alternatives would need: someone bringing a task that was not designed around the project's own categories and comparing the component against an ordinary alternative, which for sev-

eral components would be a short conventional script. Nothing in this paper makes that comparison.

A provenance review, among the five evaluations shown here, bears most directly on the origin story: a trusted person examining selected private material against the public collection under confidentiality. Prior public commitments could provide another kind of provenance evidence. Even completed, this review would transfer trust rather than remove it: public readers would be relying on the reviewer's access, competence, and report instead of on mine, and the review itself could not be published. The privacy cost may reasonably never be paid. If it is not, the origin story remains testimony, and this paper is written so that its argument does not depend on it.

A falsifiable comparison

The next study should compare two presentations of the same frozen public items. The baseline would provide ordinary documentation, code, outputs, and tests. The comparison condition would additionally provide the explicit claim, test case, expected result, decision rule, run record, and inference limit used here. Evaluators who did not design the items would choose from the frozen public list, inspect deliberately mismatched claim–rule pairs, use fresh inputs where an expected result can be justified independently, and publish failures without author substitution.

The useful observations would be whether readers detect a mismatch, how often they wrongly accuse a matching item, how long they take to locate the point of disagreement, whether they reproduce the run, and how well their confidence tracks those outcomes. That study would measure reviewer behaviour on the public items. It would still not measure the private system, the unseen selection pool, or correctness beyond the independently justified cases. No such comparison is reported in this paper.

From an open question to a falsifiable handoff

The five evaluations still leave an outsider to infer what the project expects, reconstruct credible alternatives, and decide which observation would matter. "Evaluate Plectis" transfers that framing work; a confident prediction can make the project's own expectation look like evidence.

The current repository, beyond the historical snapshot examined above, therefore includes a small, read-only *hypothesis-handoff* format. A packet names one declared known gap and refuses to treat it as a complete inventory of unknown questions. It then names the current wall, a tentative leading hypothesis, evidence for it, evidence against or still missing, serious alternatives, and discriminators whose possible outcomes point to named hypotheses and have stated interpretations. It also names an exact expert return, decisive and route-only effects, repository-relative landing targets with validators, and the order by which a return could enter an authority record and public release. Run

```
plectis hypothesis-handoff --input packet.json --format text
```

validates that structure and renders it as a briefing. It does not call a model, infer probabilities, judge an expert, or change a claim. This later interface is not evidence for the historical snapshot analysis.

The worked public packet asks whether evaluator-selected cases would expose a different failure profile from the author-selected fixtures. Its working hypothesis is that they would expose more failures and a broader taxonomy. The reasons are visible: I selected the existing cases and rules, and a later cold route exposed one real wiring defect. The counterweight is equally visible: there is no independent comparison corpus, and one defect estimates no rate. Two alternatives remain live: the current fixtures may approximate the independent profile, or environment and dependency failures may dominate the difference. A preregistered case-selection rule and a failure taxonomy fixed before outcomes are inspected would separate those accounts better than another author-selected example.

This format does not make the leading hypothesis independent. It is the project's expected answer, in the same sense that Section 4's expected output is project-supplied. Its contribution is different: it makes the prior, alternatives, and update rule inspectable before an expert answer is known. A reader can reject the leading hypothesis, add a missing alternative, or dispute a discriminator without first reverse-engineering the project's private reasoning. If an answer arrives, the packet also prevents the answer from becoming a public conclusion merely because it came from an expert. The return remains advisory until its evidence is reproduced or independently checked, its intended meaning is reviewed, the authoritative record is updated, and the declared release checks pass.

The useful claim is therefore narrow. A hypothesis handoff can reduce the cost of informed disagreement and preserve what would change the project's mind. It cannot establish that the option set is complete, that the leading hypothesis is well calibrated, that the requested evidence is obtainable, or that any expert will agree. Those are precisely the kinds of finding the handoff is meant to make easier to return.

How the record should age

The evaluations above may expose failures. Repairs should add facts, not erase them. The appendix keeps three failures at the pinned commit although a later commit repairs one; that repair does not subtract the first fact. If criticism shows a validator too weak or a claim too broad, record the revision's date and cause instead of making the project appear always to have meant the narrower claim. A test rewritten after its outcome is known is a different test and may be better. The known case can show whether the same failure returns; fresh confirmation requires new cases whose answers were not used to design the repair. Disappointment is in scope: if an evaluation finds a component trivial or no better than a short ordinary script, that conclusion belongs on the record with the same standing as a pass. Any outside evaluator's report should remain under that evaluator's control, cited rather than absorbed. If the repository later falls short of this paragraph, the shortfall is itself a finding, and Section 9 says where findings go.

These evaluations do not pair one-for-one with the distinctions. They address the runner, public-case selection, tested-case correctness, usefulness, and origin. None repairs a validator–claim mismatch, proves representativeness, or turns risk reduction into a guarantee.

9 Conclusion

Plectis makes selected public claims testable; it does not verify the private system as a whole. Its fixed cases, procedures, outputs, rules, and stated limits let a stranger repeat a result and identify exactly what they dispute. A pass is no stronger than the test oracle—the rule and expected answer—or than the basis for believing that oracle.

The worked audio calculation demonstrates the hierarchy. Repeating the stored result supports computational reproducibility for that public case. Deriving one expected number by arithmetic adds evidence that the implementation agrees with the stated formula on that case. Neither step establishes that the formula is the right audio rule, that the code came from the private system, or that the selected case represents it.

More author-controlled components enlarge the inventory without changing who chose the cases or rules. Moving a relevant choice out of the author's hands can strengthen evidence about that choice; it does not repair every gap.

A Dated reproduction record

Except where a paragraph gives a later date, this appendix records one run on 17 July 2026 under macOS 15.6.1 on Apple silicon (arm64) and Python 3.13.7. The repository version was commit `57ffb7f5830cc446a2cbd8083d5d80bcab2af563`, abbreviated `57ffb7f5830c` elsewhere. Later commits or machines may differ; the body does not depend on them.

A typed hypothesis handoff makes the next outside contribution more specific: it exposes the project's tentative answer, alternatives, and update rule before the outcome is known. That is a better invitation to disagree, not stronger evidence that the project is right.

A first review. Use the appendix's no-install block and select the version analysed here. The tour describes the checkout but does not choose a component claim. Ask the repository to find the audio example with

```
PYTHONPATH=src python3 -m plectis comprehend --first-action "audio level calculation" --format text
```

The response identifies the audio component, but do not run its first suggested command: it writes to the repository's saved receipt paths. Run the paper's exact no-install command near the start of Section 4 instead; it writes only to `/tmp` and leaves the checkout unchanged. Under `/tmp/plectis-audio-rms/`, open `batch8_audio_level_rms_port_result.json`; JSON is a plain-text field-and-value format. Open `exercise`. In each `reference_cases` item, compare `expected_level` with `observed_level`. The top-level `anti_claim` says what a pass does not establish; `claim_ceiling` inside `exercise` gives the project's strongest conclusion. The project wrote both cautions; no independent reviewer checked their limits. To challenge rather than repeat the supplied case, make a disposable copy of the fixture input:

```
cp -R fixtures/first_wave/batch8_audio_level_rms_port/input/. /tmp/plectis-audio-rms-input
```

In the copied `batch8_audio_level_rms_port_probe_manifest.json`, change the first case's second sample from `0.05` to `0.06`, but leave `expected_level` unchanged. Then run:

```
PYTHONPATH=src python3 -m plectis batch8-audio-level-rms-port run \
  --input /tmp/plectis-audio-rms-input --out /tmp/plectis-audio-rms-probe \
  --acceptance-out /tmp/plectis-audio-rms-check.json; echo $?
```

The final number printed should be `1`; in the acceptance file, `status` should be `blocked` and `accepted` should be `false`. This means the validator caught the deliberate mismatch; the exercise did not crash. Report any different result in `CONTRIBUTING.md`. Treat a pass as evidence only for the published rule.

Names. Repository: github.com/wcook04/plectis. Formerly Microcosm, Plectis retains package `microcosm_core`, page `ORGANS.md` ("organs" means components), and the `microcosm` alias. This paper uses command `plectis`.

Getting the code and running without installing.

```
git clone https://github.com/wcook04/plectis
cd plectis
git checkout 57ffb7f5830c
PYTHONPATH=src python3 -m plectis tour --format text .
```

The final full stop means "the current folder". `git checkout` pins the exact state described here; omit it to run the newest version, whose dated numbers may differ.

The clean-clone tour read 5,018 files and chose the README-first order (internally `readme_onboarding_route`) from five reading orders. It wrote 21 generated files under `.microcosm/` without changing the examined source. Those files can be rebuilt; the README's different count is another dated run, not a stable project size.

Installing.

```
python3 -m pip install .
plectis tour --format text .
```

No runtime library is required; `pip` may fetch `setuptools` to build. No check proves the absence of network or model calls, so inspect the source.

The worked example. A fresh Section 4 rerun reproduced all displayed passes and values and overwrote `/tmp/plectis-audio-rms`. A stale field reports a copied private module, while the saved import record reports none because public code replaced it; the discrepancy remains.

A second Python version. On 18 July 2026, a clean macOS clone under Python 3.12.7 matched the tour and example. Its offline install used only the machine's `setuptools` 80.9.0 (minimum 77.0.3). Only versions 3.12.7 and 3.13.7 were tested.

Project-wide checks. The project-wide Lean trust scan checks 58 files for six text patterns but does not run Lean or verify proofs: unfinished placeholders; custom axioms (unproved assumptions); compiled calculations whose acceptance relies on more than Lean's small proof-checking core; `partial` definitions (runnable but not unfoldable in proofs); `unsafe` definitions (barred from theorems); and removed computation limits. No pattern appeared. The scan does not show whether files compile, proofs are valid, or statements express their authors' intent [12]. Separate components run Lean on small test projects; their results apply only to those projects. Registry and trust checks passed; the 58 files are distinct from the 20 mathematics components. The limited `make test` run covered 32 public test files, not every test file: 356 of 361 tests passed, two skipped, and three failed. The failures were an `AGENTS.md` line-break expectation; outdated line or byte counts in copied-source manifests; and root-layout rules that did not yet allow the `paper` directory or root PDF; the last was later repaired without altering this pinned run. Exact failure names, environment, and counts are in `paper/public-test-receipt.json`. `make ci` would add smoke and package-install checks, but was not green because `make test` failed. These checks cover only the public checkout; this is my run, not an independent repetition. The paper's own `scripts/check_public_system_paper.py` fails if counts, pinned facts, the example, cautions, or citations disagree. It checks historical facts against the pinned commit separately from current prose; with one maintainer, that is internal consistency, not external truth.

Statements and declarations

Declaration of generative AI use. Large-language-model agents were used throughout development to draft and revise prose, formal proofs, and software. The author set the objectives and acceptance criteria, selected and reviewed the claims, and approved the published version. The author assumes responsibility for the accuracy, interpretation, and presentation of the work. Generative systems are production tools; they are not authors and supply no independent authority.

Authority boundary. The account of private production is testimony about origin, not independent evidence about the system's correctness, representativeness, or reliability. The public commands and records establish only the cases and limits stated in this paper.

References

[1] Per Runeson and Martin Höst. "Guidelines for Conducting and Reporting Case Study Research in Software Engineering." *Empirical Software Engineering*, 14(2):131–164, 2009. doi:10.1007/s10664-008-9102-8.
[2] National Academies of Sciences, Engineering, and Medicine. *Reproducibility and Replicability in Science*. Washington, DC: The National Academies Press, 2019. doi:10.17226/25303.
[3] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The Oracle Problem in Software Testing: A Survey." *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi:10.1109/TSE.2014.2372785.
[4] Robert Rosenthal. "The File Drawer Problem and Tolerance for Null Results." *Psychological Bulletin*, 86(3):638–641, 1979. doi:10.1037/0033-2909.86.3.638.
[5] Object Management Group. *Structured Assurance Case Metamodel (SACM)*, 2.3, formal/23-05-08, October 2023.
[6] SCSC Assurance Case Working Group (ACWG). *Goal Structuring Notation Community Standard, Version 3*. SCSC-141C, May 2021. SCSC-141C.
[7] National Information Standards Organization. *Reproducibility Badging and Definitions*, NISO RP-31-2021, approved 15 Jan. 2021.
[8] Association for Computing Machinery. "Artifact Review and Badging—Current," v1.1, 24 Aug. 2020; accessed 18 July 2026.
[9] National Institute of Standards and Technology. *Secure Hash Standard*, FIPS PUB 180-4, 2015. NIST FIPS 180-4.
[10] National Institute of Standards and Technology. "Hash Functions." Updated 9 September 2024. Accessed 18 July 2026.
[11] National Aeronautics and Space Administration. *Software Assurance and Software Safety Standard*, NASA-STD-8739.8B, Section 4.4.1.2, p. 48, 2022. NASA-STD-8739.8B.
[12] Lean Project. *Lean Language Reference* sections "Validating a Lean Proof" and "Partial and Unsafe Definitions." Accessed 18 July 2026.