

Your tests pass. That is not the same as your tests working.

TestGuard takes the promises your code makes, breaks each one on purpose, and reports every promise your tests did not notice breaking.

THE PROBLEM

Coverage tells you a line ran. It never tells you anyone checked the result. So a codebase can be fully covered and completely undefended — and nothing in CI will say a word.

```
A REAL TEST, FROM THIS PROJECT'S OWN FIXTURE
expect(store.writeAudit).toHaveBeenCalledWith(
  expect.objectContaining({ action: 'MASK', scope: 'g1', ruleCount: 1 }),
); // `content` is never named — so nothing checks it
```

`objectContaining` ignores keys it does not list. Swap the redacted text for the **raw secret** and this test still passes. Coverage of that line: 100%. The audit log now leaks the very thing it exists to protect.

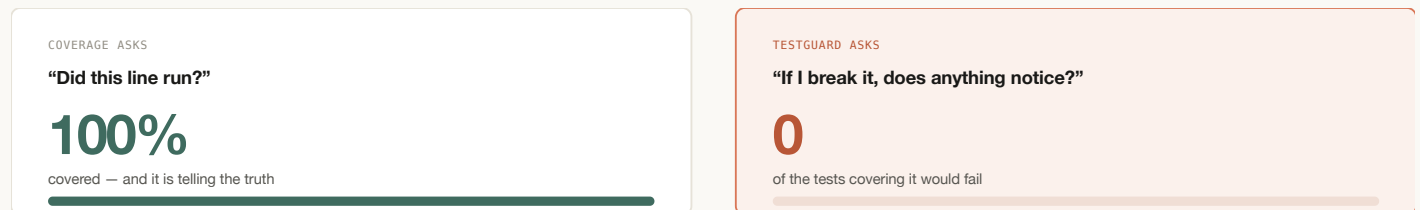


FIG. 1 — THE SAME LINE OF CODE, UNDER TWO DIFFERENT QUESTIONS. ONLY ONE OF THEM IS WORTH KNOWING.

THE METHOD

killed

You broke it — a test failed. **Good.** That behaviour is genuinely defended.

SURVIVED

You broke it — everything stayed green. **A blind spot.**

The word trips everyone once: it is the *fault* that survived, not the test. SURVIVED is the bad news. A healthy report is full of *killed*.

A tool that reports everything as caught is worse than no tool at all — because nobody questions good news.

THE DESIGN CONSTRAINT EVERYTHING ELSE FOLLOWS FROM

FIELD REPORT A - AI-AUTHORED PRODUCTION CODEBASE, ~4,900 TESTS

coverage on the compliance path that leaked

FIELD REPORT B – A DIFFERENT CODEBASE, 458 TESTS, 24 SECURITY CLAIMS, 39 FAULTS INJECTED

of them critical — super-admin gating, cookie flags, the whole authorization callback

The suite was not bad.
It was unexamined.

● KILLED — DEFENDED ● SURVIVED — A BLIND SPOT

FIG. 2 - MEASURED, NOT MODELLED. EACH DOT IS ONE INJECTED FAULT.

The largest gap ran through a compliance-critical path at 100% coverage, where the one assertion that mattered used `objectContaining` and omitted the field carrying the data. Exactly the exhibit on page one.

WHY THE QUESTION HAS TO BE ASKED ADVERSARIALLY

ONE FAULT, START TO VERDICT

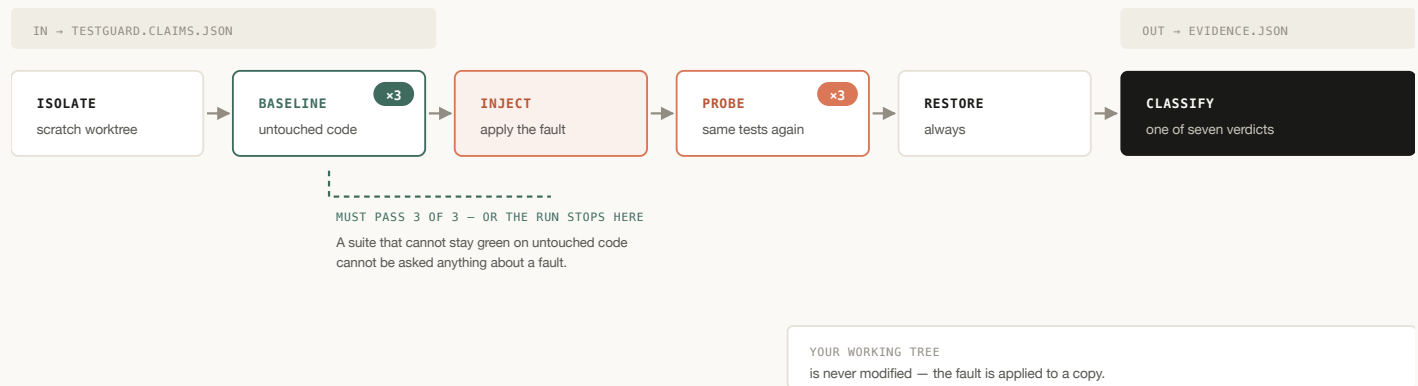
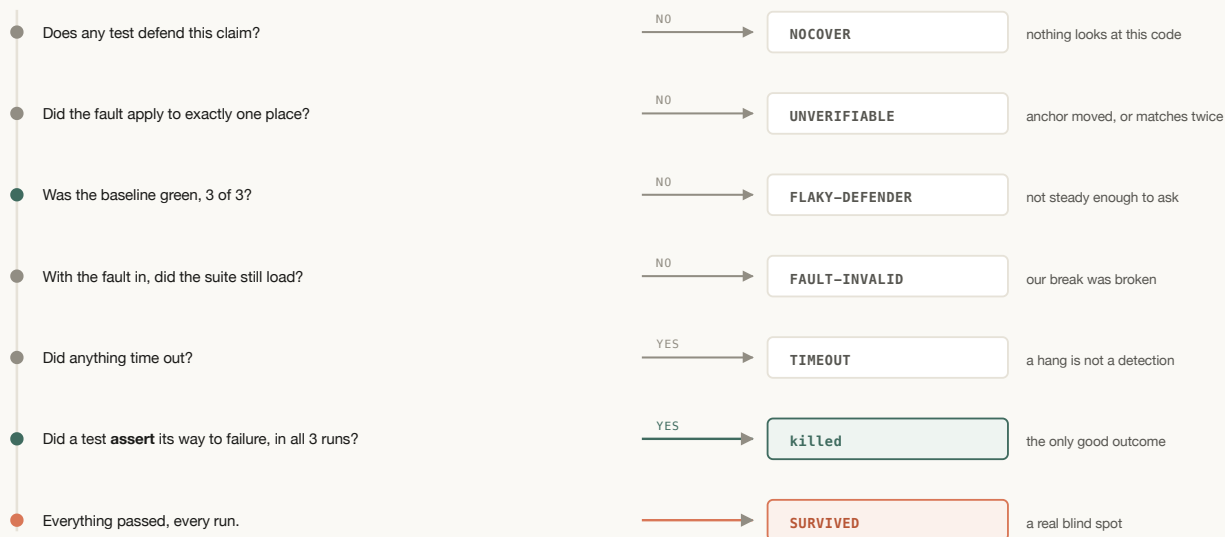


FIG. 3 — THE PROBE LOOP. EVERY STEP REPEATS PER FAULT; THE BASELINE IS CACHED PER DEFENDER SET.

HOW A VERDICT IS DECIDED

The order is the specification. Each check short-circuits the ones below it, so the tool never reaches an optimistic answer through a question it should not have asked.



ANYTHING IN BETWEEN — SOME RUNS KILLED, SOME PASSED — IS FLAKY-DEFENDER, NEVER A KILL.

FIG. 4 — VERDICT ORDER. EVERY AMBIGUITY RESOLVES DOWNWARD, TOWARD "UNPROVEN".

WHAT A RUN COSTS, AND WHY IT IS PREDICTABLE

Wall clock is not a mystery. For each claim:

$$(\text{baseline runs} + \text{faults} \times N) \times \text{cost of the claim's defender set}$$

The defender set is the unit, not the file — a run executes all of a claim's tests at once. So one slow acceptance test named by five claims is paid for thirty times over, and the report names that file rather than leaving you to guess.

Baselines are cached per defender set, and a verdict is reused when the source, the defenders *and* the fault are all unchanged.

SEVEN VERDICTS, BECAUSE “DID THE TESTS FAIL?” IS A SLOPPY QUESTION

VERDICT	WHAT IT MEANS
killed	A test body ran and rejected the behaviour. The only good outcome.
SURVIVED	Everything passed. A real blind spot in the tests.
NOCOVER	No test even looks at this code. Not “weak tests” — <i>no</i> tests.
UNVERIFIABLE	The fault could not be applied: its anchor moved, or matches twice. Nothing was learned.
FAULT-INVALID	The break itself was broken — it did not compile. Our fault, not yours.
TIMEOUT	The suite hung. A hang is not a detection.
FLAKY-DEFENDER	The tests are not reliable enough on untouched code to be asked the question.

WHERE THE PIECES SIT

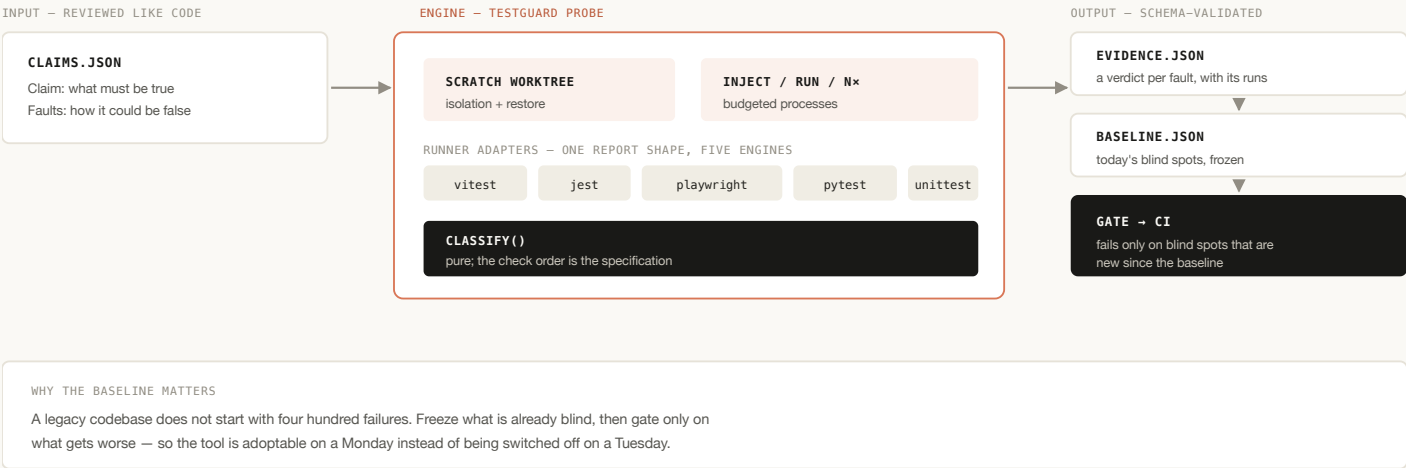


FIG. 5 – CLAIMS IN, EVIDENCE OUT, AND A GATE THAT MEASURES THE DELTA RATHER THAN THE DEBT.

EVERY AMBIGUITY ROUNDS TOWARD “UNPROVEN”

- Three runs, never one.
- A green baseline before any fault is injected.
- A timeout is never a kill.
- A suite that fails to load is never a kill.
- Mixed results across runs are not a kill.
- A test that mocks what it defends is not a defender.

WHAT IS NOT NEW

Breaking code to test your tests is **mutation testing**, and it dates to the 1970s. Stryker, PIT, mutmut and Cosmic Ray all do it. Anyone claiming that part is novel is selling something.

WHAT IS DIFFERENT: THE QUESTION BEING ASKED

CLASSIC MUTATION TESTING Mutates everything, mechanically "How good are my tests?" → Mutation score: 73% A number. Not actionable, not auditable, and it cannot tell you which promise is at risk.	TESTGUARD Binds every fault to a stated claim "Is <i>this specific promise</i> defended?" → "Your project says a missing scope fails closed. Nothing checks that." A finding. An assessor reads your claims — reviewable like code — not a score.
---	--

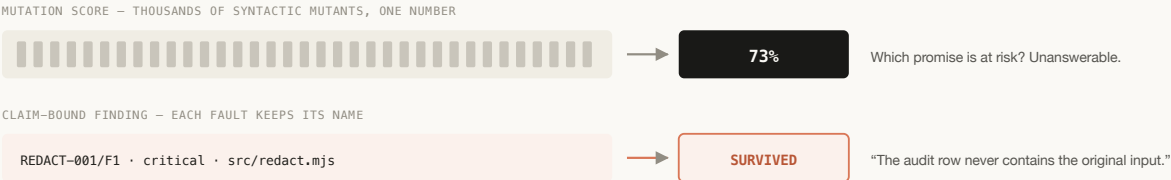


FIG. 6 — A SCORE COMPRESSES EVERYTHING INTO ONE NUMBER; A CLAIM KEEPS THE FINDING ATTRIBUTABLE.

That is the difference between a **metric** and an **audit**, and it is the whole design.

THREE MORE THINGS THAT FOLLOW FROM IT

Paranoid about itself

Most tools mark a mutant killed when the test command exits non-zero — conflating “a test caught it” with “everything exploded.” Here the verdict set is closed and each member means something different.

Evidence, not output

Schema-validated JSON with stable fingerprints, shared with the other Guard tools. Freeze today’s blind spots, gate CI on *new* ones.

Built for AI-written code

When an agent writes the tests, the failure mode is not “no tests.” It is confident, plausible tests that assert nothing. That is precisely what this detects.

Coverage tells you a line ran. TestGuard tells you which of your promises nobody is actually defending.

GITHUB.COM/RACCIOLY/TESTGUARD · NPM TESTGUARD-CLI · MIT