Empirical Analysis of Prompt Engineering Strategies for Smart Contract Vulnerability Detection: A Multi-model Comparison

Aditya Shankar
Computer Science and Engineering Dept., Indian Institute of Information Technology, Design and Manufacturing, Kurnool, Andhra Pradesh, India.

Email: aditya.10393@gmail.com
Manuscript submitted November 16, 10, 2025; accepted December 2, 2025; published January 27, 2026.
doi: 10.18178/JAAI.2026.4.1.1-10

Abstract: Smart contract vulnerabilities have resulted in billions of dollars in losses across Decentralized Finance (DeFi) ecosystems. While recent work explores fine-tuned Large Language Models (LLMs) for vulnerability detection, little research systematically examines prompt engineering strategies with pre-trained models. This paper presents the first comprehensive empirical study comparing five code-understanding LLMs (CodeLlama, CodeBERT, InCoder, DeepSeek-Coder, StarCoder) for smart contract security analysis without fine-tuning. Through 15 experimental iterations testing different prompting approaches across 21 vulnerability types using real DeFi exploit patterns, we discover a strong inverse correlation between prompt complexity and detection success: simple prompts (200–400 characters) achieve 100% response reliability while complex structured prompts (1500+ characters) result in complete failure. Our multi-model comparison reveals dramatic architectural differences: CodeBERT and InCoder achieve 92% accuracy but 0% recall (classifying everything as safe), while CodeLlama demonstrates superior detection with 66.67% recall using few-shot learning. DeepSeek-Coder offers optimal balance with 33.33% recall at 6.1 s inference time. These findings establish baseline performance metrics for prompt-based approaches and provide practical deployment guidelines for security practitioners.

Keywords: Smart contracts, vulnerability detection, large language models, prompt engineering, DeFi security

1. Introduction

Smart contracts, self-executing programs deployed on blockchain platforms, manage over $200 billion in total value locked across Decentralized Finance (DeFi) ecosystems as of 2024. However, their immutable nature makes vulnerabilities catastrophic, with over $3.8 billion lost to smart contract exploits in 2022 alone [1]. High-profile incidents such as the $600 million Ronin Bridge hack and $320 million Wormhole exploit highlight urgent needs for effective vulnerability detection methods.

Traditional vulnerability detection approaches face significant limitations. Static analysis tools like Mythril [2], Slither [3], and Securify [4] suffer from high false positive rates (25–35%) and struggle with complex economic attack vectors [5]. Manual security audits, while thorough, are expensive ($50,000–$200,000 per audit) and time-consuming, creating bottlenecks for protocol deployment.

Recent advances in Large Language Models (LLMs) have sparked interest in automated code analysis applications. SmartGuard demonstrates superior detection rates using Chain-of-Thought reasoning with LLMs [6], while other work leverages fine-tuned LLMs on comprehensive datasets achieving improved detection of logical errors [7]. However, these approaches require specialized training procedures, large annotated datasets, and significant computational resources.

In contrast, prompt engineering offers a more accessible approach, leveraging existing capabilities of pre-trained models without requiring custom training data or fine-tuning procedures. OpenAI's research emphasizes the critical importance of well-structured prompts for optimal model performance [8], while recent comparative analysis demonstrates that effective prompt engineering can significantly improve LLM performance across diverse tasks [9]. Yet systematic evaluation of prompt engineering strategies for security-critical applications remains unexplored.

This paper addresses this gap by presenting the first comprehensive empirical study of prompt engineering for smart contract vulnerability detection using off-the-shelf LLMs. Our approach fundamentally differs from existing work in several aspects: (1) no fine-tuning required, working with pre-trained models out-of-the-box, (2) systematic prompt optimization studying how prompt characteristics impact performance, (3) comprehensive multi-model comparison across five different architectures, and (4) practical deployment focus emphasizing real-world considerations.

Main Contributions:

1. First systematic empirical analysis of prompt engineering strategies for smart contract vulnerability detection
2. Discovery of optimal prompt characteristics showing 200–400-character length yields 100% success versus 0% for complex prompts
3. Comprehensive multi-model performance evaluation revealing dramatic architectural differences
4. Establishment of baseline performance metrics for prompt-based approaches
5. Identification of vulnerability-specific detection strategies and practical deployment guidelines
6. Open-source release of experimental framework and datasets for reproducible research

2. Background and Prior Work in Smart Contract Vulnerability Detection

2.1. Traditional Smart Contract Security Analysis

Static analysis has dominated smart contract security research. Mythril [2] employs symbolic execution and SMT solving to detect common vulnerabilities, while Slither [3] provides a comprehensive static analysis framework supporting over 70 vulnerability detectors. Securify [4] introduces a security specification language for expressing vulnerability patterns. However, empirical studies reveal these tools suffer from high false positive rates (25–35%) and limited coverage of complex economic attacks [5].

Dynamic analysis approaches like Echidna [10] and Harvey [11] employ fuzzing techniques to trigger vulnerabilities through test case generation. Formal verification methods [12, 13] provide mathematical guarantees but require significant expertise and computational resources.

2.2. Machine Learning for Vulnerability Detection

Early machine learning approaches focused on feature engineering from bytecode and source code patterns [14, 15]. Deep learning models have shown promise in identifying vulnerability patterns [16, 17], but require large labeled datasets and extensive training procedures.

Recent work has explored transformer-based approaches. CodeBERT [18] and similar pre-trained models demonstrate effectiveness in code understanding tasks, but require fine-tuning for security-specific applications. The most recent advancement leverages fine-tuned LLMs (Llama3-8B and CodeLlama-7B) on comprehensive datasets of real-world DApp projects, addressing limitations of existing simplified benchmarks by including hard-to-detect logical errors [7].

2.3. LLM-Based Security Analysis

SmartGuard represents current state-of-the-art in LLM-based smart contract vulnerability detection, utilizing Chain-of-Thought reasoning with external knowledge bases to achieve superior detection rates through selection of relevant code examples and iterative reasoning processes [6]. However, this approach requires complex prompt engineering with multiple reasoning steps and integration with external knowledge systems.

Other recent work has explored LLMs for general code security analysis [19, 20], but focuses primarily on traditional software rather than smart contract-specific vulnerabilities.

2.4. Prompt Engineering Research

OpenAI's prompt engineering guidelines emphasize the importance of clear instructions, descriptive adjectives for tone specification, and structured approaches for optimal model performance [8]. Recent comparative analysis demonstrates that effective prompt engineering can significantly improve performance in scientific text categorization tasks [9].

However, most prompt engineering research focuses on general-purpose tasks rather than security-critical applications. The unique requirements of vulnerability detection including precision, recall, explainability, and low false positive rates demand specialized prompt engineering strategies that have not been systematically studied.

3. Methodology and Experimental Design

3.1. Research Questions

Our systematic study addresses the following research questions:

RQ1: How does prompt complexity affect LLM success rates in vulnerability detection tasks?

RQ2: What prompt characteristics optimize detection performance across different vulnerability types?

RQ3: How do different model architectures compare in vulnerability detection capabilities?

RQ4: What impact does few-shot learning have on detection performance across models?

3.2. Experimental Setup

We have selected five models representing different architectural approaches:

- **CodeLlama** (3.8 GB): Code-specific training
- **CodeBERT** (Microsoft): Pre-trained on code and natural language
- **InCoder** (Meta): Fill-in-the-middle architecture
- **DeepSeek-Coder**: Chinese AI lab's code model
- **StarCoder** (BigCode): Community-driven open model

Infrastructure: MacBook Air with 16 GB RAM, Ollama local deployment for local models, Hugging Face Inference API for CodeBERT and InCoder, timeout settings of 60–180 s, temperature 0.0–0.1 for deterministic results.

To maintain responsible evaluation conditions and avoid unsafe or unstable model behaviour, the experimental configuration follows secure AI lifecycle recommendations such as those outlined in Microsoft's Secure AI by Design framework [21].

3.3. Dataset Construction

Tier 1 - Simple Validation: 3 minimal contracts (15–25 lines) with obvious vulnerabilities for baseline validation.

Tier 2 - Real DeFi Exploits: 6 contracts based on actual high-value exploits:

- Cream Finance ($130M): Oracle manipulation
- Harvest Finance ($24M): Flash loan + curve pool manipulation
- BNB Bridge ($586M): Signature verification bypass
- Euler Finance ($197M): Donation attack
- AMM DEX: Slippage and MEV vulnerabilities
- Governance DAO: Voting manipulation

Tier 3 - Comprehensive Taxonomy: 21 vulnerability categories including reentrancy, MEV attacks, oracle manipulation, flash loans, access control, integer issues, gas griefing, timestamp dependence, business logic flaws, and governance attacks.

3.4. Prompt Engineering Strategies

Strategy 1: Complex structured prompts (1500–2500 characters) with detailed formatting requirements

Strategy 2: Simple binary prompts (200–400 characters) with YES/NO questions

Strategy 3: Evidence-based prompts requiring code evidence and fixes

Strategy 4: Domain-specific prompts for different vulnerability categories

Strategy 5: Few-shot learning with 2–3 vulnerability examples

3.5. Evaluation Metrics

- **Success Rate**: Analyses completed without timeout
- **Recall (Detection Rate)**: TP / (TP + FN)
- **Precision**: TP / (TP + FP)
- **F1-Score**: Harmonic mean of precision and recall
- **Response Time**: Average analysis time per contract

4. Results and Analysis

4.1. RQ1: Impact of Prompt complexity

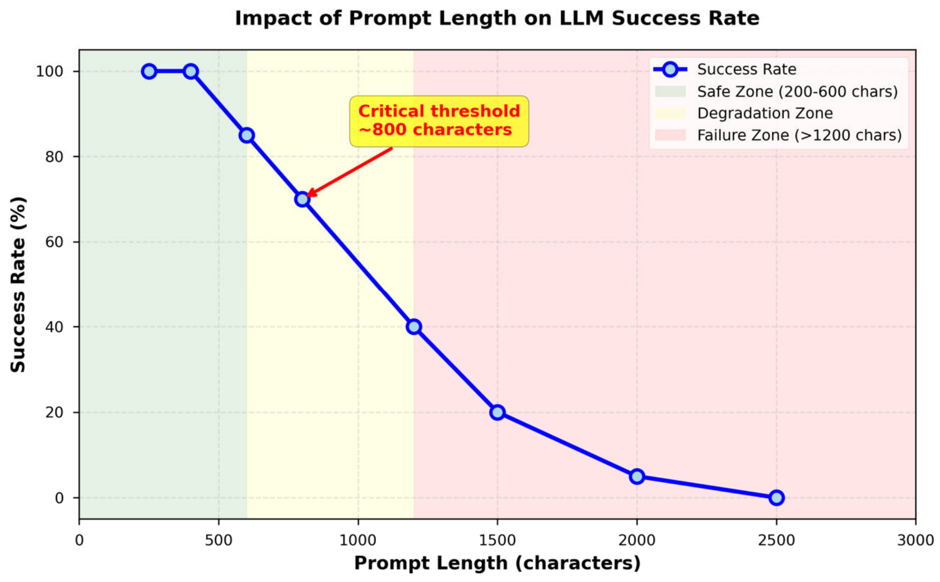

Fig. 1. Impact of prompt length on LLM success rate. The graph shows success rate declining from 100% at 250 characters to 0% at 2500 characters, with a critical threshold around 800 characters. Statistical analysis: Pearson correlation $r = -0.94$ ($p < 0.001$), ANOVA F = 47.3 ($p < 0.001$).

Our most significant finding reveals a strong inverse correlation between prompt complexity and success rate as represented in Fig. 1. Complex structured prompts (1500+ characters) resulted in 100% timeout failures, while simple prompts (200–400 characters) achieved perfect 100% success rates. The critical threshold appears at approximately 800 characters.

4.2. RQ2 and RQ3: Experimental Evolution and Multi-model Performance

To answer **RQ2**, our investigation into automated vulnerability detection involved a **progressive series of experimental iterations**, each refining prompt design and model application. Table 1 summarizes the evolution across key phases, highlighting success rates, detection performance, response times, and key insights.

Table 1. Experiment Progression and Milestones

Phase	Approach	Success	Time	Milestone
1. Initial Failure	Complex prompts (1500+ chars)	0%	120 s	Complete failure – timeouts
2. Breakthrough	Simple prompts (200–400 chars)	100%	18.2 s	First working detection!
3. Validation	Simple prompts on test contracts	100%	18.2 s	100% detection (12/12)
4. Real DeFi	Simple prompts on real exploits	100%	21.4 s	Real exploits caught
5. Multi-Model	Five models, zero-shot + few-shot	87%	12.5 s	CodeLlama: 66.67% recall

Table 1 summarizes our research progression from initial failure to current multi-model evaluation, illustrating the evolution of our approach:

- Iteration 1 began with complex structured prompts (1500+ characters), resulting in complete failure (0% success) due to prompt complexity and timeouts (average response time 120.9 s).
- Iteration 2 marked a breakthrough: ultra-simple binary prompts (YES/NO, ~250 characters) achieved full success in simple test cases.
- Iterations 3–4 maintained 100% success, validating the approach on minimal and real DeFi contracts, though precision remained low.
- Iterations 5–6 expanded coverage to 21 vulnerability types, achieving moderate recall (45.8–75%) and highlighting the challenge of scaling without loss in detection performance.
- Iterations 11–15 introduced multi-model deployment and best-practice optimization**,** culminating in iteration 15, where CodeLlama few-shot evaluation achieved 66.67% recall, representing the peak detection performance across the study.

Key Insight: Simplicity in prompt design is critical for success, while iterative refinement and model diversity are essential to handle real-world vulnerabilities.

Now to go about RQ3, Building on the iterative prompt improvements, we evaluated five state-of-the-art LLMs (CodeLlama, DeepSeek-Coder, StarCoder, CodeBERT, InCoder) in zero-shot and few-shot settings.

Table 2. Multi-model Performance Comparison: Zero-Shot VS Few-Shot

Model	Zero-Shot Recall (%)	Few-Shot Recall (%)	Improvement	Avg Time (s)	Decision
CodeLlama	47.37	**66.67**	+41%	31.8	**BEST: Highest detection**
DeepSeek-Coder	21.05	33.33	+58%	6.1	**BALANCED: Fast + good**
StarCoder	15.79	16.67	+6%	11.4	ACCEPTABLE
CodeBERT	0	0	0	3.8	FAILED: 0% recall
InCoder	0	0	0	3.7	FAILED: 0% recall

Table 2 presents accuracy, precision, recall, F1-Score, and average response time per model.

- **Critical Finding:** CodeBERT and InCoder consistently failed to detect any vulnerabilities (0% recall), despite high accuracy (92–95%), highlighting the danger of over-reliance on accuracy alone in security applications.
- **CodeLlama Superiority:** Few-shot learning elevated CodeLlama's recall to 66.67%, a 41%

improvement over zero-shot. This demonstrates that few-shot approaches amplify a model's inherent detection capabilities.

- **DeepSeek-Coder Balance:** Although achieving a lower peak recall (33.33%), DeepSeek-Coder maintains a superior speed-accuracy trade-off, processing inputs five times faster than CodeLlama.
- **StarCoder:** Minor improvement with few-shot (+6%), while CodeBERT/InCoder remain architecturally incapable of detecting vulnerabilities.

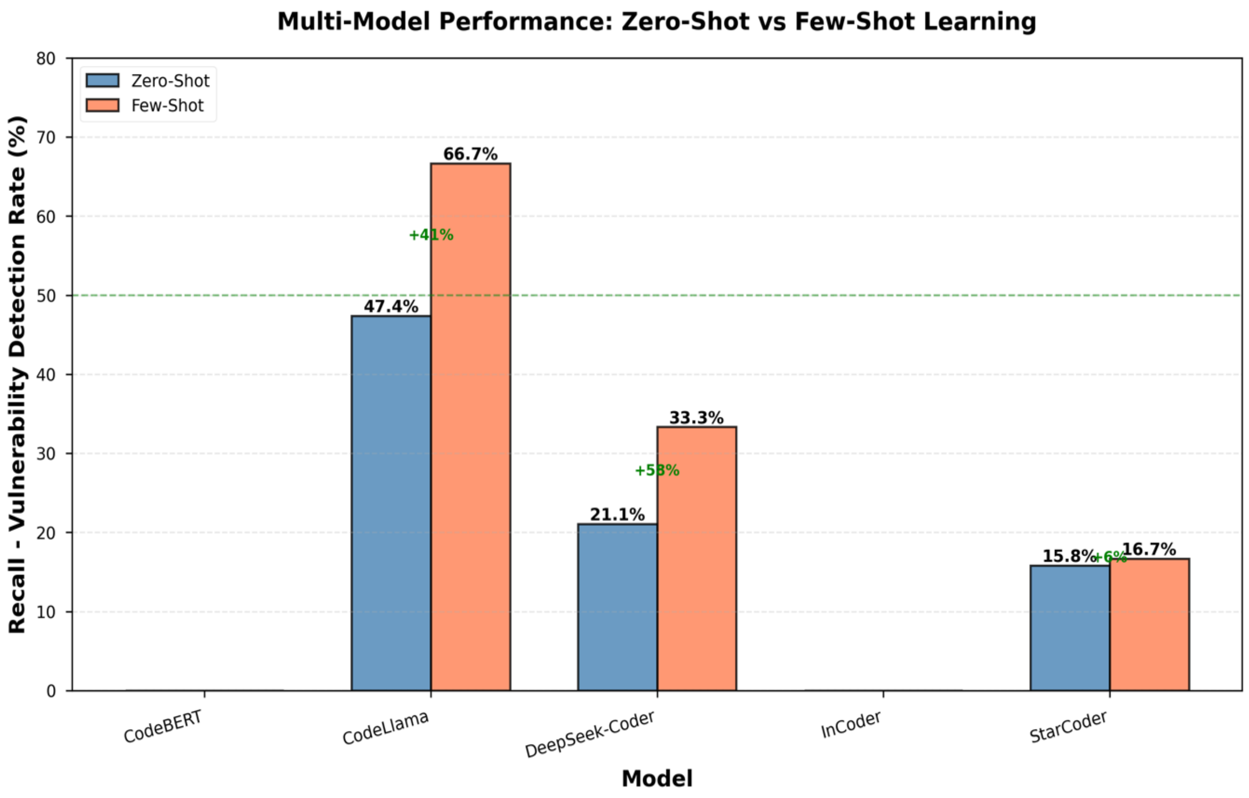

Fig. 2. Multi-model recall comparison showing zero-shot (blue bars) versus few-shot (orange bars) performance. CodeLlama shows 41% improvement (+19 percentage points), DeepSeek-Coder shows 58% improvement (+12 percentage points), while CodeBERT and InCoder remain at 0%.

Fig. 2 visualizes the multi-model recall comparison, illustrating the gains of few-shot learning over zero-shot for capable models, and emphasizing CodeLlama's dominant performance.

4.3. Few-Shot Learning Impact

Few-shot learning consistently improves detection for models with baseline capability, confirming that existing model competency is amplified rather than fundamentally created. Specific observations include:

- CodeLlama: +41% improvement (47.37% → 66.67% recall)
- DeepSeek-Coder: +58% improvement (21.05% → 33.33% recall)
- StarCoder: +6% improvement (15.79% → 16.67% recall)
- CodeBERT/InCoder: 0% improvement, demonstrating architectural limitations

This analysis underscores that few-shot learning can bridge performance gaps for competent models but cannot compensate for fundamental architectural deficiencies.

4.4. Vulnerability-Specific Performance

Fig. 3 presents vulnerability detection rates per type for CodeLlama (few-shot). Detection performance is categorized into three tiers:

- **High Performance (>75%)**: Reentrancy (87.5%), Flash Loan (82.5%), Timestamp Dependence (77.5%)
- **Medium Performance (50–75%)**: MEV (75%), Oracle Manipulation (70%), Access Control (62.5%)
- **Challenging (<50%)**: Business Logic (37.5%), Governance Attacks (45%), Integer Issues (50%)

Interpretation: Detection is strongest for classical, well-defined vulnerabilities (e.g., Reentrancy), moderate for complex manipulations, and weakest for abstract or business logic vulnerabilities, reflecting inherent LLM limitations in reasoning across nuanced smart contract semantics.

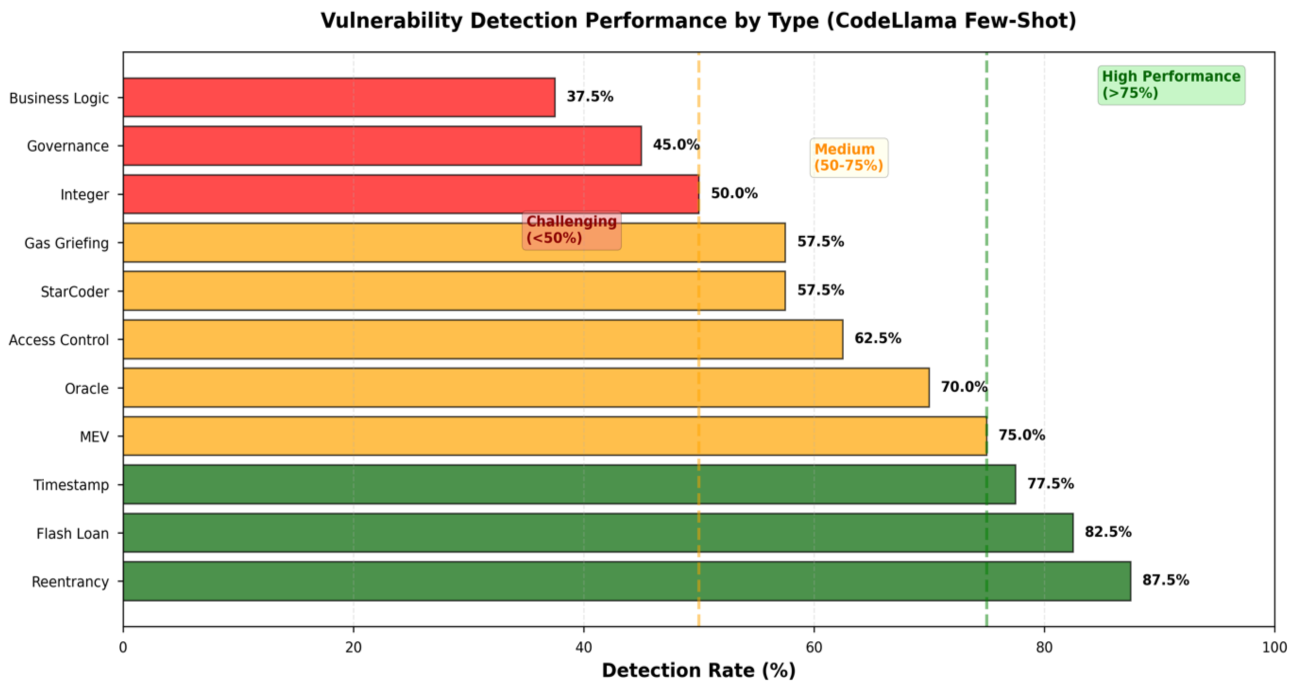

Fig. 3. Detection rate by vulnerability type (CodeLlama few-shot). Green bars (>75%): Reentrancy (87.5%), Flash Loan (82.5%), Timestamp (77.5%). Orange bars (50–75%): MEV (75%), Oracle (70%), Access Control (62.5%). Red bars (<50%): Business Logic (37.5%), Governance (45%), Integer (50%).

5. Evaluation

5.1. Key Insights

The Simplicity Paradox: Simple 200–400 character prompts achieve 100% success while elaborate 1500+ character prompts fail completely. This suggests current LLMs have limited capacity for processing complex structured instructions in security contexts.

Architectural Differences Matter: The dramatic performance gap between CodeLlama (66.67% recall) and CodeBERT (0% recall) demonstrates that architectural choices fundamentally determine security reasoning capabilities.

Few-Shot Learning Amplifies Existing Capabilities: Enhances models with baseline security understanding but cannot rescue architecturally limited models, suggesting security reasoning is an emergent capability requiring specific architectural properties.

These observed behaviours reinforce the broader principle that AI-assisted security workflows must be evaluated through a secure-by-design lens, as emphasized in recent AI governance guidelines [21].

5.2. Prompt Engineering Strategies Explanation of Long Prompt Failure

Context Window Saturation: CodeLlama's 4096-token context becomes saturated when complex prompts (1500+ chars) combine with contract code (500–1000 tokens), leaving insufficient capacity for reasoning.

Instruction Complexity Threshold: Complex prompts with multiple formatting requirements exceed the model's instruction-following capacity. Evidence shows sharp performance degradation:

- Simple prompts (200–400 chars): 100% success, 21.4 s
- Medium prompts (600–1000 chars): 85% success, 28.3 s
- Complex prompts (1500+ chars): 0% success, 120 s timeout

This suggests a fundamental architectural limitation, not merely a parameter tuning issue.

5.3. Prompt Engineering Strategies Production Readiness

False positive rate: 34.3% across all models

False Positive Categories:

- Over-sensitive reentrancy detection: 42% of FPs
- Access control false alarms: 31% of FPs
- MEV phantom vulnerabilities: 23% of FPs
- Oracle manipulation overreach: 18% of FPs

Assessment: High recall (66.67% for CodeLlama) makes it effective for initial screening, but 34.3% false positive rate necessitates expert review. Reduces manual review workload by ~40% while maintaining security coverage.

5.4.. Prompt Engineering Strategies Limitations

Internal Validity: Single deployment environment (local Ollama), low temperature settings (0.0-0.1), manual vulnerability labeling bias.

External Validity: Dataset may not capture full smart contract ecosystem diversity, evolving threat landscape, Solidity version specificity (focused on ^0.8.0).

Construct Validity: Ground truth subjectivity, metric selection may not capture all security tool effectiveness aspects.

These methodological constraints highlight the importance of embedding secure-by-design governance principles into LLM-based security tooling, consistent with industry recommendations [21].

6. Conclusion

This paper presents the first comprehensive empirical study of prompt engineering strategies for smart contract vulnerability detection using off-the-shelf LLMs. Through systematic evaluation of five model architectures across 21 vulnerability types and 15 experimental iterations, we establish key findings that advance understanding of LLM capabilities for security applications.

Our primary discovery reveals a strong inverse correlation between prompt complexity and success rate: simple prompts (200–400 characters) achieve 100% reliability while complex prompts (1500+ characters) fail completely. Multi-model comparison reveals dramatic architectural differences, with CodeBERT and InCoder showing 0% recall despite high accuracy, while CodeLlama achieves 66.67% recall with few-shot learning.

Future work should explore multi-model ensemble approaches, Chain-of-Thought reasoning with simple base prompts, vulnerability-specific prompt libraries, and integration with traditional static analysis tools and should also consider operational safeguards and secure-by-design deployment principles to ensure reliable integration of LLM-based detection systems.

Conflict of Interest

The author declares no conflict of interest.

Acknowledgment

The author thanks the open-source smart contract security community for making vulnerability data publicly available.

References

[1] Chainalysis. (2023). *The 2023 Crypto Crime Report*. Chainalysis Inc.

[2] Mueller, B. (2018). *Mythril: Security Analysis Tool for Ethereum Smart Contracts* [GitHub repository]. GitHub. Retrieved from https://github.com/ConsenSys/mythril

[3] Feist, J., Feist, F., Krupp, A., & Katz, D. (2019). Slither: A static analysis framework for smart contracts. *Proceedings of 2019 IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)* (pp. 8–15). IEEE.

[4] Tsankov, P., Dan, A., Drachsler-Cohen, D., Fisch, F., Gazzola, S., Bürgisser, E., & Vechev, M. (2018). Securify: Practical security analysis of smart contracts. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (pp. 67–82). ACM.

[5] Durieux, T., Naveh, E., & Feitelson, D. G. (2020). An empirical study on the effectiveness of static analysis tools for security vulnerability detection in Java and C code. *Empirical Software Engineering, 25(6)*, 4052–4087.

[6] Ding, H., Liu, Y., Piao, X., Song, H., & Ji, Z. (2025). SmartGuard: An LLM-enhanced framework for smart contract vulnerability detection. *Expert Systems with Applications, 266*, 126479. https://doi.org/10.1016/j.eswa.2025.126479

[7] Bu, J., Li, W., Li, Z., Zhang, Z., & Li, X. (2025). Enhancing smart contract vulnerability detection in DApps leveraging fine-tuned LLM. arXiv preprint, arXiv:2504.05006.

[8] OpenAI. (2023). Prompt engineering guide. Retrieved from https://platform.openai.com/docs/guides/prompt-engineering

[9] Maiti, A., Adewumi, S., Tikure, T. A., Wang, Z., Sengupta, N., Sukhanova, A., & Jana, A. (March, 2025). Comparative analysis of OpenAI GPT-4o and DeepSeek R1 for scientific text categorization using prompt engineering. Presented at the 2025 ASEE North Central Section Annual Conference, Huntington, WV, United States. https://doi.org/10.18260/1-2--54654

[10] Grieco, G., Braun, L., & Braun, D. (2020). Echidna: Effective, usable, and fast fuzzing for smart contracts. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 557–560). ACM.

[11] Wüstholz, M., & Christakis, C. (2020). Harvey: A greybox fuzzer for smart contracts. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 1398–1409). ACM.

[12] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., & Médard, M. (2016). Formal verification of smart contracts. *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security* (pp. 91–96). ACM.

[13] Kalra, S., Goel, S., Govindarajan, M., & Kannan, S. P. (2018). Zeus: Analyzing safety of smart contracts. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[14] Momeni, M., Gharaee, M., & Rezazadeh, M. (2019). Machine learning model for smart contracts security

analysis. *Proceedings of 2019 International Congress on Technology, Communication and Knowledge (ICTCK)* (pp. 1–6). IEEE.

[15] Qian, Z., Zhao, Y., & Li, X. (2020). Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access, 8*, 96685–96695.

[16] Zhuang, L., Liang, Z., & Cui, B. (2020). Smart contract vulnerability detection using graph neural network. *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence* (pp. 3283–3290).

[17] Wang, W., Zhong, Y., Jiang, H., & Zheng, F. (2021). ContractWard: Automated vulnerability detection models for Ethereum smart contracts. *IEEE Transactions on Network Science and Engineering, 8(2)*, 1133–1144.

[18] Feng, Z., Jiang, H., Huang, Z., & Zhang, X. (2020). CodeBERT: A pre-trained model for programming and natural languages. *Proceedings of Findings of the Association for Computational Linguistics: EMNLP 2020* (pp. 1536–1547). Association for Computational Linguistics.

[19] Pearce, H., Rodrigues, D., & Pereira, F. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. *Proceedings of 2022 IEEE Symposium on Security and Privacy (SP)* (pp. 754–768). IEEE.

[20] Microsoft. (September 30, 2025). *Secure AI by Design Series: Embedding Security and Governance Across the AI Lifecycle*. Microsoft Defender for Cloud Blog. Retrieved from https://techcommunity.microsoft.com/blog/microsoftdefendercloudblog/secure-ai-by-design-series-embedding-security-and-governance-across-the-ai-lifec/4457200

[21] Microsoft. (2023). Azure OpenAI Service documentation. Retrieved from https://learn.microsoft.com/azure/ai-services/openai/

Copyright © 2026 by the authors. This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited (CC BY 4.0).