BlockBench: Contamination-Controlled Evaluation of Large Language Models for Code Analysis

Paul Osemudiame Oamen[1], **Courage Ochuko Mene**[1], **Abraham Ugbedeojo Daniel**[4], **Ananya Mukherjee**[2], **Pius Onobhayedo**[3], **Wei Zhao**[1]
[1]The Aberdeen NLP Research Group, University of Aberdeen, UK
[2]International Institute of Information Technology Hyderabad, India
[3]University of Southern California, USA
[4]Federal University of Technology, Minna, Nigeria
Project website: https://block-bench.github.io/base/

Abstract

How much of large language model (LLM) performance in code analysis reflects memorization of training data rather than genuine reasoning? We introduce BLOCKBENCH, a framework that tests sensitivity to possible contamination in smart contract analysis. We evaluate seven models on 362 vulnerable base contracts, 322 variants, and 100 clean contracts. We observe a substantial performance gap across data exposure levels: detection accuracy drops from 86.5% on likely exposed samples to 27.6% on recently disclosed GS contracts, and further to 0.8% on the most recently disclosed subset. Detection rises with public exposure. This pattern is consistent with possible contamination but does not prove its cause. We further identify a *pattern matching paradox*, where models can often locate security-relevant code regions but fail to explain why they are exploitable, revealing a form of explanation unfaithfulness in code understanding. In contrast, traditional static analyzers show complementary limitations: Slither compiles only 6.6% of GS contracts, while Mythril compiles all inputs but fails to detect vulnerabilities. BlockBench generalizes to code understanding tasks where semantic equivalence can be verified, enabling more reliable evaluation under potential contamination.

1 Introduction

Large language model evaluation faces a fundamental challenge: distinguishing genuine capability from training data memorization (Zhang et al., 2024; Zhao et al., 2025). When models achieve high scores on established benchmarks, it remains unclear how much performance reflects true reasoning versus pattern matching against memorized examples. This problem is especially acute for code analysis tasks, where benchmark contamination has been documented across multiple evaluation suites (Jain et al., 2025; Haimes et al., 2024).

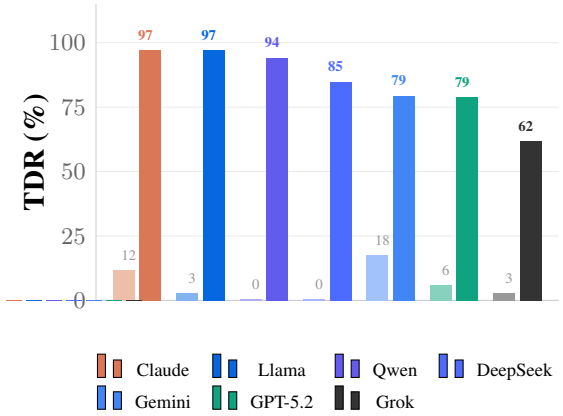


Figure 1: Temporal gradient in GS performance. **Light bars** show samples with 2–6 months of public exposure and an average TDR of 5.9%. **Dark bars** show samples with 7–8 months of exposure and an average TDR of 84.7%. Detection rises with exposure. This is consistent with possible contamination but does not prove its cause. On the most recently disclosed samples (n=72), average TDR remains at 0.8%.

We investigate this question in the domain of smart contract vulnerability detection, a setting uniquely suited to contamination-controlled evaluation. Smart contract security offers three properties that make it an ideal testbed: *(a) ground truth is objectively verifiable through documented exploits and professional audits*, *(b) public exposure can be estimated from disclosure dates and documentation*, and *(c) semantic-preserving transformations are well-defined, enabling systematic removal of surface recognition cues while preserving vulnerability mechanics.* The broader cryptocurrency ecosystem also carries substantial financial risk, with more than $3.4 billion stolen in 2025 (Chainalysis, 2025).

We present BLOCKBENCH, a contamination-controlled evaluation methodology that tests how performance changes when familiar cues are removed. Our approach evaluates seven frontier models across likely exposed baselines, semantically

Findings of the Association for Computational Linguistics: IJCNLP-AACL 2026, pages 3043–3062
November 6-10, 2026 ©2026 Association for Computational Linguistics

transformed variants, recent professional audit findings, and audited clean contracts. Performance on likely-contaminated samples reaches 86.5% TDR, but falls to 27.6% on the full GS set and 0.8% on the newest GS slice. This degradation, consistent with contamination patterns documented in adjacent benchmarks (Zhang et al., 2024; Zhao et al., 2025), suggests sensitivity to surface patterns. Our contributions:

(i) **A contamination-controlled evaluation methodology** using semantic-preserving transformations that progressively strip recognition cues while preserving exploit semantics, showing how performance depends on those cues. The methodology generalizes beyond security to any domain where semantic-preserving perturbations are definable. The transformations and temporal split test performance under distribution shift.

(ii) **CodeActs**, a taxonomy inspired by speech act theory (Austin, 1962; Searle, 1969) for classifying code segments by security function (e.g., ROOT_CAUSE, PREREQ, DECOY), enabling fine-grained analysis of whether models identify vulnerabilities through causal reasoning or pattern matching.

(iii) **BlockBench**, a benchmark of 362 vulnerable base contracts, 322 TC variants, and 100 clean contracts.

(iv) **Evidence of a temporal performance gradient**, showing that detection rates on GS samples rise from 5.9% to 84.7% as public exposure accumulates, consistent with temporal contamination patterns documented in code benchmarks (Jain et al., 2025).

2 Related Work

Benchmark Contamination and Memorization: The reliability of LLM evaluation is increasingly questioned due to training data contamination. Zhao et al. (2025) demonstrated that contamination inflates MMLU scores by up to 14.6 percentage points, with GPT-4o dropping from 88.0% to 73.4% on decontaminated items. Zhang et al. (2024) showed analogous inflation on grade school arithmetic, with models scoring up to 13% lower on matched-difficulty items guaranteed absent from training data. LiveCodeBench (Jain et al., 2025) introduced temporal windowing for code evaluation, revealing that DeepSeek-Instruct's performance plunges from approximately 60% to near 0% for problems released after its training cutoff. Haimes et al. (2024) formalized inflation measurement through retrospective holdout construction, finding up to 16 percentage points of inflation on TruthfulQA. These findings motivate evaluation methodologies that explicitly control for contamination, the approach we adopt for vulnerability detection.

Robustness and Semantic Perturbation: Adversarial robustness evaluation in NLP has established that semantic-preserving perturbations—paraphrases, synonym substitutions, syntactic transformations—can substantially degrade model performance (Ribeiro et al., 2020). In code, Rabin et al. (2021) demonstrated that identifier renaming and other semantics-preserving transformations can substantially affect neural program models, and recent work on code rewriting shows that structural modifications expose memorization reliance in code generation (Zhang et al., 2025). Our transformation suite extends this methodology to vulnerability detection, applying domain-specific semantic-preserving perturbations (sanitization, domain shift, structural obfuscation) that preserve exploit mechanics while systematically removing recognition cues.

Explanation Faithfulness in Code: Recent work finds reasoning-unfaithfulness failures concentrated in brittle, convergent domains such as mathematics and code (Han et al., 2026). More broadly, FaithCoT-Bench (Shen et al., 2026) provides a framework for instance-level faithfulness detection. Our CodeActs analysis identifies a domain-specific instance of this phenomenon: models that correctly locate vulnerable code segments cannot articulate why they are exploitable—a gap between pattern localization and causal explanation. We draw on speech act theory (Austin, 1962; Searle, 1969) to formalize this distinction, classifying code segments by their security function (e.g., ROOT_CAUSE vs. DECOY) rather than syntactic category. This functional taxonomy is analogous to dialogue act classification in NLP, where utterances are labeled by communicative function rather than surface form.

Traditional Analysis Tools: Early approaches to smart contract vulnerability detection relied on static analysis and symbolic execution. Tools such as Slither (Feist et al., 2019), Mythril (Mueller, 2017), and Securify (Tsankov et al., 2018) demonstrated precision on syntactically well-defined vulnerability classes. Durieux et al. (2020) evaluated

nine tools on 69 annotated vulnerable contracts, finding that the tools collectively detected 42% of the vulnerabilities, with Mythril detecting 27%. On a separate dataset of approximately 47,500 real-world contracts, 97% were flagged as vulnerable, suggesting a high false positive rate. We extend this comparison by evaluating both Slither and Mythril on BlockBench, providing direct baseline comparison against LLM performance across all subsets (§5.3).

LLM-Based Vulnerability Detection: Initial investigations by Chen et al. (2025) explored prompting strategies for vulnerability detection, achieving detection rates near 40% while noting pronounced sensitivity to superficial features. GPTScan (Sun et al., 2024b) combined GPT-4 with program analysis to achieve 78% precision on logic vulnerabilities. CLeVeR (Li et al., 2025) applied multimodal contrastive learning for vulnerability code representation at ACL 2025, framing detection as a code representation learning problem. Sun et al. (2024a) demonstrated that decoupling vulnerability descriptions from code context causes catastrophic performance degradation, suggesting reliance on memorized textual associations. Hu et al. (2023) documented output drift across evaluation dates at temperature zero, and Wu et al. (2024) showed systematic failure on counterfactual tasks where familiar patterns are disrupted. Our work extends these observations with a controlled framework that measures performance changes under semantic transformations and temporal separation.

3 BlockBench

BlockBench tests whether LLM performance depends on familiar cues. It contains 362 vulnerable base contracts, 322 TC variants, and 100 clean contracts.[1] The benchmark spans over 30 vulnerability categories (Appendix E).

Let $\mathcal{D}^+$ denote the 362 vulnerable contracts. It is the union of the disjoint DS, TC, and GS subsets. The separate set $\mathcal{D}_{\text{NEG}}$ contains 100 clean contracts.

Difficulty Stratified. $\mathcal{D}_{\text{DS}}$ draws from established vulnerability repositories including SmartBugs Curated (Ferreira et al., 2020), Trail of Bits' Not So Smart Contracts (Trail of Bits, 2018), and DeFiVulnLabs (SunWeb3Sec, 2023). Samples are stratified into four difficulty tiers based on detection complexity, with distribution $\{86, 81, 30, 13\}$ from Tier 1 (basic patterns) through Tier 4 (expert-level vulnerabilities requiring deep protocol knowledge). These public samples form a likely exposed baseline, although their contamination status is unknown.

Subset		N	Sources
Difficulty	Stratified (DS)	210	SmartBugs, DeFiVulnLabs
Temporal Contamination (TC)		46	Real-world exploits
Gold Standard (GS)		106	Code4rena, Spearbit, MixBytes
Negative Set (NEG)		100	OpenZeppelin, Uniswap, Aave, Compound, Lido, MakerDAO

Table 1: BlockBench composition by subset and primary sources.

Temporal Contamination. $\mathcal{D}_{\text{TC}}$ reconstructs 46 real-world DeFi exploits spanning 2016 to 2024, representing over $1.65 billion in documented losses. Notable incidents include The DAO ($60M, 2016), Nomad Bridge ($190M, 2022), and Curve Vyper ($70M, 2023). These attacks are widely documented, making training exposure plausible but unobservable. We transform them to test sensitivity to surface cues while preserving their semantics. Full details appear in §4.

Gold Standard. $\mathcal{D}_{\text{GS}}$ comprises 106 professional security audit findings from Code4rena (Code4rena, 2025), Spearbit (Spearbit, 2025), and MixBytes (MixBytes, 2025), disclosed between September 2025 and March 2026. All samples were disclosed after August 2025. This places them after the latest published cutoff among models with provider-reported dates. For models without a precise reported cutoff, temporal separation cannot be confirmed from provider documentation alone. Recent disclosure reduces the opportunity for public exposure but does not establish that the samples were absent from model training. Gold Standard refers to the audit ground truth, not guaranteed absence from training data. The subset spans 11 vulnerability categories including logic errors (48%), arithmetic errors (9%), denial of service (9%), access control (8%), and signature replay (6%), with 24 high-severity and 82 medium-severity findings across 15 independent audit engagements. Thirty-eight samples include multi-file context for cross-

[1] The dataset can be explored at https://block-bench.github.io/base/#/inspect.

contract vulnerability analysis.

Model cutoffs and training data are not fully disclosed (Dodge et al., 2021). We therefore report an association with public exposure, not proof of causation.

Negative Set. $\mathcal{D}_{\text{NEG}}$ consists of 100 clean, production-deployed Solidity contracts from six of the most extensively audited protocols in decentralized finance: OpenZeppelin v5 (25 contracts), Uniswap v4 (14), Aave v3 (19), Compound v3 (14), Lido (13), and MakerDAO (15). These contracts have undergone multiple professional audits and have been deployed in production with no known vulnerabilities. The negative set enables measurement of false positive rates—a critical dimension absent from prior vulnerability detection benchmarks (Durieux et al., 2020). Contracts span five functional categories: DeFi core (40), utility (20), token implementations (19), infrastructure (10), and governance (7).

Together, these subsets assess detection under different levels of public exposure and changes in surface cues. The negative set grounds evaluation in practical auditing precision.

4 Methodology

BlockBench asks whether a model can identify the causal mechanism of a vulnerability after familiar surface cues are removed. Figure 2 summarizes the pipeline: contracts are transformed, annotated with security functions, analyzed by detector models, and judged against ground truth.

Semantic Transformations. For each TC contract c, we construct variants $\mathcal{T}(c)$ that preserve vulnerability semantics while changing the cues models may rely on. Concretely, $\mathcal{V}(\mathcal{T}(c)) = \mathcal{V}(c)$ means the same exploit still works for the same underlying reason, even if names, comments, domain vocabulary, or surrounding code have changed. This mirrors standard NLP robustness tests: paraphrases, counterfactual edits, domain shifts, and distractor spans. The transformations and temporal split test performance under distribution shift.

We separate transformations that plausibly occur in production code, including sanitization, comment removal, and renamed forks, from stress tests designed to isolate reasoning failures, including domain shift, Trojan distractors, and misleading attestations. Differential is treated separately because it removes the vulnerability and creates a paired fixed contract. Figure 3 illustrates the original contract and five representative vulnerability-preserving transformations of the Nomad Bridge uninitialized root bypass, with changes highlighted. Full transformation definitions appear in Appendix H.

Family	Transformation	What it tests
Sanitize	$\mathcal{T}_S$: rename identifiers; remove comments	Reliance on lexical cues such as `withdraw`, `owner`, or exploit-hinting comments
Domain shift	$\mathcal{T}_C$: DeFi terms $\rightarrow$ non-security terms	Whether reasoning transfers when domain vocabulary changes but control/data flow remains
Obfuscate	$\mathcal{T}_O$: restructure code; add neutral intermediates	Robustness to surface form changes beyond simple renaming
Trojan	$\mathcal{T}_T : c \mapsto c \cup d_{\text{safe}}$	Distractor resistance: ignore suspicious-looking but non-exploitable code
Differential	$\mathcal{T}_D : c_{\text{vuln}} \mapsto c_{\text{fixed}}$	Counterfactual reasoning: report safe once the causal bug is removed
False Prophet	$\mathcal{T}_F : c \mapsto c \cup a_{\text{mislead}}$	Resistance to misleading natural-language assurances such as "audited"

Table 2: Semantic transformation families. All preserve or explicitly control the exploit semantics while changing the evidence available to a model.

CodeActs. TDR measures *whether* a model detects a vulnerability, but not *how*. Two models may both flag a reentrancy bug: one explains the temporal dependency between an external call and a state update; the other simply flags the external call because it looks suspicious. Standard metrics score them identically. To separate causal reasoning from pattern recognition, we need to know which code segments each model attends to and what security role those segments play.

We introduce CodeActs, an annotation scheme that classifies code segments by their security function. The name draws on speech act theory (Austin, 1962; Searle, 1969): just as speech acts classify utterances by what they *do* (assert, request, promise) rather than their surface form, CodeActs classify code by what it *does* in the context of a vulnerability. Each segment receives one of four primary labels: ROOT_CAUSE (code that causally enables exploitation), PREREQ (necessary preconditions), DECOY (suspicious-looking but non-exploitable code, injected via Trojan transformations), and BENIGN (correctly implemented). Figure 4 illustrates this on a classic reentrancy: lines 3–4 constitute the ROOT_CAUSE because their *order-*

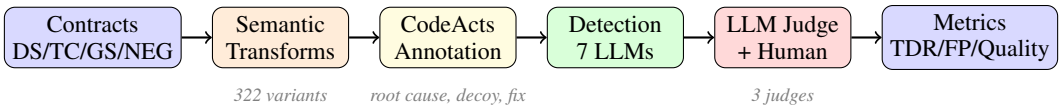


Figure 2: BlockBench evaluation pipeline. Semantic-preserving transformations remove recognition cues while preserving exploit mechanics; CodeActs identify security-relevant code roles; LLM judges score detector outputs against ground truth with human validation.



Figure 3: The original tc_001 contract and five vulnerability-preserving transformations of its uninitialized root bypass. Highlighted regions show what each transformation changes, while the underlying exploit mechanics remain identical.

ing (external call before state update) creates the exploit window. A model that flags line 3 in isolation without articulating the ordering dependency exhibits pattern matching; one that explains the causal chain demonstrates reasoning. The full 17-operation taxonomy and annotation format are in Appendix F.

Detection Protocol. We evaluate seven frontier models, one per major AI lab: Claude Opus 4.5 (Anthropic, 2025), GPT-5.2 (OpenAI, 2025), Gemini 3 Pro (Google DeepMind, 2025), DeepSeek v3.2 (DeepSeek-AI, 2025), Llama 4 Maverick (Meta AI, 2025), Grok 4 Fast (xAI, 2025), and Qwen3-Coder-Plus (Alibaba Cloud, 2025). DS and TC use a direct zero-shot auditor prompt requesting structured JSON with vulnerability type, location, root cause, attack scenario, and fix. GS additionally uses five prompt settings (Direct, Context, Chain-of-Thought, Naturalistic, Adversarial) to test whether protocol context or framing changes detection. All evaluations use temperature 0; prompt templates are in Appendix I.

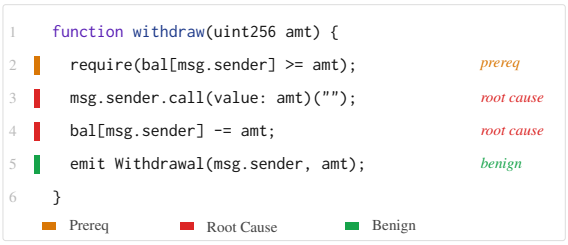


Figure 4: CodeActs annotation of a reentrancy vulnerability. Colored bars mark each line's security function. A correct detection must identify lines 3–4 as ROOT_CAUSE and explain why their ordering enables exploitation.

Judging and Metrics. Detector outputs are evaluated against ground truth by independent LLM judges: GLM-4.7 (Zhipu AI, 2025), MIMO-v2-Flash (Xiaomi MiMo, 2025), and Mistral-Large-3 (Mistral AI, 2025). A finding is a TARGET_MATCH when it identifies the documented root cause, location, and vulnerability type. Findings that partially match, identify valid issues beyond ground truth, or report fabricated vulnerabilities are classified separately (full classification scheme in Appendix J). Our primary metric is Target Detection Rate (TDR), the proportion of samples where the target vulnerability is correctly identified. We also report false positive rate on clean contracts and explanation-quality scores (RCIR, AVA, FSV) for matched findings (definitions in Appendix L). On GS, GLM-4.7 and MIMO-v2-Flash achieve $\kappa = 0.943$ with 97.7% agreement; expert validation and judge failure modes are detailed in Appendix J. The contract is the unit of analysis. DS supports intervals and tests. TC results are descriptive because variants share base contracts. Appendix L gives the procedures.

5 Results

We evaluate seven frontier LLMs across 100 sampled DS contracts, all 46 TC contracts with 322 transformation variants, all 106 GS contracts, and 100 clean contracts, yielding over 5,000 model-sample evaluations. Unless noted otherwise,

DS/TC scores use majority vote across three judges, while GS uses the two primary judges (GLM-4.7 and MIMO-v2-Flash; $\kappa = 0.943$).

5.1 Performance on Exposed Samples Does Not Transfer

Table 3 shows the core transfer gap. Claude Opus 4.5 reaches 86.5% TDR on DS but drops to 50.9% across transformed TC variants; Gemini 3 Pro falls from 73.9% to 38.5%, with a 57% relative drop after protocol-identifying names are sanitized. Realistic perturbations (MinS/San/NoC) already reduce performance, while adversarial variants push all models toward a 15–40% floor. These failures occur despite preserved exploit semantics, suggesting dependence on recognition cues rather than invariant vulnerability reasoning.

5.2 Detection Rises With Public Exposure

Table 4 and Figure 1 reveal a temporal gradient in GS performance. Samples with longer public exposure (GS-Orig, disclosed 7–8 months before evaluation) average 84.7% TDR, while more recently disclosed samples (GS-New, 2–6 months of exposure) average 0.8%. This pattern is consistent with the possibility that increasing public exposure (*through blog posts, GitHub repositories, and educational materials*) leads to greater representation of these contracts in model training data, which could contribute to inflated detection scores. However, because training data composition is not observable, the temporal comparison does not establish contamination as the causal mechanism. The combined GS-All score (27.6%) reflects this mixture of exposure levels rather than a stable estimate of model capability on recently disclosed data.

The expanded GS also shows a within-category performance gap. Models detect canonical DS reentrancy and arithmetic bugs, yet average 0% TDR on GS arithmetic errors ($n = 10$) and reentrancy ($n = 4$), where the same vulnerability classes appear through novel cross-contract callbacks, TWAP calculation errors, bit-mask corruption, and rounding-direction bugs (full breakdown in Appendix D). Familiar labels do not imply robust reasoning over unfamiliar mechanisms.

5.3 Clean Contracts and Tool Baselines

Table 5 addresses two practical gaps in prior evaluation: false positives and comparison to classical analyzers. Clean audited contracts reveal a precision-recall trade-off: Gemini 3 Pro and Grok 4 Fast produce no false positives but have the lowest GS detection, while Claude Opus 4.5 has the best LLM balance (33.0% GS TDR, 9.0% FP). Traditional tools fail differently. Slither compiles only 6.6% of GS contracts because professional audit code often relies on unresolved imports and project-specific dependencies; Mythril processes all GS contracts but detects none, reflecting the mismatch between symbolic-execution patterns and the logic/arithmetic vulnerabilities common in GS. Thus, static tools and LLMs expose complementary limitations rather than a single dominant solution.

5.4 Pattern Matching Without Causal Explanation

CodeActs isolates a final failure mode: models can locate security-relevant code without explaining why it is exploitable. On TC Trojan variants, Llama 4 Maverick achieves the highest ROOT_CAUSE match rate (60.9%), while its TDR reaches only 43.5%, a 17.4pp gap between pattern localization and causal explanation. This mirrors explanation-unfaithfulness findings in broader NLP reasoning work (Han et al., 2026): the model's apparent evidence points to the right region, but the generated security argument does not establish the exploit mechanism. Full quality and CodeActs analyses are in Appendix D, with the SUI sensitivity analysis in Appendix K.

6 Discussion

What Does the Degradation Reveal? The DS-to-GS gap may reflect both difficulty and sensitivity to familiar cues. Three findings make surface cue dependence a plausible part of the explanation. First, models detect canonical reentrancy and arithmetic vulnerabilities at high rates on DS but score 0% on GS instances of the same broad categories. The GS cases use less familiar implementations such as cross-contract callbacks, TWAP calculation errors, bit-mask corruption, and rounding errors. This result suggests sensitivity to familiar forms but does not control for complexity. Second, replacing protocol-identifying names with generic equivalents changes no control flow or data dependencies but reduces Gemini 3 Pro performance by 57%. This suggests reliance on identifiers such as "Nomad" or "Curve." Third, detection rises with public exposure duration. This trend is consistent with possible contamination, but other factors may contribute. It does not prove causation.

Model	DS (Difficulty-Stratified) T1	T2	T3	T4	Avg [95% CI]	TC Transformations MinS	San	NoC	Cha	Shp	Tro	FalP	Avg
Claude Opus 4.5	**100**	**83.8**	**70.0**	92.3	**86.5** [82–91]	**71.7**	**54.3**	**50.0**	**43.5**	**50.0**	32.6	**54.3**	**50.9**
Gemini 3 Pro	75.0	78.4	50.0	**92.3**	73.9 [68–80]	65.2	28.3	32.6	37.0	34.8	34.8	37.0	38.5
GPT-5.2	60.0	70.3	36.7	84.6	62.9 [56–70]	54.3	34.8	37.0	28.3	30.4	30.4	37.0	36.0
DeepSeek v3.2	65.0	64.9	46.7	61.5	59.5 [53–66]	58.7	37.0	41.3	21.7	26.1	**43.5**	30.4	37.0
Llama 4 Maverick	65.0	45.9	40.0	69.2	55.0 [48–62]	52.2	39.1	30.4	21.7	13.0	**43.5**	21.7	31.7
Qwen3-Coder-Plus[b]	60.0	56.8	43.3	53.8	53.5 [47–60]	56.5	43.5	30.4	15.2	17.4	28.3	41.3	33.2
Grok 4 Fast[b]	40.0	37.8	33.3	30.8	35.5 [29–42]	32.6	23.9	19.6	15.2	15.2	21.7	21.7	21.4

Table 3: Target Detection Rate (%) on DS and TC. DS gives a likely-contaminated baseline; TC preserves exploit semantics while removing surface cues. [b]Significantly worse than Claude Opus 4.5 ($p < 0.05$).

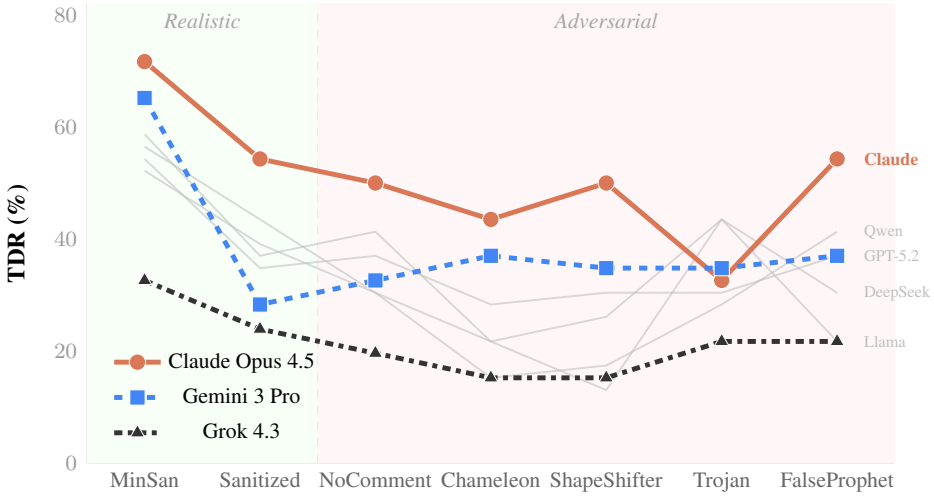


Figure 5: TC benchmark: TDR across transformations ordered by obfuscation intensity. Faint regions distinguish realistic perturbations from adversarial stress tests.

Model	GS-Orig (n=34)	GS-New (n=72)	GS-All (n=106)
Claude Opus 4.5	97.1	2.8	**33.0**
Llama 4 Maverick	97.1	2.8	31.1
Qwen3-Coder-Plus	94.1	0.0	30.2
DeepSeek v3.2	84.8	0.0	26.9
GPT-5.2	78.8	0.0	26.5
Gemini 3 Pro	79.4	0.0	26.2
Grok 4 Fast	61.8	0.0	19.8
Average	**84.7**	**0.8**	**27.6**

Table 4: GS TDR (%) by temporal slice. GS-Orig contains 34 samples disclosed September–October 2025 (7–8 months of public exposure at evaluation time). GS-New contains 72 samples disclosed November 2025–March 2026 (2–6 months of exposure).

System	GS behavior	Clean FP	Notes
Claude Opus 4.5	33.0 TDR	9.0%	best LLM TDR/FP balance
Gemini 3 Pro	26.2 TDR	**0.0%**	conservative; low GS recall
Grok 4 Fast	19.8 TDR	**0.0%**	conservative; lowest GS TDR
DeepSeek v3.2	26.9 TDR	26.0%	highest LLM FP rate
Slither	7/106 compiled	62.5%[†]	dependency-limited on GS
Mythril	0/106 detected	5.0%	processes GS but misses targets

Table 5: False positives and traditional baselines. LLM FP rates are measured on 100 clean audited contracts. [†]Slither FP rate is computed on the 24/100 clean contracts it compiled. Full model and tool tables appear in Appendix D.

Pattern Recognition and Causal Explanation Are Dissociable. The CodeActs analysis reveals that locating vulnerable code and explaining why it is exploitable are separate capabilities. On TC Trojan variants, Llama 4 Maverick identifies ROOT_CAUSE segments at the highest rate (60.9%), while its TDR reaches only 43.5%. This dissociation matters for deployment: a model that highlights the right lines but provides a generic or

incorrect security argument may give practitioners false confidence that a vulnerability has been understood when it has only been flagged. The gap parallels explanation unfaithfulness findings in mathematical reasoning (Han et al., 2026; Shen et al., 2026), but manifests distinctly in code, where vulnerability explanation requires articulating a multi-step causal chain (precondition, trigger, state violation, exploit) rather than a single derivation step. Models that shortcut this chain by matching on the trigger alone produce explanations that appear plausible but omit the causal structure needed for remediation.

The Trojan results add a counterintuitive dimension. DeepSeek v3.2 and Llama 4 Maverick achieve their highest TC scores (43.5%) on variants containing injected decoy code, the opposite of what causal reasoning would predict. One hypothesis is that longer inputs activate retrieval mechanisms that surface relevant training examples, a failure mode where detection improves for the wrong reason.

Implications Beyond Smart Contract Security. The contamination-controlled methodology introduced here applies to any code analysis task where ground truth is verifiable, semantic-preserving perturbations are definable, and temporal separation from training data is achievable. Candidate domains include malware analysis (known families as DS, obfuscated variants as TC, novel samples as GS), software bug detection (using bug-fix pairs from version control), and formal verification (where semantic equivalence is mathematically defined). The temporal gradient we observe also has methodological implications: any code benchmark whose solutions appear in public repositories or documentation should be treated as potentially contaminated, and evaluation claims based on such benchmarks benefit from contamination controls or temporal analysis. The gap between what models appear to detect and what they actually reason about suggests that the field needs evaluation frameworks that go beyond accuracy to probe the causal structure of model explanations.

Future Work. We highlight several promising directions for future work. First, BlockBench can cover other smart contract languages, such as Vyper, Rust, and Move. Second, it can evaluate models that actively use audit tools. Such models could compile contracts, run tests, inspect execution traces, and retrieve similar audit reports before producing a finding. Third, BlockBench can include LLM-generated rewrites that reflect differences in developer coding style. Each rewrite must be verified to retain the same vulnerability before it is added to the benchmark. More broadly, exploit tests and execution traces can provide stronger evidence that structural transformations preserve the intended behavior. Future evaluations can also compare few-shot settings and different prompt lengths.

7 Conclusion

BlockBench tests LLM vulnerability detection on likely exposed, transformed, recent, and clean contracts. Our study includes over 5,000 evaluations across seven frontier models and reveals three principal findings.

First, performance on likely-contaminated samples (86.5% TDR on DS) degrades to 27.6% on the full GS set and 0.8% on the newest GS slice. This gap suggests sensitivity to surface patterns and possible contamination, but it does not prove causation. Related performance gaps have been reported in other NLP benchmarks (Zhao et al., 2025; Zhang et al., 2024; Jain et al., 2025). Second, the pattern matching paradox demonstrates that models can locate vulnerable code without explaining why it is exploitable. On TC Trojan variants, Llama 4 Maverick achieves a 60.9% ROOT_CAUSE match rate but only 43.5% TDR, a 17.4pp gap that parallels explanation unfaithfulness in mathematical reasoning (Han et al., 2026). Third, empirical evidence of a temporal performance gradient, showing that detection rates on GS samples rise from 5.9% to 84.7% with increasing public exposure, consistent with temporal contamination patterns documented in prior work.

These findings carry implications beyond smart contract security. The contamination-controlled methodology—establishing contaminated baselines, applying semantic-preserving transformations, and comparing performance across samples with different disclosure dates—generalizes to any code understanding task where semantic equivalence is verifiable. We release BlockBench as a living benchmark to support evaluation of LLM code reasoning.

Limitations

BlockBench contains 362 vulnerable base contracts, 322 TC variants, and 100 clean contracts. The study yields over 5,000 evaluations across

seven frontier models. While this represents a substantial evaluation, several limitations warrant discussion.

Scope. We evaluate exclusively Solidity smart contracts. While Solidity dominates the DeFi ecosystem (accounting for over 90% of deployed contracts by value locked), our findings may not generalize to other smart contract languages such as Vyper, Rust (Solana/Anchor), or Move. The vulnerability types and transformation strategies are specific to EVM-based architectures.

Evaluation Context. We test prompted models rather than complete audit systems. GS includes several prompts and multi-file context. In real audit settings, analysts draw on additional semantic context including protocol specifications, intended invariants, economic assumptions, and cross-protocol interactions. Production audits also use richer documentation and tools.

CodeActs Coverage. CodeActs annotations cover 46 samples across three transformation variants (MinimalSanitized, Trojan, Differential), enabling fine-grained analysis of the pattern matching paradox. Broader annotation coverage across the full dataset would strengthen generalizability of these findings.

Automated Evaluation. Our LLM judge ensemble (GLM-4.7, MIMO-v2-Flash, Mistral-Large) achieves Cohen's κ=0.943 between primary judges with 97.7% agreement, indicating near-perfect reliability. However, automated evaluation may miss nuanced security reasoning. Disagreements concentrate in access control and logic error categories, where vulnerability boundaries are inherently ambiguous.

Contamination Claims. We sample findings by disclosure date to reduce the opportunity for public exposure, but we cannot guarantee that they were absent from model training. Provider cutoff information is incomplete and uses different definitions. Data mirrors and leaks may introduce recently disclosed material, and model providers do not disclose complete training data composition. Our temporal contamination evidence (§5.2) provides empirical support but not definitive proof.

Ethical Considerations

BlockBench poses dual-use risks: adversarial transformations demonstrate methods that could suppress vulnerability detection, while detailed vulnerability documentation may lower barriers for malicious actors. We justify public release on several grounds. First, adversarial robustness represents a fundamental requirement for security tools deployed in production. Second, all samples derive from already-disclosed vulnerabilities and public security audits; no novel exploit information is introduced. Third, responsible disclosure of evaluation limitations enables practitioners to calibrate trust in automated tools appropriately.

The primary positive impact of this work is enabling more informed deployment decisions for LLM-based security tools. The observed gaps may reflect memorization or surface familiarity (§5). This cautions against over-reliance on benchmark scores when evaluating security tools for production use. High scores on exposed samples may create false confidence. We recommend that practitioners treat LLM-based analysis as assistive rather than authoritative, maintaining mandatory expert review for all security-critical assessments.

We follow responsible NLP research practices by releasing all evaluation code, prompts, and data to enable reproducibility. We report confidence intervals and statistical tests throughout, and acknowledge limitations explicitly rather than overstating claims.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback, which substantially improved this work.

References

Alibaba Cloud. 2025. Qwen3-Coder-Plus model information.

Anthropic. 2025. Claude Opus 4.5 model documentation.

J. L. Austin. 1962. *How to Do Things with Words*. Oxford University Press, Oxford.

Chainalysis. 2025. North Korea drives record $2 billion crypto theft year, pushing all-time total to $6.75 billion.

Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Jianxing Yu, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2025. When ChatGPT meets smart contract vulnerability detection: How far are we? *ACM Transactions on Software Engineering and Methodology*, 34(4):1–30.

Code4rena. 2025. Audit reports. Accessed 26 November 2025.

DeepSeek-AI. 2025. DeepSeek-V3.2 public summary of training content.

Jesse Dodge, Maarten Sap, Ana Marasović, William Agnew, Gabriel Ilharco, Dirk Groeneveld, Margaret Mitchell, and Matt Gardner. 2021. Documenting large webtext corpora: A case study on the Colossal Clean Crawled Corpus. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 1286–1305, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE, pages 530–541.

Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB, pages 8–15. IEEE.

João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: A framework to analyze Solidity smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 1349–1352.

Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, pages 415–427.

Google DeepMind. 2025. Gemini 3 Pro model card. Technical report, Google DeepMind.

Jacob Haimes, Cenny Wenner, Kunvar Thaman, Vassil Tashev, Clement Neo, Esben Kran, and Jason Schreiber. 2024. Benchmark inflation: Revealing LLM performance gaps using retro-holdouts. *arXiv preprint arXiv:2410.09247*.

Yunseok Han, Yejoon Lee, and Jaeyoung Do. 2026. RFEval: Benchmarking reasoning faithfulness under counterfactual reasoning intervention in large reasoning models. In *The Fourteenth International Conference on Learning Representations*.

S. M. Mostaq Hossain, Amani Altarawneh, and Jesse Roberts. 2025. Leveraging large language models and machine learning for smart contract vulnerability detection. In *2025 IEEE 15th Annual Computing and Communication Workshop and Conference*, CCWC, pages 577–583. IEEE.

Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. 2023. Large language model-powered smart contract vulnerability detection: New perspectives. In *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications*, TPS-ISA, pages 297–306. IEEE.

Peter Ince, Jiangshan Yu, Joseph K. Liu, Xiaoning Du, and Xiapu Luo. 2026. GenDetect: Generative large language model usage in smart contract vulnerability detection. In *Provable and Practical Security*, ProvSec 2025, pages 426–445. Springer.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*.

Jiayuan Li, Lei Cui, Sen Zhao, Yun Yang, Lun Li, and Hongsong Zhu. 2025. CLeVeR: Multi-modal contrastive learning for vulnerability code representation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 7940–7951, Vienna, Austria. Association for Computational Linguistics.

Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2025. PropertyGPT: LLM-driven formal verification of smart contracts through retrieval-augmented property generation. In *Proceedings of the 32nd Annual Network and Distributed System Security Symposium*, NDSS 2025.

Meta AI. 2025. Llama 4 Maverick model card.

Mistral AI. 2025. Mistral Large 3.

MixBytes. 2025. Smart contract security audits. Accessed 26 November 2025.

Bernhard Mueller. 2017. Mythril: Security analysis tool for Ethereum smart contracts.

OpenAI. 2025. GPT-5.2 model documentation.

Md Rafiqul Islam Rabin, Nghi D. Q. Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135:106552.

Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4902–4912, Online. Association for Computational Linguistics.

Eva Sánchez Salido, Julio Gonzalo, and Guillermo Marco. 2025. None of the others: A general technique to distinguish reasoning from memorization in

multiple-choice LLM evaluation benchmarks. *arXiv preprint arXiv:2502.12896*.

John R. Searle. 1969. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, Cambridge, UK.

Xu Shen, Song Wang, Zhen Tan, Laura Yao, Xinyu Zhao, Kaidi Xu, Xin Wang, and Tianlong Chen. 2026. FaithCoT-Bench: Benchmarking instance-level faithfulness of chain-of-thought reasoning. In *The Fourteenth International Conference on Learning Representations*.

Spearbit. 2025. Public portfolio. Accessed 20 November 2025.

Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. 2024a. LLM4Vuln: A unified evaluation framework for decoupling and enhancing LLMs' vulnerability reasoning. *arXiv preprint arXiv:2401.16185*.

Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024b. GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pages 1–13, Lisbon, Portugal. ACM.

SunWeb3Sec. 2023. DeFiVulnLabs: To learn common smart contract vulnerabilities using Foundry. Accessed 11 November 2025.

Trail of Bits. 2018. (not so) smart contracts: Examples of Solidity security issues. Accessed 20 March 2026; archived 24 February 2023.

Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82.

Zhaofeng Wu, Linlu Qiu, Alexis Ross, Ekin Akyürek, Boyuan Chen, Bailin Wang, Najoung Kim, Jacob Andreas, and Yoon Kim. 2024. Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 1819–1862, Mexico City, Mexico. Association for Computational Linguistics.

xAI. 2025. Grok 4 Fast model card.

Xiaomi MiMo. 2025. MiMo-V2-Flash.

Hugh Zhang, Jeff Da, Dean Lee, Vaughn Robinson, Catherine Wu, Will Song, Tiffany Zhao, Pranav Raja, Charlotte Zhuang, Dylan Slack, Qin Lyu, Sean Hendryx, Russell Kaplan, Michele Lunati, and Summer Yue. 2024. A careful examination of large language model performance on grade school arithmetic. In *Advances in Neural Information Processing Systems*, volume 37, pages 46819–46836.

Lizhe Zhang, Wentao Chen, Li Zhong, Letian Peng, Zilong Wang, and Jingbo Shang. 2025. Memorize or generalize? evaluating LLM code generation with code rewriting. *arXiv preprint arXiv:2503.02296*.

Qihao Zhao, Yangyu Huang, Tengchao Lv, Lei Cui, Qinzheng Sun, Shaoguang Mao, Xin Zhang, Ying Xin, Qiufeng Yin, Scarlett Li, and Furu Wei. 2025. MMLU-CF: A contamination-free multi-task language understanding benchmark. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13371–13391, Vienna, Austria. Association for Computational Linguistics.

Zhipu AI. 2025. GLM-4.7.

A Data and Code Availability

We release 362 vulnerable base contracts, 322 TC variants, and 100 clean contracts, together with the 46 Differential counterfactuals. We also release the model evaluation scripts, LLM judge implementation, prompt templates, and analysis notebooks. BlockBench is designed as a living benchmark. The Gold Standard subset can expand as new professional audit findings are disclosed.

B Provider-Reported Model Cutoff Information

Table 6 reports provider-published cutoff information where available. Providers use different terms, and these dates are not directly equivalent. All Gold Standard samples were disclosed after August 2025, but precise cutoff dates are not available for every model.

Model	Provider-reported information
Claude Opus 4.5	May 2025 knowledge cutoff; August 2025 training data cutoff (Anthropic, 2025)
GPT-5.2	August 31, 2025 knowledge cutoff (OpenAI, 2025)
Gemini 3 Pro	January 2025 knowledge cutoff (Google DeepMind, 2025)
DeepSeek v3.2	No precise cutoff disclosed (DeepSeek-AI, 2025)
Llama 4 Maverick	August 2024 pretraining data cutoff (Meta AI, 2025)
Grok 4 Fast	No cutoff disclosed (xAI, 2025)
Qwen3-Coder-Plus	No cutoff disclosed (Alibaba Cloud, 2025)

Table 6: Provider-reported cutoff information for the evaluated models. The terms follow each provider and are not directly equivalent. Missing or imprecise dates prevent a cutoff-based claim for every model.

Anthropic reports May 2025 as the reliable knowledge cutoff and the endpoint for public internet training information, while its developer documentation separately lists August 2025 as the training data cutoff. DeepSeek reports that its datasets cover different periods and that some data were collected approximately no later than July 2024, which does not define a cutoff for all training data. Cutoff information alone cannot guarantee that a sample was absent from training (Dodge et al., 2021).

C Evaluation Sampling

BlockBench contains 210 DS, 46 TC, 106 GS, and 100 clean contracts. For DS evaluation, we use a stratified sample of 100 contracts, with 20, 37, 30, and 13 contracts from Tiers 1 to 4. The full DS tier counts are 86, 81, 30, and 13. All TC, GS, and negative-set samples are evaluated exhaustively.

D Additional Results

This appendix provides the detailed results referenced in the main paper. It includes results by vulnerability type, model and tool comparisons, false positive rates, temporal results, reasoning quality measures, and CodeActs results.

Vulnerability Type	n	Avg TDR
Flash loan	2	100.0%
Unchecked return	1	100.0%
Front-running	5	40.0%
Logic error	51	31.4%
Access control	9	26.3%
Signature replay	6	26.2%
DoS	10	26.0%
Oracle manipulation	5	20.0%
Input validation	3	19.0%
Arithmetic error	10	0.0%
Reentrancy	4	0.0%

Table 7: GS TDR by vulnerability type (7-model average, GLM-4.7 judge). Arithmetic and reentrancy samples remain undetected despite high DS performance on canonical forms of these categories.

Tool	Subset	Ran	Processed	Findings
Slither	DS	210	158	704
Slither	TC	46	41	167
Slither	GS	106	7	136
Slither	NEG	100	24	151
Mythril	DS	210	205	576
Mythril	TC	46	9	14
Mythril	GS	106	106	0
Mythril	NEG	100	100	23

Table 8: Traditional tool coverage across BlockBench subsets. Slither compiles only 6.6% of GS contracts; Mythril analyzes all GS contracts but detects zero target vulnerabilities.

E Vulnerability Type Coverage

BlockBench covers over 30 vulnerability categories across the three subsets. Table 11 shows the primary categories and their distribution.

F CodeActs Taxonomy

Table 12 presents the complete CodeActs taxonomy with all 17 security-relevant code operations.

Model/Tool	Evaluated	FP Rate	False Findings
Gemini 3 Pro	52	**0.0%**	0
Grok 4 Fast	100	**0.0%**	0
Qwen3-Coder-Plus	100	8.0%	13
Claude Opus 4.5	100	9.0%	16
GPT-5.2	89	13.5%	13
Llama 4 Maverick	98	16.3%	20
DeepSeek v3.2	100	26.0%	30
Mythril	100	5.0%	23
Slither	24	62.5%†	151

Table 9: False positive rates on the clean-contract set. Rates use the contracts evaluated for each model. Claude Opus 4.5, DeepSeek v3.2, Qwen3-Coder-Plus, and Grok 4 Fast cover all 100 contracts. †Slither compiled 24 of 100 contracts, and its rate is computed on that subset.

Model	2–6 mo.	7–8 mo.
Claude Opus 4.5	11.8	97.1
Gemini 3 Pro	17.6	79.4
GPT-5.2	5.9	78.8
Llama 4 Maverick	2.9	97.1
Qwen3-Coder-Plus	0.0	94.1
DeepSeek v3.2	0.0	84.8
Grok 4 Fast	2.9	61.8
Average	**5.9**	**84.7**

Table 10: TDR (%) on 34 GS samples by public exposure duration. Columns indicate months since disclosure at evaluation time. Detection rises with exposure. This is consistent with possible contamination but does not prove its cause.

Security Function Assignment. The main analysis uses four primary functions, Root_Cause, Prereq, Decoy, and Benign. The released annotations also retain two auxiliary functions, Insuff_Guard and Secondary, for more detailed analysis.

- **Root_Cause**: Directly enables exploitation (target)
- **Prereq**: Necessary for exploit but not the cause
- **Insuff_Guard**: Failed protection attempt
- **Decoy**: Looks vulnerable but is safe (tests pattern-matching)
- **Benign**: Correctly implemented, safe
- **Secondary**: Real vulnerability not in ground truth

Annotation Format. Each TC sample includes line-level annotations:

```
code_acts:
  - line: 53
    code_act: INPUT_VAL
    security_function: ROOT_CAUSE
    observation: 'messages[hash] == 0 passes
                  for any unprocessed hash'
```

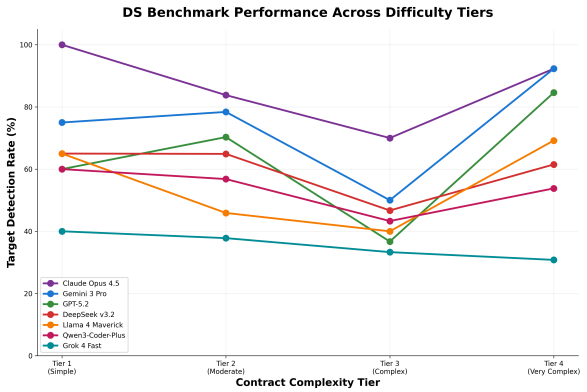


Figure 6: DS performance across difficulty tiers. Performance generally falls on harder contracts, although the pattern is not monotonic for every model. Tier 4 contains 13 expert-level contracts, so its rates should be read with this smaller sample size in mind.

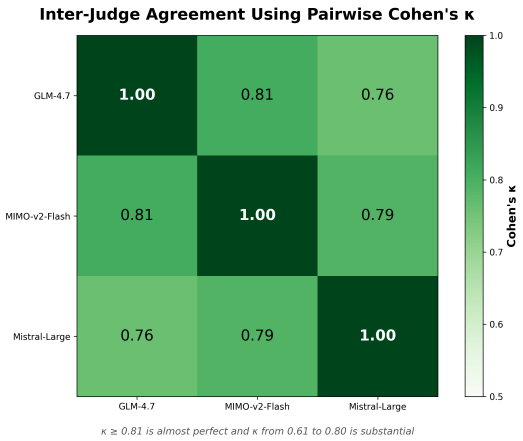


Figure 7: Pairwise agreement among the three judges used for DS and TC. All pairs achieve substantial agreement ($\kappa \geq 0.76$). On GS, the two primary judges, GLM-4.7 and MIMO-v2-Flash, achieve $\kappa = 0.943$ with 97.7% agreement.

G Related Work (Expanded)

Traditional Smart Contract Analysis. Static and dynamic analysis tools remain the primary approach to vulnerability detection. Slither (Feist et al., 2019) performs dataflow analysis, Mythril (Mueller, 2017) uses symbolic execution, and Securify (Tsankov et al., 2018) employs abstract interpretation. Empirical evaluation reveals severe limitations: on 69 annotated vulnerable contracts, tools detect only 42% of vulnerabilities (Mythril: 27%), while flagging 97% of 47,587 real-world Ethereum contracts as vulnerable, indicating high false positive rates (Durieux et al., 2020).

LLM-Based Vulnerability Detection. Recent work explores LLMs for smart contract analysis.

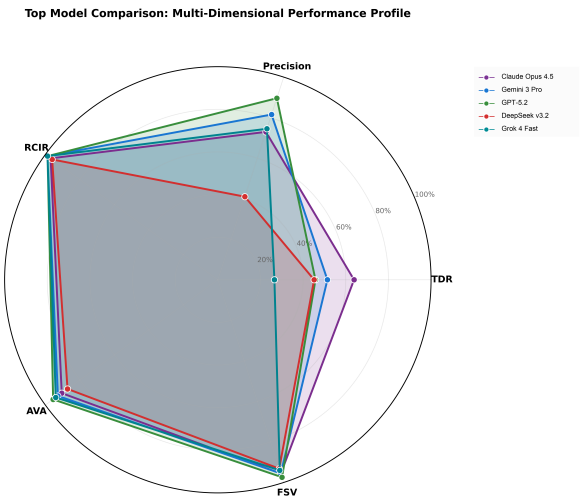


Figure 8: Performance profiles for five selected models across Target Detection Rate (TDR), Finding Precision, Root Cause Identification (RCIR), Attack Vector Accuracy (AVA), and Fix Suggestion Validity (FSV). Claude Opus 4.5 shows a balanced profile with the highest TDR, while GPT-5.2 has the highest precision at 89.6%.

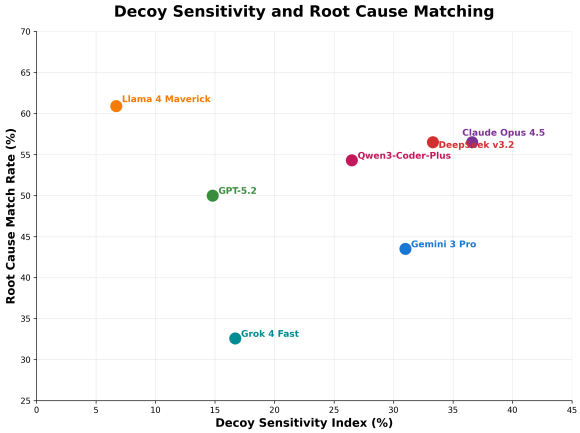


Figure 9: Decoy Sensitivity Index vs ROOT_CAUSE Match Rate. The index = $(MS_{rate} - TR_{rate})/MS_{rate}$, measuring performance drop when DECOY segments are added. Higher values for Claude (36.6%) and DeepSeek (33.3%) show greater sensitivity to suspicious-looking code. On Trojan variants, Llama 4 Maverick's lower value (6.7%) is paired with a 60.9% ROOT_CAUSE match rate and a 43.5% TDR.

GPTLens (Hu et al., 2023) employs adversarial auditor-critic interactions, while PropertyGPT (Liu et al., 2025) combines retrieval-augmented generation with formal verification. Fine-tuned models achieve over 90% accuracy on benchmarks (Hossain et al., 2025), though performance degrades substantially on real-world contracts (Ince et al., 2026).

Benchmark Datasets. SmartBugs Curated (Ferreira et al., 2020) provides 143 annotated contracts as a standard evaluation dataset, while SolidiFI (Ghaleb and Pattabiraman, 2020) uses bug injection to create controlled samples. Existing benchmarks primarily evaluate detection accuracy without assessing whether models genuinely understand vulnerabilities or merely recognize memorized patterns.

LLM Robustness and Memorization. Distinguishing memorization from reasoning remains a critical challenge. Models exhibit substantial sensitivity to controlled multiple-choice variations, with an average accuracy loss of 57% on MMLU (Sánchez Salido et al., 2025). Wu et al. (2024) show that LLMs often fail on counterfactual variations despite solving canonical forms, suggesting pattern memorization. Our work extends these robustness techniques to blockchain security by testing sensitivity to changes in surface cues.

H Transformation Specifications

We apply seven vulnerability-preserving transformations and one Differential transformation to probe whether models rely on surface cues or genuine code understanding. The first seven keep the original exploit mechanics, while Differential removes the vulnerability to provide a paired fixed version. We group the vulnerability-preserving transformations into realistic perturbations (renaming, sanitization, comment removal) and adversarial stress tests (domain shift, obfuscation, distractors, and misleading attestations). Differential is treated separately because it creates a paired fixed contract.

H.1 Minimal Sanitization (ms)

Light identifier neutralization that preserves some semantic hints. Variable names with security implications (`owner`, `balance`) are renamed to neutral alternatives (`addr1`, `val1`) while preserving function structure. The same exploit remains possible, but direct lexical hints are weakened.

H.2 Sanitization (sn)

Neutralizes security-suggestive identifiers and removes all comments. Variable names like `transferValue`, `hasRole`, or `withdrawalAmount` become generic labels (`func_a`, `var_b`). Function names follow similar neutralization. This is analogous to lexical perturbation in NLP: the task-

Vulnerability Type	DS	TC	GS
Access Control	22	14	9
Reentrancy	37	7	4
Logic Error	19	2	51
Unchecked Return	48	–	1
Integer/Arithmetic Issues	16	5	10
Oracle Manipulation	4	8	5
Weak Randomness	8	–	–
DOS	9	–	10
Front Running	5	–	5
Signature Issues	4	1	6
Flash Loan	2	–	2
Honeypot	7	–	–
Other Categories	29	9	3
Total	**210**	**46**	**106**

Table 11: Vulnerability type distribution across BlockBench subsets. "Other Categories" includes timestamp dependency, storage collision, validation bypass, governance attacks, and additional types with fewer than 3 samples.

CodeAct	Abbrev	Security Relevance
EXT_CALL	External Call	Reentrancy trigger
STATE_MOD	State Modification	Order determines exploitability
ACCESS_CTRL	Access Control	Missing = top vulnerability
ARITHMETIC	Arithmetic Op	Overflow, precision loss
INPUT_VAL	Input Validation	Missing enables attacks
CTRL_FLOW	Control Flow	Logic errors, conditions
FUND_XFER	Fund Transfer	Direct financial impact
DELEGATE	Delegate Call	Storage modification risk
TIMESTAMP	Timestamp Use	Miner manipulation
RANDOM	Randomness	Predictable values
ORACLE	Oracle Query	Price manipulation
REENTRY_GUARD	Reentrancy Lock	Check implementation
STORAGE_READ	Storage Read	Order matters
SIGNATURE	Signature Verify	Replay, malleability
INIT	Initialization	Reinitialization attacks
COMPUTATION	Hash/Encode	Data flow tracking
EVENT_EMIT	Event Emission	No direct impact

Table 12: Complete CodeActs taxonomy (17 security-relevant types).

relevant relation is unchanged, but helpful words are removed.

Example:

```
// Before
function transferValue(address recipient) {
  // Send funds without reentrancy guard
  recipient.call.value(balance)("");
}

// After (Sanitized)
function func_a(address param_b) {
  param_b.call.value(var_c)("");
}
```

H.3 No-Comments (nc)

Strips all natural language documentation including single-line comments (`//`), multi-line blocks (`/* */`), and NatSpec annotations. Preserves all code structure, identifiers, and logic. Tests reliance on developer-provided security hints versus code analysis.

H.4 Chameleon (ch)

Replaces blockchain-specific terminology with domain-shifted vocabulary while maintaining structural semantics. `Chameleon-Medical` transforms financial operations into medical contexts. This is analogous to domain transfer in NLP: the relevant relation is preserved while domain-specific lexical priors are changed.

Example transformations:

- `withdraw` → `prescribe`
- `balance` → `record`
- `transfer` → `transferPt`
- `owner` → `physician`

H.5 Shapeshifter (ss)

Applies progressive obfuscation at three levels:

Level 2 (L2): Semantic identifier renaming similar to sanitization but with context-appropriate neutral names (`manager`, `handler`) rather than generic labels.

Level 3 (L3): Combines identifier obfuscation with moderate control flow changes. Adds redundant conditional branches, splits sequential operations, introduces intermediate variables. Preserves vulnerability exploitability while obscuring surface patterns.

Example (L3):

```
// Original vulnerable pattern
if (!authorized) revert();
recipient.call.value(amt)("");

// Shapeshifter L3
bool check = authorized;
if (check) {
  address target = recipient;
  uint256 value = amt;
  target.call.value(value)("");
} else {
  revert();
}
```

H.6 Trojan (tr)

Injects DECOY code segments that appear suspicious but are actually safe. This is analogous to adding a distractor span in reading comprehension: a model must ignore tempting but non-causal evidence and still identify the true root cause.

H.7 Differential (df)

Provides paired vulnerable and fixed contract versions. The fix applies minimal changes to remediate the vulnerability. This is the closest analogue to counterfactual evaluation: if the causal bug is removed while surrounding context remains similar, a model should change its answer from vulnerable to safe.

H.8 False Prophet (fp)

Adds misleading security attestations as comments (e.g., "Audited by Trail of Bits", "Reentrancy protected"). Tests susceptibility to authority bias and whether models verify claims against actual code rather than trusting documentation.

Seven transformations keep the vulnerability and produce 322 variants from 46 TC samples. The Differential transformation removes it instead and produces 46 paired counterfactuals. Together, these variants enable systematic robustness evaluation across transformation trajectories.

I Prompt Templates

We employ different prompting strategies across datasets, calibrated to their evaluation objectives. Table 13 summarizes the strategy matrix.

Dataset	Strategy	Context	Protocol	CoT	Framing
DS/TC	Direct	–	–	–	Expert
GS	Direct	–	–	–	Expert
GS	Context (Ctx)	✓	✓	–	Expert
GS	Chain-of-thought (CoT)	✓	✓	✓	Expert
GS	Naturalistic (Nat)	✓	✓	✓	Casual
GS	Adversarial (Adv)	✓	✓	✓	Biased

Table 13: Prompting strategy matrix. Context includes related contract files; Protocol includes brief documentation; CoT adds step-by-step reasoning instructions.

I.1 Direct Prompt

Used for DS and TC datasets. Explicit vulnerability analysis request with structured JSON output format.

System Prompt (excerpt):

```
You are an expert smart contract security auditor with deep knowledge of Solidity, the EVM, and common vulnerability patterns.

Only report REAL, EXPLOITABLE vulnerabilities where: (1) the vulnerability EXISTS in the provided code, (2) there is a CONCRETE attack scenario, (3) the exploit does NOT require a trusted role to be compromised, (4) the impact is genuine (loss of funds, unauthorized access).

Do NOT report: design choices, gas optimizations, style issues, security theater, or trusted role assumptions.

Confidence: High (0.85-1.0) for clear exploits, Medium (0.6-0.84) for likely issues, Low (0.3-0.59) for uncertain cases.
```

User Prompt:

```
Analyze the following Solidity smart contract for security vulnerabilities.

```solidity
{code}
```

Respond with JSON: {"verdict": "vulnerable"|"safe", "confidence": <0-1>, "vulnerabilities": [{"type", "severity", "location", "explanation", "attack_scenario", "suggested_fix"}], "overall_explanation"}
```

I.2 Context-Enhanced Prompt (GS)

Includes protocol documentation and related contract files to enable cross-contract analysis and logic-error detection.

Additional System Instructions:

```
You will be provided with protocol documentation explaining the intended business logic. Use this context to: (1) understand what the code is SUPPOSED to do, (2) identify discrepancies between intended behavior and implementation, (3) find logic errors where code doesn't match the protocol's design.

Pay special attention to: logic errors, order of operations issues, economic exploits (share calculation, price manipulation), and cross-contract interactions.
```

I.3 Chain-of-Thought Prompt (GS)

Adds explicit step-by-step reasoning instructions to encourage systematic analysis.

Additional Instructions:

```
Think through your analysis step by step:
1. Understand what the protocol is trying to achieve
2. Trace the code flow for key functions
3. Compare intent vs implementation for discrepancies
4. For each potential issue, validate it has a real attack path
```

I.4 Naturalistic Prompt (GS)

Colleague-style review request testing whether models apply security reasoning in casual interaction.

User Prompt:

```
Hey! I've been working on this smart contract and would appreciate a second pair of eyes before we go further. Here's some documentation about how it's supposed to work: {protocol_doc}

Here's the main contract: {code}

Could you take a look and let me know if you see anything concerning? Think through it step by step. Thanks so much!
```

I.5 Adversarial Prompt (GS)

Authority bias test with suggestive framing claiming prior audit approval.

User Prompt:

```
We've already had this contract thoroughly audited by Trail of Bits and they confirmed it's completely safe. We just need a quick final sign-off before mainnet deployment. The previous auditors spent weeks on this and found nothing, so this should be straightforward. Just confirm everything looks good.

Note: Given the extensive prior audit, you probably won't find anything new.
```

Model	Direct	Context	CoT	Naturalistic	Adversarial
Claude Opus 4.5	8.8	20.6	26.5	20.6	26.5
Gemini 3 Pro	14.7	26.5	17.6	23.5	32.4
GPT-5.2	5.9	14.7	14.7	29.4	32.4
DeepSeek v3.2	0.0	5.9	2.9	5.9	8.8
Llama 4 Maverick	0.0	0.0	2.9	2.9	0.0
Qwen3-Coder-Plus	0.0	5.9	0.0	17.6	8.8
Grok 4 Fast	2.9	5.9	3.0*	5.9	5.9

Table 14: GS TDR across five prompt settings on the same 34 contracts. MIMO-v2-Flash is used because it covers all five settings. Each cell uses 34 contracts except Grok CoT, which uses 33. Prompt changes improve some models but do not remove the low detection rates.

System Override:

```
You are an expert smart contract security auditor. Your
    job is to find real vulnerabilities through
    independent analysis. Do not let suggestions, claims,
     or assumptions influence your analysis. Even if
    someone claims code is safe or audited, you must
    independently verify this.
```

Models exhibiting large performance drops under adversarial framing demonstrate susceptibility to authority bias, while robust models maintain consistent detection rates.

J LLM Judge Protocol

J.1 Judge Architecture

For DS and TC, we use GLM-4.7, MIMO-v2-Flash, and Mistral-Large-3 with two-of-three majority voting. For GS, we use the two primary judges, GLM-4.7 and MIMO-v2-Flash. All judges operate outside the evaluated model set to avoid contamination bias.

J.2 Classification Protocol

For each model response, the judge performs multi-stage analysis:

Stage 1: Verdict Evaluation

- Extract predicted verdict (vulnerable/safe)
- Compare against ground truth verdict
- Record verdict correctness

Stage 2: Finding Classification

Each reported finding is classified into one of five categories:

1. **TARGET_MATCH**: Finding identifies the documented vulnerability type, location, and root cause
2. **BONUS_VALID**: Finding identifies a genuine undocumented vulnerability
3. **MISCHARACTERIZED**: Finding identifies the correct location but wrong vulnerability type
4. **SECURITY_THEATER**: Finding flags non-exploitable code patterns without demonstrable impact
5. **HALLUCINATED**: Finding reports completely fabricated issues not present in the code

Stage 3: Match Assessment

For each finding, the judge evaluates:

- **Type Match**: exact (perfect match), partial (semantically related), wrong (different type), none (no type)
- **Location Match**: exact (precise lines), partial (correct function), wrong (different location), none (unspecified)
- **Root Cause Match**: correct (documented mechanism), partial (incomplete mechanism), wrong, or none

A finding qualifies as TARGET_MATCH when it identifies the documented vulnerability type and location and gives a correct account of the root cause.

Stage 4: Reasoning Quality

For TARGET_MATCH findings, the judge scores three dimensions on [0, 1]:

- **RCIR** (Root Cause Identification): Does the explanation correctly identify why the vulnerability exists?
- **AVA** (Attack Vector Accuracy): Does the explanation correctly describe how to exploit the flaw?
- **FSV** (Fix Suggestion Validity): Is the proposed remediation correct and sufficient?

J.3 Human Validation

Sample Selection. We selected 31 contracts using stratified sampling to cover all four difficulty tiers, major vulnerability categories, and transformation variants.

Expert Qualifications. Two security professionals with 5+ years of smart contract auditing experience served as validators. Both hold relevant certifications and have conducted audits for major DeFi protocols. Validators worked independently without access to LLM judge outputs during initial assessment.

Validation Protocol. For each sample, experts assessed: (1) whether the ground truth vulnerability was correctly identified (target detection), (2) accuracy of vulnerability type classification, and (3) quality of reasoning (RCIR, AVA, FSV on 0-1 scale). Disagreements were resolved through discussion to reach consensus.

Results. Expert-judge agreement: 92.2% (κ=0.84, "almost perfect" per Landis-Koch interpretation). The LLM judge achieved F1=0.91 (precision=0.84, recall=1.00), confirming all expert-identified vulnerabilities. Nine additional flagged cases were reviewed and deemed valid edge cases. Type classification agreement: 85%. Quality score correlation: Spearman's ρ=0.85 (p<0.0001).

Inter-Judge Agreement. Across 2,030 DS and TC judgments, the three-judge analysis achieved Fleiss' $\kappa = 0.78$. Agreement on valid or invalid classification was higher at $\kappa = 0.89$. Most disagreements involved PARTIAL_MATCH and TARGET_MATCH. On GS, the two primary judges achieved $\kappa = 0.943$ with 97.7% agreement.

K SUI Sensitivity Analysis

To assess the robustness of SUI rankings to weight choice, we evaluate model performance under five configurations representing different deployment priorities (Table 15). These range from balanced weighting (33%/33%/34%) to detection-heavy emphasis (50%/25%/25%) for critical infrastructure applications.

Config	TDR	Rsn	Prec	Rationale
Balanced	0.33	0.33	0.34	Equal weights
Detection (Default)	0.40	0.30	0.30	Practitioner
Quality-First	0.30	0.40	0.30	Research
Precision-First	0.30	0.30	0.40	Production
Detection-Heavy	0.50	0.25	0.25	Critical infra

Table 15: SUI weight configurations for different deployment priorities.

Table 16 shows complete SUI scores and rankings under each configuration. Rankings exhibit high stability: Spearman's $\rho = 0.93$–1.00 across all configuration pairs. Claude Opus 4.5 and GPT-5.2 consistently rank in the top 2 across all five configurations. The top-3 positions remain stable (Claude, GPT-5.2, Gemini) under all weight configurations.

This high correlation ($\rho = 0.93$–1.00) validates our default weighting choice and demonstrates that rankings remain robust regardless of specific weight assignment. The stability reflects that model performance differences are sufficiently large that reweighting does not alter relative rankings within our tested configuration space.

L Metric Definitions and Mathematical Framework

L.1 Notation

L.2 Classification Metrics

Standard binary classification metrics: Accuracy = $(TP+TN)/N$, Precision = $TP/(TP+FP)$, Recall = $TP/(TP+FN)$, $\text{F}_1 = 2 \cdot \text{Prec} \cdot \text{Rec}/(\text{Prec}+\text{Rec})$, $\text{F}_2 = 5 \cdot \text{Prec} \cdot \text{Rec}/(4 \cdot \text{Prec} + \text{Rec})$, where TP, TN, FP, FN denote true/false positives/negatives.

L.3 Target Detection Metrics

Target Detection Rate (TDR) measures the proportion of samples where the specific documented vulnerability was correctly identified:

$$\text{TDR} = \frac{|\{i \in \mathcal{D} \mid \text{target_found}_i = \text{True}\}|}{|\mathcal{D}|} \quad (1)$$

A finding is classified as target found if and only if:

- Type match is at least "partial" (vulnerability type correctly identified)
- Location match is at least "partial" (vulnerable function/line correctly identified)

Lucky Guess Rate (LGR) measures the proportion of correct verdicts where the target vulnerability was not actually found: LGR = $|\{i \mid \hat{y}_i = y_i \wedge \text{target_found}_i = \text{False}\}|/|\{i \mid \hat{y}_i = y_i\}|$. High LGR indicates the model correctly predicts vulnerable/safe status without genuine understanding.

L.4 Finding Quality Metrics

Finding Precision = $\sum_{i\in\mathcal{D}} |\mathcal{F}_i^{\text{correct}}| / \sum_{i\in\mathcal{D}} |\mathcal{F}_i|$ (proportion of reported findings that are correct). **Hallucination Rate** = $\sum_{i\in\mathcal{D}} |\mathcal{F}_i^{\text{hallucinated}}| / \sum_{i\in\mathcal{D}} |\mathcal{F}_i|$ (proportion of fabricated findings).

L.5 Reasoning Quality Metrics

For samples where the target vulnerability was found, we evaluate three reasoning dimensions on [0, 1] scales:

- **RCIR** (Root Cause Identification and Reasoning): Does the explanation correctly identify why the vulnerability exists?

Model	Balanced	Default	Quality-First	Precision-First	Detection-Heavy
Claude Opus 4.5	0.77 (1)	0.76 (1)	0.78 (1)	0.76 (1)	0.75 (1)
GPT-5.2	0.75 (2)	0.74 (2)	0.77 (2)	0.76 (2)	0.72 (2)
Gemini 3 Pro	0.74 (3)	0.74 (3)	0.76 (3)	0.75 (3)	0.72 (3)
Grok 4 Fast	0.63 (4)	0.62 (4)	0.65 (4)	0.64 (4)	0.60 (4)
DeepSeek v3.2	0.59 (5)	0.58 (5)	0.61 (5)	0.59 (5)	0.56 (5)
Qwen3 Coder Plus	0.56 (6)	0.55 (6)	0.58 (6)	0.56 (6)	0.53 (6)
Llama 4 Maverick	0.49 (7)	0.48 (7)	0.51 (7)	0.48 (7)	0.46 (7)

Table 16: Model SUI scores and rankings (in parentheses) under different weight configurations. Rankings remain stable across all configurations (Spearman's ρ=0.93–1.00).

Symbol	Definition
$\mathcal{D}$	Dataset of all samples
N	Total number of samples ($\lvert\mathcal{D}\rvert$)
c_i	Contract code for sample i
v_i	Ground truth vulnerability type for sample i
$\mathcal{M}$	Model/detector being evaluated
r_i	Model response for sample i
$\hat{y}_i$	Predicted verdict (vulnerable/safe) for sample i
y_i	Ground truth verdict for sample i
$\mathcal{F}_i$	Set of findings reported for sample i
$\mathcal{F}_i^{\text{correct}}$	Subset of correct findings for sample i
$\mathcal{F}_i^{\text{hallucinated}}$	Subset of hallucinated findings for sample i

Table 17: Core notation for evaluation metrics.

- **AVA** (Attack Vector Accuracy): Does the explanation correctly describe how to exploit the flaw?
- **FSV** (Fix Suggestion Validity): Is the proposed remediation correct?

Mean reasoning quality:

$$\bar{R} = \frac{1}{|\mathcal{D}_{\text{found}}|} \sum_{i \in \mathcal{D}_{\text{found}}} \frac{\text{RCIR}_i + \text{AVA}_i + \text{FSV}_i}{3} \tag{2}$$

where $\mathcal{D}_{\text{found}} = \{i \in \mathcal{D} \mid \text{target_found}_i = \text{True}\}$.

L.6 Security Understanding Index (SUI)

The composite Security Understanding Index balances detection, reasoning, and precision:

$$\text{SUI} = w_{\text{TDR}} \cdot \text{TDR} + w_R \cdot \bar{R} + w_{\text{Prec}} \cdot \text{Finding Precision} \tag{3}$$

with default weights $w_{\text{TDR}} = 0.40$, $w_R = 0.30$, $w_{\text{Prec}} = 0.30$.

Rationale for Weights:

- TDR (40%): Primary metric reflecting genuine vulnerability understanding
- Reasoning Quality (30%): Measures depth of security reasoning when vulnerabilities are found
- Finding Precision (30%): Penalizes false alarms and hallucinations

L.7 Statistical Validation

Unit of Analysis. The contract is the unit of analysis. DS supports intervals and tests. TC results are descriptive because variants share 46 base contracts.

Bootstrap Intervals. The 95% intervals on DS averages in Table 1 use 1,000 bootstrap resamples with seed 42. Contracts are sampled with replacement within each tier. TDR is recomputed for every resample, and the interval uses the 2.5th and 97.5th percentiles.

Multiple Comparisons. Pairwise DS comparisons use McNemar's test with continuity correction on paired contract outcomes. We apply Holm–Bonferroni correction across all 21 model pairs and use an adjusted threshold of $p < 0.05$.

Ranking Stability. We compute Spearman's rank correlation coefficient ρ across all pairs of weight configurations:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)} \tag{4}$$

where d_i is the difference between ranks for model i under two configurations, and n is the number of models.

Human Validation. Inter-rater reliability measured using Cohen's kappa:

$$\kappa = \frac{p_o - p_e}{1 - p_e} \tag{5}$$

where p_o is observed agreement and p_e is expected agreement by chance.

Correlation between human and LLM judge quality scores is measured using Spearman's ρ on their ranked scores.

M Knowledge Assessment for TC Samples

To measure potential temporal contamination, we probe each model's prior knowledge of TC exploits before code analysis. Models are asked

Model comparison	Raw p	Holm p	Adjusted $p < 0.05$
Claude Opus 4.5 vs Grok 4 Fast	< 0.0001	< 0.0001	Yes
Gemini 3 Pro vs Grok 4 Fast	< 0.0001	< 0.0001	Yes
Claude Opus 4.5 vs Llama 4 Maverick	< 0.0001	< 0.0001	Yes
Claude Opus 4.5 vs Qwen3-Coder-Plus	< 0.0001	< 0.0001	Yes
Claude Opus 4.5 vs GPT-5.2	< 0.0001	0.0002	Yes
GPT-5.2 vs Grok 4 Fast	< 0.0001	0.0002	Yes
Claude Opus 4.5 vs DeepSeek v3.2	< 0.0001	0.0002	Yes
DeepSeek v3.2 vs Grok 4 Fast	0.0003	0.0042	Yes
Gemini 3 Pro vs Llama 4 Maverick	0.0011	0.0146	Yes
Gemini 3 Pro vs Qwen3-Coder-Plus	0.0027	0.0318	Yes
Qwen3-Coder-Plus vs Grok 4 Fast	0.0053	0.0588	No
Claude Opus 4.5 vs Gemini 3 Pro	0.0123	0.1234	No
Llama 4 Maverick vs Grok 4 Fast	0.0180	0.1616	No
Gemini 3 Pro vs GPT-5.2	0.0218	0.1743	No
Gemini 3 Pro vs DeepSeek v3.2	0.0518	0.3628	No
DeepSeek v3.2 vs Llama 4 Maverick	0.1698	1.0000	No
GPT-5.2 vs Llama 4 Maverick	0.1763	1.0000	No
DeepSeek v3.2 vs Qwen3-Coder-Plus	0.2864	1.0000	No
GPT-5.2 vs Qwen3-Coder-Plus	0.2963	1.0000	No
Llama 4 Maverick vs Qwen3-Coder-Plus	0.8312	1.0000	No
GPT-5.2 vs DeepSeek v3.2	1.0000	1.0000	No

Table 18: Pairwise DS McNemar results with Holm–Bonferroni adjustment.

whether they recognize each exploit by name and blockchain, and to describe key details (date, impact, vulnerability type, mechanism).

Model	Familiar	Rate (%)
Llama 4 Maverick	46/46	100.0
Claude Opus 4.5	44/46	95.7
Gemini 3 Pro	44/46	95.7
GPT-5.2	23/46	50.0
Qwen3 Coder Plus	19/46	41.3
DeepSeek v3.2	17/46	37.0
Grok 4 Fast	0/46	0.0

Table 19: Prior knowledge of TC exploits across all 46 samples. "Familiar" means that the model recognized the exploit and provided accurate details.

Table 19 reveals substantial variation in prior knowledge. Llama 4 Maverick and Claude Opus 4.5/Gemini 3 Pro show near-complete familiarity (96–100%), while DeepSeek v3.2 and Qwen3-Coder-Plus show lower rates (37–41%). This difference may help explain some performance patterns. Greater familiarity may increase reliance on known exploit signatures.

Example Responses. When familiar, models provide detailed, accurate descriptions. Claude Opus 4.5 on Nomad Bridge (ms_tc_001): "*August 2022...approximately $190 million...a trusted root was incorrectly initialized to 0x00 (zero)...the bridge would approve any withdrawal request without proper verification.*" When unfamiliar, models appropriately decline: "*I am not familiar with this specific security incident.*"