What AI Auditors Miss: Diagnosing and Closing the Gap Between AI and Human Web3 Security Audits

Zhiju Yang[1][0000-0003-2831-7424], Yue Xue[2], and Xiecongyou Yang[2]

[1] Seattle University, Seattle, WA, USA
zhijuyang@seattleu.edu
[2] CertiK, New York, NY, USA
{yue.xue,xiecongyou.yang}@certik.com

Abstract. AI code auditors now run in production, and they still miss real bugs. We study one commercial smart-contract AI auditor against 298 completed human audits holding 5162 findings. The auditor covers 69% of the 1766 findings rated critical, major, or medium; we label the 548 misses with a ten-class why-missed taxonomy. Project characteristics explain only 3% of the coverage variance. How it fails, by contrast, is structured: in 71% of the misses the auditor had already reported a different bug in the same file (a reliability gap); in the other 29% it never engaged the file (a blind spot). On 141 reliability-gap misses with resolvable source, re-running the same model on the same file recovers 18% and an eight-sample ensemble 25%; retrieved audit rules never beat a plain rescan in any of three injection framings, so knowledge was not this auditor's bottleneck. Thirteen detectors, each targeting one vulnerability family and selected per function, recover 46% of those 141, 52% running all thirteen; re-scoring credited findings against another project's bug credits none, putting the measured false-recovery floor at zero. The cost is false positives: the union emits a median of 35 findings per function at near 5% precision, and the filtering stage a deployment needs is outside this evaluation. We report what is recoverable, not a deployable scanner. For this system, reliability and family depth close the gap; more rules do not.

Keywords: Smart contract security · AI code auditing · Large language models · Vulnerability detection.

1 Introduction

Web3 auditing companies now use AI auditors that read a codebase and return a vulnerability report to assist human auditors, and the tools find real bugs at a useful rate. They also miss real bugs, including critical ones. The interesting question is no longer whether an AI auditor misses, but what it misses, why, and what a scanner designer should build next; a benchmark score says the tool caught some fraction of a fixed set and nothing about the structure of the

complement. A knowledge gap wants a new detector, a localization failure wants better planning, a sampling failure wants a second draw. Treating all misses as one undifferentiated mass wastes engineering effort on the wrong axis, and we will show that the intuitively appealing axis, giving the model more rules, is the wrong one.

We study the misses of one production AI auditor for smart contracts: a multi-stage pipeline over large language models that has run on hundreds of paid engagements, so its output reflects a deployed configuration rather than a lab setup. For each engagement we hold the human audit report and the AI report for the same code; matching the two yields the findings the AI should have reported and did not. The catalog holds 5162 human findings across 298 projects. We focus on its high-severity core, where the AI covers 69.0% and the 548 not strictly covered form our benchmark of misses.

We label each miss with a ten-class taxonomy of why it was missed and ask what predicts a miss. Project characteristics barely predict coverage: a cluster-robust model over project metadata explains 3% of the coverage variance against a project fixed-effects ceiling of 27%, so roughly 73% of most of the variance is within-project and finding-specific. The single robust project-level signal is a cliff at the bottom of the complexity scale, where trivially simple projects are covered at 34%; several other apparent signals dissolve under cluster-robust inference.

What is predictable is the shape of the miss. For 71% of the misses the AI reported a different bug in the same file, and in 77% of those the same function; the code entered the scan window, necessary for detection but not sufficient. We call these reliability gaps, and the remaining 29% blind spots. Unlike coverage, this split is strongly structured. The capability claim rests on a second measurement that uses no location information at all: for 86.5% of misses the AI demonstrably catches the same why-missed class elsewhere in the catalog. The dominant failure mode of a production AI auditor is not ignorance. It is unreliability.

The diagnosis makes testable predictions, and the second half of the paper tests them. Our first four scanners all failed to recover more than a small fraction of the misses. The turning point was a control condition: re-running the same model on the same file recovers 18% of reliability-gap misses, and an eight-sample ensemble 25%; those rates lower-bound the share of misses that needed no new capability rather than measuring why the rest were missed. Injecting retrieved audit rules never beats the plain rescan, and at volume it hurts; but rules do win inside two mechanical categories, which points at the right axis: match the intervention to the bug's family. We build thirteen deep single-family detectors, route each function to the families its code suggests, and union the results, recovering 46% of the misses, about three times the rescan. The cost: about 35 findings fired per function, roughly two of them real, so a shippable version still needs deduplication and a precision stage.

This paper makes four contributions.

– A benchmark of 548 misses, matched against human audit reports under a frozen LLM-judge rubric, with a ten-class why-missed taxonomy (Section 2).

- A measurement of what predicts a miss, with cluster-robust inference: coverage is finding-specific and weakly tied to project characteristics, while the failure mode, reliability gap versus blind spot, is strongly structured. Several plausible claims from naive analysis are shown to be artifacts (Section 3).
- A reliability-versus-capability diagnosis, reframing the engineering problem from "make the model smarter" to "make the pipeline more reliable, and add depth only where capability genuinely fails" (Section 4).
- An intervention study with matched negative and positive results: four failed scanner designs, a rule-injection null spanning three framings, a variance-reduction ladder, and a routed mixture of deep family detectors that triples recovery at a zero false-recovery floor (Section 5).

2 Background and Dataset

2.1 The AI auditor under study

The system we study is CertiK's production AI auditor, a multi-stage LLM pipeline for smart-contract security review, whose reports were produced between January and June 2026. A loader ingests and parses the project; a planner identifies the files and functions that matter, extracts call-graph context, and selects relevant security checklists; a scanner stage runs several LLM-based scanning strategies over the planned targets, largely at file and function granularity; a deduplication stage merges overlapping candidates; a validator re-examines each candidate to cut false positives; and an enhancer assigns severity, category, description, and recommendation to the survivors.

The authors include CertiK employees; CertiK builds and sells the system under study. That affiliation gave us production access and the AI and human reports from hundreds of paid engagements; it also means we grade our own product, so every comparison runs through the judge rubric of Section 2.3, frozen before any intervention, and the adversarial control of Section 5.8.

The stage structure is not idiosyncratic: published LLM auditing systems also split candidate generation from a separate verification, filtering, or ranking stage, using program-analysis confirmation in GPTScan [15], an adversarial critic in GPTLens [8], and decoupled reasoning components in LLM4Vuln [14]. Our results bear on that shared shape and claim nothing wider: the interventions of Section 5 are model-agnostic in design, but the percentages are this system's.

2.2 Ground truth and the finding catalog

Ground truth is the completed human audit: for each of 298 projects, the final report written by a professional audit team, 5162 findings in total. The catalog spans fourteen named source languages plus a residual *other* label: 247 of the 298 projects include Solidity and carry 3748 of the 5162 findings, while the remaining 51 are led by Rust (23 projects), TypeScript (7) and Go (6), with Haskell, Java, JavaScript, Move, Aiken, FunC, Python, C++, Kotlin and Swift

appearing in ones and twos. Section 3 tests language as a coverage predictor; the intervention targets of Section 5.2 are Solidity only. Human reports are imperfect ground truth: auditors miss bugs too, and a bug the human team and the AI both missed never enters the catalog, so we measure coverage of the professional standard, an upper bound on coverage of the bugs actually present (Section 7). They remain the strongest available reference and the standard the tool is sold against. We split the catalog into a core tier, the 1766 findings rated critical, major, or medium, where security value concentrates, and an extended tier of 3396 mostly informational, minor, optimization, and centralization notes. Unless stated otherwise, every number in this paper is over the core tier. Full-catalog coverage over all 5162 findings is 54%; the gap to the core's 69% is mostly the AI declining low-value housekeeping findings, acceptable behavior rather than misses worth engineering against.

2.3 Matching AI findings to human findings

Deciding whether an AI finding and a human finding describe the same bug is the load-bearing step in benchmark construction. Titles, line numbers, and framing differ, and hand-matching five thousand findings does not scale, so we use an LLM judge with a frozen rubric. The rubric defines four verdicts. **Exact**: the same root cause, code elements, and failure mechanism. **Root cause**: the same underlying cause at the same component, with framing, extent, or emphasis differing. **Partial**: overlapping but not equal, such that an auditor reading the AI finding would plausibly arrive at the reported issue. **None**: no shared mechanism. A human finding is *covered strictly* if some AI finding matches it at exact or root-cause level, and *covered partially* if the best match is partial. The rubric text is hashed and pinned across every experiment, so a recovery claimed in Section 5 is judged by exactly the standard that defined a miss. The judge is `gpt-5.4` of the GPT-5 family, whose parameter count the vendor does not disclose; it is one of the models the production pipeline itself ran during the study window, so the judge is no stronger a reader than the system it grades. Every match decision is the majority of three independent judge calls, calibrated before freezing against a human-labeled sample of 120 items, 60 pairing one finding against one AI candidate and 60 pitting a finding against a whole AI catalog, where the judge reproduced the human verdict on 95% and 85% of them respectively; Section 5.8 adds an adversarial check for the headline result.

Applying the frozen matcher to the core tier gives 1218 findings covered strictly, 180 covered partially, and 368 uncovered. Our miss set is the 548 findings not covered strictly, the uncovered plus the partials, because a partial credit still means the client would not have received the actual finding: 32 critical, 109 major, 407 medium. Coverage is flat across the value scale, 70.6% on criticals, 71.0% on criticals and majors pooled, 69.0% on the full core.

Table 1. The 548 core misses by why-missed class. "Same file" is the share of the class where the AI reported a different bug in the same file (Section 3.3).

Why-missed class	Count	Same file	What the AI would have needed
Checklist pattern	125	64%	apply a known audit rule to the code
Business-logic spec	99	82%	spot deviations from intended behavior
Deep state-machine	93	83%	reason over a multi-step stateful flow
Cross-contract reasoning	89	73%	compose facts across files, contracts
External context	68	47%	use off-chain context: docs, deployment
Numerical precision	26	81%	track rounding, scaling, and decimals
Economic incentive	21	81%	model attacker profit and game theory
Upgrade / deploy	11	91%	reason about proxies, initializers
Gas / DoS	10	60%	bound loops, gas, and resource growth
Validation loss	6	0%	none: the pipeline dropped it

2.4 The why-missed taxonomy

Coverage measures how much is missed and says nothing about why. For each of the 548 misses we therefore ask what capability the AI would have needed, encoded in a ten-class taxonomy: a labeling pass, prompted with the human finding, the capability required to detect it, and the relevant source, assigned one primary class and optional secondary classes. Table 1 lists the classes with their counts.

Three properties of the labels shape how we use them. First, the top classes are not clean silos: deep state-machine and business-logic co-occur as primary and secondary labels on 47 misses, cross-contract and business-logic on 42, and cross-contract and state-machine on 42. We therefore treat the big reasoning classes as one soft cluster and avoid leaning on small count differences between them. Second, a blind second labeling pass reproduced the primary label 50% of the time raw and 75% when secondary labels are credited, with disagreements being defensible adjacent overlaps; we therefore use the taxonomy as a soft spine, and no headline claim depends on the exact boundary between two reasoning classes. Third, "validation loss" is a property of the pipeline rather than of the model's reasoning. It is real and worth fixing, but we exclude those six misses when we reason about capability.

3 What Predicts a Miss

The obvious hypothesis is that some projects are just hard for the AI: big ones, complex ones, unusual languages. If that were true, coverage would track project metadata and a vendor could route hard projects to humans. We tested it with some care, and it is mostly false.

3.1 Project characteristics barely predict coverage

We fit a logistic model of per-finding coverage on project features: log size, complexity, language, severity, and protocol type, over the 1748 core findings

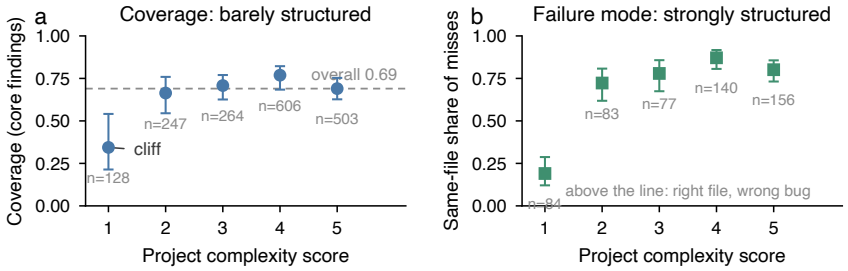


Fig. 1. The central contrast of Section 3. (a) Coverage by project complexity with cluster-bootstrap intervals: a cliff at complexity 1, no gradient above it (feature model $R^2 = 0.03$). (b) The same-file share of misses: strongly structured ($R^2 = 0.26$).

in 231 projects with complete labels. All inference is cluster-robust by project, sandwich estimator plus a project-resampling bootstrap, and this choice is not a formality: findings cluster strongly within projects. Under naive inference, size "significantly" predicts coverage at $p = 0.009$; under cluster-robust inference the same coefficient has $p = 0.26$. Most of the apparent structure in this data is manufactured by clustering.

The model with project features reaches a McFadden pseudo-R^2 of 0.031. A project fixed-effects model, the ceiling for anything expressible at project level, reaches 0.275. Observed features therefore capture about a ninth of the between-project explainable variance, and roughly 73% of coverage variation is within-project: coverage is a property of the specific bug, not of the project it lives in. The individual terms agree. Size does not predict coverage (odds ratio per tenfold increase 1.12, cluster-robust interval [0.92, 1.36]); language does not (Solidity versus non-Solidity under five points, $p = 0.41$); protocol type does not as a block (joint Wald $p = 0.076$); severity does not (OR ∼1.21, [0.91, 1.60]).

One term survives, and it points the opposite way from intuition. Complexity predicts coverage, but as a low-end cliff rather than a gradient (Figure 1a). Projects rated trivially simple, complexity one on a five-point scale, are covered at 34.4%, against 69% overall and 71.9% for everything above, a gap of −0.375 with interval [−0.514, −0.168] and $p < 0.001$, robust to size and severity controls, to trimming dominant projects, and to the cluster bootstrap. Above complexity one there is nothing: levels two through five sit flat between 66% and 77%, and an ordinal test excluding the bottom level is null (β 0.044, $p = 0.635$). The correct statement is not "complex code is harder for the AI" but "trivially simple projects are covered worst": these are small token contracts whose defining risk is adversarial intent, hidden fees, owner backdoors, and rug mechanics, not reasoning depth.

3.2 Measurement pitfalls: claims that did not survive

Earlier passes over the same data produced several publishable-looking claims that turned out to be artifacts; three retractions are instructive. A *size effect*,

coverage zigzagging across size quartiles, reproduces only under a size metric that double-counts files across audit commits, in buckets dominated by single large projects (one contributing 31% of a quartile); deduplicated, the curve is flat, project-weighted it is monotone, and the cluster-robust regression is null. A *language effect*, non-Solidity projects appearing to have a much lower same-file rate (53% versus 81%), vanishes after controlling for size and complexity (language OR 0.98, $p = 0.97$): those projects are simply bigger and differently distributed. And the complexity-one cliff was initially reported as "simple adversarial tokens are covered at 34%," which is wrong: real ERC-20 tokens at complexity one are covered at 55.6%, the 34% figure is dragged down by large non-token engagements in the same band, and only the cliff itself is robust, surviving jackknife removal of the worst project. We kept the corrected claim and dropped the story. A fourth pattern, low-complexity projects skewing toward checklist misses (27.0% versus 19.9%), does not survive correction (BH $q = 0.129$) and we treat it as descriptive only.

3.3 The failure mode is structured, even when coverage is not

Coverage is finding-specific, but the way the AI fails is not. For each miss we join back to the AI's own report and ask whether the AI reported any finding, on any topic, in the same file, after normalizing paths. The answer is yes for 71.0% of the 548 misses, cluster interval [0.61, 0.80]. Among misses whose location resolves at all, the rate is 82%, and within those files the AI named the same contract 90% of the time and the same function 77% of the time. For most misses the AI was standing on the bug and reported its neighbor. The join locates the failure; it does not establish ability. That the file entered the scan window is necessary for the AI to report the bug and not sufficient, and the capability half of the diagnosis is carried by the class-recall measurement of Section 4, which uses no locations.

We call the 71% reliability gaps and the 29% blind spots, and the split is structured in exactly the way coverage is not. A model of the same-file indicator on project features reaches $R^2 = 0.26$, roughly eight times the coverage model, with complexity pushing strongly toward reliability gaps (OR 2.61 per step) and size pushing toward blind spots (OR 0.43 per tenfold), both $p < 10^{-4}$. Figure 1b shows the gradient: the same-file share climbs from 19% at complexity one to 87% at complexity four. The one robust compositional shift completes the picture: cross-contract misses rise from 10% to 22% of the miss mix as complexity grows (BH $q = 0.002$). The bridge to everything that follows: for complex, high-value DeFi the AI is engaged but unreliable, and the fix should make it fire consistently where it already looks; for trivially simple and very large repositories the AI is disengaged, and the fix is coverage engineering, not intelligence.

4 Reliability Versus Capability

4.1 Two measurements that agree

The same-file join is one line of evidence, and it bounds engagement rather than ability. A second shares none of its assumptions: across the whole catalog, at what rate does the AI cover findings of the same why-missed class?

The answer is that every class is demonstrated at least once, and weighted by class recall the AI catches the classes behind its misses at a mean rate of 63.5%, bootstrap interval [61.3, 64.8]. Drawing a threshold at one half, 86.5% of the misses fall in within-capability classes and 13.5% in genuine capability classes: external context, which the AI catches only 21% of the time because the needed information is often not in the repository at all, and validation loss. Everything else sits between 57% and 79%. The two measurements come from different data along different logic and carry different halves of the claim: the same-file rate, 71 to 82%, shows only that the code was in front of the model, necessary for reporting the bug and not sufficient; the 86.5% within-capability rate establishes the ability, and uses no locations. Read together, the model usually has the ability and usually looks in the right place, but does not fire consistently. This inverts the field's default instinct, smarter models and more knowledge, and Section 5 shows the inversion is not just interpretive.

4.2 Where capability genuinely fails

The reliability reading does not mean capability gaps are negligible, and at critical severity they have a specific address. Over the 109 critical findings, per-family catch rates (for families with at least six criticals) show the AI strong exactly where the money moves, 96% on fund-flow accounting (22/23), 90% on business-logic invariants (19/21), and 85% on access control (17/20), with input validation at 5/9 and generic code-quality findings at 0/6. It is weak on two families: critical oracle-manipulation findings are caught 1 time in 6, and critical denial-of-service findings 7 in 16. The oracle weakness is severity-specific, since at major severity oracle findings are caught at 93%; what the AI misses is precisely the catastrophic variant, where a manipulable price feeds liquidation or minting at scale.

The blind spots decompose too: of the 153 misses where the AI reported nothing in the file (at the stricter location join), 83 have no resolvable code location at all, dominated by external-context and spec-style findings; no amount of scanning reaches them, since they need documentation, deployment configuration, or the client's intent as input. Of the 70 that do resolve, 67 are in-scope files the scanner should have read, 41 of those in repositories with more than one hundred files, which points at coverage decay at scale rather than at reasoning. This also fixes the scope of the intervention study: we evaluate recovery on reliability-gap misses, where the file is known and the failure is consistency, the dominant mode and the one a scanner-side fix can reach.

5 Interventions

The diagnosis makes predictions: if misses are dominated by variance, resampling should recover a meaningful share and knowledge injection should not add much; if a minority are family-specific capability gaps, family-specialist depth should recover what breadth cannot. This section tests those predictions, beginning with what we actually built first, because the failures forced the diagnosis-driven design.

5.1 First attempts: four scanners that failed

Our first instinct, like most of the field's, was to build smarter scanners; we built four, all judged by the same frozen rubric on held-out targets. The *memorized-rule scanner* mined sixteen concrete rules from checklist-class training misses and recovered nothing held out, for a structural reason: sixteen of the nineteen held-out misses required a rule the training misses never produced. Checklist-class bugs form a long tail of near-unique patterns, and a finite mined rule list cannot cover the tail. The *structural-property scanner* replaced mined rules with six general properties to verify interprocedurally and reached 11% on the same nineteen held-out targets, its recoveries exactly the cases where a property forced source-to-sink tracing. Reasoning about the code beats matching patterns in it, but a handful of generic properties inherits the same coverage problem one level up.

The *reasoning scanner* dropped rules entirely and asked the model to reason about cross-contract and state-machine misses with the whole project available. On a powered, leakage-clean evaluation of 203 units it recovered 15% of cross-contract misses in project mode and 9% of state-machine misses, rising to 19% and 17% when pointed at the right entrypoint file. The localization tax, four to eight points, is real but modest, so finding the right file is not the main obstacle; and even with the file given, four fifths of the hard misses stayed missed. An *agentic variant* that hypothesized, investigated with tools, and verified recovered 7% on a fifteen-target pilot, no better than a single pass; the bottleneck was not orchestration but that the model never generated the right hypothesis. Four designs, one pattern: everything hovered between zero and twenty percent regardless of machinery. That flat ceiling sent us back to measurement and produced the reliability diagnosis of Sections 3 and 4.

5.2 Experimental protocol

All remaining experiments share one protocol. The target set is 141 reliability-gap misses spanning 91 projects: misses where the AI had reported some other bug in the same file, and where that file resolves to scannable source. Every scanner variant sees the code and nothing else, with no access to the human finding, its location, or any derivative of it; scanners read a window of up to 22k characters around the target function's file. A target counts as recovered if any emitted finding matches the human finding at exact or root-cause level (strict)

Table 2. Recovery of the 141 reliability-gap misses (91 projects). Strict = exact or root cause under the frozen three-vote judge; lenient adds partial. McNemar is the paired test against the rescan control on the same targets.

Lever	Intervention	Strict	95% CI	Lenient	McNemar p
baseline	generic focused rescan (control)	18.4%	[12.3, 25.0]	28.4%	
rules	forced 26-check engine	14.9%	[9.4, 21.2]	21.3%	worse, p=0.44
rules	checklist, substitutive, K=15	4.3%	[2.0, 9.0]	7.8%	loses, p<0.001
rules	checklist, substitutive, K=100	16.3%	[11.1, 23.3]	23.4%	tied, p=0.74
rules	checklist, additive, K=15	16.3%	[11, 23]		tied, p=0.65
rules	checklist, additive, K=100	14.2%	[9, 21]		worse, p=0.26
variance	ensemble, pass@3	21.4%	[15.7, 27.6]	31.8%	
variance	ensemble, pass@8	25.5%	[19.1, 32.2]	36.9%	
depth	routed deep-detector union	**46%**	[38, 54]	57%	
depth	all-13 union (ceiling)	**52%**	[44, 60]	63%	

or at least partial level (lenient) under the frozen three-vote judge of Section 2.3. Intervals are Wilson or project-cluster bootstrap; paired comparisons between arms on the same targets use McNemar's test. We report strict throughout and give lenient in Table 2; no conclusion changes under lenient scoring.

Every arm, from the focused rescan to the thirteen family detectors, runs on the same model, `gpt-5.4` of the GPT-5 family, reached over an OpenAI-compatible internal gateway, so a difference between arms is a difference in intervention and not in model; the vendor does not disclose its parameter count. The router in front of the detectors runs on the same model at medium reasoning effort, the detectors at high. Decoding is left at the library defaults, temperature 1 with top-p and maximum output length unset, one sample per call; only the ensembles of Section 5.4 (3 and 8 samples of the identical prompt) and the routed union of Section 5.7 (three embedded generic rescans) draw more. The harness is Python 3.13 calling the provider SDK directly, with Pydantic schemas for structured output and no orchestration framework. Scan cost is about $0.012 per target for the rescan control and $0.099 for pass@8; we did not meter the routed union, which fires a mean of 9.7 detector lanes per target against 15 for the all-13 arm.

5.3 The baseline is the finding: a rescan recovers 18%

The control condition for everything else is the cheapest intervention imaginable: take the file the AI already scanned, and scan it again with a focused prompt and the same model, adding nothing. This control recovers 18.4% of the misses strictly. These are bugs the production system missed in a file it had already read, with a validation stage behind it. Each recovery is a miss the same model lands on a second pass over the same bytes, so 18.4% lower-bounds the share of misses that need no new capability rather than measuring variance's share of all failures; the misses a second draw leaves behind stay unattributed here. The number disciplines the rest of the section: a scanner that cannot beat a plain rescan has added nothing but cost. Our first diagnosis-era design took

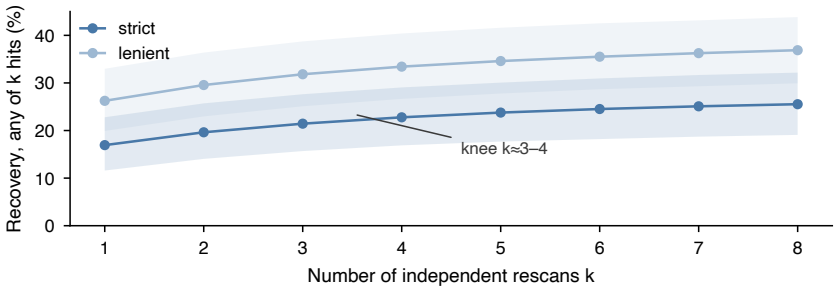


Fig. 2. The pass@k ensemble curve, with bootstrap bands: same prompt, more samples, recovery rising from 17% to 25% strict with a knee near $k = 3$.

the reliability reading literally, forcing the model to be exhaustive: an engine that walks a fixed list of 26 structural checks, fixed for every target so there is nothing to leak. It recovers 14.9%, below the control ($p = 0.44$ strictly, $p = 0.099$ leniently), because walking 26 checks spends the model's attention on generic near-misses and the file's one specific bug gets a thinner slice of the forward pass.

5.4 Ensembling recovers more, cheaply

If a miss is a bad draw, draw again. We run the identical rescan prompt k times independently and count a target recovered if any sample lands it, the pass@k protocol from code generation applied to audit recall. Figure 2 shows the curve, with the marginal sample decaying fast enough that the knee sits near three or four. Recovery climbing smoothly with samples under an identical prompt isolates a variance component, and the cost is a few cents per target; the curve also bounds that component, flattening with three quarters of the targets still unrecovered at eight samples. The cheapest fix today is to ensemble the existing scanner on the files it already reads, and take the union through the existing validation stage.

5.5 Injecting rules does not beat a rescan

The intuitive fix says the model missed a known bug class, so give it the knowledge. The knowledge source is a curated corpus of 1318 concrete audit rules spanning 45 vulnerability categories, written by the audit organization independently of this benchmark; for each target function we embed the code and every rule with a local embedding model and retrieve the top K rules by cosine similarity. We audited the corpus for leakage against the benchmark answers before running anything: the highest code-to-rule similarity is 0.873, and the top pairs are generic patterns, not restatements of answers.

The *substitutive* framing rebuilds the prompt around the retrieved rules, asking the model to verify each against the code. It behaves monotonically in K, 4.3% at $K = 15$, 7.8% at $K = 40$, 16.3% at $K = 100$, so retrieval is functioning,

but the curve saturates at a tie with the plain rescan (16.3% versus 18.4%, McNemar $p = 0.74$) and never crosses it. The *additive* framing is the fairer test of the folk claim that relevant hints can only help: keep the exact rescan prompt, append the retrieved rules as optional hints, and leave the output free. It ties the control at $K = 15$ and then degrades as hints grow, 16.3%, 15.6%, 14.2% at $K = 100$, monotonically worse in the direction more-context-should-help. The conclusion surprised us: In this system, over this 1318-rule corpus, and across the three injection framings we tried, added rule context does not substitute for re-sampling: what retrieval adds, sampling already had. Whether knowledge binds in another auditor, or against a corpus written for a different failure distribution, our data cannot say. And the additive result is sharper than a null: a block of relevant-looking hints anchors generation toward hint-shaped generic findings and away from the file's one specific bug. For scanner designers this deprioritizes a whole design corridor: rule libraries, retrieval pipelines, and checklist prompts, in any mixture we tried, are dominated by re-running the model on this benchmark.

5.6 Where rules do help: inside mechanical categories

If checklists encode mechanical knowledge, they should pay off exactly on mechanical bug classes, so we built category-specific target pools, 49 numerical-precision misses and 21 oracle-usage misses, and ran the checklist arms against a matched rescan control inside each. On numerical misses the retrieved checklist recovers 31% against the control's 8%, a paired win at McNemar $p = 0.007$; on oracle misses 29% against 10%, the same direction without significance at this pool size ($p = 0.22$). The additive-hint framing helps neither pool, consistent with the anchoring reading. A well-selected checklist beats re-sampling only where the bug is a known pattern instantiated in code; on reasoning-heavy classes it is dead weight. The general principle: interventions pay when they match the bug's family, and the way to scale that is not one checklist but a bank of family specialists with a router in front.

5.7 A routed mixture of deep family detectors

We built thirteen detectors, one per family, listed in Table 3. *Deep* here means the depth of threat-model reasoning within one vulnerability family, not deep learning: every detector is an LLM prompt over the same model as the rescan control of Section 5.3, nothing was trained or fine-tuned for this work , and what separates these from the failed rule scanners of Section 5.1 is prompt depth, not model class. Each detector is a specialist prompt built around a single threat model, not a list of patterns: it enumerates the family's failure classes (five for the oracle detector), instructs the model to reason interprocedurally, tracing every source its family cares about to every sink it can corrupt, and grounds each finding in a concrete function and mechanism, reporting nothing when the family does not apply. Each detector was validated on one real miss from its family before the suite ran.

Table 3. The thirteen deep family detectors. Credited counts the strict recoveries (of 73 in the all-13 arm) crediting that detector; a recovery can credit several arms, and the embedded generic rescans and checklist contribute 7 each.

Family	Threat model	Credited
Fee / parameter bounds	unbounded fees and rates, privileged setters	11
ERC-20 integration	fee-on-transfer, missing return values	8
Reward accounting	share and reward bookkeeping invariants	8
Access control	missing or wrong authorization on transitions	6
Input validation	unchecked arguments and degenerate values	6
Liquidity / slippage	AMM interaction, sandwich and MEV exposure	6
DoS / unbounded work	unbounded loops, growth, blocked control flow	5
Arithmetic / rounding	rounding direction, truncation, precision loss	4
Oracle manipulation	manipulable or stale price and valuation sources	3
Reentrancy / CEI	writes after external calls, read-only reentrancy	1
Initialization / upgrade	proxies, initializers, storage layout, migrations	1
Decimals / scaling	decimal mismatches across tokens and feeds	0
Signature replay	signed-message integrity, replay, malleability	0

Routing decides which specialists read which function. The router inspects the target window for each family's lexical and semantic signals, pool and price-feed reads for oracle manipulation, and selects every family with any support, deliberately over-selecting: a wasted detector call costs pennies, a missed family costs a missed bug. It selects about ten of thirteen families on average, achieved perfect routing recall on a labeled sample, and falls back to all thirteen when uncertain. Every target also receives three generic rescans, so the mixture strictly contains the ensemble baseline, and the mechanical families add the retrieved checklist. All findings pool into one union with no deduplication, scored by the standard protocol.

The routed union recovers 46% of the misses strictly. Running all thirteen detectors on every target, the architecture's recall ceiling, reaches 52% strict. Against the 18.4% control and the 25.5% ensemble, the mixture roughly triples the baseline, and the router preserves 88% of the ceiling at three quarters of the compute. Attribution across the 73 strictly recovered targets is broad (Table 3): eleven of the thirteen specialists are credited on at least one recovery, so the gain is the mixture itself rather than one strong detector wearing twelve disguises.

The per-category decomposition shows why the mixture wins where the checklist could not. On mechanical classes the specialists dominate, numerical precision rising from 0% to 56% (5/9) and checklist-pattern misses from 28% to 78% (25/32). But the lift extends into the reasoning classes: deep state-machine from 10% to 43% (13/30), cross-contract from 16% to 42% (8/19), business logic from 16% to 40% (10/25), and, on small pools, economic incentive from 44% to 67% and upgrade misses from 33% to 50%. A shallow rule names a pattern; a family specialist carries a threat model and license to trace it through the code, apparently enough to recover a large minority of bugs whose why-missed label says reasoning. The residual is not uniform: external context stays low at 22%, because the missing input is often outside the repository, and the surviv-

ing state-machine and economic misses are the multi-transaction lifecycle bugs of Section 6. The router's one notable weakness is cross-contract, where it captures 26% against the ceiling's 42%, unsurprising since cross-contract signals live outside the window the router reads; routing on call-graph context is the obvious next step.

5.8 Is the recovery real? A card-swap control

A 52% recovery from a union of thirty-odd findings per target invites a specific suspicion: perhaps the union fires so much that a lenient judge can always find something to credit. We ran the direct experiment. For every target in the all-13 arm we took the exact findings the judge had credited and re-verified each twice under the same three-vote protocol: once against the target's own human finding, and once against a human finding from a different project, assigned by derangement so no target sees its own project, both comparisons using the same reduced finding text. Specific findings should re-match their own card and fail the foreign card; generic near-anything findings should match both. The credited findings re-match their own card 48% of the time under the reduced text, and they match the foreign card exactly zero times in 141 targets. The floor is not low; it is empty. Nor did the judge hedge: all 73 strict recoveries were unanimous three-vote verdicts. The reduced-text own-card rate sits below the full-context 52% as expected, but the quantity that matters is the gap between own and foreign, and that gap is the entire signal. We consider this the paper's strongest validity check and suggest it as a standard control for any LLM-judged recovery claim: swap the cards, and see whether the number survives.

5.9 Targeted detectors at the true blind spots

Section 4.2 identified critical oracle manipulation and DoS as the two genuine capability gaps, and the family bank tests capability engineering in isolation. On oracle-family miss targets the standalone oracle detector recovers 50% against a 35% rescan control (fourteen targets, so the interval is wide but the direction consistent), and against the production system's 17% catch rate on critical oracle bugs. The DoS detector shows no lift, 29% against a 32% control, which also fits: the DoS baseline was middling rather than blind, leaving less specific incapacity for a specialist to supply. Targeted capability engineering pays where measurement says the model is weak, and not elsewhere.

5.10 Cost and synthesis

Recall came from firing widely, and we report what that costs. The union emits a mean of 37 findings per target, median 35, of which roughly two on average are real by the judge's accounting, so precision sits near 5%. A routed target also runs about ten family detectors and three generic rescans against the control's single pass. This is a research configuration for measuring how much of the miss

distribution is recoverable in principle, not a shippable scanner. The right reading of (52% recall, 5% precision) is a ceiling and a bill: about half the reliability-gap misses are reachable by depth and routing, and the rest of the distance is a deduplication and validation stage that we did not build or evaluate, so we make no claim that this configuration is deployable. What recall survives such a stage is the open question.

Two levers close the reliability gap, and they compose. Variance reduction, re-running what already exists, recovers up to a quarter of the misses for cents. Family depth with routing recovers half, including a large minority of the reasoning-heavy misses, at the price of a precision stage. Rule knowledge in context, in the framings we tested, recovers nothing beyond re-sampling in aggregate and helps only when narrowed to mechanical families. Extrapolating with caution, since the 141 targets are the resolvable, Solidity-centric slice of the reliability gaps: the union's 52% of the 71% reliability share suggests roughly a third of all 548 misses is recoverable by architecture alone, before any model improvement, with the ensemble supplying the cheap half and the mixture behind a deduplication and validation stage we did not evaluate. The remaining misses, concentrated in external context, multi-transaction state machines, and economic design, are the genuine capability frontier, and no configuration we tested moves them.

6 Case Studies

Three real misses from the benchmark, two recovered by our interventions and one that resists everything we built, trace the boundary between the recoverable and the frontier.

6.1 Recovered: a spot price used as a valuation

A vault helper values tokens from `slot0().sqrtPriceX96`, the instantaneous Uniswap V3 spot price, so an attacker can move the pool, deposit at a corrupted valuation, and unwind; the human report rates it medium. The AI reported other findings in the same file but never propagated the tainted quote into the deposit accounting three calls away. The oracle specialist recovers it, tracing the manipulable read to seven sinks; the plain rescan control does not, so the recovery is family depth, not re-sampling.

6.2 Recovered: top-up deposits dilute everyone's rewards

A staking contract tracks `userShares` per depositor and a `totalShares` accumulator that denominates reward distribution. On top-up, the code updates the user's balance correctly but increases `totalShares` by the full post-update balance rather than the delta, silently double-counting the old shares: every top-up dilutes `accRewardPerShare` and shifts rewards away from everyone else, amplifiable by splitting deposits. The buggy line is syntactically indistinguishable from a correct accumulator update, so nothing pattern-shaped can catch it; detection

requires carrying the invariant that `totalShares` equals the sum of `userShares` across repeated transitions. The reward-accounting specialist recovers it as a root-cause match, accumulator-versus-sum consistency being one of its enumerated failure classes. This deepest recovered case marks how far the family-depth lever reaches: single-contract stateful invariants are within range when the family is recognized.

6.3 Missed: exit-cap recycling

A yield protocol enforces a lifetime exit cap, ejecting a user once they have earned three times their principal, and a per-user deposit limit. On exit, it deletes the user's deposit and claim history; a returning user is therefore new in the contract's eyes, and the lifetime cap becomes a per-cycle cap an address can recycle indefinitely, rated major. No function is individually wrong, and the deletion looks like hygienic cleanup. Seeing the problem requires simulating a multi-transaction lifecycle, deposit, accrue, exit, re-deposit, and recognizing that the recycled state defeats an economic invariant the code only implies. This hardest species, deep state-machine entangled with economic incentive, survived the rescans, the checklist, the reasoning scanner, the agentic loop, and the full thirteen-specialist union. The 43% of state-machine misses the union does recover are the invariant-shaped ones like the top-up case; the lifecycle-shaped ones remain open and plausibly require stateful simulation rather than more reading.

7 Threats to Validity

Ground truth. Human audit reports can themselves miss bugs, so our coverage is against the professional standard rather than all bugs present, and misses shared by both human and AI are invisible. This is the right yardstick for the question we ask, but absolute numbers should be read as relative to it.

Judge validity. Every match and recovery is decided by an LLM judge. We defend at four levels: a frozen rubric shared by benchmark construction and all interventions; three-vote majorities on every decision; human calibration before freezing; and the card-swap control of Section 5.8, bounding the false-recovery floor at zero out of 141. Absolute levels could shift under a different judge family, but the comparative claims are within-judge.

Construct validity of the same-file join. "Reliability gap" is operationalized as the AI reporting any finding in the miss's file, a proxy. Within same-file misses the AI named the same contract 90% and the same function 77% of the time, and the independent class-recall measurement of Section 4 reaches the same conclusion without using locations at all.

Taxonomy softness. Inter-rater agreement is 50% raw and 75% with secondary credit, so per-class counts carry real uncertainty. The claims that depend on the taxonomy use coarse mechanical-versus-reasoning distinctions that survive the measured label noise.

Selection of the intervention targets. The 141 targets are reliability-gap misses with resolvable Solidity source, the dominant but not the whole miss population. Recovery on blind spots, non-Solidity code, and location-less findings would be lower; the extrapolation in Section 5.10 says a third of all misses, not of all bugs. The corpus leakage audit (maximum similarity 0.873, generic patterns) limits the risk that retrieval arms were quietly fed answers.

Precision and cost. The union's 5% precision is an estimate over the evaluated targets, not a false-positive study on clean code. A deployment would need the dedup and validation stages we did not evaluate, and those could eat part of the recall we report.

Generality. One production system and, in our own interventions, one model family for scanning and judging, over one language ecosystem for the detectors. The system under study is itself less homogeneous than that: across the window its scanner lanes ran five models from three vendors, so the miss distribution we characterise is not one model's artifact, though the recovery rates we measure are one model's. Every percentage we report, from the 71/29 split to the union's 46% and its 5% precision, profiles that one configuration and should be expected to move under a different model family, pipeline, or ecosystem. What transfers is the method: the why-missed taxonomy for any miss distribution, the same-file and class-recall pair as independent reliability-versus-capability tests, the card-swap false-recovery floor for LLM-judged recovery, and the principle that an intervention pays only where it matches the bug's family. Replicating this on a second auditor and ideally a non-EVM ecosystem is the obvious next step; until then the quantitative findings should be assumed to hold for this system alone.

8 Related Work

Program analysis for smart contracts. The first generation of contract security tooling was symbolic and static: Oyente introduced symbolic execution over EVM bytecode [11], Securify checked violation patterns over dependency graphs [16], Mythril added taint tracking [12], Slither made static analysis fast enough for CI [4], and Echidna established property-based fuzzing [7]. These tools are precise on the classes their analyses encode and blind outside them; our rule-scanner failure reproduces that boundary in the LLM era.

Empirical evaluations and benchmarks. Durieux et al. ran nine analyzers over 47,587 contracts and found low mutual agreement and modest recall [3], SmartBugs standardized the execution harness [5], and SolidiFI injected known bugs to measure detection [6]. A study of 2,421 manual blockchain audit reports found logic-related issues to be the distinctive product of human auditing [18]. Our subject is a deployed commercial AI pipeline rather than open-source analyzers, and the reference is the paid human audit of the same engagement rather than an injected or curated bug set.

LLM-based vulnerability detection. GPTScan pairs GPT with program analysis [15], GPTLens structures detection as generator and critic roles [8], LLM4Vuln decouples LLM vulnerability reasoning [14], fine-tuned open models

serve as auxiliary vulnerability flaggers [19], and David et al. measure partial LLM coverage of real exploits [2]. These studies report what models catch; ours is a production-scale anatomy of what a deployed system does not catch, why, and which interventions convert that diagnosis into recall.

Sampling, ensembles, and judging. The pass@k analysis follows the code-generation evaluation of Chen et al. [1], and the recovery-through-resampling effect parallels self-consistency in reasoning tasks [17]. The additive-hints degradation connects to evidence that models use long contexts unevenly [10]. Our evaluation leans on LLM-as-judge methodology [20]; the card-swap control we add is, to our knowledge, a new validity check for LLM-judged recovery claims.

Mixtures and routing. Routing inputs to specialists descends from mixtures of experts [9] and its modern sparse instantiations [13]. Our mixture operates at the task level, specialist prompts over a shared base model, and inverts the usual purpose: MoE routes for efficiency at fixed quality, while we route for coverage of a heterogeneous bug distribution at a cost we accept. Our results suggest that specialist decomposition is a real axis for LLM-based analysis tools, distinct from scale and from sampling.

9 Conclusion

Production AI auditors miss bugs, and this paper gave those misses a structure a designer can act on. From 298 human-audited engagements we assembled 548 core misses and found that size, language, protocol type, and severity say almost nothing about whether a finding is covered, and almost three quarters of coverage variance is specific to the individual bug. What is predictable is the shape of failure: in 71% of misses the AI had already reported a different bug in the same file, usually the same function, and in 86.5% it demonstrably catches the same class elsewhere. A production AI auditor's dominant failure is not what it cannot do. It is what it does not do reliably.

That diagnosis changes what is worth building. Re-running the model on files it already read recovers 18% of reliability-gap misses; an eight-sample ensemble recovers 25% for cents. The intuitive fix, giving the model retrieved audit rules, never beat the rescan in any of the three framings we tried and degraded as retrieval grew; it won only inside two mechanical categories, so in this system what retrieval supplied, re-sampling already had. What beats everything is matching depth to family: thirteen specialist detectors recover 46% of the misses, 52% running all thirteen, at a false-recovery floor measured at zero. The cost is precision, near 5% before filtering, locating the remaining engineering in a deduplication and validation stage this study does not evaluate. Three directions follow: routing on call-graph context; treating the blind-spot minority as the scheduling and input problems it decomposes into; and attacking the frontier that survived everything with stateful simulation and specification recovery rather than more reading. The lesson we did not hold when we started is that in this deployed system reliability engineering and specialist decomposition bought

more recall than knowledge did; whether that ordering holds for other auditors is for replication to settle, but the place to learn what a deployed system needs is not its benchmark score but the anatomy of its misses.

References

1. Chen, M., Tworek, J., Jun, H., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
2. David, I., Zhou, L., Qin, K., Song, D., Cavallaro, L., Gervais, A.: Do you still need a manual smart contract audit? arXiv preprint arXiv:2306.12338 (2023)
3. Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE) (2020)
4. Feist, J., Grieco, G., Groce, A.: Slither: A static analysis framework for smart contracts. In: IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB) (2019)
5. Ferreira, J.F., Cruz, P., Durieux, T., Abreu, R.: Smartbugs: A framework to analyze solidity smart contracts. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE) (2020)
6. Ghaleb, A., Pattabiraman, K.: How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) (2020)
7. Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: Effective, usable, and fast fuzzing for smart contracts. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) (2020)
8. Hu, S., Huang, T., Ilhan, F., Tekin, S.F., Liu, L.: Large language model-powered smart contract vulnerability detection: New perspectives. In: IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA) (2023)
9. Jacobs, R.A., Jordan, M.I., Nowlan, S.J., Hinton, G.E.: Adaptive mixtures of local experts. Neural Computation **3**(1) (1991)
10. Liu, N.F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., Liang, P.: Lost in the middle: How language models use long contexts. Transactions of the Association for Computational Linguistics (TACL) (2024)
11. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS) (2016)
12. Mueller, B.: Smashing ethereum smart contracts for fun and real profit. HITB SECCONF Amsterdam (2018)
13. Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., Dean, J.: Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In: International Conference on Learning Representations (ICLR) (2017)
14. Sun, Y., Wu, D., Xue, Y., Liu, H., Ma, W., Zhang, L., Liu, Y., Li, Y.: LLM4Vuln: A unified evaluation framework for decoupling and enhancing LLMs' vulnerability reasoning. arXiv preprint arXiv:2401.16185 (2024)
15. Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., Liu, Y.: GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. In: Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE) (2024)

16. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.: Securify: Practical security analysis of smart contracts. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS) (2018)
17. Wang, X., Wei, J., Schuurmans, D., Le, Q.V., Chi, E.H., Narang, S., Chowdhery, A., Zhou, D.: Self-consistency improves chain of thought reasoning in language models. In: International Conference on Learning Representations (ICLR) (2023)
18. Yang, Z., Man, G., Yue, S.: Understanding security audits on blockchain. In: Proceedings of the International Conference on Blockchain Technology and Applications (ICBTA) (2022)
19. Yang, Z., Man, G., Yue, S.: Automated smart contract vulnerability detection using fine-tuned large language models. In: Proceedings of the International Conference on Blockchain Technology and Applications (ICBTA) (2023)
20. Zheng, L., Chiang, W.L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E.P., Zhang, H., Gonzalez, J.E., Stoica, I.: Judging LLM-as-a-judge with MT-Bench and chatbot arena. In: Advances in Neural Information Processing Systems (NeurIPS) (2023)