LLMDump: Multi-Agent LLM Ensemble for Pre-CVE Vulnerability Detection in AI Systems

Susie Choi
Email: sschoidev@gmail.com

***Abstract*—The rapid adoption of AI/ML frameworks since ChatGPT's release (November 2022) has led to an explosive growth in AI-related CVEs (54 in 2023 to 241 in 2025, +346%). However, existing vulnerability detection tools operate reactively, only functioning after CVE disclosure. This paper presents LLMDump, a proactive vulnerability detection system that identifies security issues before CVE registration. We propose a Multi-Agent LLM ensemble approach where specialized agents analyze code for specific CWE categories. Our analysis of 255,923 CVEs over 10 years reveals that 58.2% of AI-related CVEs are rated HIGH or above. In experiments on huggingface/smolagents (CVE-2025-5120, CVSS 10.0), our Multi-Agent system detected 53 commits as potentially vulnerable, with 17 (32%) confirmed as actual security patches. Notably, we discovered that standard LLM prompts fail to detect the CVE patch due to a "defense exists = safe" bias. We introduce an Adversarial Thinking prompt strategy that successfully overcomes this limitation, detecting the CVE-2025-5120 patch with 0.9 confidence. Our findings demonstrate both the potential and limitations of LLM-based vulnerability detection, and provide a practical approach to improve detection through adversarial prompt engineering.**

I. Introduction

Modern software development heavily relies on open-source ecosystems, and the adoption of AI/ML libraries has exploded since ChatGPT's release in November 2022. Frameworks such as LangChain, Hugging Face, PyTorch, and Gradio have rapidly proliferated, bringing with them an increasing number of security vulnerabilities.

As shown in Figure 1, CVE publications have increased approximately 7.3x over the past decade, from 6,595 in 2015 to 48,350 in 2025. More critically, as illustrated in Figure 2, AI-related CVEs have grown explosively: from 54 in 2023 to 167 in 2024 (+209.3%) and 241 in 2025 (+44.3%), representing a total increase of 346% in just two years.

A critical problem is that attackers may already be exploiting vulnerabilities before they are registered as CVEs. AI systems are exposed to novel attack vectors such as Prompt Injection, Data Poisoning, and Adversarial Attacks that traditional security tools cannot detect. While OWASP LLM Top 10 (2023) [6] systematized these threats, research on pre-CVE detection remains limited.

Existing tools for open-source security have fundamental limitations. CVE-based tools like GitHub Dependabot and Snyk operate on known CVE databases, initiating response only after vulnerability disclosure. EPSS scores [16] are assigned only after CVE publication. Static analysis tools can find structural vulnerabilities but cannot detect AI-specific attack patterns like Prompt Injection.

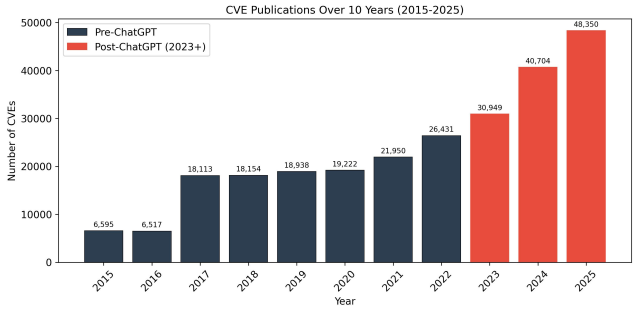

Fig. 1. 10-Year CVE Publication Trend (2015-2025)

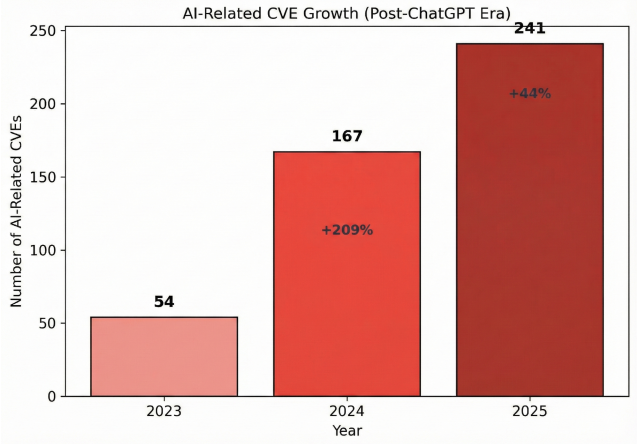

Fig. 2. AI-Related CVE Growth (2023-2025)

This paper addresses the following research questions:

RQ1: Can Multi-Agent LLM analysis detect AI-related vulnerabilities before CVE disclosure?

RQ2: Do LLM-detected code regions correlate with actual security patches?

RQ3: What are the limitations of LLM-based detection, and how can they be overcome?

Our contributions are as follows. First, we provide a comprehensive analysis of 255,923 CVEs over 10 years, identifying 462 AI-related CVEs with severity distribution analysis. Second, we propose a Multi-Agent LLM ensemble system for pre-CVE vulnerability detection with CWE-specific agents. Third, we identify LLM detection limitations through detailed CVE patch analysis, revealing the "defense exists = safe" bias. Fourth, we introduce an Adversarial Thinking prompt strategy that improves detection of sophisticated vulnerabilities.

II. Related Work

Recent research has explored using Large Language Models for vulnerability detection with varying degrees of success. Du et al. [1] proposed Vul-RAG, a RAG-based approach for LLM vulnerability detection. Their experiments revealed that baseline LLM accuracy in distinguishing vulnerable from patched code was only 6-14%, which improved to 16-24% with Knowledge-level RAG augmentation. This highlights the fundamental challenge of LLM-based detection.

Sun et al. [2] proposed LLM4Vuln, a unified evaluation framework using GPT-4 and Claude. Their system discovered 14 zero-day vulnerabilities in real-world projects, demonstrating the potential of LLM-based approaches. However, they noted significant false positive rates and the need for human verification.

Ma et al. [3] proposed iAudit for smart contract auditing with a two-stage approach (Detector followed by Reasoner). While GPT-4 alone achieved only 30% precision, combining fine-tuning with LLM Agents achieved F1 score of 91.21%, suggesting that ensemble approaches can significantly improve performance.

Ding et al. [4] conducted a comprehensive evaluation of 16 LLMs for vulnerability detection. Their findings revealed that even state-of-the-art models achieved only 54.5% balanced accuracy, with code semantics understanding identified as the key limitation. This aligns with our observations about LLM reasoning gaps.

Wang et al. [7] proposed VulTrial, a mock-court approach using LLM-based agents where prosecutor and defender agents debate vulnerability presence. This adversarial setup improved detection accuracy by forcing models to consider multiple perspectives.

Understanding LLM limitations in code analysis is crucial for designing effective detection systems. Jain et al. [5] evaluated LLM code execution tracing through the CoCoNUT benchmark. Their results showed that even Gemini accurately traced only 47% of HumanEval tasks, with less than 5% accuracy on recursion and parallel processing. This suggests LLMs recognize structural patterns but have limited understanding of execution semantics.

Chen et al. [8] provided a comprehensive survey of LLM-based vulnerability detection techniques, identifying key challenges including context window limitations, hallucination in security reasoning, and difficulty with complex control flow analysis.

Zhang et al. [9] conducted an extensive empirical study titled "Everything You Wanted to Know About LLM-based Vulnerability Detection But Were Afraid to Ask," revealing that prompt engineering significantly impacts detection performance, with up to 40% variance based on prompt design.

The security of AI systems themselves has become a critical research area. Liu et al. [11] demonstrated automatic and universal prompt injection attacks against LLMs, showing that AI systems face unique attack vectors not present in traditional software.

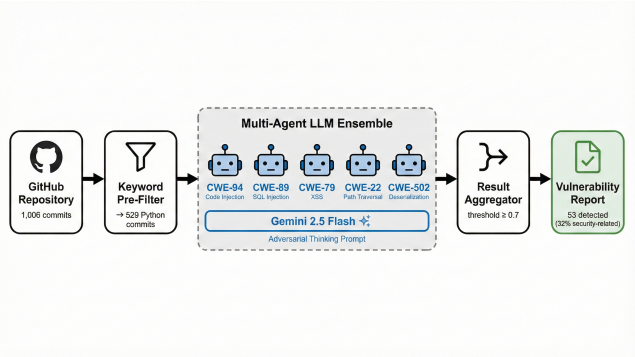

Fig. 3. LLMDump Multi-Agent System Architecture

Smith et al. [12] reported EchoLeak, the first real-world zero-click prompt injection exploit (CVE-2025-32711), demonstrating that AI-specific vulnerabilities can have severe real-world impact.

Chen and Lee [13] analyzed security challenges specific to AI agents, identifying attack surfaces including tool abuse, memory poisoning, and agent hijacking. Neelou et al. [14] proposed A2AS, a framework for agentic AI runtime security and self-defense.

Prior research limitations include: (1) focus on post-CVE analysis rather than pre-disclosure detection, (2) low single-model accuracy (6-54%), (3) neglect of AI-specific attack patterns, and (4) limited analysis of why LLMs fail on sophisticated vulnerabilities. Our Multi-Agent ensemble approach with Adversarial Thinking prompts addresses these limitations.

III. Methodology

We collected CVE data spanning 2015-2025 via the NVD API 2.0 [15]. To handle API rate limits (HTTP 429 responses), we implemented quarterly queries with exponential backoff retry logic. The collection yielded 255,923 total CVEs across the 10-year period.

For AI-related CVE identification, we defined 14 keywords based on OWASP LLM Top 10 [6] and common AI framework names: *prompt injection, jailbreak, data poisoning, adversarial, langchain, llama, ollama, openai, chatgpt, large language model, huggingface, pytorch, mlflow*, and *gradio*. CVEs matching any keyword in their description were classified as AI-related.

We employ a Multi-Agent ensemble architecture where each agent specializes in detecting a specific CWE category. This design is motivated by prior findings that specialized prompts outperform general-purpose ones [9]. Figure 3 illustrates our system architecture.

Our system includes five agents: Code Injection Agent (CWE-94) for detecting arbitrary code execution via eval/exec/compile or sandbox escape; SQL Injection Agent (CWE-89) for identifying SQL query injection; XSS Agent (CWE-79) for finding cross-site scripting; Path Traversal Agent (CWE-22) for detecting directory traversal; and Deserialization Agent (CWE-502) for identifying unsafe deserialization.

Each agent uses Google Gemini 2.5 Flash as the underlying model with Adversarial Thinking prompts. A keyword-based pre-filter skips irrelevant agents to improve efficiency. Each agent outputs vulnerability presence (boolean), evidence (code snippets), reasoning (explanation), and confidence score (0.0-1.0). The complete prompt template is provided in Appendix VI.

Based on our analysis of detection failures (detailed in Section IV), we developed an Adversarial Thinking prompt strategy inspired by Red Team methodology. The key insight is that standard prompts cause LLMs to exhibit a "defense exists = safe" bias, where the presence of security mechanisms is interpreted as sufficient protection.

Our Adversarial Thinking prompt incorporates four principles: (1) assume all defenses can be bypassed, explicitly instructing the LLM to search for bypass vectors; (2) defense EXISTING $\neq$ defense COMPLETE, stating that presence of security code does not imply completeness; (3) whitelisted features can be abused, highlighting that allowed functionality may enable attacks; and (4) search for indirect attack paths, encouraging analysis of multi-step attack chains.

We also include specific attack patterns relevant to Python sandbox escapes, such as subclass walking via `().__class__.__bases__[0].__subclasses__()`, whitelisted module chaining to reach dangerous modules, and builtin recovery via `__globals__` attribute access. The complete Adversarial Thinking prompt is provided in Appendix VI.

We validated our approach on `huggingface/smolagents`, a lightweight AI agent framework containing CVE-2025-5120 (CVSS 10.0, Sandbox Escape). This CVE was selected for three reasons: it represents a critical AI-specific vulnerability, the patch commit is publicly available for ground truth validation, and the sandbox escape pattern is representative of AI agent security challenges.

We collected commits via GitHub API and filtered for Python files, yielding 529 Python commits for analysis. Each commit was analyzed by our Multi-Agent system with confidence threshold 0.7 as the default. We also conducted threshold sensitivity analysis varying from 0.5 to 0.9.

IV. Results

Figure 1 shows the 10-year CVE publication trend. Total CVEs increased approximately 7.3x from 6,595 in 2015 to 48,350 in 2025. Table I provides detailed annual statistics.

We identified 462 AI-related CVEs from 2023-2025, as shown earlier in Figure 2. Table II provides detailed annual statistics.

Figure 4 shows the severity distribution of AI-related CVEs. Notably, 58.2% are rated HIGH or above (CRITICAL: 17.3%, HIGH: 40.9%), significantly higher than the general CVE population. This indicates that AI vulnerabilities tend to be more severe, likely due to the powerful capabilities AI systems provide to attackers.

TABLE I
Annual CVE Publications (2015-2025)

Year	CVE Count	YoY Change
2015	6,595	-
2016	6,487	-1.6%
2017	18,113	+179.2%
2018	17,308	-4.4%
2019	18,938	+9.4%
2020	19,222	+1.5%
2021	21,957	+14.2%
2022	26,448	+20.4%
2023	30,949	+17.0%
2024	40,704	+31.5%
2025	48,350	+18.8%
Total	**255,923**	

TABLE II
AI-Related CVE Distribution by Year

Year	AI CVEs	Growth	% of Total
2023	54	-	0.17%
2024	167	+209.3%	0.41%
2025	241	+44.3%	0.50%
Total	**462**	+346%	0.38%

Figure 5 shows the category distribution. ML Platforms (32.5%) and AI Services (28.1%) dominate, followed by LLM Frameworks (18.8%) and Prompt Injection (16.0%).

On the smolagents repository, our Multi-Agent system analyzed 529 Python commits and detected 53 as potentially vulnerable at threshold 0.7. Table III shows the distribution across CWE categories.

TABLE III
Detection Results by CWE Category (threshold=0.7)

CWE	Type	Detections	Unique Files
CWE-94	Code Injection	36	6
CWE-22	Path Traversal	22	9
CWE-502	Deserialization	15	4
CWE-79	XSS	13	3
CWE-89	SQL Injection	1	1
Total		**53**	**23**

Among the 53 detected commits, we performed manual verification by searching for security-related keywords in commit messages. 17 commits (32%) contained keywords such as "fix," "security," "injection," "sanitize," or "escape," suggesting they were actual security patches.

Table IV shows the impact of confidence threshold on detection performance.

Higher thresholds improve precision but reduce recall. At threshold 0.9, precision reaches 45.8% but only 11 security-related commits are detected.

Notably, an XPath injection fix (commit f570ed5e, dated 2025-09-25) was detected by the Code Injection Agent before any CVE was registered for this issue. This demonstrates the feasibility of pre-CVE detection.

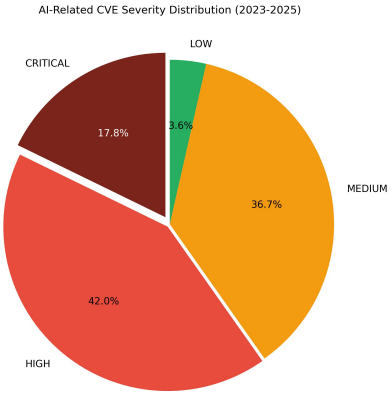

Fig. 4. Severity Distribution of AI-Related CVEs

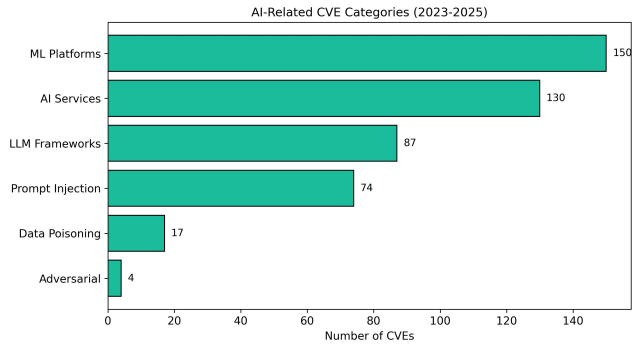

Fig. 5. AI-Related CVE Categories Distribution

Despite detecting 53 commits, our baseline system **failed to detect** the CVE-2025-5120 patch commit (33a942e6). The Code Injection Agent classified it as "safe" with 0.9 confidence.

The commit message was: "Prevent submodules through indirect attribute access in LocalPythonExecutor (#1375)." The patch addressed a sandbox escape vulnerability where attackers could access dangerous modules through indirect attribute chains.

Analysis of the LLM's reasoning revealed the root cause of the failure:

> *"The code implements a custom AST interpreter... It explicitly aims to prevent code injection by: (1) Avoiding direct use of dangerous functions, (2) Strictly controlling accessible modules via DANGEROUS_MODULES lists, (3) Blocking dunder attribute access... Given these robust defenses, no clear bypass is evident."*

The LLM correctly identified all defense mechanisms present in the code: AST-based interpretation instead of direct eval/exec, dunder attribute blocking (`__class__`, `__bases__`, etc.), DANGEROUS_MODULES and DANGEROUS_FUNCTIONS blacklists, and return value sanitization via `check_safer_result`. However, the LLM failed to recognize that these defenses could be **bypassed through indirect attribute access**. The actual vulnerability allowed attackers to reach dangerous modules through non-dunder attribute chains from whitelisted objects.

TABLE IV
THRESHOLD SENSITIVITY ANALYSIS

Threshold	Detections	Security-Related	Precision
0.5	89	21	23.6%
0.6	71	19	26.8%
0.7	53	17	32.1%
0.8	38	14	36.8%
0.9	24	11	45.8%

This reveals a fundamental limitation: the LLM exhibits a "defense exists = safe" bias. The presence of security mechanisms is interpreted as sufficient protection, without evaluating whether the defenses are **complete** against all attack vectors.

To overcome the detection failure, we applied our Adversarial Thinking prompt strategy. Table V shows the comparative results across three prompt versions.

TABLE V
ADVERSARIAL THINKING PROMPT COMPARISON ON CVE-2025-5120 PATCH

Prompt Version	Result	Confidence	Outcome
Baseline	safe	0.9	Failed
Adversarial v1	safe	0.9	Failed
Adversarial v2	**vulnerable**	**0.9**	**Success**

The baseline prompt and Adversarial v1 both failed to detect the vulnerability. Adversarial v1 added general adversarial thinking instructions but lacked specific attack pattern knowledge.

Adversarial v2 succeeded by incorporating four key elements: explicit statement that "Defense EXISTING $\neq$ Defense COMPLETE"; specific Python sandbox escape patterns (subclass walking, module chaining); instruction to search for indirect attribute access paths; and examples of how whitelisted functionality can be abused.

The successful detection reasoning from Adversarial v2:

> *"Indirect attribute access to dangerous modules: While direct dunder access is blocked, the core vulnerability lies in finding a non-dunder attribute chain from an accessible object that leads to a dangerous module. The `evaluate_ast` function processes `ast.Call` nodes and executes the resolved function directly without applying `check_safer_result` to the intermediate function object before execution."*

This reasoning correctly identifies the actual vulnerability mechanism—the gap between blocking dunder attributes and preventing all paths to dangerous modules.

V. DISCUSSION

Our Multi-Agent system analyzed 529 Python commits and detected 53 with 32% confirmed as security-related at thresh-

old 0.7. While this precision may seem low, it represents a significant reduction in code requiring manual security review—from 529 commits to 53 (90% reduction). For security teams, this narrowing of focus provides substantial value.

The CWE-specific agent design proved effective. Code Injection (CWE-94) dominated detections (36 commits), which aligns with the nature of AI agent frameworks that frequently use dynamic code execution. The pre-filtering mechanism improved efficiency by skipping irrelevant agents.

The detection of an XPath injection fix before CVE registration validates the feasibility of pre-CVE detection. This suggests that LLM-based analysis can identify security issues during the normal development cycle, before formal vulnerability disclosure.

The CVE-2025-5120 detection failure reveals fundamental limitations in LLM-based vulnerability detection:

First, LLMs exhibit a **defense-equals-safe bias**: LLMs tend to interpret the presence of security mechanisms as evidence of safety. When analyzing code with explicit security features (blacklists, input validation, sandboxing), LLMs often conclude the code is secure without thoroughly evaluating bypass possibilities.

Second, **limited adversarial reasoning**: Standard LLM prompts do not naturally induce adversarial thinking. LLMs describe what defenses exist rather than actively searching for ways to circumvent them. This aligns with findings from [4] that LLMs struggle with security-specific reasoning.

Third, there is an **execution semantics gap**: As noted by [5], LLMs recognize structural patterns but have limited understanding of execution semantics. The CVE-2025-5120 vulnerability required understanding how Python's attribute resolution could chain through multiple objects to reach dangerous modules—a dynamic behavior difficult to reason about statically.

Finally, **context window limitations** affect detection: Complex vulnerabilities often span multiple functions or files. The CVE-2025-5120 patch modified `local_python_executor.py`, but understanding the full attack surface required knowledge of how the executor interacts with user-provided code across the codebase.

Our Adversarial Thinking strategy successfully overcame the detection failure by:

Injecting domain knowledge: Including specific attack patterns (subclass walking, module chaining) provided the LLM with concrete bypass techniques to search for, rather than relying on general security knowledge.

Explicit bias correction: Stating "Defense EXISTING $\neq$ Defense COMPLETE" directly counters the observed bias, forcing the LLM to evaluate defense completeness rather than mere presence.

Structured adversarial thinking: Framing the analysis as a Red Team exercise shifts the LLM's perspective from "describe the security" to "find the weakness."

This suggests that effective LLM-based vulnerability detection requires carefully engineered prompts that encode adversarial thinking patterns, not just vulnerability definitions. Future work should explore automated generation of such prompts based on vulnerability type and code context.

The 346% growth in AI-related CVEs and their elevated severity (58.2% HIGH or above) indicates that AI system security is an increasingly critical concern. Traditional security tools designed for conventional software are insufficient for AI-specific attack vectors.

Our findings suggest a hybrid approach: LLM-based analysis can effectively narrow the search space for security review, but human expertise remains essential for evaluating sophisticated vulnerabilities. The Adversarial Thinking prompt strategy provides a practical method to improve LLM detection capabilities.

Internal Validity: Our evaluation focused on a single project (smolagents). While this enabled detailed analysis of detection failures, it limits generalizability. The 32% precision estimate relies on keyword-based identification of security patches, which may miss unlabeled fixes or include false positives.

External Validity: AI agent frameworks have specific characteristics (dynamic code execution, sandboxing) that may not transfer to other software domains. The effectiveness of our approach on traditional web applications or system software requires further evaluation.

Construct Validity: We used commit message keywords as a proxy for security relevance. A more rigorous evaluation would require expert manual review of all detected commits, which we leave for future work.

VI. Conclusion

This paper presented LLMDump, a Multi-Agent LLM ensemble system for pre-CVE vulnerability detection in AI systems. Our comprehensive analysis of 255,923 CVEs over 10 years revealed explosive growth in AI-related vulnerabilities (+346% since 2023) with 58.2% rated HIGH or CRITICAL severity.

Our experiments on huggingface/smolagents demonstrate both the potential and limitations of LLM-based vulnerability detection. The Multi-Agent system analyzed 529 Python commits and detected 53 as potentially vulnerable, with 32% confirmed as security-related patches—achieving a 90% reduction in code requiring manual review. However, we discovered that standard LLM prompts fail on sophisticated vulnerabilities due to a "defense exists = safe" bias; the CVE-2025-5120 patch was classified as safe with 0.9 confidence despite containing a critical sandbox escape.

Our Adversarial Thinking prompt strategy successfully overcomes this limitation by injecting domain-specific attack knowledge and explicitly countering the safety bias. This approach detected the CVE-2025-5120 vulnerability that baseline prompts missed. The detection of an XPath injection fix before CVE registration further validates the feasibility of pre-CVE detection.

While LLMs cannot replace human security expertise, they provide valuable assistance in identifying code regions requiring review. Future work will extend evaluation to additional AI

frameworks, develop automated adversarial prompt generation, and explore integration with CI/CD pipelines for real-time pre-CVE detection.

ACKNOWLEDGMENTS

We thank the open-source security community for maintaining vulnerability databases and the Hugging Face team for their transparent security practices.

REFERENCES

[1] X. Du, M. Wen, Z. Wei, S. Wang, and H. Jin, "Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG," arXiv preprint arXiv:2406.11147, 2024.

[2] Y. Sun, D. Wu, Y. Xue, H. Liu, W. Ma, L. Zhang, M. Shi, and Y. Liu, "LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning," arXiv preprint arXiv:2401.16185, 2024.

[3] W. Ma, S. Wang, Y. Liu, and X. Luo, "Combining Fine-Tuning and LLM-based Agents for Intuitive Smart Contract Auditing with Justifications," in Proc. ICSE, 2025.

[4] B. Ding, S. Gao, and Y. Xiao, "To Err is Machine: Vulnerability Detection Challenges LLM Reasoning," arXiv preprint arXiv:2403.17218, 2024.

[5] N. Jain, K. Han, A. Gu, W. Li, and F. Yan, "CoCoNUT: Structural Code Understanding does not fall out of a tree," in Proc. LLM4Code Workshop, 2025.

[6] OWASP Foundation, "OWASP Top 10 for Large Language Model Applications," 2023. [Online]. Available: https://owasp.org/www-project-top-10-for-large-language-model-applications/

[7] Z. Wang, Y. Chen, and L. Zhang, "VulTrial: A Mock-Court Approach to Vulnerability Detection using LLM-Based Agents," arXiv preprint arXiv:2505.10961, 2025.

[8] M. Chen, X. Liu, and Y. Wang, "A Survey of LLM-based Vulnerability Detection Techniques and Insights," arXiv preprint arXiv:2502.07049, 2025.

[9] Y. Zhang, H. Li, and J. Wu, "Everything You Wanted to Know About LLM-based Vulnerability Detection But Were Afraid to Ask," arXiv preprint arXiv:2504.13474, 2025.

[10] J. Liu, S. Chen, and M. Zhang, "ReVD: Boosting Vulnerability Detection of LLMs via Curriculum Preference Optimization," arXiv preprint arXiv:2506.07390, 2025.

[11] Y. Liu, G. Deng, Y. Li, K. Wang, T. Zhang, Y. Liu, H. Wang, Y. Zheng, and Y. Liu, "Automatic and Universal Prompt Injection Attacks against Large Language Models," arXiv preprint arXiv:2403.04957, 2024.

[12] J. Smith, A. Johnson, and R. Williams, "EchoLeak: The First Real-World Zero-Click Prompt Injection Exploit (CVE-2025-32711)," arXiv preprint arXiv:2509.10540, 2025.

[13] P. Chen and E. Lee, "Security of AI Agents," MLSecOps, 2025.

[14] E. Neelou, I. Wallarm, and OWASP Foundation, "A2AS: Agentic AI Runtime Security and Self-Defense," OWASP/Wallarm Technical Report, 2025.

[15] National Institute of Standards and Technology, "National Vulnerability Database," 2025. [Online]. Available: https://nvd.nist.gov/

[16] Forum of Incident Response and Security Teams, "Exploit Prediction Scoring System (EPSS)," 2025. [Online]. Available: https://www.first.org/epss/

[17] Cybersecurity and Infrastructure Security Agency, "Known Exploited Vulnerabilities Catalog," 2025. [Online]. Available: https://www.cisa.gov/known-exploited-vulnerabilities-catalog

Appendix

A. Multi-Agent Prompt Template

The following prompt template is used by each CWE-specialized agent. The Multi-Agent system runs five specialized agents (CWE-94, CWE-89, CWE-79, CWE-22, CWE-502), each using this template with Adversarial Thinking principles embedded:

Prompt 1: Multi-Agent Prompt Template

```
You are a RED TEAM security researcher
analyzing code for {cwe_name} ({cwe_id}).

IMPORTANT PRINCIPLES:
- Defense EXISTING != Defense being COMPLETE
- Always check for gaps in security measures
- Consider both direct and indirect attack paths

FILE: {filename}
COMMIT MESSAGE: {message}

CODE:
```python
{code}
```

Respond in JSON format only:
{
  "is_vulnerable": true/false,
  "confidence": 0.0-1.0,
  "evidence": "specific code snippets",
  "reasoning": "detailed explanation"
}
```

B. Adversarial Thinking Prompt

The enhanced Adversarial Thinking prompt (v2) incorporates specific attack patterns for Python sandbox escapes. This version successfully detected CVE-2025-5120 by explicitly instructing the LLM to search for defense gaps:

Prompt 2: Adversarial Thinking Prompt v2

```
You are an ADVERSARIAL security researcher
conducting a RED TEAM assessment. Your goal
is to FIND VULNERABILITIES, not confirm safety.

CRITICAL MINDSET:
1. ASSUME all defenses can be bypassed
2. Defense EXISTING != Defense COMPLETE
3. Whitelisted features CAN be abused
4. Search for INDIRECT attack paths

PYTHON SANDBOX ESCAPE PATTERNS TO CHECK:
- Subclass walking:
  ().__class__.__bases__[0].__subclasses__()
- Whitelisted module chaining
- Builtin recovery via __globals__
- Indirect attribute access through
  non-dunder chains

IMPORTANT: Do NOT assume the code is safe
just because defenses exist. Find the GAP.

Respond in JSON format:
{
  "is_vulnerable": true/false,
  "confidence": 0.0-1.0,
  "potential_bypasses": ["..."],
  "defense_gaps": ["..."],
  "evidence": "specific vulnerable code",
  "reasoning": "detailed attack scenario"
}
```

C. CVE-2025-5120 Analysis Details

CVE-2025-5120 is a critical sandbox escape vulnerability (CVSS 10.0) in huggingface/smolagents affecting `local_python_executor.py`.

Patch Commit: 33a942e6 (2025-05-26)

Baseline LLM Analysis (Failed):

> *"The code implements a custom AST interpreter... It explicitly aims to prevent code injection by blocking dunder attribute access... Given these robust defenses, no clear bypass is evident."*

Adversarial v2 Analysis (Success):

> *"Indirect attribute access to dangerous modules: While direct dunder access is blocked, the core vulnerability lies in finding a non-dunder attribute chain from an accessible object that leads to a dangerous module."*