AUDITGPT: A MULTI-AGENT FRAMEWORK FOR ENHANCING STATIC ANALYSIS

Junze Hu[1,2], *Yizhe Zeng*[1,2], *Guoli Zhao*[1,2], *Kaiyu Xie*[1,2], *Yimo Ren*[1], *Dahui Li*[1]✉, *Hongsong Zhu*[1,2]✉

[1] Institute of Information Engineering, Chinese Academy of Sciences, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, China
{hujunze, zengyizhe, zhaoguoli, xiekaiyu, renyimo, lidahui, zhuhongsong}@iie.ac.cn

ABSTRACT

Traditional manual auditing of large-scale software has become impractical. While static analysis tools like CodeQL offer scale, they are constrained by predefined rules, leading to significant false positives and negatives. Similarly, emerging Large Language Models (LLMs)-based approaches are impeded by context window limitations, restricting the holistic, multi-file analysis essential for complex vulnerability detection. To address these challenges, we introduce AUDITGPT, a multi-agent framework that orchestrates collaboration between LLMs and static analysis for comprehensive vulnerability detection. In 10 well-known open-source projects from GitHub, the tool identifies 2 previously unknown vulnerabilities. Meanwhile, our approach enhances CodeQL's precision, reducing its false positive rate by 56.3% at least while achieving a 57.4% detection coverage validated by Proofs-of-Concept (PoC).

Index Terms— static analysis, large language models, proofs-of-concept, multi-agent

1. INTRODUCTION

The prevalence of security vulnerabilities in Java applications presents a significant challenge. The attack surface is broad, with studies indicating that at least 42% of Java projects contain one or more security flaws [1]. A recent report from Datadog [2] found that a critical 44% of production Java services harbor at least one known-exploited vulnerability, posing a direct and immediate threat. The sheer scale of this problem renders manual auditing impractical and makes automated detection an absolute necessity.

One prominent line of research has leveraged deep learning models [3, 4, 5]. While these methods show promise at the code snippet level, they struggle to scale to the complex inter-procedural dependencies and contextual nuances required for project-level analysis, limiting their practical applicability in real-world audits.

In practice, the predominant approach remains Static Application Security Testing (SAST), with leading tools like CodeQL [6, 7], Checker Framework [8], Snyk Code [9], and SonarQube [10] relying on static taint analysis. However, this methodology is prone to generating a high volume of both false positives (FPs) [11], which waste significant developer effort, and false negatives (FNs), which introduce insidious security risks. These inaccuracies stem from fundamental limitations inherent in static analysis. To remain tractable, the analysis must over-approximate complex control flows (branches, loops, arrays) [1] and often relies on models that fail to recognize custom sanitizers [12]. More critically, even a technically valid data-flow path may lack an exploitable context [13], leading to a deluge of FPs. Conversely, tools often create FNs by hastily assuming that any sanitizer guarantees security, when in reality it may be flawed [12]. This deep-rooted tension, whereby attempts to reduce FNs by expanding definitions, as in IRIS [13], inevitably surges FPs—underscores the critical need for a more precise, context-aware paradigm.

LLMs have recently emerged as a promising tool in cybersecurity, demonstrating potential across a wide range of tasks [14]. Consequently, researchers have begun applying them to vulnerability detection [15, 16, 17]. However, recent studies demonstrated that LLMs show limited success in identifying security flaws in complex, real-world codebases [18, 19]. Severe methodological flaws further hamper attempts to build upon this shaky foundation for automated PoC generation. For instance, an approach relies on feeding the LLM isolated call chains from a Code Property Graph (CPG) [1]. Because this method strips away project-level context, generating effective PoC for vulnerabilities involving complex data flows is difficult.

To address these limitations, we propose AUDITGPT, a novel framework where vulnerability detection and validation are performed by a cascade of specialized LLMs-powered agents. The process is initiated by a Sink Identification Agent, which comprehensively identifies potential sinks by embedding project knowledge. Subsequently, after performing a backward path reconstruction from each sink, a dedicated Path Pruning Agent analyzes the resulting candidate data flows, filtering out traces originating from uncontrollable sources. Finally, the refined candidates are passed to a Verification multi-agent System. Guided by our novel Audit Task Graph (ATG) and aided by a tool library, these verification agents conduct deep contextual analysis, culminating in automatically generating a verifiable PoC.

Contributions. The contributions are outlined as follows:

- We design a method that integrates CodeQL with Agent to systematically extract all candidate vulnerable call chains from an entire Java project.
- We introduce the ATG, a novel reasoning substrate that models and orchestrates the complex workflow of vulnerability validation. The ATG is uniquely characterized by its stateful, dynamic structure and a failure-driven refinement mechanism, which allows our multi-agent system to learn from failed attempts, accumulate evidence, and adaptively converge on a valid exploit strategy.
- We design a multi-agent framework where agents use Retrieval-Augmented Generation (RAG) [20]. This capability lets them search the project's entire source code. As a result, they can look beyond abstract call chains and provide deep, contextual analysis.

✉: Corresponding author

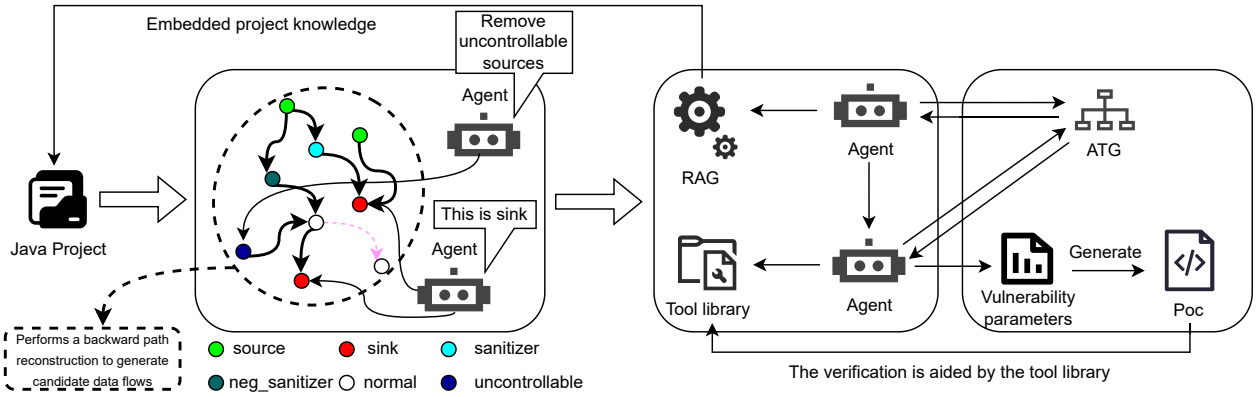

Fig. 1: Overview of AUDITGPT. The system performs static analysis on a Java project to identify candidate data flows via backward path reconstruction (dashed circle), discarding infeasible paths (pink dashed line). A multi-agent framework, guided by the ATG, then leverages RAG and a tool library to verify these candidates and generate a final PoC.

2. DESIGN AND IMPLEMENTATION

The architecture of AUDITGPT, illustrated in Fig. 1, is a multi-stage pipeline that systematically identifies and validates vulnerabilities by emulating an expert's reasoning process. The pipeline executes three primary phases: (1) high-precision sink identification, (2) backward path reconstruction and pruning, and (3) multi-agent validation via an ATG.

Initially, the pipeline's sink identification phase employs an agent to locate critical endpoints semantically across all project methods. This is followed by a backward path reconstruction from each sink to generate candidate data-flow paths, which are immediately pruned based on source controllability. The final and core phase is the multi-agent validation. Here, a team of specialized agents collaborates by operating on a dynamic ATG to perform deep contextual analysis, false positive adjudication, and PoC generation for confirmed findings.

2.1. Extract Candidate Vulnerable Call Chains

Let $G = (\mathcal{N}, \mathcal{E})$ be the Data Flow Graph (DFG) derived from taint analysis. Our methodology first extracts a broad set of Initial Candidate Paths ($\mathcal{P}_{cand}$), which are then formally classified into two mutually exclusive subsets: Vulnerable Paths ($\mathcal{P}_{vuln}$) and Benign Paths ($\mathcal{P}_{benign}$).

An Initial Candidate Path ($\mathcal{P}_{cand}$) is defined as any data-flow path from a potential source node to an identified sink node:

$$\mathcal{P}_{cand} \triangleq= \mathcal{N}_{source} \rightarrow \cdots \rightarrow \mathcal{N}_{sink} \tag{1}$$

From this initial set, a candidate path is classified as a Vulnerable Path ($\mathcal{P} \in \mathcal{P}_{vuln}$) if and only if it satisfies both of the following conditions:

$$(\mathcal{N}^*_{path(\mathcal{P})} \cap \mathcal{N}_{sanitizer}) = \emptyset \tag{2}$$

$$(\mathcal{N}^*_{path(\mathcal{P})} \cap \mathcal{N}_{neg\_sanitizer}) \neq \emptyset \tag{3}$$

Conversely, a candidate path is classified as a Benign Path ($\mathcal{P} \in \mathcal{P}_{benign}$) if it satisfies the following logical condition:

$$(\mathcal{N}_{source} \in \mathcal{N}_{uncontrollable}) \vee ((\mathcal{N}^*_{path} \cap \mathcal{N}_{sanitizer}) \neq \emptyset) \tag{4}$$

The sets used in these definitions are categorized as follows:

- $\mathcal{N}^*_{path}$: Represents the set of intermediate nodes in a given path $\mathcal{P}$.
- $\mathcal{N}_{sanitizer}$: Represents the set of effective sanitization routines.
- $\mathcal{N}_{neg\_sanitizer}$: Represents the set of known bypassable or flawed sanitizers.
- $\mathcal{N}_{uncontrollable}$: Represents the set of non-user-controllable source nodes (entry points).

Our methodology begins with a foundational phase of high-precision sink identification. We leverage CodeQL [6] to enumerate all methods within a project (encompassing both custom and third-party functions) and decompose each into its signature and body. This structured representation is then fed into an agent for semantic classification as a potential sink. Based on these identified sinks, we then leverage CodeQL to perform a backward path reconstruction, extracting all data-flow paths that terminate at these critical nodes. To prune the resulting candidates, an agent conducts a final analysis on each path's source node, discarding any chain that originates from a non-user-controllable entry point ($\mathcal{N}_{uncontrollable}$).

2.2. Audit Task Graph

To orchestrate the complex, non-linear workflow of vulnerability validation, we introduce the Audit Task Graph (ATG), as shown in Fig. 2, a stateful, directed graph of interdependent sub-tasks. A key design principle of the ATG is its hierarchical structure; the graph originates from a high-level root task (Validate Call Chain), which is progressively decomposed into more granular sub-tasks for analysis and evidence gathering. This hierarchical decomposition allows for a structured and methodical exploration of the solution space. Unlike a simple tree, the ATG topology supports multi-dependency convergence and incorporates feedback loops, which are essential for its core innovation: a failure-driven refinement mechanism.

This dynamic and hierarchical topology is explicitly designed to overcome the limitations of linear reasoning in Chain-of-Thought (CoT) [21] and the static, pre-defined plans of Hierarchical Task Networks (HTN) [22]. Furthermore, the ATG advances on general-purpose models like Tree/Graph of Thoughts (ToT/GoT) [23, 24] in two critical ways. First, whereas ToT/GoT nodes are generic blocks of text, ATG utilizes structured data fields specifically tailored for security analysis. This allows it to precisely record key information, such as a specific vulnerability hypothesis, the payload used to test it, and the resulting evidence. Second, the aforementioned refinement loop systematically learns from empirical, external feedback (a failed PoC), a more robust mechanism than the intrinsic self-

evaluation and simple path pruning common in ToT.

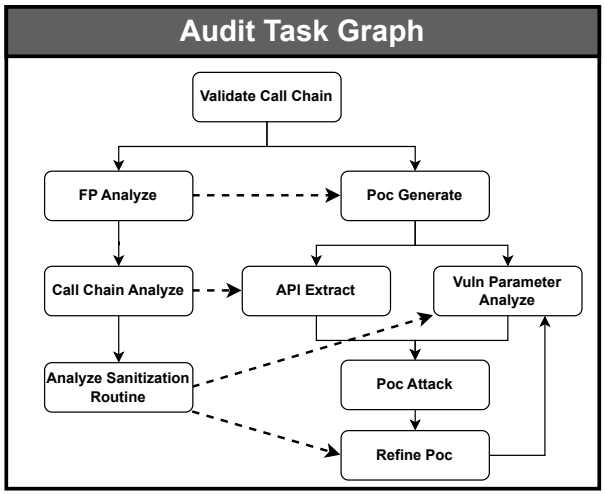

Fig. 2: ATG Concept

2.3. Architecture of the multi-agent Framework

Our multi-agent framework is architected around a central state-management structure—the ATG.

According to Algorithm 1, the process is initiated by a Planner agent, which performs an initial triage on the path of the received candidate. It establishes a strategic validation objective for the path, determining whether the end-goal is simple False Positive Adjudication or comprehensive End-to-End Validation, culminating in PoC generation. This mirrors an expert's strategic decision on how deeply to investigate a potential finding. The core of our framework's intelligence resides in the Analysis agent. It utilizes a RAG capability over the project's entire source code when tasked with deep code comprehension (`Analyze Sanitization Routine`). This allows it to move beyond the call chain's abstractions to reason about fine-grained implementation details. The agent's final output is a classification of whether the path is a false positive, which is recorded in the ATG. If a path is confirmed as benign, its branch in the tree is terminated. Finally, for high-confidence candidates slated for full validation, the Exploitation agent is activated. It begins by performing route extraction and parameter instantiation from the source node. Equipped with an Abstract Syntax Tree (AST) analysis tool, the agent then performs two critical sub-tasks: (1) it precisely tracks the data flow of the tainted parameter to the sink, and (2) it resolves path constraints by assigning appropriate values to non-tainted parameters to ensure execution reaches the correct program branch. The outcome is a structured JSON object detailing the exploitation requirements, which is then used to synthesize a final `curl` command. For blind vulnerabilities (RCE, SSRF), the framework leverages an external DNS-based utility for out-of-band verification. Critically, our framework features a feedback loop: a failed attempt is not an endpoint. The agent analyzes the failure response, updates the ATG with this new knowledge, and re-engages the `Refine Payload` sub-task. This iterative refinement, guided by the evolving state of the ATG, is key to the framework's high success rate.

3. EVALUATION

In this section, we evaluate AUDITGPT's effectiveness and efficiency by addressing the following key research questions (RQs):

Algorithm 1 multi-agent Validation Framework

Require: A candidate path $\mathcal{P}_{cand}$
Ensure: Validation result (vulnerable or not), a generated PoC if vulnerable

function VALIDATEPATH($\mathcal{P}_{cand}$)
  ATG ← InitializeATG($\mathcal{P}_{cand}$) Create the Audit Task Graph
  Phase 1: False Positive Analysis
  isFP ← AnalyzeFP_WithRAG(ATG, $\mathcal{P}_{cand}$)
  ATG.update(node='FP_Analysis', status='Completed', result=isFP)
  if isFP **then**
    return (False, **null**)
  end if
  Phase 2: PoC Generation
  details ← GenerateDetails_WithAST(ATG, $\mathcal{P}_{cand}$)
  ATG.update(node='PoC_Details', evidence=details)
  poc ← SynthesizePoC(details)
  ATG.update(node='Initial_PoC', poc=poc)
  Phase 3: Iterative Verification & Refinement
  for each attempt in 1 . . . MAX_ATTEMPTS **do**
    response ← SendRequest(poc)
    isSuccess ← Verify(response)
    ATG.update(node='Verification', status=isSuccess, evidence=response)
    if isSuccess **then**
      return (True, poc)
    else
      reason ← AnalyzeFailure(response)
      ATG.update(node='Failure_Analysis', evidence=reason)
      poc ← RefinePoC(poc, reason) Refine PoC based on new evidence
      ATG.update(node='Refined_PoC', poc=poc)
    end if
  end for
  return (False, poc)
end function

- **RQ1 (Efficiency):** How effective is our framework at generating and pruning candidate call chains compared to state-of-the-art (SOTA) tools?
- **RQ2 (Precision):** How effectively does our framework reduce false positives from baseline static analysis?
- **RQ3 (Effectiveness):** What is the success rate of our automated PoC generation?

To answer our research questions, we evaluate AUDITGPT, powered by both DeepSeek-V3 [25] and GPT-4o [26], against three state-of-the-art (SOTA) baselines—CodeQL [6], IRIS [13], and VULkiller [1] on a benchmark of 10 real-world Java applications (e.g., *Jeecg-Boot*, *jshERP*) containing diverse vulnerabilities.

3.1. Efficiency of Candidate Path Analysis

For **RQ1**, we conduct a comparative analysis of the candidate paths generated by our framework against the baseline tool, CodeQL, and the state-of-the-art tool, IRIS. The results, summarized in Table 1, reveal a critical trade-off between path coverage and precision. While the baseline CodeQL generates a manageable but incomplete set of paths, leading to potential false negatives, IRIS exhibits the oppo-

site problem. Our analysis indicates that IRIS's permissive source identification strategy—which includes a vast number of potential entry points without sufficient controllability checks—causes it to generate an excessive number of candidate paths. Many of these paths are fragments of benign data flows or originate from non-user-controllable sources. As quantified in Table 1, this results in a substantial volume of irrelevant candidates, imposing a significant overhead on subsequent verification efforts and diminishing its practical utility.

Table 1: Comparison of Candidate Path Generation and Pruning Efficiency across different tools.

Tool	Candidate Paths	Vuln Path	True Path Coverage
CodeQL	511	39	7.6% (39/511)
IRIS	2344	**60**	2.6% (60/2344)
AUDITGPT	**251**	54	**21.5% (54/251)**

Table 2: Performance of AUDITGPT as an end-to-end framework and as a post-processing filter for SOTA tools.

Analysis Configuration	Final False Positive Rate (%)
Scenario 1: On CodeQL's Candidate Paths	
CodeQL (Baseline)	91.7(433/472)
with AUDITGPT Filter	**22.0(104/472)**
Scenario 2: On IRIS's Candidate Paths	
IRIS (SOTA)	44.8(1025/2284)
with AUDITGPT Filter	**35.4(809/2284)**
Scenario 3:* AUDITGPT *End-to-End Performance Without RAG	
AUDITGPT (w/ DeepSeek V3)	**23.8(47/197)**
AUDITGPT (w/ GPT-4o)	28.4(56/197)
Scenario 3:* AUDITGPT *End-to-End Performance	
AUDITGPT (w/ DeepSeek V3)	**7.1(14/197)**
AUDITGPT (w/ GPT-4o)	**10.6(21/197)**

3.2. Precision and False Positive Reduction

For **RQ2**, we benchmark its false positive (FP) reduction capabilities in two distinct modes: as a post-processing filter for SOTA tools and as a standalone, end-to-end framework. The primary challenge in this domain is that many candidate paths are neutralized by complex sanitization routines whose defensive logic is not fully captured by abstract call chain representations.

Our false positive analysis agent leverages a RAG model, indexed on the entire codebase, to reason about the fine-grained implementation of potential sanitizers, moving beyond call-chain analysis to accurately identify even subtle yet effective sanitization routines.

The empirical results, presented in Table 2, validate the efficacy of this approach across all tested scenarios.

- **As a Post-Processing Filter:** When applied to the raw output of CodeQL (Scenario 1), our framework reduces the final false positive rate from a staggering **91.7%** down to **22.0%**. Similarly, for IRIS (Scenario 2), it reduces the FP rate from **44.8%** to **35.4%**. This demonstrates our framework's capacity to significantly enhance the precision of existing SAST tools.
- **As an End-to-End Framework:** When operating independently (Scenario 3), AUDITGPT achieves a low false positive rate of **7.1%**; this rate drastically increases to **23.8%** when our RAG-based contextual analysis is disabled.

These findings confirm the versatility and effectiveness of our methodology. However, we note that our framework still faces challenges with certain highly abstract frameworks (MyBatis), where sanitization occurs implicitly through framework-internal parameterized queries. This process can leave insufficient semantic clues in the direct data-flow path for our RAG-based analysis to confirm safety, resulting in some false positives and providing a clear direction for future work.

3.3. Effectiveness of PoC Generation

To answer **RQ3**, we evaluate our framework's end-to-end effectiveness on the high-confidence candidates that passed our precision analysis. As detailed in Table 3, our framework achieved an overall PoC success rate of **57.4%**. This high success rate is largely attributable to our multi-agent framework, which decomposes the complex exploitation task. For instance, to validate blind vulnerabilities (out-of-band SSRF), one agent was responsible for payload delivery while another monitored an external DNS log for interaction, confirming exploits where traditional response-based checks would fail.

While the framework is highly effective, we observe nuances in its performance on specific vulnerability types. We guide the agent to use specific techniques, such as error-based SQL injection, but this does not yield a significant performance increase over generic methods. The framework's primary limitation, however, remains with Java Deserialization (CWE-502), where the success rate was 0%. This is a known hard problem, as successful exploitation requires constructing complex "gadget chains" that are highly dependent on the application's environment—a task our current data-flow-based approach is not well-equipped to handle. We leave the enhancement for this specialized challenge as a key direction for future work.

Table 3: Effectiveness of Automated PoC Generation.

CVE/CWE-ID	Type	VULkiller	AUDITGPT (Ours)
CWE-89	SQL Inj.	**58.3%**	41.6%
CWE-78	OS Cmd Inj.	60%	**100%**
CWE-22	Path Inj.	63.7%	**81.8%**
CWE-74	Multi Inj.	33.3%	**66.6%**
CWE-79	Xss.	0%	**70%**
CWE-502	Deserialize.	0%	0%
Average		38.8%	**57.4%**

4. CONCLUSION

In this paper, we address the inherent limitations of traditional static analysis and monolithic LLM approaches in vulnerability detection. We introduce AUDITGPT, a novel multi-agent framework that synergistically integrates deep code analysis with LLM-driven semantic reasoning. Our experiments demonstrate that AUDITGPT significantly enhances detection precision by effectively filtering false positives from baseline tools, while simultaneously achieving a high success rate in automated PoC generation. This dual capability for high-precision analysis and effective end-to-end validation underscores the superiority of our decomposed, multi-agent approach. Future work will focus on extending our framework to handle more complex vulnerability classes, such as those requiring gadget chain construction, and validating its generalizability on a larger scale.

5. ACKNOWLEDGEMENT

This work was supported in part by the Excellent Talent Program of the Institute of Information Engineering, Chinese Academy of Sciences (No. E5YY07111K4).

6. REFERENCES

[1] Xingchen Chen, Baizhu Wang, Mengjun Zhang, Yaqin Cao, and Qixu Liu, "Vulkiller: Java web vulnerability detection with code property graph and large language models," in *ICASSP 2025 - 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2025, pp. 1–5.

[2] Datadog, "State of devsecops," https://www.datadoghq.com/state-of-devsecops/, Accessed: July 2025.

[3] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray, "Deep learning based vulnerability detection: Are we there yet?," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2022.

[4] David Hin, Andrey Kan, Huaming Chen, and M. Ali Babar, "Linevd: statement-level vulnerability detection using graph neural networks," in *Proceedings of the 19th International Conference on Mining Software Repositories*, New York, NY, USA, 2022, MSR '22, p. 596–607, Association for Computing Machinery.

[5] Benjamin Steenhoek, Hongyang Gao, and Wei Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, New York, NY, USA, 2024, ICSE '24, Association for Computing Machinery.

[6] GitHub, "CodeQL," https://codeql.github.com/, Accessed: July 2025.

[7] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer, "QL: Object-oriented Queries on Relational Data," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, Shriram Krishnamurthi and Benjamin S. Lerner, Eds., Dagstuhl, Germany, 2016, vol. 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 2:1–2:25, Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[8] "Checker Framework," https://checkerframework.org/, Accessed: July 2025.

[9] Snyk, "Snyk Code," https://snyk.io/product/snyk-code/, Accessed: July 2025.

[10] SonarSource, "SonarQube," https://www.sonarsource.com/products/sonarqube/, Accessed: July 2025.

[11] Hong Jin Kang, Khai Loong Aw, and David Lo, "Detecting false alarms from automatic static analysis tools: how far are we?," in *Proceedings of the 44th International Conference on Software Engineering*, New York, NY, USA, 2022, ICSE '22, p. 698–709, Association for Computing Machinery.

[12] Youkun Shi, Yuan Zhang, Tianhao Bai, Xuenan Feng, Jiarun Dai, Fengyu Liu, Lei Zhang, Xiapu Luo, and {Yang, Min}, "Xssky: Detecting xss vulnerabilities through local path-persistent fuzzing," in *Proceedings of the 34th USENIX Security Symposium*, 2025.

[13] Ziyang Li, Saikat Dutta, and Mayur Naik, "Llm-assisted static analysis for detecting security vulnerabilities," in *International Conference on Learning Representations*, 2025.

[14] Jie Zhang, Haoyu Bu, Hui Wen, Yongji Liu, Haiqiang Fei, Rongrong Xi, Lun Li, Yun Yang, Hongsong Zhu, and Dan Meng, "When llms meet cybersecurity: A systematic literature review," *Cybersecurity*, vol. 8, no. 1, pp. 1–41, 2025.

[15] Vishwanath Akuthota, Raghunandan Kasula, Sabiha Tasnim Sumona, Masud Mohiuddin, Md Tanzim Reza, and Md Mizanur Rahman, "Vulnerability detection and monitoring using llm," in *2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE)*, 2023, pp. 309–314.

[16] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024.

[17] Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, and Bill Chu, "Software vulnerability detection using large language models," in *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2023, pp. 112–119.

[18] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Hengbo Tong, Swarna Das, Earl T Barr, and Wei Le, "To err is machine: Vulnerability detection challenges llm reasoning," *arXiv preprint arXiv:2403.17218*, 2024.

[19] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo, "Large language model for vulnerability detection and repair: Literature review and the road ahead," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 5, May 2025.

[20] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2020, NIPS '20, Curran Associates Inc.

[21] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al., "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24824–24837, 2022.

[22] Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman, "Shop2: An htn planning system," *Journal of artificial intelligence research*, vol. 20, pp. 379–404, 2003.

[23] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *Advances in neural information processing systems*, vol. 36, pp. 11809–11822, 2023.

[24] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al., "Graph of thoughts: Solving elaborate problems with large language models," in *Proceedings of the AAAI conference on artificial intelligence*, 2024, vol. 38, pp. 17682–17690.

[25] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al., "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[26] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al., "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.