ProgramBench Measures Recall

Execute-only binaries and hidden tests reward prior knowledge of algorithms, formats, and reference traces.

June Kim
Independent Researcher
june@june.kim
ORCID 0009-0005-3153-9396

June 27, 2026

Abstract

ProgramBench grades software agents on rebuilding a program from its compiled binary and documentation, with the source withheld and the binary execute-only, on the premise that the binary is a queryable oracle, so any graded behavior is discoverable by running it. Discoverable is not reconstructable: some graded behaviors are the exact output of a hash, a cipher, a codec, a compressor, which no source-blind, offline solver can reproduce without the algorithm's specification the artifact never carries. This paper reads the benchmark's published test suites and classifies every exact-output assertion by how a source-blind solver could obtain the value it checks. At least twenty-one programs pin a value that only recall of a published algorithm supplies, and because the primary metric is conjunctive, one such test puts a whole program beyond reconstruction; the reported floor of zero resolved tasks is consistent with a recall gate. Recall tracks the kind of a program's core and its implementation language's standard library, not its size. The result for each audience is concrete: a model-blind, replayable list of programs a runner should exclude, and a five-class rule a maker can apply to screen tests.

1 The claim

We defend one proposition. Because ProgramBench supplies the reference as an execute-only binary and denies the solver internet, a source-blind solver can only run the binary and read the bundled documentation; for some graded behaviors no such solver can reconstruct the answer, because it is the output of a function over an open input domain that no finite set of oracle queries identifies. The information that would replace the search is the algorithm's specification, withheld and unfetchable, so the only feasible channel to those behaviors is prior information, and because the metric is conjunctive, one such behavior in a task's suite puts the whole program beyond reconstruction.

The argument is an existence proof, and it reduces to a single witness: one graded behavior that no finite sample identifies and no supplied document supplies is enough to put a task beyond reconstruction. It binds every solver at once, turning on no model's weakness and no scaffold's limit, because what is missing is information. We use impossible in the feasibility sense, not the computability sense, and bound it: infeasible with classical computing within the foreseeable future. Every graded behavior is computable; the reference computes it, and a reconstruction exists in principle. But computable is not feasible: the reconstruction we deny is not undecidable. It is unreachable by any classical search a solver could run in this hardware era. Existence is not reachability, and the escape from the search is not capability; it is holding the program's specification in advance.

ProgramBench (Yang et al., 2026, arXiv:2605.03546; the paper hereafter) gives the agent a compiled binary `B` and usage documentation `D`. The source `S` that produced `B` is stripped and withheld, and `B` is set to execute-only permissions, a restriction the paper imposes, in its own words, to "prevent reading or reverse engineering of the binary with a tool like ghidra" (§2.2, Build an inference environment). A solver may run `B` and observe outputs; it may not read its bytes, disassemble it, or instrument it. We assume the sandbox enforces this against all byte access, including runtime memory extraction and tracing; if it does not, the binary can be dumped and the benchmark's own anti-reverse-engineering guarantee fails, which only sharpens the same conclusion. The solver also has no internet: the paper enforces this by running the container without a network, and its system prompt tells the model that "the only source of truth about what the executable does is the executable itself and its bundled documentation" and that it "must not search the internet, package registries, or any external source" (§2.1, §A.2). The same prompt adds a revealing instruction: "even if you recognize what the executable is, you must reimplement it from behavioral observation alone" (§A.2). The clause presumes recognition is common and tries to manage it, which concedes that prior possession of the program is a live channel the benchmark must suppress by instruction.

The solver's inputs are therefore exactly two, the queryable binary and the bundled documentation; anything else, a public format specification, an RFC, a published algorithm, is available only from training, which is recall rather than reconstruction from the artifact. The agent emits a program `P` and a build script. `P` is graded by a hidden suite `H` of behavioral tests, each a pair `(x, B(x))` asserting that `P(x)` matches the original on input `x`. The primary metric, % Resolved, credits a task only when `P` matches `B` on every test in `H`. Suites run from hundreds to tens of thousands of tests per task, 248,853 in total with a median near 770 (§2.3).

Two properties of `H` carry the argument, both stated in the benchmark's own construction. First, `H` is produced by a generator with source access: it explores the program, its source, its tests, and its documentation, and uses line coverage of `S` as its objective, adding tests to reach code paths not yet exercised. Second, grading is conjunctive: a single failed test forfeits the task. The solver must reproduce a behavior whose graded points were chosen by reading a source it cannot see, and reproduce all of them at once.

3 Aims and construction

The aim is specific and worth stating in its own terms. Existing benchmarks, the paper argues, measure "focused, limited tasks such as fixing a single bug or developing a single, specified feature"; ProgramBench instead asks an agent to architect and implement a codebase that matches the reference executable's behavior, offered as "a test of software design, not just implementation alone" (§2.4). The construction is careful where its predecessors were not. Grading is determinate: the reference is a complete, queryable function with one ground-truth answer per behavior, so there is no maintainer's arbitrary choice for a faithful solver to recover and fail on. Grading is implementation-agnostic, checking behavioral equivalence and admitting any language or architecture (§2.1). The suites are filtered: an assertion linter cuts the dummy-pass rate from 18.5% to 3.7% (§5.1), tests that the reference itself fails are discarded, and generated coverage averages 79.7% against the projects' own 56.8% (§5.1, Figure 7). The reporting is candid: a finite suite "necessarily under-approximates" the specification (§2.2), and "% Tests Passed also does not reliably correlate with percentage of working functionality" (§3). These are real improvements, and they address the failures that retired SWE-bench Verified.

We take all of it as given. The argument that follows is narrow and orthogonal: determinacy is not the property that decides solvability. A specification that is complete, and queryable anywhere, is unusable if the solver cannot know where to query it. The gap is between a behavior being determined and a behavior being findable.

4 Terms

We fix four terms used throughout. They sort a graded behavior by how a source-blind solver could obtain the value the hidden test expects.

Discoverable. The expected value is obtainable from the materials at hand: the bundled docs, or running the reference and inspecting its outputs and files (generate-and-inspect), or probing an enumerable, numeric, or pointer-valued input. One observation generalizes, because the value is a constant or a rule the solver reads off once and reproduces. zoxide's database version byte is discoverable: run the reference, read the file it writes, see the 3.

Brute-searchable. The expected value lies in a finite space the solver could in principle enumerate, but the space is too large to traverse within the budget. The answer is reachable by search and only by search, so feasibility turns on the size of the space. An undocumented magic constant or a checksum gate on an input the reference will not emit is of this kind.

Recall-only. The expected value is the output of a function, a hash, a cipher, a codec, a compressor, that a solver cannot reproduce from feasible observation of the reference, because the outputs on chosen inputs carry no information that fixes the output on an unseen one. The function is perfectly determined, by its published specification; the point is not that the value is unknowable, it is that the only feasible channel to it is prior possession of that specification, recalled from training rather than recovered from the artifact. That is recall, not reconstruction. It is distinct from brute-searchable, because no larger search budget closes the gap, and it is the paper's spine.

Benchable and unbenchable. A program is benchable when every graded behavior in its hidden suite is discoverable. It is unbenchable when at least one graded behavior is recall-only or brute-searchable beyond budget. Because the primary metric is conjunctive, one such test makes the task unresolvable by reconstruction, so the verdict attaches to the whole program, which is the unit a benchmark runner can cite and exclude.

5 The test oracle problem

A test needs two things, an input to run and a way to judge the output. The second is the oracle, the source that says whether a result is correct (Howden 1978). Where that source comes from is a settled subject in testing, surveyed under the name the oracle problem (Barr et al. 2015). A specified oracle derives the expected output from a description of the task, so it judges any implementation against the contract. When no specification is at hand, or the correct output is too hard to determine independently, the program is non-testable in Weyuker's sense (Weyuker 1982), and testing falls back to a pseudo-oracle, the output of a trusted reference taken as the standard (Davis and Weyuker 1981).

The two kinds of oracle look identical until two correct programs disagree. For a behavior with one right answer, `echo hello`, the reference's captured output and the contract coincide, and a pseudo-oracle is as good as a specified one. The gap opens wherever a task admits more than one correct output. Picture a serializer that emits the same object with its keys in another order, or two renderers that write byte-different files displaying the same page: each is correct, and each fails a pseudo-oracle that pinned one reference's exact bytes. This benchmark furnishes the real version. typst's `test_emphasis_basic` compares a compiled PDF, byte for byte, against the reference's, and ditaa's compares an exact PNG; a faithful reimplementation that renders the identical document but lays its bytes out differently fails, because the golden pins the reference's file with no check on the page's contract. A pseudo-oracle measures agreement with one implementation; only a specified oracle measures the contract, and the two diverge exactly where the contract leaves the output free.

specified oracle judges against the contract contract-correct outputs candidate golden pseudo-oracle judges against one reference run contract-correct outputs candidate golden The contract is a region; the captured golden is a single point. A candidate correct by the contract but off the golden's bytes fails the pseudo-oracle.

ProgramBench runs on a pseudo-oracle and nothing else. It withholds the source a specification would come from and grades against `B`'s captured output, so its goldens are the reference's own trace, validated into a contract by no independent check. The generated tests show it in the open, writing the golden from the reference run and then asserting against it:

```
if not golden.exists():
    golden.write_text(result.stdout)
assert result.stdout == golden.read_text()
```

The oracle problem proper is the implementation-pinned case: where the contract leaves the output free, the captured golden fixes one reference's incidentals, and a program that meets the contract through different bytes fails. That is a defect in the oracle itself. The benchmark's other hard cases are not oracle defects; they sit beside this one because the same withheld source produces them. For a hash or a codec the golden is a valid oracle and the value is exactly determined; what fails is the solver, which cannot compute that value without the specification the benchmark denies it. That is an information barrier, recall, and §6 is its proof. And because the suite is hidden and conjunctive, even valid, reachable oracles are demanded all at once across a surface too large to cover (§7). The three are distinct: an oracle that overspecifies, the case

this section makes; a value the solver cannot reach, the recall barrier §6 proves; a surface too large to clear blind, the coverage barrier §7 measures. They fall out of one decision, to grade against a captured reference with the source withheld. Reading the assertions in the public suites and sorting each into one of the three is the whole of our method.

6 Limits of black-box probing

Because `B` is execute-only, the solver's only operation on it is to run it: submit an input, observe the output. Disassembly, symbolic execution, and instrumentation, the tools that would recover a constant or a branch condition, all require reading the binary and are denied by the permission the benchmark sets to block ghidra. The right model of the solver is an algorithm with black-box oracle access, which is the benchmark's actual setting.

What the oracle yields is finite. In its budget a solver runs `B` on finitely many inputs and observes finitely many pairs `(x, B(x))`. A program's behavior has a head, the documented contract that `D` and convention specify and competent synthesis reproduces, and a tail, the accumulated handling of particular cases. For the head, and for any behavior that is a constant or a simple rule, a finite sample suffices: one run reveals it and it generalizes, the discoverable case of §4. The hidden suite, concentrated by its source-guided selection on the tail, is where finite sampling fails, and it fails in two ways.

The first is a gradient-free gate: a branch entered only when some bits of the input hit an exact value, an internal checksum or a sentinel, where every near miss returns the same response. Nothing in the transcript, output, error, exit, length, timing, points toward the value, so an undirected search over a `d`-bit discriminator costs on the order of `2^d` queries and no black-box method does better save by chance. For deep gates this is already past any feasible budget. This is the brute-searchable case; we note it and pass on, because the second is stronger.

The second is recall-only: the graded behavior is the output of a function the solver cannot reproduce from feasible observation, a hash, a cipher, a codec, a compressor. Running `b3sum` on a thousand chosen inputs teaches nothing about its value on the thousand-and-first, because the digests of chosen inputs carry no information that fixes the digest of an unseen one. The function is perfectly determined, by the BLAKE3 specification; the point is not that the value is unknowable, it is that the only feasible channel to it is prior possession of that specification, which is not in the binary's output, not in the usage docs, and, with no internet, not fetchable. A solver that produces the right output on an unseen input has recalled the algorithm. It has not recovered it from the artifact.

input probed inputs: every function agrees hidden test input Many functions fit every probed input and part at the one the hidden test grades. No finite sample picks out the function.

The paper's own analogy concedes the gap. The executable, it writes, "resembles how a developer queries a product manager" (§2.4); but a product manager answers the question the developer did not know to ask, while the oracle answers only the question posed. It returns `f(x)` for a fully formed `x`; it does not reveal which inputs carry an unsignposted degree of freedom, nor supply a function no finite sample identifies. Queryability reduces output uncertainty and leaves untouched the two that decide the hard tasks: which behaviors exist to be matched, and, for the recall-only ones, a value the oracle cannot teach.

7 Completeness vs. coverage

The grading scheme decides what is being asked. A coverage scheme credits a reconstruction for the behaviors it gets right and rises as more are rebuilt, which is the percent of tests passed that the benchmark reports and then downgrades. A completeness scheme credits it only when nothing is wrong, which is the conjunction the benchmark makes primary as % Resolved. The metric itself scores a finite thing, equality with `B` on every case in the hidden suite. But because the suite is hidden, the solver cannot aim at that finite set; its effective target is the surface the set is drawn from, and one uncovered behavior anywhere is total failure, so the metric puts the solver under the pressure of full behavioral equivalence without asking for it by name. Equivalence is a bar testing was never built to clear, a test showing a behavior wrong but never every behavior right (Dijkstra 1970), and undecidable in the general program setting (Rice 1953).

The benchmark's defense is that the solver need match only the finite suite `H`, not the whole function `f`. The reduction holds for whoever knows `H`. The solver does not. Because `H` is hidden and the submission is one-shot, the solver cannot condition its implementation on the graded cases, nor repair after seeing a failure, so any behavior that might be in `H` must be treated as if it will be graded. Under a pass-all metric an uncovered graded behavior fails the whole task, and the solver's effective target is no longer some

representative behavior. It is the union of every behavior the hidden suite could plausibly test. Hiddenness converts test-passing into blind coverage; the conjunction makes any missed point fatal.

7.1 The blast radius

How large that target is, the benchmark's own suites measure. Count, per program, the distinct exact-output obligations its hidden tests pin, each an input and the output the solver must reproduce on it. The count runs from a handful for a small tool, nine for `cmatrix`, nineteen for `tty-clock`, to the thousands for a real program: a median of 204, then `sqlite` at 1,171, `pandoc` at 2,746, `gdal` at 4,064. This is not a count of independent concepts, since one correct rule can satisfy many obligations at once; it is the size of the hidden target surface, the number of places an omitted or slightly wrong behavior can lose the task. Call it the blast radius.

Read against the conjunctive metric, surface size is the exposure. The solver cannot see which obligations are graded, so it must treat them all as live, and a single miss anywhere fails the task. The benchmark reports a % Resolved floor of zero across nine models (§4), and a hidden surface in the hundreds to thousands under a conjunction reaches that floor on size alone, with no appeal to a missing algorithm. What loses the task is the source-shaped tail, the obligations the documentation never named. The documented head a competent solver covers; the tail it cannot see.

7.2 Search still has to look

The model counts the odds of covering. It says nothing about the cost of finding what to cover, and the finding is the harder wall. A solver that will not guess has to search, and search is not magic, it has to look. A behavior the program performs only on an exact input, a magic number, a checksum match, a value the documentation never names, hides from every near miss; finding it is an undirected search over the `d`-bit value that gates it, about `2^d` probes, no black-box method beating chance (the gradient-free gate of §6).

Put numbers on the looking. Give the search a top-spec prosumer machine, and grant it a generous million runs of the reference a second, well above what real process execution sustains. The wall-clock:

value width	probes `2^d`	wall-clock at a million runs/s	pinned in the suites
64-bit	1.8×10^{19}	580,000 years	a CRC64; XXH64 (7zip)
128-bit	3.4×10^{38}	`10^25` years, about `10^15` times the age of the universe	an MD5; ffmpeg's frame hashes
256-bit	1.2×10^{77}	past any physical clock	BLAKE3 (blake3); GOST (php-src)

Real process execution runs slower, so these are if anything generous. And they are only the brute-searchable case, where search is at least possible. A recall function admits none, since a hash output is not reachable by probing at all, so even these numbers sit below the recall witnesses. A more capable model does not move them, because the bottleneck is the looking, and the looking has nowhere to fit.

That coverage cannot be reached by luck; it has to be imposed. To gain confidence rather than fortune, the solver has to black-box check its own program against `B` over the target surface, and checking means choosing which inputs to exercise, which is building a suite. For a real program the coverage that suite must reach approaches the reference's own. To be sure it has missed no graded behavior the solver has to exercise its reconstruction about as thoroughly as the project exercises itself, and then wider, because it cannot see where the graded boundary falls. SQLite's maintainers built that coverage over years with the source in hand; the solver is asked to approach it blind, in one submission, and because a single uncovered behavior fails the task, it cannot stop at the common cases. It must chase the tail to the end, and the chase has no certificate even there. Finite observation does not single out a behavior: many non-equivalent programs agree on every probed input and part on the next, so a solver that has matched every input it ran still cannot know it matches the one it did not, the inductive gap that identification from examples has studied since Gold 1967.

7.3 Reasonable and unreasonable

This is what makes "reconstruct the program" a quantity rather than a verdict. The behaviors a small specification can be tested for are few, derivable from the spec, coverable: reasonable. The behaviors `sqlite` can be tested for, its query planner, its type coercions, its collations, its error paths, run to the thousands

the benchmark itself counts: unreasonable. The recall witnesses are the other barrier, where one behavior's value cannot be obtained at all; this is the coverage barrier, where every value is obtainable and there are too many to cover blind in one shot. A whole-program reconstruction benchmark meets both, and the size of the hidden target surface decides where the second one bites.

8 The existence proof

Because resolution is conjunctive, the task-level claim reduces to a single behavior: if a task's suite contains even one recall-only test, no source-blind solver resolves that task, and the verdict attaches to the whole program. We exhibit such tests directly, because the benchmark's generated suites are public and we read the assertions.

The cleanest witness is a graded test that compares the program's output, byte for byte, against a bundled golden asset only the reference algorithm produces. blake3's suite is the model. Its `test_chunk_boundary_1024_bytes_exact` runs `b3sum` on a bundled 1024-byte file and asserts that standard output equals a bundled `hash_1024.golden`, the reference's exact digest. The input is not handed to the solver at inference, the digest of an unseen 1024-byte file is not interpolable from any finite set of other digests, and the BLAKE3 specification is neither in the usage docs nor fetchable offline. The test is recall-only, and one such test forecloses resolution by reconstruction.

Matching only the finite hidden suite does not lower the bar. The solver is not given `H`, and a recall-only value does not generalize from its neighbors, since the digest of one input carries no information about the digest of another. One such test in the suite is enough, and the solver cannot know which test it is.

The reading is confirmed against bodies, not inferred, and it cuts both ways. jp2a, which renders a JPEG to ASCII, asserts its output equals a bundled fixture produced by decoding the image through libjpeg, whose inverse-DCT rounding no standard-library decoder reproduces: recall-only. The reverse is as clear. zoxide's apparent "exact-format" tests, which grade a database version byte, a null-byte separator, and an error template, are all discoverable: the solver runs the reference, reads the file it writes and the message it prints, and reproduces them. zoxide is benchable; blake3 and jp2a are not. The appendix carries the program-level verdict with one witness test apiece, read from the suites themselves.

9 Source-guided test selection

The asymmetry follows from the selection objective. The generator maximizes line coverage of `S`, which steers test generation toward paths a generic input rarely exercises, the low-frequency tail where the recall-only behaviors sit. The procedure does not target them by name; its reward correlates with reaching them. The benchmark grades a source-blind solver against a source-guided selector whose objective tracks the behaviors the solver cannot reach from the artifact. And it compounds with thoroughness: each test the generator adds is another chance to land on such a behavior, so the more thorough the grader, the larger the share of its tasks that carry one.

This bears on the coverage figure the benchmark reports. Its generated suites reach line coverage near eighty percent, comparable to the projects' own (§5.1, Figure 7). That figure was obtained with source access, by the generator. The figure relevant to solvability is the coverage a source-blind solver can reach from the binary and documentation within budget, and it is not reported. The reported number measures the party holding the map.

10 Predictors of unbenchability

A maker reading this will want to know which programs to expect trouble from, and the obvious guess is wrong. Recall-unbenchability is orthogonal to program size. Across the twenty-one recall programs the graded surface runs from `fasttext` at twenty-nine obligations to `gdal` at four thousand, and the largest recall-free program, `pandoc`, pins more obligations (2,746) than all but one of them. The medians barely separate, 268 against 194. Size predicts almost nothing about whether a program is recall-gated.

What predicts it is the kind of the program's core. A program is recall-gated when one graded behavior is an un-inferable function, a hash, a cipher, a codec, a compressor, a binary format, however small or large the program around it. `fasttext` is a small embedder and recall; `pandoc` is a large document converter and recall-free, because its formats, a docx that is a zip of XML, are parseable from the supplied materials and an embedding is not.

The second predictor is the implementation language, through its standard library. The recall rate is several times higher in C and C++ than in Go and Rust:

language	recall / total
C	9 / 33
C++	4 / 12
Go	4 / 46
Rust	4 / 107

Part of that is domain, since the heavy media and crypto tools tend to be written in C. But part is the standard library, and it is size-independent: the same function is recall in a language whose stdlib lacks the algorithm and benchable in one that carries it. The appendix holds the matched pair: `chafa` in C decodes an image and earns a witness, while `ascii-image-converter` in Go does the same job through the standard library's `image/png` and stays benchable. Compression shows the rule again: a wrapper around zstd, lz4, or brotli is recall because no standard library carries them, while gzip and xz pass through the stdlib's zlib and lzma. The predictor is what the implementation language's standard library carries.

Size has its own barrier, but it is the other axis: the coverage pressure of §7, which the largest programs run into whether or not any single behavior is recall. A program can be caught by either axis or both. `gdal` is caught by both, a checksum core inside four thousand obligations; `pandoc` by scale alone; `fasttext` by kind alone. The maker's screen, then, is the core's kind and the implementation language first, and size as a separate, second filter.

11 The fairness audit

The paper audits all 200 tasks and reports that no test invokes a flag or subcommand absent from the documentation, with five instances where implementation-dependent output could plausibly appear, none realized (§2.2, detailed in §A.3.4). Its premise is that "any behavior a behavioral test expects is discoverable by running that same command" (§2.2). We take the audit at face value. It establishes that the entry points are documented; it does not establish that the graded behavior at those entry points is determined by anything the solver receives. A named flag shows a door exists. It does not fix which room behind it is graded, and the leak is one level below the flag: a behavior at a documented entry point, triggered by a value or combination recorded in the source and absent from the documentation.

FFmpeg makes the gap concrete on two axes. By value: the full manual documents that `-ss` exists, and even that its position relative to `-i` selects input versus output seeking, so we do not claim that semantics is unwritten. We claim the narrower thing the audit cannot: a flag-presence check fixes neither which positional reading a test grades, nor, for the interactions the manual leaves to the source, whether they are written at all. By volume: `ffmpeg -h` prints a curated subset of options, the full listing needs `-h full`, and per-codec options appear only under targeted queries such as `-h encoder=libx264`. A complete index is not a determined behavior.

The paper raises feasibility directly and answers it, and the answer checks the same kind of property a second time. Its feasibility review (§A.2.4) asks whether a graded behavior is discoverable: whether it is "not communicated or documented via any observable channel," whether it is "entirely absent from the task instance's start state." It reviews all 200 repositories, finds none, and concedes only the case it can already rule out, an executable with "important but entirely undiscoverable functionality." We grant the finding and deny that it settles feasibility, because discoverability is not reconstructability. A hash is observable: run `b3sum`, observe a digest, the behavior is plainly present in an observable channel. It is not reconstructable: the digest of one input fixes nothing about the digest of another, so observing the behavior does not let a solver reproduce it on the hidden test input. The audit rules out behavior with no observable instance; it does not test for behavior that is observable yet not reproducible from feasible observation, and the recall-only class lives exactly there. "This does not mean the task is impossible," the section writes (§A.2.4), and we agree: the task is not impossible, it is recall-gated, which a discoverability review was not built to detect.

12 The grader and the product-manager analogy

The product-manager analogy describes a process: query, hypothesize, infer behavior "in the absence of complete specifications" (§2.4). The primary metric scores a result: % Resolved credits a task only when the candidate matches the reference on every hidden test (§2.1). These answer different questions. The analogy

justifies the difficulty of the search; it says nothing about which results are admissible, and the grader admits exactly one, byte-identity with the reference on the tested inputs.

Inference under partial specification, taken seriously, would credit a solution correct in contract though not identical in detail, which is what implementation-agnostic grading is said to buy. For a hash, byte-exact equality is the contract, but the contract is the algorithm's specification, which the benchmark does not supply, so identity with the reference is reachable only by holding that specification in advance. The conjunctive byte-check cannot tell "inferred the right behavior" from "reproduced the exact reference," because for these behaviors the two are the same bytes or they are failure. The metric collapses the distinction the analogy depends on.

The paper's own ranking shows which side it stands on. It makes % Resolved primary and downgrades percent of tests passed to a relative measure, noting that "even a single failed test can imply a fundamental flaw" (§3). A benchmark built to reward partial inference would lead with the partial metric; this one leads with the least partial-credit object available. The likeliest explanation is not confusion. It is expedience: a golden-output check is the grader one can build and run at scale, and a contract-level grader that honored the analogy is the one reached for when the deadline has passed. The fork holds either way. If the measured skill is inference under incomplete specification, % Resolved is the wrong primary metric; if % Resolved is right, the product-manager framing is the wrong description. The paper cannot keep both.

13 Prior information as the only channel

No solver restricted to running an execute-only binary, reading the bundled docs, and working offline reconstructs a recall-only behavior: a finite sample does not identify the function, and the specification is withheld and unfetchable. What remains is prior information, and it divides in two. For a behavior fixed by a public standard, a hash, a codec, a published format, general knowledge of that standard supplies the value, but with no internet that knowledge is recalled from training. For a behavior recorded only in the program's own source, program-specific memory is required, and no public standard supplies it either. The benchmark does not distinguish the two, and the argument does not need it to: both are carried from training rather than recovered from the artifact, and both convert the task from reconstruction to lookup.

Therefore, for a recall-only behavior, prior information is the only feasible channel; a solver that matches it has recalled it. Full resolution demands every graded behavior at once, so a task with one such behavior sits at its floor unless training supplies it, and the partial metric measures, in part, how much of the tail a model has memorized, confounded with documentation quality, scope, and program size, which themselves move with prevalence.

Prior information is not a confound to be scrubbed away here; for the recall-only behaviors it is the channel the metric reads.

14 Corroborating evidence

The witness receipts in the appendix carry the argument: a referee who grants one row has granted a recall-gated task, and the case does not rest on what follows. We add three published observations consistent with the reading, as corroboration and not proof, each individually confounded. The benchmark reports a % Resolved floor of zero across all nine models (§4, Table 2), which a large enough task size would also produce. It reports widely cloned utilities scoring highest and large idiosyncratic systems lowest, difficulty largely model-agnostic (§4, Figure 5), though prevalence is confounded with documentation quality and scope.

The strongest is the internet ablation. With the network open, source-code lookup was the dominant behavior, "accounting for 79–95% of flagged runs," with twenty to thirty-six percent of tasks flagged (§4.1, Table 3). The paper reads this as cheating to be blocked; read as solvability it measures how often a model needs the specification, and cutting the internet does not remove the need, it removes the legitimate channel and leaves recall. None of the three separates the reading from "these are merely large, rare programs." The receipts do, and they stand alone.

15 The benchmark's defenses

We have argued the general case; we now hold it against the paper's specific defenses, each quoted and located. Five claims carry its fairness and its reading of the results, and none survives the no-internet, source-blind condition the paper itself imposes.

Execute-only binaries and hidden tests reward prior knowledge of algorithms, formats, and reference traces.

The paper's defense (quoted, located)	Where it fails under the no-internet, source-blind condition
The executable is "a comprehensive but opaque oracle" and "any behavior a behavioral test expects is discoverable by running that same command" (§2.4, §2.2).	Holds for documented flags, fails for the value and algorithm space where the grading concentrates. An un-inferable codec or hash is not learnable from any feasible set of input-output pairs, and with no internet the specification cannot be fetched (§2.1).
Overspecified tests, "undesirable because they make a ProgramBench task instance infeasible to successfully complete," are pruned (§A.3.4).	The only feasibility check run is that no test invokes a flag or subcommand absent from the docs (§A.3.4). A test grading H.265 encoding uses the documented codec flag, passes the check, and is infeasible all the same, because the flag does not carry the codec.
The different-language ablation forces "deep understanding of program behavior rather than surface-level recall" (§4.1).	Switching language defeats verbatim reproduction of the reference's source. It does not defeat recall of its behavior, which a model re-emits in any language. The ablation's own numbers are mixed, some models falling and others rising (§4.1).
The bundled "test assets that a model cannot reasonably synthesize on its own" mitigate a conceded "unfair asymmetry" (§2.2, §A.2).	Bundling settles input availability and nothing else. Given the H.265 file, matching the reference's decode still requires the codec; the input is supplied and the algorithm, the half that was never reconstructable, withheld.
The zero floor reads as difficulty: "no model fully solves any single ProgramBench task instance," with "meaningful progress" besides (§4).	A floor of zero across nine models, on a set whose recall-only tasks no source-blind solver can reconstruct, is consistent with a recall gate. It is not a capability frontier. It stays at zero for any solver that does not already hold the missing specifications, because what is absent is information the artifact does not carry.

16 Oracle provenance

16.1 The reference as contract

Both defects fall out of one decision. With the source withheld, the reference is pressed into service as the contract (§5), and that choice surfaces in two ways. Recall is information the solver lacks: the contract encodes a function, a hash or a codec, that no finite probing recovers. Provenance is authority the contract lacks: the expected value may be nothing more than what one reference build happened to emit. A test can carry either or both.

16.2 The self-capturing golden

The expected value is captured from the reference, then asserted against it. At least twenty-nine programs ship graded tests of this shape, more than carry a recall witness, seven of them the same programs, where the captured golden is the very digest or decompressed blob the solver was never going to reproduce.

A capture is not automatically wrong; the bytes it records may be the intended behavior. What it is not is a contract. A golden earns its authority as an oracle only when something independent of the reference has confirmed that the captured bytes are the behavior that matters, with no incidental artifact of one build, the object order a library happened to emit, a timestamp, a path. Nothing in the pipeline performs that confirmation, so the provenance is the reference grading itself: it cannot fail a test whose answer key is its own output, while a candidate that meets the same contract through different bytes does. Twelve write the golden only `if not golden.exists()`; the goldens ship, so that branch is dormant, and the pattern stands as the clearest evidence of how every golden was made, by capturing the reference.

A graded test that writes its own answer key is the defect a validation pass exists to catch, and the pattern survives across the set. SWE-bench Verified names such a pass: a human confirmed the gold patch resolves each task and that its tests admit it.

The captured golden does not condemn the suite, where roundtrip checks, observable constants, and return codes stay valid; but under a conjunctive metric one such golden zeroes a sound task, and the headline blends four oracles it never separates, valid but unreconstructable, implementation-pinned, weak-provenance capture, and genuinely contractual, with no word on which a task failed on.

The benchmark is not careless. Its test hygiene is substantial and its grading is sound on the points it checks. The defect is upstream, in the framing. The absence of a specification is presented as a virtue, implementation-agnostic and memorization-resistant; it is also the defect, because an artifact carries no ranking of which behaviors matter, and a grader that imports that ranking from the withheld source grades against information the solver was denied. What the primary metric reports is ordinary synthesis on the documented head, where strong models already perform and where the metric does not concentrate, and prior information, recalled algorithm specifications and program-specific memory, on the recall-only tail, which is what produces the floor and the prevalence-graded partial signal. Where the oracle is itself a reference capture, it scores a third thing, identity with the reference's own bytes, as if that were the contract.

Under all three is one fact of provenance: the golden came before the measurement. The suite still carries the capture logic that produced it, writing the golden when it was absent and asserting against it once present. That is strong evidence that the oracle was generated by running a reference rather than derived from the task, and the faulty-oracle patterns that survive show no effective validation pass of the kind SWE-bench Verified applies by hand. So the metric measures agreement with a captured reference trace. It measures task fidelity only where that trace is a correct and complete expression of the task, and the capture fossils mark exactly where the assumption breaks. Recall, implementation-pinning, and coverage are three ways the distance to a captured golden parts from the distance to the task.

time run the reference observe its output O golden := O saved as the standard grade candidate P P == golden ? The standard is set at the middle step, the reference's own output, before anything checks that it is the task's contract.

Where exactly the admissible set ends is an open problem, and we do not pretend to draw its boundary. The benchmark needs determinism, since an exact behavioral check presumes the reference is a function, so it rightly excludes or pins the non-deterministic cases; but determinism is necessary, not sufficient. A hash is deterministic and recall-only. The admissible set, the programs every one of whose graded behaviors a source-blind solver could reconstruct from the artifact within budget, lies inside the deterministic ones, and its boundary is not a property of the program alone: it moves with the supplied documentation, the hidden suite's coverage, and the comparator's strictness, the same three-way dependence that makes a single benchable-or-not verdict ill-posed in the general case. We claim only the sound one-sided direction. A recall-only witness is sufficient to place a program outside the set, which is all an exclusion needs; the complete characterization of what belongs inside we leave open.

Excluding the recall-witnessed programs is necessary, not sufficient. It removes the cases where one behavior cannot be reached at all; it does not touch the coverage barrier, which bites hardest on the largest programs left in the set, nor the selection bias, since the generator drew its tests by reading the source and aimed them at code paths the documentation never names. The benchable subset is a floor, the programs no witness has yet excluded. It is not a certificate that source-blind reconstruction suffices for them. The recall exclusion corrects the central conflation; the pressure a source-guided, hidden, conjunctive suite puts on a source-blind solver persists across the programs that remain.

The defect is upstream of grading, and it is repairable without giving up behavioral equivalence, the part that works.

18 Recommendations

The repair differs by who is using the benchmark. We separate two audiences, because they have different powers and different incentives.

18.1 For a benchmark runner

Reporting model scores, the actionable unit is the whole program. A runner cannot drop tests to taste: choosing which to exclude moves the very score being reported, a conflict of interest. Removing the discretion dissolves the conflict, with no deference to an authority, and three properties remove it. The exclusion is model-blind, read from the test bodies and fixed before any model runs, so it cannot be reverse-engineered to flatter a score. It is replayable, each verdict re-derivable by grep over the public suites, so the runner is accountable to a check rather than to anyone's say-so. And it is citable, the same fixed list for everyone, so excluding it is a property of the benchmark, not a choice about one model. The appendix gives that list, the programs whose hidden suite contains at least one recall-only test, read from the test bodies. Excluding them, a runner reports % Resolved against the benchable subset rather than all 200 tasks, and the headline figure stops conflating reconstruction with recall. The same list is a skip-list: a run against a program no

source-blind solver can resolve spends the build budget on a foregone fail, so excluding it saves the compute as well as correcting the score. One recall-only test is enough to move a program onto the list, because the metric is conjunctive.

18.2 For a benchmark maker

The unit is the test, and four changes make a program benchable. First, admit a graded behavior only when the supplied documentation determines it: a competent solver should reach it from the help and manual alone, without prior exposure to the program. This is the fairness test the paper states informally, that interacting with the binary "resembles how a developer queries a product manager" (§2.4), made into an admission rule. Second, prefer assertions a source-blind solver can satisfy. A roundtrip or self-consistency check, compress then decompress, write then read, grades the contract without pinning the reference's exact bytes; xz and pigz pass exactly because their suites do this, decompressed through the standard library's lzma and zlib. A check against an observable constant grades a format the solver can read off the reference. The pattern to avoid is the byte-exact comparison against a golden asset only the reference produces, whether that asset is a hash digest, a rendered image, or any byte carrying the reference's own incidentals. Third, where a program's core is a published algorithm, a hash, a cipher, a codec, a compressor, score it on a separate track: holding BLAKE3's compression function in advance is a real capability, but it is knowledge, and no usage page will ever contain it. Fourth, never let a test author its own oracle: a golden captured from the reference run certifies only provenance, and one written only when the file is absent will pass any output at all. Validate the goldens the way SWE-bench Verified validates its tasks, a pass that confirms each expected value is a contract a second implementation could meet. It must not be a byte the reference happened to emit.

18.3 The triage rule

The triage is automatable, which matters at 248,853 tests: the maker reaches for the same LLM assistance the suites were generated with. One pass per test classifies how a source-blind, offline solver would obtain the value each assertion checks:

class	how a source-blind solver obtains the value	action
Self-consistent	checked against its own output: a roundtrip, compress then decompress, write then read	keep; any correct implementation passes
Observable	a constant or rule read off one reference run: a version byte, an error template, a separator, a full-dump listing	keep; discoverable
Contract	a documented property: structure, count, range, exit code	keep
Recall	byte-exact against a golden only an un-inferable function produces, a hash, cipher, codec, or compressor, on an input the solver cannot pre-query	remove, or move to a knowledge track
Pinned	an implementation detail of the reference itself: an exact rendering or a captured frame	normalize or remove

The one question that separates Recall from Observable is whether the expected value would change on an unseen input in a way the reference's observable outputs do not reveal. A hash answers yes and is recall; a version byte answers no and is observable. The rule is mechanical enough to run at the suite's scale and specific enough that two agents applying it should agree, which is the property the generation pipeline already relies on.

We do not run that audit here; one recall-only witness per program is the existence proof, the appendix carries one apiece, and the full triage is the maker's task. The repair keeps determinacy, implementation-agnostic grading, and the test hygiene the paper already has, and gains the construct validity it claims.

The suites reach eighty percent coverage, so search does find the branches. With source access, by the generator. The solver's reachable coverage, source-blind and within budget, is the relevant quantity and is unreported.

The binary is given, so reverse-engineer it. It is execute-only. Reading, disassembling, and instrumenting it are denied by the permission that blocks ghidra. The solver's only operation on the binary is to run it, which is the model the argument assumes.

Recall is a real capability, so measuring it is legitimate. Agreed, and a benchmark of how faithfully a model reproduces known software from prior exposure is a coherent instrument. The benchmark describes a different one and disclaims recall in its own words: its prompt orders the solver, "even if you recognize what the executable is," to "reimplement it from behavioral observation alone" (§A.2), and it casts the different-language ablation as forcing "deep understanding of program behavior rather than surface-level recall" (§4.1). It set out to measure reconstruction and to suppress recall, and we show the construction cannot deliver that.

The binary is queryable, so probe it to resolve any ambiguity. This holds for the option space and not the value space, and the distinction is the argument. Flags and subcommands are documented and enumerable: a solver reads the help, tries the handful of options, even runs an order-sensitive flag both ways, and observes the reference's choice, so ambiguity in what a documented flag does is self-resolving by probing. Untyped inputs are not enumerable. A string, a byte stream, a number at its edges has no finite set of cases to walk, and the values that trigger a distinct behavior are sparse and unsignposted in an unbounded space. The oracle answers any value submitted, but the solver cannot enumerate which values matter, and the source that marks them is withheld. Probing settles the flags; it does not settle the values, and the value space is where the recall witnesses of the appendix live.

The benchmark bundles the inputs, so an H.265 video or an obscure binary asset is provided, not synthesized. True, and the paper does this on purpose, conceding that without it "evaluation uses assets that … models are unable to generate on their own" and that this is an "unfair asymmetry" (§2.2, §A.2). But a bundled input does not make the behavior reconstructable. Given the H.265 file, matching the reference's decode requires the H.265 codec; given a corpus, matching its output requires the compressor. The asset is the input; the behavior on it is fixed by an algorithm the supplied docs do not contain, that input-output pairs do not reveal, and that no internet can fetch. Bundling settles input availability and leaves the recall barrier untouched.

20 Witness receipts

The benchmark's generated suites are public (`programbench/ProgramBench-Tests`), so a task's verdict need not be inferred; we read its assertions. For each program we look for one recall-only test, a graded assertion no source-blind, offline solver can satisfy, and stop at the first, because the conjunctive metric lets one such test sink the task. The unit a runner excludes is the program; the receipt is the assertion.

What the assertions confirm comes in two kinds.

The first is an un-inferable function over an open domain: the assertion compares output, byte for byte, against a bundled golden the reference algorithm produced, on an input the solver cannot pre-query, where the function is a hash, a cipher, a codec, or a compressor. No finite sample identifies it and the specification is offline. The repair is to ship the specification or to score the program on a separate knowledge track. The second is an implementation detail pinned in a golden: a value that is the reference's own, not the contract's, such as an exact rendered file or terminal frame, which no independent implementation reproduces. That is a test-quality defect, repairable by normalizing the golden or asserting the contract instead of the trace.

The contrast is the discipline. A roundtrip test grades a compressor without pinning its bytes: xz and pigz compress with the candidate and decompress with the standard library's lzma and zlib, so any valid implementation passes, and both are benchable. The difference from blake3 is not the domain; it is whether the assertion demands an output only the reference can produce. The line is exact: the standard library carries zlib, gzip, bz2 and lzma, so gzip, bzip2 and xz are roundtrippable, while zstd, lz4 and brotli are absent from it, so a golden that pins their output is recall.

Table A1. Verdict summary. The audit is a one-sided search. It confirms a program unbenchable by exhibiting one recall witness, but it cannot certify benchability, which would require exhausting a suite of

up to 14,645 tests against a strict comparator. The count below is therefore a floor, and the residual is "no witness found," not a clean bill. The unit is the program: one recall witness forecloses resolution.

Verdict	Programs	Basis
Unbenchable, recall witness verified	21	a graded test demands an un-inferable function (hash, cipher, codec, compressor, opaque binary format, Unicode width table) compared to a bundled golden; a positive existence proof
Unbenchable, golden pins an implementation detail	3	a byte-exact render (ditaa PNG, typst PDF) or per-frame hash (ffmpeg) the contract does not fix; a test-quality defect rather than recall
No witness found	176	no witness surfaced in a complete read of the exact-match assertions; not a certificate of benchability
Unread	1	test bodies absent from the released set (`calculator`, a placeholder fixture task)
Total	201	

The twenty-one are a floor, not a count of the unfair set, but they survive a complete read of the published suites rather than a sample. The search stays one-sided, so "no witness found" remains the absence of a surfaced witness. A contestable tier sits just outside the floor.

Table A2. Recall witnesses. The spine. Each row names one graded test whose expected value is the output of a function no source-blind, offline solver can reproduce. Formally, each is a hidden assertion demanding `g(x*)` for a function `g` whose specification is neither supplied nor inferable from any feasible set of black-box evaluations on inputs other than `x*`, so observing `g` elsewhere fixes nothing about `g(x*)`. The test is not defective: it grades a real skill, recalling a published algorithm, and the only repair is to rescope, by shipping the specification or scoring the program on a separate knowledge track.

Program	Witness test	Graded behavior, and why recall
blake3	`test_chunk_boundary_1024_bytes_exact`	`b3sum` of a bundled file vs a `.golden` digest; BLAKE3 is not inferable from input-output pairs and its spec is offline
php-src	`test_phpt_file[gost]`	a GOST hash output vs the `.phpt` expected vector
7zip	`test_scrc_hash_functions_xxh64`	an XXH64 digest vs a golden; xxHash is not in any standard library
age	`test_decrypt_existing`	decrypts a bundled `.age` file; requires X25519, ChaCha20-Poly1305 and scrypt, none in stdlib
brotli	`test_binary_data_decompression`	decompresses a bundled `.compressed` to exact bytes; brotli is not in stdlib
zstd	`test_golden_decompression`	decompresses a bundled `.zst` to exact bytes; zstd is not in stdlib
lz4	`test_cli_golden`	decompressor output vs a bundled golden; lz4 is absent from the standard library, where gzip and xz are present
sox	`test_lpc10_statistical_properties`	statistics of an LPC10-decoded asset vs a golden; the LPC10 speech codec is not in stdlib
jp2a	`test_ext_width_default_palette`	ASCII render vs a libjpeg-decoded fixture; the decoder's rounding is in no standard library
pixterm	`test_webp_format_support`	render of a WebP image vs a golden; WebP decode is not in stdlib

Execute-only binaries and hidden tests reward prior knowledge of algorithms, formats, and reference traces.

Program	Witness test	Graded behavior, and why recall
chafa	`test_loaders`	ANSI render of a decoded image vs a golden; the implementation language (C) carries no standard image decoder
fasttext	`test_print_sentence_vectors_basic`	printed sentence vectors vs a golden; the fastText embedding algorithm
gdal	`test_raster_info_checksum`	the image checksum (e.g. `4672`) from GDALChecksumImage, which is not in stdlib
samtools	`test_depth_basic_output_format`	`depth` over a bundled BAM vs a golden; the BAM binary record format is not in stdlib
bedtools2	`test_bamtobed_t1_one_block_no_split`	`bamtobed` of a bundled BAM vs a golden BED; the same BAM format, decoded
dsq	`test_parquet_type_preservation`	an exact column value from a bundled `.parquet` (and a `.avro` union in its sibling test); neither format is in stdlib
duckdb	`test_import_parquet`	query output over a bundled `.parquet` vs a golden; Parquet decode is not in stdlib
parqeye	`test_tui_data_display`	a terminal render of a bundled `.parquet`; the view exists only by decoding it
elfcat	`test_elf32`	HTML of an ELF binary's structure vs a `.golden`; the ELF binary layout is not in stdlib
ov	`test_zstd_decompression`	a screen render that requires zstd-decompressing a bundled fixture; zstd is not in stdlib
csview	`test_cjk_emoji_with_padding`	a CJK and emoji table padded into aligned columns vs a golden; the Unicode width table is not in the Rust stdlib

Every row is re-derivable from the task's tarball in `programbench/ProgramBench-Tests`: the witness test names the assertion and the bundled golden it cites. The list grows only by addition, since each row is an independent existence proof. The full per-program verdict over all 201 directories, the complete exact-output inventory the witnesses were drawn from, the audit's withdrawn-screen history, and the re-runnable pipeline that regenerates every verdict are in the companion repo, github.com/kimjune01/program-bench-audit.

Adjacent but not recall: the implementation-detail tier. Three programs fail reconstruction for the second reason above, and Table A1 counts them apart. ditaa and typst compare a rendered artifact, a PNG and a PDF, byte for byte against a golden, and ffmpeg pins per-frame MD5s of a rendered clip. Matching any of them demands the reference's exact rasterizer, font embedding or frame pipeline, bytes that are the reference's own, which no independent implementation reproduces. Unlike a recall witness, the repair is local: normalize the golden or assert the contract. Counting them as recall would overstate the floor, so they are tabled separately; the spine stays the un-inferable functions.

Excluded from the floor: the contestable tier. Some programs grade a structured exact output a competent solver could derive from examples plus general engineering knowledge, so we do not count them as recall.

program	graded output	why derivable, not recall
proj	coordinate-projection math	reconstructable from examples and engineering knowledge
tinycc, quickjs, luajit	machine code or bytecode	a known compilation transform
tree-sitter	compiled WebAssembly	a known compilation transform
pandoc	a `.docx`	a ZIP of XML the standard library opens

Execute-only binaries and hidden tests reward prior knowledge of algorithms, formats, and reference traces.

program	graded output	why derivable, not recall
ascii-image-converter	a decoded PNG	`image/png` is in the Go standard library; its C sibling chafa has none and earns a witness above
tuc, solar	cut codepoints, a truncated ASCII line	the standard library suffices, where csview pads CJK and emoji, needs the width table, and earns a witness above

A referee could contest the remainder either way; we leave them out rather than inflate the floor. The twenty-one are what survives that conservatism.

There is an uncomfortable reading of grader thoroughness in this. Each test the generator adds is another chance to assert against a golden only the reference produces, so the more thorough the suite, the more of its tasks pass out of reach of reconstruction and into reach only of recall. Thoroughness and unfairness compound.

LLM use

This work used a large language model (Claude, Anthropic) in two roles. For the audit, the model read the public test suites and proposed a classification for each graded assertion; the author adjudicated every verdict, and each is a re-fetchable receipt, the cited test and the bundled golden it checks, that a reader can re-derive by grep over the public suites without trusting the model or the author. The prose was drafted and revised with the same model under the author's editing. The model's contribution is extraction and drafting; the warrant for every claim is the re-derivable receipt, not the model's say-so.

References

Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S., 2015. The Oracle Problem in Software Testing: A Survey. IEEE Transactions on Software Engineering 41(5):507-525. doi:10.1109/TSE.2014.2372785

Davis, M.D., and Weyuker, E.J., 1981. Pseudo-oracles for non-testable programs. Proceedings of the ACM '81 Conference, 254-257. doi:10.1145/800175.809889

Dijkstra, E.W., 1970. Notes on Structured Programming (EWD249). Circulated manuscript. "Testing shows the presence, not the absence of bugs."

Gold, E.M., 1967. Language Identification in the Limit. Information and Control 10(5):447-474. doi:10.1016/S0019-9958(67)91165-5

Howden, W.E., 1978. Theoretical and Empirical Studies of Program Testing. IEEE Transactions on Software Engineering 4(4):293-298. doi:10.1109/TSE.1978.231514

Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K., 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? ICLR 2024. arXiv:2310.06770

OpenAI, 2024. Introducing SWE-bench Verified. openai.com. A 500-task, human-validated subset of SWE-bench; deprecated February 2026 for test flaws and training-data contamination.

Rice, H.G., 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. Transactions of the American Mathematical Society 74:358-366. doi:10.1090/S0002-9947-1953-0053041-6

Weyuker, E.J., 1982. On Testing Non-testable Programs. The Computer Journal 25(4):465-470. doi:10.1093/comjnl/25.4.465

Yang et al., 2026. ProgramBench. arXiv:2605.03546. The benchmark under review; sections cited inline as (§n).