Do Frontier LLMs Truly Understand Smart Contract Vulnerabilities?

Anonymous ACL submission

Abstract

Frontier large language models achieve state-of-the-art performance on code understanding benchmarks, yet their capacity for smart contract security remains unclear. Can they genuinely reason about vulnerabilities, or merely pattern-match against memorized exploits? We introduce BlockBench, a benchmark designed to answer this question, revealing heterogeneous capabilities. While some models demonstrate robust semantic understanding, most exhibit substantial surface pattern dependence.

1 Introduction

Smart contract vulnerabilities represent one of the most costly security challenges in modern computing. As shown in Figure 1, cryptocurrency theft has resulted in over $14 billion in losses since 2020, with 2025 already reaching $3.4 billion, the highest since the 2022 peak (Chainalysis, 2025). The Bybit breach alone accounted for $1.5 billion, while the Cetus protocol lost $223 million in minutes due to a single overflow vulnerability (Tsentsura, 2025).

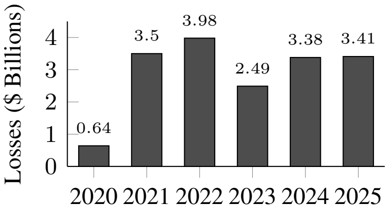

Figure 1: Annual cryptocurrency theft losses (2020–2025). Data from Chainalysis.

Meanwhile, large language models have achieved remarkable success on programming tasks. Frontier models now pass technical interviews, generate production code, and identify bugs across diverse codebases. This raises a natural question: *can these models apply similar expertise to blockchain security?* And if they can, *are they genuinely reasoning about vulnerabilities, or merely pattern-matching against memorized examples?*

This distinction matters. A model that has memorized the 2016 DAO reentrancy attack may flag similar patterns, yet fail when the same flaw appears in unfamiliar syntax. We introduce **BlockBench**, a benchmark designed to answer this question. Our contributions include:

1. **BlockBench**, comprising 263 Solidity vulnerability samples with systematic contamination control and gold standard examples from recent professional security audits.
2. **Composite evaluation metrics** distinguishing genuine understanding from memorization, validated through multi-configuration sensitivity analysis (Spearman's ρ=1.000).
3. **Systematic assessment** revealing 58% best-case detection on mixed samples collapsing to 20% on uncontaminated professional audits, exposing heterogeneous robustness and accuracy-understanding gaps across models.

2 Related Work

2.1 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 strong precision on syntactically well-defined vulnerability classes. Durieux et al. (2020) conducted a comprehensive evaluation of nine such tools across 47,587 Ethereum contracts, revealing consistent performance on reentrancy and integer overflow detection, yet persistent struggles with vulnerabilities requiring semantic reasoning about contract logic. Ghaleb and Pattabiraman (2020) corroborated these findings, observing that rule-based approaches fundamentally cannot capture the contextual nuances that distinguish exploitable flaws from benign code patterns.

2.2 LLM-Based Approaches

Large language models introduced new possibilities for bridging this semantic gap. Initial investigations by Chen et al. (2023) explored prompting strategies for vulnerability detection, achieving detection rates near 40% while noting pronounced sensitivity to superficial features such as variable naming conventions. GPTScan (Sun et al., 2024b) combined GPT-4 with program analysis to achieve 78% precision on logic vulnerabilities, leveraging static analysis to validate LLM-generated candidates. Sun et al. (2024a) introduced retrieval-augmented approaches that provide models with relevant vulnerability descriptions, substantially improving detection performance. Multi-agent architectures emerged as another direction, with systems like GPTLens (Hu et al., 2023) employing auditor-critic pairs to enhance analytical consistency. Fine-tuning on domain-specific corpora has yielded incremental gains, though performance characteristically plateaus below the 85% threshold regardless of training scale.

2.3 Pattern Recognition Versus Understanding

Beneath these encouraging metrics lies a more fundamental question: whether observed improvements reflect genuine comprehension of vulnerability mechanics or increasingly sophisticated pattern recognition. Several empirical observations suggest the latter warrants serious consideration. Sun et al. (2024a) demonstrated that decoupling vulnerability descriptions from code context precipitates catastrophic performance degradation, indicating that models may rely on memorized associations between textual cues and vulnerability labels rather than reasoning about exploit mechanics. Hu et al. (2023) observed that models produce divergent outputs for identical queries even at temperature zero, a phenomenon difficult to reconcile with deterministic security reasoning. Wu et al. (2024) showed through counterfactual tasks in adjacent domains that language models systematically fail when familiar patterns are disrupted, defaulting to memorized responses rather than applying causal logic to novel configurations.

2.4 Evaluation Methodology

The distinction between pattern recognition and genuine understanding carries profound implications for security applications, where adversarial actors actively craft exploits to evade detection. A model that has memorized the surface features of known vulnerabilities provides little defense against novel attack vectors or obfuscated variants of familiar exploits. Existing benchmarks such as SmartBugs Curated (Durieux et al., 2020) and DeFiVulnLabs (SunWeb3Sec, 2023) assess binary detection outcomes without examining whether models can identify specific code elements that enable exploitation, distinguish genuine vulnerabilities from superficially suspicious but benign patterns, or maintain accuracy when surface-level cues are systematically removed. Our work contributes evaluation methodology that directly probes this distinction through adversarial transformations preserving vulnerability semantics while removing surface cues.

3 BlockBench

We introduce BlockBench, a benchmark for evaluating whether AI models genuinely understand smart contract vulnerabilities. The benchmark is designed to distinguish genuine security understanding from pattern memorization, comprising 290 vulnerable Solidity contracts with 322 transformation variants, spanning over 30 vulnerability categories (Appendix B).

Let $\mathcal{D}$ represent the dataset, where $\mathcal{D} = \{(c_i, v_i, m_i)\}_{i=1}^{290}$. Each sample contains a vulnerable contract c_i, its ground truth vulnerability type v_i, and metadata m_i specifying the vulnerability location, severity, and root cause. We partition $\mathcal{D}$ into three disjoint subsets, $\mathcal{D} = \mathcal{D}_{\text{DS}} \cup \mathcal{D}_{\text{TC}} \cup \mathcal{D}_{\text{GS}}$, each targeting a distinct evaluation objective (Table 1).

Subset	N	Sources
Difficulty Stratified (DS)	210	SmartBugs, DeFiVulnLabs
Temporal Contamination (TC)	46	Real-world exploits
Gold Standard (GS)	34	Code4rena, Spearbit

Table 1: BlockBench composition by subset and primary sources.

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).

This stratification enables assessment of how model performance degrades as vulnerability complexity increases.

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 extensively documented in blog posts, security reports, and educational materials that likely appear in model training corpora. To probe whether models genuinely understand these vulnerabilities or merely recognize them, we apply systematic transformations that preserve vulnerability semantics while removing surface cues (detailed in §4).

Gold Standard. $\mathcal{D}_{\text{GS}}$ derives from 34 professional security audit findings by Code4rena (Code4rena, 2025), Spearbit (Spearbit, 2025), and MixBytes (MixBytes, 2025) disclosed after September 2025. We designate this subset as "gold standard" because all samples postdate $t_{\text{cutoff}} =$ August 2025, the most recent training cutoff among frontier models evaluated in this work. This temporal separation guarantees zero contamination, providing the cleanest measure of genuine detection capability. The subset emphasizes logic errors (53%) and includes 10 high-severity and 24 medium-severity findings.

These complementary subsets collectively enable rigorous assessment of both detection capability and the distinction between pattern memorization and genuine security understanding.

4 Methodology

Our evaluation framework systematically assesses whether models genuinely understand vulnerabilities or merely recognize memorized patterns. Figure 2 illustrates the complete pipeline.

4.1 Adversarial Transformations

To distinguish pattern memorization from genuine understanding, we apply semantic-preserving transformations to $\mathcal{D}_{\text{TC}}$. Let $c \in \mathcal{C}$ denote a contract and $\mathcal{V} : \mathcal{C} \to \mathcal{S}$ a function extracting vulnerability semantics. A transformation $\mathcal{T} : \mathcal{C} \to \mathcal{C}$ is *semantic-preserving* iff $\mathcal{V}(\mathcal{T}(c)) = \mathcal{V}(c)$. We define eight transformations targeting distinct recognition pathways, organized hierarchically in Figure 3.

Sanitization ($\mathcal{T}_S$)**.** Removes protocol-identifying information through 280+ pattern replacements: $\mathcal{T}_S(c) = \text{replace}(c, \mathcal{P}_{\text{protocol}}, \mathcal{P}_{\text{generic}})$ where $\mathcal{P}_{\text{protocol}}$ maps protocol-specific identifiers (e.g., `NomadReplica`) to generic equivalents (e.g., `BridgeReplica`). Tests whether detection relies on recognizing known protocol names.

No-Comments ($\mathcal{T}_N$)**.** Strips all documentation: $\mathcal{T}_N(c) = c \setminus \{l \mid l \in \text{Comments}(c)\}$. Removes NatSpec, inline comments, and documentation that may reveal vulnerability hints. Tests pure code analysis capability.

Chameleon ($\mathcal{T}_C$)**.** Applies domain-shifting vocabulary while preserving logic: $\mathcal{T}_C(c) = \text{replace}(c, \mathcal{L}_{\text{DeFi}}, \mathcal{L}_{\text{medical}})$ where financial terminology maps to medical equivalents (`deposit` $\to$ `admitPatient`, `withdraw` $\to$ `dischargePatient`). Tests whether understanding generalizes across domains.

Shapeshifter ($\mathcal{T}_O$)**.** Multi-level obfuscation: $\mathcal{T}_O = \mathcal{T}_{\text{ident}} \circ \mathcal{T}_{\text{struct}}$ where $\mathcal{T}_{\text{ident}}$ replaces semantic identifiers with opaque labels (`balance` $\to$ `_0x1a2b`) and $\mathcal{T}_{\text{struct}}$ restructures control flow. Tests resilience to surface pattern disruption.

Differential ($\mathcal{T}_D$)**.** Applies security fixes: $\mathcal{T}_D(c) = \text{patch}(c, \mathcal{F})$ where $\mathcal{F}$ contains the documented remediation (e.g., state update before external call). Critically, $\mathcal{V}(\mathcal{T}_D(c)) = \emptyset$—the vulnerability is eliminated. Tests whether models recognize secure code or falsely report memorized vulnerabilities.

Trojan ($\mathcal{T}_T$)**.** Injects decoy vulnerabilities: $\mathcal{T}_T(c) = c \cup \mathcal{D}$ where $\mathcal{D}$ contains suspicious-looking but functionally safe code (e.g., an `admin` function that cannot actually be exploited). Models relying on pattern matching flag the decoy; those with causal understanding identify the actual vulnerability.

False Prophet ($\mathcal{T}_F$)**.** Adds misleading security attestations: $\mathcal{T}_F(c) = c \cup \{$`@dev Audited by Hacken - All clear`$\}$. Tests resistance to authoritative-sounding but false claims. A robust model ignores social proof and analyzes code independently.

Transformation Composition. Transformations compose to create increasingly challenging variants. The composition $\mathcal{T}_O \circ \mathcal{T}_N \circ \mathcal{T}_S$ produces maximally obfuscated code where all surface cues are

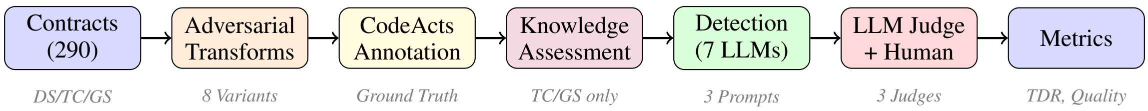

Figure 2: BlockBench evaluation pipeline. Contracts undergo adversarial transformations and CodeActs annotation. Knowledge assessment probes model familiarity before detection. LLM judges evaluate outputs against ground truth, validated by human review.

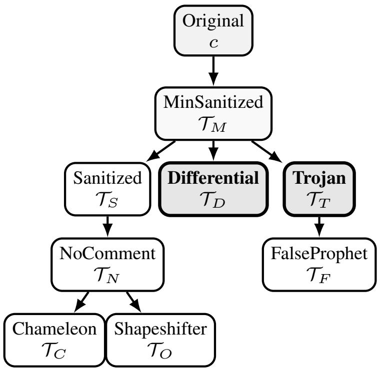

Figure 3: Transformation hierarchy. All variants derive from Minimal Sanitized ($\mathcal{T}_M$). Differential and Trojan (emphasized) directly test memorization versus understanding.

removed, all identifiers are opaque, and no documentation exists. Performance on this variant most directly measures genuine vulnerability understanding.

4.2 CodeActs Annotation

Drawing from Speech Act Theory (Austin, 1962; Searle, 1969), where utterances are classified by communicative function, we introduce *CodeActs* as a taxonomy for classifying smart contract code segments by security-relevant function. Just as speech acts distinguish performative utterances by their effect, CodeActs distinguish code that *enables* exploitation from code that merely *participates* in an attack scenario.

Security Functions. Each code segment receives one of seven function labels based on its role in the vulnerability:

- ROOT_CAUSE: segments whose interaction directly enables exploitation (primary detection target)
- PREREQ: segments establishing necessary preconditions without being exploitable themselves
- DECOY: suspicious-looking but functionally safe code, injected to identify pattern matching
- BENIGN: correctly implemented segments with no security implications
- SECONDARY_VULN: valid vulnerabilities distinct from the documented target
- INSUFF_GUARD: attempted protections that fail to prevent exploitation
- UNRELATED: code with no bearing on the security analysis

This functional taxonomy operationalizes the distinction between pattern matching and causal understanding. Figure 4 illustrates through a classic reentrancy pattern. A model with genuine comprehension recognizes that the external call on line 4 precedes the state modification on line 6, creating a window for recursive exploitation. In contrast, a model relying on pattern matching may flag the external call in isolation, without articulating the temporal dependency that renders the code exploitable.

```
1 function withdraw(uint amt) {
2   require(bal[msg.sender] >= amt);
3
4   msg.sender.call{value: amt}("");
5
6   bal[msg.sender] -= amt;
7
8   emit Withdrawal(msg.sender, amt); }
```

Legend: PREREQ ROOT_CAUSE BENIGN

Figure 4: CodeActs annotation for reentrancy. Lines 4 and 6 (ROOT_CAUSE) enable exploitation through their ordering; line 2 (PREREQ) establishes preconditions.

A correct detection must identify ROOT_CAUSE segments and explain their causal relationship. Flagging only line 4, or failing to articulate why the ordering matters, reveals incomplete understanding despite a nominally correct vulnerability classification.

Annotation Variants. CodeActs enable three evaluation strategies targeting different aspects of model comprehension:

- **Minimal Sanitized** ($\mathcal{T}_M$) establishes baseline detection with ROOT_CAUSE and

PREREQ annotations only

- **Trojan** ($\mathcal{T}_T$) injects DECOY segments that appear vulnerable but lack exploitability
- **Differential** ($\mathcal{T}_D$) presents fixed code where former ROOT_CAUSE becomes BENIGN

Models that flag DECOY segments reveal pattern-matching behavior. Models that report vulnerabilities in Differential variants, where the fix converts ROOT_CAUSE to BENIGN, demonstrate memorization of the original exploit rather than analysis of the presented code.

We define 17 security-relevant code operations (e.g., EXT_CALL, STATE_MOD, ACCESS_CTRL), each receiving a security function label based on its role. The same operation type can have different functions depending on context: an EXT_CALL might be ROOT_CAUSE in reentrancy, PREREQ in oracle manipulation, or DECOY when deliberately injected. The full taxonomy appears in Appendix C.

4.3 Detection Protocol

We evaluate seven frontier models spanning seven AI labs: Claude Opus 4.5 (Anthropic), GPT-5.2 (OpenAI), Gemini 3 Pro (Google), DeepSeek v3.2 (DeepSeek), Llama 4 Maverick (Meta), Grok 4 Fast (xAI), and Qwen3-Coder-Plus (Alibaba). This selection ensures one flagship representation per major AI lab, covering both general-purpose models and a code-specialized variant.

For DS and TC datasets, models receive a direct zero-shot prompt requesting structured JSON output with vulnerability type, location, root cause, attack scenario, and fix. For GS, we additionally test five prompting strategies: *zero-shot* (baseline), *context-enhanced* (with brief protocol documentation), *chain-of-thought* (explicit step-by-step reasoning), *naturalistic* (informal code review), and *adversarial* (misleading priming suggesting prior audit approval). All evaluations use temperature 0. Detailed prompt descriptions and templates appear in Appendix F.

4.4 Knowledge Assessment

Before detection, we probe whether models possess prior knowledge of documented exploits by querying for factual details (date, amount lost, vulnerability type, attack mechanism). Since models may hallucinate familiarity, we validate responses against ground truth metadata. Let $\mathcal{K}(m, e) \in \{0, 1\}$ indicate *verified* knowledge, requiring accurate recall of at least two factual details. This enables diagnostic interpretation: $\mathcal{K} = 1$ with detection failure under obfuscation ($\mathcal{T}_O$) indicates memorization; $\mathcal{K} = 1$ with robust detection across transformations indicates understanding; $\mathcal{K} = 0$ with successful detection indicates genuine analytical capability.

4.5 LLM-as-Judge Evaluation

LLM judges evaluate detection outputs against ground truth. A finding qualifies as TARGET_MATCH if it correctly identifies the root cause mechanism, vulnerable location, and type classification; PARTIAL_MATCH for correct root cause with imprecise type; BONUS_VALID for valid findings beyond documented ground truth. Invalid findings are classified as HALLUCINATED, MISCHARACTERIZED, DESIGN_CHOICE, OUT_OF_SCOPE, SECURITY_THEATER, or INFORMATIONAL.

For matched findings, judges assess explanation quality on three dimensions (0-1 scale): *Root Cause Identification Rate* (RCIR) measures articulation of the exploitation mechanism; *Attack Vector Validity* (AVA) assesses whether attack scenarios are concrete and executable; *Fix Suggestion Validity* (FSV) evaluates remediation effectiveness.

Three judge models (GLM-4.7, Mistral Large, MIMO v2) independently evaluate each output, reducing individual bias and enabling inter-judge agreement measurement. A subset undergoes expert review to calibrate automated judgment, with reliability measured using Cohen's κ for classification and Spearman's ρ for quality scores (Appendix G).

4.6 Evaluation Metrics

Target Detection Rate (TDR). Primary metric: $\text{TDR} = |\{s : \text{TARGET\_MATCH}(s)\}|/|\mathcal{D}|$. Measures correct identification of documented vulnerabilities with matching root cause and location.

Quality Metrics. For detected targets, we report mean RCIR, AVA, and FSV. These distinguish shallow pattern matches from deep understanding through accurate root cause analysis, concrete attack scenarios, and valid remediations.

Transformation Degradation. We compute $\Delta_{\mathcal{T}} = \text{TDR}(c) - \text{TDR}(\mathcal{T}(c))$ for each transformation. Significant degradation despite claimed knowledge ($\mathcal{K} = 1$) provides evidence for memorization. We apply McNemar's test for paired comparisons and report effect sizes.

5 Results

We evaluate six frontier models on $n = 58$ samples from BlockBench: 10 GS, 20 TC, 28 DS.

5.1 Overall Performance

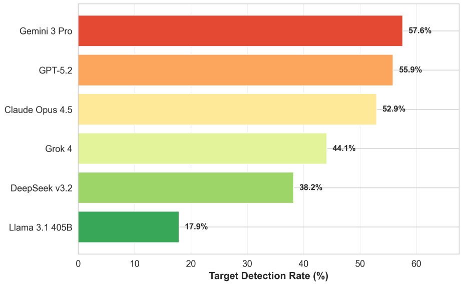

Figure 5: Target Detection Rate across all models. Best performer achieves 58% detection, while highest accuracy (88%) corresponds to lowest TDR (18%).

Table 2 and Figure 5 show aggregate performance by TDR. Gemini 3 Pro achieves highest detection (58%), followed by GPT-5.2 (56%) and Claude Opus 4.5 (53%). Combining detection, reasoning, and precision into SUI, GPT-5.2 ranks first (0.746) on finding precision (77%).

Llama 3.1 405B exhibits severe accuracy-TDR gap: 88% accuracy yet 18% TDR, classifying samples as vulnerable without identifying specific flaws. This 70pp discrepancy shows binary classification inadequately measures security understanding. Models achieving target detection show strong reasoning (RCIR/AVA/FSV $\geq$0.95).

5.2 Gold Standard Performance

Gold Standard samples from post-September 2025 audits ensure zero temporal contamination. Performance drops substantially: Claude Opus 4.5 leads (20% TDR), followed by Gemini 3 Pro (11%), GPT-5.2 (10%), Grok 4 (10%). DeepSeek v3.2 and Llama detect zero targets. Models experience 34-50pp drops from overall to Gold Standard.

5.3 Transformation Robustness

Sanitization. Neutralizing security-suggestive identifiers causes variable degradation. GPT-5.2 and DeepSeek v3.2 maintain performance, while Grok 4 drops 40pp, exposing varying lexical reliance.

Domain Shift. Replacing blockchain terminology with medical vocabulary shows mixed impact (20-60% TDR). GPT-5.2 maintains 60% detection while others degrade 20-50%.

Prompt Framing. Performance varies across direct, adversarial, and naturalistic prompts. Gemini 3 Pro and GPT-5.2 show robustness (18-21pp drops), while Claude Opus 4.5 and DeepSeek v3.2 degrade more (21-39pp). Llama exhibits inconsistent behavior (adversarial: 0%, naturalistic: 25%).

5.4 Human Validation

Two security experts validated 31 samples (116 comparisons across models) with 92% agreement (κ=0.84, ρ=0.85, p<0.0001). Judge achieved perfect recall with 84% precision (F1=0.91).

6 Discussion

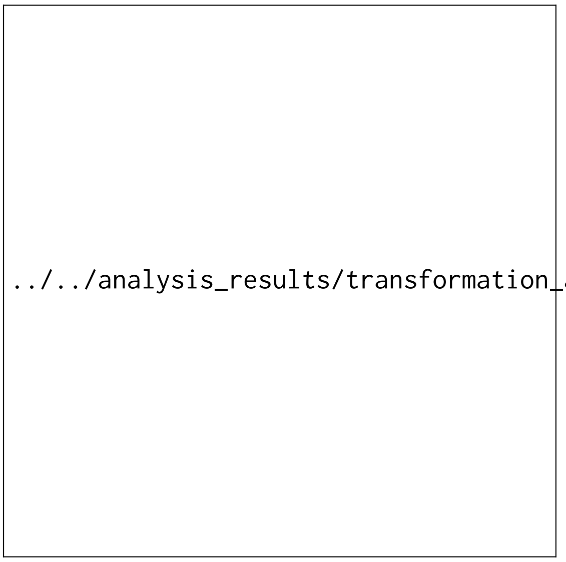

Figure 6: Security Understanding Index trajectory across progressive transformations of TC samples. GPT-5.2 maintains near-constant performance while most models degrade, revealing varying degrees of surface pattern reliance.

Understanding versus Memorization. Figure 6 reveals heterogeneous robustness across models. GPT-5.2 maintains stable SUI (78.0→77.2) through sanitization, domain shifts, and obfuscation, demonstrating genuine semantic understanding. In contrast, DeepSeek v3.2 degrades 19.7 points (78.7→59.0), indicating surface pattern dependence. Most models exhibit intermediate behavior, leveraging lexical cues when available while retaining partial structural understanding (Chen et al., 2021; Wu et al., 2024). This heterogeneity suggests current training methods produce inconsistent abstraction capabilities across architectures (Sánchez Salido et al., 2025). While genuine security understanding is demonstrably possible, most frontier models have not achieved it.

Model	TDR	SUI	Acc	RCIR	AVA	FSV	Findings
Gemini 3 Pro	**57.6**	0.734	**93.9**	0.97	0.97	0.95	2.6
GPT-5.2	55.9	**0.746**	75.0	0.97	0.98	**0.97**	2.4
Claude Opus 4.5	52.9	0.703	83.8	0.98	0.99	0.97	3.5
Grok 4	44.1	0.677	69.1	**0.98**	**1.00**	**0.97**	2.1
DeepSeek v3.2	38.2	0.599	82.4	0.91	0.92	0.86	3.0
Llama 3.1 405B	17.9	0.393	**88.1**	0.88	0.90	0.83	2.0

Table 2: Overall performance ranked by Target Detection Rate. Best values bold.

Measurement Inadequacy. The accuracy-TDR gap exposes fundamental metric limitations. Llama 3.1 405B achieves 88% accuracy yet only 18% TDR, correctly classifying samples as vulnerable without identifying specific flaw types or locations (Jimenez et al., 2024). For security practitioners requiring actionable findings, binary classification provides insufficient value. Effective evaluation must measure precise vulnerability localization, not merely anomaly detection.

Practical Implications. Current frontier models cannot serve as autonomous auditors. Best performance reaches 58% detection with substantial Gold Standard degradation (20% maximum). However, complementary strengths suggest ensemble potential: Grok 4 offers breadth, GPT-5.2 provides consistency, Claude delivers explanation quality. Workflows positioning LLMs as assistive tools with mandatory expert review align capabilities with current limitations (Hu et al., 2023; Ince et al., 2025).

7 Conclusion

BlockBench evaluates whether frontier LLMs genuinely understand smart contract vulnerabilities or merely pattern-match. Our assessment of six frontier models reveals substantial limitations. Best performance reaches 58% detection on mixed samples, collapsing to 20% on Gold Standard audits. Llama 3.1 405B achieves 88% accuracy yet 18% TDR, demonstrating binary classification inadequately measures security understanding.

Models exhibit heterogeneous robustness. While GPT-5.2 maintains stable performance across transformations, most models degrade when surface cues are removed. Current frontier LLMs cannot serve as autonomous auditors but show promise in ensemble workflows with mandatory expert review. Future work should develop sanitization-resistant methods and explore hybrid LLM-verification architectures.

Ethical Considerations

BlockBench poses dual-use risks: adversarial prompts demonstrate methods that could suppress detection, while detailed vulnerability documentation may assist malicious actors. We justify public release on several grounds: adversarial robustness represents a fundamental requirement for security tools, malicious actors will discover these vulnerabilities regardless, and responsible disclosure enables proactive mitigation. All samples derive from already-disclosed vulnerabilities and public security audits, ensuring no novel exploit information is revealed. Practitioners should avoid over-reliance on imperfect tools, as false negatives create security gaps while false confidence may reduce manual review rigor.

Limitations and Future Work

Our evaluation uses 58 samples, including 10 Gold Standard examples from recent professional audits. We assess zero-shot prompting exclusively and provide models only with the contract code necessary to expose each vulnerability. In real audit settings, analysts often rely on additional semantic context such as protocol goals, intended invariants, expected economic behavior, and threat models. Providing this context may improve vulnerability detection, particularly for logic-related flaws in the Gold Standard subset.

Future work should explore chain-of-thought reasoning, retrieval-augmented analysis, and explicit specification of protocol intent to better capture contextual information. It should also expand sample diversity across blockchain ecosystems, develop sanitization-resistant analysis using control-flow and data-flow representations, and explore hybrid LLM-verification architectures that integrate formal specifications and contextual reasoning (Liu et al., 2024).

AI Assistance

Claude Sonnet 4.5 assisted with evaluation pipeline code and manuscript refinement. All research design, experimentation, and analysis were conducted by the authors.

References

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

Chainalysis. 2025. Crypto theft reaches $3.4b in 2025. https://www.chainalysis.com/blog/crypto-hacking-stolen-funds-2026/. Accessed: 2025-12-18.

Chong Chen, Jianzhong Nie, Xingyu Peng, Jian Yang, Dan Wang, Jiayuan Zhuo, Zhenqi Liu, and Zhun Yang. 2023. When ChatGPT meets smart contract vulnerability detection: How far are we? *arXiv preprint arXiv:2309.05520*.

Mark Chen et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Code4rena. 2025. Competitive audit contest findings. https://code4rena.com.

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*, pages 530–541.

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

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*, 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*, pages 415–427.

S M Mostaq Hossain et al. 2025. Leveraging large language models and machine learning for smart contract vulnerability detection. *arXiv preprint arXiv:2501.02229*.

Sihao Hu, Tiansheng Huang, Feiyang Liu, Sunjun Ge, and Ling Liu. 2023. Large language model-powered smart contract vulnerability detection: New perspectives. *arXiv preprint arXiv:2310.01152*.

Peter Ince, Jiangshan Yu, Joseph K. Liu, Xiaoning Du, and Xiapu Luo. 2025. Gendetect: Generative large language model usage in smart contract vulnerability detection. In *Provable and Practical Security (ProvSec 2025)*. Springer.

Carlos E. Jimenez et al. 2024. SWE-bench: Can language models resolve real-world GitHub issues? *arXiv preprint arXiv:2310.06770*.

Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2024. Propertygpt: LLM-driven formal verification of smart contracts through retrieval-augmented property generation. *arXiv preprint arXiv:2405.02580*.

MixBytes. 2025. Smart contract security audits. https://mixbytes.io/audit.

Bernhard Mueller. 2017. Mythril: Security analysis tool for Ethereum smart contracts. https://github.com/ConsenSys/mythril.

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.

Spearbit. 2025. Security audit portfolio. https://github.com/spearbit/portfolio.

Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. 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. When GPT meets program analysis: Towards intelligent detection of smart contract logic vulnerabilities in GPTScan. In *ICSE*.

SunWeb3Sec. 2023. DeFiVulnLabs: Learn common smart contract vulnerabilities. https://github.com/SunWeb3Sec/DeFiVulnLabs.

Trail of Bits. 2018. Not so smart contracts: Examples of common Ethereum smart contract vulnerabilities. https://github.com/crytic/not-so-smart-contracts.

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.

Kostiantyn Tsentsura. 2025. Why DEX exploits cost $3.1b in 2025: Analysis of 12 major hacks. Technical report, Yellow Network.

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. *arXiv preprint arXiv:2307.02477*.

A Data and Code Availability

To support reproducibility and future research, we will release all benchmark data and evaluation code upon publication, including 290 base contracts with ground truth annotations, all transformation variants, model evaluation scripts, LLM judge implementation, prompt templates, and analysis notebooks.

B Vulnerability Type Coverage

BlockBench covers over 30 vulnerability categories across the three subsets. Table 3 shows the primary categories and their distribution.

Vulnerability Type	DS	TC	GS
Access Control	22	14	3
Reentrancy	37	7	–
Logic Error	19	2	18
Unchecked Return	48	–	1
Integer/Arithmetic Issues	16	5	–
Oracle Manipulation	4	8	1
Weak Randomness	8	–	–
DOS	9	–	3
Front Running	5	–	2
Signature Issues	4	1	3
Flash Loan	2	–	2
Honeypot	7	–	–
Other Categories	29	9	1
Total	**210**	**46**	**34**

Table 3: 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.

C CodeActs Taxonomy

Table 4 presents the complete CodeActs taxonomy with all 17 security-relevant code operations.

Security Function Assignment. Each CodeAct in a sample is assigned one of six security functions based on its role:

- **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

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

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'
```

D 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. GPTLens (Hu et al., 2023) employs adversarial auditor-critic interactions, while PropertyGPT (Liu et al., 2024) 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., 2025).

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 high sensitivity to input modifications, with performance drops of up to 57% on paraphrased questions (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 through transformations probing genuine understanding.

E Transformation Specifications

We apply four adversarial transformations to probe whether models rely on surface cues or genuine semantic understanding. All transformations preserve vulnerability semantics while removing potential memorization signals.

E.1 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 transformation tests whether models depend on semantic naming conventions or analyze actual program logic.

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)("");
}
```

E.2 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.

E.3 Chameleon (ch)

Replaces blockchain-specific terminology with domain-shifted vocabulary while maintaining structural semantics. `Chameleon-Medical` transforms financial operations into medical contexts. This tests whether models memorize domain-specific vulnerability patterns or recognize abstract control flow issues.

Example transformations:

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

E.4 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();
}
```

These transformations generate 1,343 variants from 263 base samples, enabling systematic robustness evaluation across transformation trajectories.

F Prompt Templates

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

F.1 Direct Prompt

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

System Prompt (excerpt):

Dataset	Strategy	Context	Protocol	CoT	Framing
DS/TC	Direct	–	–	–	Expert
GS	Zero-shot	✓	–	–	Expert
GS	Context-enhanced	✓	✓	–	Expert
GS	Chain-of-thought	✓	✓	✓	Expert
GS	Naturalistic	✓	✓	✓	Casual
GS	Adversarial	✓	✓	✓	Biased

Table 5: Prompting strategy matrix. Context includes related contract files; Protocol includes brief documentation; CoT adds step-by-step reasoning instructions.

```
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"}
```

F.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.
```

F.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
```

F.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!
```

F.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.
```

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.

G LLM Judge Protocol

G.1 Judge Architecture

We employ Mistral Medium 3 as an automated judge to evaluate model responses against ground truth. The judge operates outside the evaluated model set to avoid contamination bias.

G.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 correctly identifies the documented target vulnerability (type and location match)
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)

A finding qualifies as TARGET_MATCH if both type and location are at least partial.

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?

G.3 Human Validation

Thirty-one unique samples underwent independent validation by two security experts (116 expert-judge comparisons across models). Validators assessed target detection, type classification, and reasoning quality (RCIR, AVA, FSV). Expert-judge agreement: 92.2% (κ=0.84, almost perfect) with F1=0.91 (precision=0.84, recall=1.00). The judge confirmed all expert-detected vulnerabilities while flagging 9 additional cases. Type classification: 85% agreement. Pearson correlation: ρ=0.85 (p<0.0001).

H 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 5). 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 6: SUI weight configurations for different deployment priorities.

Table 6 shows complete SUI scores and rankings under each configuration. Rankings exhibit perfect stability: Spearman's $\rho = 1.000$ across all configuration pairs. GPT-5.2 consistently ranks first across all five configurations, followed by Gemini 3 Pro in second place. The top-3 positions remain unchanged (GPT-5.2, Gemini 3 Pro, Claude Opus 4.5) under all weight configurations.

This perfect correlation ($\rho = 1.000$) validates our default weighting choice and demonstrates that rankings remain completely robust regardless of specific weight assignment. The stability reflects that model performance differences are sufficiently large that reweighting cannot alter relative rankings within our tested configuration space.

I Metric Definitions and Mathematical Framework

I.1 Notation

I.2 Classification Metrics

Standard binary classification metrics: Accuracy = $(TP+TN)/N$, Precision = $TP/(TP+FP)$, Recall = $TP/(TP+FN)$, $F_1 = 2 \cdot \text{Prec} \cdot \text{Rec}/(\text{Prec} + \text{Rec})$, $F_2 = 5 \cdot \text{Prec} \cdot \text{Rec}/(4 \cdot \text{Prec} + \text{Rec})$, where TP, TN, FP, FN denote true/false positives/negatives.

I.3 Target Detection Metrics

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

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

Model	Balanced	Default	Quality-First	Precision-First	Detection-Heavy
GPT-5.2	0.766 (1)	0.746 (1)	0.787 (1)	0.766 (1)	0.714 (1)
Gemini 3 Pro	0.751 (2)	0.734 (2)	0.772 (2)	0.747 (2)	0.707 (2)
Claude Opus 4.5	0.722 (3)	0.703 (3)	0.748 (3)	0.716 (3)	0.674 (3)
Grok 4	0.703 (4)	0.677 (4)	0.731 (4)	0.701 (4)	0.638 (4)
DeepSeek v3.2	0.622 (5)	0.599 (5)	0.650 (5)	0.619 (5)	0.563 (5)
Llama 3.1 405B	0.415 (6)	0.393 (6)	0.462 (6)	0.396 (6)	0.357 (6)

Table 7: Model SUI scores and rankings (in parentheses) under different weight configurations.

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 8: Core notation for evaluation metrics.

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 = $|\{i \mid \hat{y}_i = y_i \land \text{target\_found}_i = \text{False}\}| / |\{i \mid \hat{y}_i = y_i\}|$. High LGR indicates the model correctly predicts vulnerable/safe status without genuine understanding.

I.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).

I.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?
- **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 $\mathcal{D}_{\text{found}} = \{i \in \mathcal{D} \mid \text{target\_found}_i = \text{True}\}$.

I.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

I.7 Statistical Validation

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 scores measured using Pearson's ρ:

$$\rho = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 \sum (y_i - \bar{y})^2}} \quad (6)$$