Research on a hybrid approach to detecting software code vulnerabilities using LLM in combination with SAST tools*

Oleh Yarema[1,*,†], Nataliya Zagorodna[1,†], Maryna Derkach[1,†], Aizhan Tokkuliyeva[2,†], and Myroslava Zagorodna[3,†]

[1] *Borys Grinchenko Kyiv Metropolitan University, 18/2 Bulvarno-Kudriavska str., 04053 Kyiv, Ukraine*

[2] *Institute of Mathematical Machines and Systems Problems of the National Academy of Sciences of Ukraine, 42 Ac. Glushkov ave., 03680 Kyiv, Ukraine*

Abstract

This paper explores the effectiveness of combining Large Language Models (LLMs) with traditional Static Application Security Testing (SAST) tools to detect vulnerabilities in source code. The study evaluates how hybrid methods mitigate the false-positive rates of static analysis tools and the "hallucinations" common in neural networks. The experimental results demonstrate that while static analyzers provide deterministic rigor, LLMs offer the semantic depth necessary to filter out irrelevant alerts and understand program logic. The proposed hybrid approach achieved better precision and recall by using SAST reports as technical context to guide the reasoning of LLM.

Keywords

Software Vulnerabilities, Code Analysis, Static Application Security Testing (SAST), Large Language Models (LLMs).

1. Introduction

Software security is becoming a critical factor in the stability of government and commercial systems. Every year, the volume and complexity of source code only increases, making manual code review for security and vulnerabilities an inefficient and slow process. Traditional static analysis (SAST) tools provide fast verification with low resource consumption, but suffer from a high rate of false positives due to their inability to distinguish between syntactic context and program logic [1, 2]. These legacy systems often rely on predefined pattern matching that fails to account for the data flow and state changes within modern software architectures. In addition, traditional security mechanisms, including intrusion detection and prevention systems (IDS and IPS) [3–5], firewalls, and other network security mechanisms [6–8], are not capable of detecting software vulnerabilities at the stage of their introduction, since they operate primarily at the level of network traffic or behavioral anomalies [9]. Such systems focus on analyzing already established data flows, attack signatures, or statistical deviations that arise during the exploitation of vulnerabilities, but do not have access to the semantics of source code and the logic of program implementation. As a result, a significant number of security defects, including improper input handling, incorrect memory management, unsafe deserialization, or secret leaks, remain undetected until they are actively exploited by an attacker. This limitation restricts the ability of network security mechanisms to provide proactive software security and necessitates the use of security analysis methods that operate directly at the source code level, including intelligent approaches based on large language models.

The emergence of large language models (LLMs) has opened up a new paradigm in code analysis, in which neural networks are able to interpret the semantics of development at a syntactic

* *CSDP'2026: Cyber Security and Data Protection, May 7, 2026, Lviv, Ukraine*

* Corresponding author.

† These authors contributed equally.

yarema.oleh.m@gmail.com (O. Yarema); zagorodna_n@tntu.edu.ua (N. Zagorodna); m_derkach@tntu.edu.ua (M. Derkach); tokkuliyeva_ak_1@enu.kz (A. Tokkuliyeva), myroslavazagorodna@gmail.com (M.Zagorodna)

0009-0009-8709-7813 (O. Yarema); 0000-0002-1808-835X (N. Zagorodna); 0000-0001-8977-2776 (M. Derkach); 0000-0002-5019-2413 (A. Tokkuliyeva); 0009-0002-3258-4904 (M. Zagorodna)

level that is close to human or at least emulates it. Unlike static tools, LLMs can ingest large amounts of contextual documentation and historical commit data to better understand the developer's intent behind a specific code block. However, using only LLMs to analyze code for vulnerabilities carries the risk of “hallucinations” and the lack of formal verification of the vulnerabilities found [10, 11]. Furthermore, the non-predetermined nature of generative AI means that the same security query might yield inconsistent results across different sessions, complicating the standardization of security reviews. This creates a need to explore hybrid approaches that combine the capabilities of artificial intelligence with the static accuracy of classical analyzers. In such a framework, the LLM acts as an engine to flag suspicious patterns, while the SAST component provides the mathematical rigor needed to confirm the existence of a flaw [2].

The relevance of this work is due to the need to find a balance between automating vulnerability detection and minimizing false positives. By cross-referencing AI insights with deterministic rule sets, organizations can significantly reduce the "noise" that typically overwhelms security teams.

An initial comparative analysis of three possible strategies, namely the separate use of only analyzers, only LLM, and their hybrid combination, should allow us to determine the most effective way for further research. This methodology ensures that the strengths of symbolic reasoning neural processing are evaluated both in isolation and in tandem. The results of this study have direct practical significance for the construction of algorithms for secure development in real-world projects. The scientific significance of this study is aimed at overcoming the gap between the speed of vulnerability detection and the quality of their analysis.

2. Related works

The current landscape of software security research shows a transition from traditional rule-based methodologies toward advanced, AI-driven intelligence. At the base of industry standards are Static Application Security Testing (SAST) tools such as SonarQube, CodeQL, and SnykCode, which are integrated into continuous-integration pipelines to flag defects before they reach production [12-14]. Academic research proves that early detection through these static scanners reduces the long-term costs of code correction and enhances learning outcomes in educational settings [15]. These traditional tools operate by examining source code without execution, utilizing control flow analysis, data flow analysis, and pattern matching against known error signatures. While they are highly effective at identifying common issues like naming inconsistencies and SQL injection, they face a persistent gap where they often miss nearly half of real-world vulnerability-contributing commits [16]. This limitation stems from their reliance on predefined rules, which prevents them from recognizing new or unusual problems that do not fit a specific, hard-coded pattern [16, 17].

In response to these limitations, recent works have introduced a new paradigm centered on the "code intelligence" of Large Language Models (LLMs). Unlike deterministic scanners, LLMs demonstrate a unique ability to reason across large contexts, identifying subtle data-flow flaws that evade traditional pattern-based rules. Research evaluations involving models like GPT-4, Mistral Large, and DeepSeek V3 indicate that these AI systems often achieve higher recall than static counterparts, successfully identifying a larger portion of existing bugs [18]. However, the scientific community also highlights specific failure modes unique to AI, such as "slop-squatting" — where models hallucinate non-existent package names — and tokenization artifacts that cause errors in pinpointing the exact location of a bug. Furthermore, while LLMs provide superior contextual understanding, they lack the deterministic reliability of rule-based systems [19, 20].

To address these challenges, the effectiveness of any automated security tool remains fundamentally tied to the quality of training data, such as the CVEfixes dataset, which provides real-world vulnerabilities and their respective fixes [20]. Projects like DiverseVul have further expanded this foundation with large-scale source code datasets designed specifically for deep learning-based detection [21-23]. Research by Shvyrov et al. specifically examines the application

of these datasets and LLMs within imperative programming languages, while other studies assess their effectiveness in specialized environments like Android [24, 25].

The trend in recent literature suggests a hybrid future for software assurance [25, 26]. Recent studies have also demonstrated the practical applicability of LLMs for assessing security controls and compliance risks in regulated environments [32]. One promising direction involves integrating LLMs with formal verification to enhance the detection of complex vulnerabilities, such as cryptographic protocol flaws [26]. By combining the "context-aware triage" of LLMs with the high-assurance verification of traditional scanners, developers can balance technological advancements with necessary safeguards [27]. This shift toward "augmented intelligence" in DevSecOps allows for the cross-referencing of data from multiple sources, such as those accessed via platforms like GitHub Models, to reach a more reliable final verdict [28, 29]. Ultimately, the goal is to develop a self-correcting security ecosystem capable of evolving alongside the increasing complexity of modern software [30].

3. Experiment methodology

The general algorithm of the experiment is based on a comparative analysis of three distinct strategies for detecting vulnerabilities in source code written in C#. At the preparatory stage, a control sample of ten unique code snippets is formed, each of which contains a certain number of documented vulnerabilities, such as SQL injections, unsafe deserialization, or secret keys left in the code, obtained from open sources. To ensure a high degree of realism, these snippets are based on open-source repositories and vulnerability databases [16, 21, 31]. This preparation involves the synthesis of test cases by merging multiple vulnerabilities into a single code block. This process requires taking isolated vulnerability patterns from different datasets and refactoring them into a unified, functional code [13]. This consolidation of vulnerabilities allows for the evaluation of whether a model can maintain focus when faced with overlapping security risks within a single context. By increasing the vulnerability density per snippet, the experiment can push the analyzers to demonstrate their ability to handle noise and complex dependencies.

The first stage of the study involves running the Roslyn Analyzers static tool to detect defects without involving neural networks. In the second stage, the same code fragments are submitted to the DeepSeek V3 model from the GitHub Models platform using specialized prompts [28]. The third stage, which is hybrid, involves transferring the static analyzer reports from the first stage as direct context for the LLM. In this scenario, the model acts as a verifier that will confirm or refute the results of the Roslyn analysis based on the semantic logic of the code.

For each approach, the results are recorded using specific performance indicators to ensure a granular assessment of each detection strategy. The number of False Negatives is calculated by subtracting the number of True Positive results from the total number of known vulnerabilities in the reference data set for each project. This metric acts as the primary indicator of missed threats. Conversely, the number of False Positives is determined by subtracting the confirmed True Positives from the total number of vulnerabilities reported by the tool or model; a false positive is thus defined as a reported vulnerability that does not correspond to a known problem in the reference data. The True Positives number is determined from the results manually by confirming tool or model findings.

The data obtained allows for the calculation of the Precision metric, which is defined as the proportion of correctly identified vulnerabilities (True Positives) among all reported findings. This value serves to illustrate how accurate the tool is and its tendency to generate "noise." The Recall metric represents completeness, measuring the proportion of actual vulnerabilities that were correctly identified by the tool.

Furthermore, the F1 score is utilized as the harmonic mean of Precision and Recall to provide a singular measure of the strategy's overall effectiveness. This metric is particularly useful for comparing tools with varying balances of precision and recall, as it offers a comprehensive view of detection performance. The results are analyzed by comparing these metrics for the Roslyn

Analyzer and DeepSeek V3 model separately, as well as for their combination within the hybrid scenario. The final stage consists of systematizing the obtained values into a comparative table to determine the most effective approach. This allows for an objective assessment of the effectiveness of using a combination of an LLM with a statistical analyzer and highlights the need for further research in this area.

4. Experimental results

The experimental results demonstrate a clear hierarchy of effectiveness among the three tested strategies, with the hybrid approach emerging as the better methodology for vulnerability detection in C# environments. Based on the consolidated data, the integration of deterministic static analysis with probabilistic neural reasoning visibly optimized both detection accuracy and completeness. While individual tools faced unique limitations, the combined system reached an Avg. F1 score of 0.94, effectively bridging the gap between raw pattern matching and semantic understanding.

From the Table 1 and the first row of the Table 4, which detail the performance of the standalone Roslyn Analyzer, we can see that it is a tool characterized by high precision but limited flexibility. The analyzer maintained a consistent Avg. Precision of 0.93, successfully identifying well-documented C# security patterns without generating excessive noise. However, its Avg. Recall of 0.87 highlights a vulnerability to false negatives when faced with non-standard or complex logic. For instance, in Snippet 3, the tool recorded two false negatives, failing to identify a significant portion of the total vulnerabilities present. Similarly, it struggled with Snippets 5, 7, and 8, missing one vulnerability in each case despite its high accuracy in other areas. These gaps suggest that while Roslyn is highly reliable for finding what its rules explicitly cover, it remains blind to flaws that require a broader understanding of program context. Furthermore, the tool generated false positives in Snippets 3, 4, and 7, indicating that syntactic matches sometimes fail to account for safe execution contexts.

Table 1
Experimental results for Roslyn Analyzers performance

Code Snippet No.	Total Found	True Positives	False Negative	False Positive
1	4	4	0	0
2	1	1	0	0
3	6	5	2	1
4	3	2	0	1
5	1	1	1	0
6	3	3	0	0
7	6	5	1	1
8	2	2	1	0
9	3	3	0	0
10	1	1	0	0

In the end, Table 1 underscores the "blind spots" of traditional SAST tools, where a lack of semantic depth results in a failure to detect 13% of the existing threats. This data serves as the baseline justification for introducing a Large Language Model to refine and expand the detection capabilities.

Analysis of Table 2 and the second row of the Table 4, which details the performance of the standalone DeepSeek V3 model, reveals a shift toward higher sensitivity but lower consistency compared to traditional analyzers. The model achieved an Avg. Recall of 0.87, identical to the Roslyn Analyzer, but its Avg. Precision was notably lower at 0.86. This discrepancy is largely due

to its higher Avg. False Positives (0.5), demonstrating the model's tendency to flag safe code as suspicious when it lacks formal verification.

The raw data shows that DeepSeek V3 struggled significantly with Snippet 3, recording 2 False Negatives and 1 False Positive, which matches the difficulties faced by the static analyzer in that specific context. However, the model showed some "hallucination" patterns in Snippets 1, 4, 7, and 9, where it generated unnecessary alerts. Interestingly, while it missed a vulnerability in Snippet 4 and Snippet 9, it succeeded in Snippet 5 where the Roslyn Analyzer failed, identifying all 2 True Positives. This suggests that the LLM processes code through a semantic lens that can occasionally bypass the rigid rule-based limitations of SAST tools.

Basically, Table 2 confirms that while DeepSeek V3 is a heuristic engine, its standalone use is hindered by a stochastic nature that produces 0.5 Average False Negatives, necessitating a more structured hybrid approach.

Table 2
Experimental results for DeepSeek performance

Code Snippet No.	Total Found	True Positive	False Negative	False Positive
1	5	4	0	1
2	1	1	0	0
3	6	5	2	1
4	2	1	1	1
5	2	2	0	0
6	3	3	0	0
7	6	5	1	1
8	3	3	0	0
9	3	2	1	1
10	1	1	0	0

Analysis of Table 3 and the third row of Table 4, which highlights the performance of the DeepSeek V3 and Roslyn Analyzer hybrid approach, illustrates a definitive leap in both detection stability and accuracy. By using the static analyzer's findings to guide the neural network's focus, the hybrid model achieved the study's highest Avg. Recall of 0.95 and an Avg. Precision of 0.93. Result in the raw data is the reduction of Avg. False Negatives to 0.2, meaning nearly all existing vulnerabilities were successfully identified. In Snippet 3, where both standalone tools previously struggled with two false negatives each, the hybrid approach successfully recovered one of those missed flaws, bringing the True Positives up to 6. Furthermore, the system completely eliminated false negatives in Snippets 5, 7, and 8, where the Roslyn Analyzer had previously failed to detect a vulnerability. The synergy was equally effective at noise reduction; for instance, the false positives generated by DeepSeek in Snippets 1 and 4 were entirely suppressed in the hybrid results. This suggests that the deterministic context of the Roslyn report acted as a logical filter, preventing the LLM from hallucinating vulnerabilities in safe code segments. In Snippet 7, the hybrid model managed to achieve 6 True Positives with zero False Negatives, a perfect score that neither standalone tool could reach. Even in the most complex case, Snippet 9, the hybrid approach maintained a high detection rate where DeepSeek alone had faltered. The data confirms that while the hybrid model still recorded a single false positive in Snippet 4 and 9, the overall Avg. False Positives dropped to 0.2, the lowest recorded in the experiment. This balanced performance resulted in a peak Avg. F1 score of 0.94, proving that the combination of tools is far more robust than the sum of its parts. Ultimately, Table 3 validates that the hybrid architecture provides the necessary formal verification to make LLM-based security analysis viable for real-world production code.

Table 3
Experimental results for hybrid approach performance

Code Snippet No.	Total Found	True Positive	False Negative	False Positive
1	4	4	0	0
2	1	1	0	0
3	6	6	1	0
4	3	2	0	1
5	2	2	0	0
6	3	3	0	0
7	6	6	0	0
8	3	3	0	0
9	3	2	1	1
10	1	1	0	0

According to the data in Table 4, the hybrid approach demonstrated a superior performance profile, achieving the highest Avg. F1 score of 0.94. This success is primarily driven by the significant reduction in Avg. False Negatives, which fell to 0.2, thereby elevating the Avg. Recall to 0.95 — the peak value across all tested strategies. Furthermore, the hybrid method successfully maintained the Avg. Precision at 0.93, matching the Roslyn Analyzer's accuracy while simultaneously cutting the Avg. False Positives to a study-low of 0.2. These metrics prove that the combination of tools effectively fills the gaps in standalone LLM logic, which previously suffered from a lower Avg. Precision of 0.86. The data confirms that integrating static reports as context allows the system to achieve a more balanced and reliable detection rate than either the deterministic analyzer or the probabilistic model could reach individually.

Table 4
Averaged results for different tools

Tool	Avg. False Negatives	Avg. False Positives	Avg. Precison	Avg. Recall	Avg. F1 score
Roslyn Analyzer	0.5	0.3	0.93	0.87	0.89
DeepSeek V3	0.5	0.5	0.86	0.87	0.87
DeepSeek V3 + Roslyn Analyzer	0.2	0.2	0.93	0.95	0.94

Conclusions

The study confirms that integrating large language models into static code analysis definitely enhances vulnerability detection in C# code. The hybrid approach, combining DeepSeek V3 and Roslyn Analyzer, demonstrated better performance, achieving the highest overall effectiveness across all tested metrics. This synergy between deterministic rule-matching and probabilistic reasoning proved effective, as the hybrid model substantially minimized false alarms, effectively filtering out "noise" that traditional analyzers might flag.

The experiment also introduced new concern — a dependency on prompt engineering. While providing the model with a structured "Chain-of-Thought" instruction increased reasoning depth, standalone LLMs still faced a noticeable rate of false alerts when operating without static context. This sensitivity suggests that current LLM-based analysis is not yet fully deterministic, emphasizing the need for standardized query templates to ensure stability and reproducibility. Furthermore, the hybrid model's ability to reduce missed vulnerabilities proves that static reports act as a necessary anchor for neural networks.

Difficulties in interpreting complex C# constructs, such as asynchronous patterns, indicate that further research should explore specialized models pre-trained on security patches to further lower

error rates. Expanding the list of static tools beyond the Roslyn Analyzer would also provide a broader spectrum of telemetry for cross-referencing. A key direction for future research is the creation of a validated dataset containing multi-layered vulnerabilities distributed across different assembly files, moving beyond standard syntactic errors. Additionally, investigating "Multi-Agent" systems, where one LLM generates a report and another attempts to falsify or verify it, could provide a deeper assessment of LLM potential. Initial explorations of AI-driven multi-agent security frameworks for software development [33] demonstrate the feasibility of such architectures. The goal is to develop a self-correcting security ecosystem capable of improving alongside the increasing complexity of modern software.

Declaration on Generative AI

During the preparation of this work, the authors used Gemini 3.0 in order to improve writing style as well as for formatting assistance. After using this AI tool, the content was reviewed and edited by authors, who therefore take full responsibility for the publication's content.

References

[1] A. Shahana, et al., AI-Driven cybersecurity: balancing advancements and safeguards, J. Comput. Sci. Technol. Stud. 6.2 (2024) 76-85. doi:10.32996/jcsts.2024.6.2.9.

[2] M. Ferrag, et al., SecureFalcon: are we there yet in automated software vulnerability detection with llms?, IEEE Trans. Softw. Eng. (2025) 1-18. doi:10.1109/tse.2025.3548168.

[3] D. Tymoshchuk, A. Sverstiuk, Y. Klots, N. Petliak, V. Titova An explainable artificial intelligence approach for detecting network attacks. CEUR Workshop Proc., 2025, 4141, pp. 38-51.

[4] D. Tymoshchuk, O. Yasniy, M. Mytnyk, N. Zagorodna, V. Tymoshchuk, Detection and classification of DDoS flooding attacks by machine learning method. CEUR Workshop Proc., (2024), 3842, pp. 184-195.

[5] N. Zagorodna, M. Stadnyk, B. Lypa, M. Gavrylov, R. Kozak, Network attack detection using machine learning methods, Chall. Natl. Def. Contemp. Geopolit. Situat. 2022.1 (2022) 55-61. doi:10.47459/cndcgs.2022.7.

[6] M. Stetsiuk, Y. Klots, D. Tymoshchuk, M. Karpinski, N. Petliak, Detection of multi-vector attacks in IoT networks: a graph attention network-based approach. CEUR Workshop Proc., 2025, 4146, pp. 296-313.

[7] Y. Klots, V. Titova, N. Petliak, D. Tymoshchuk, N. Zagorodna, Intelligent data monitoring anomaly detection system based on statistical and machine learning approaches. CEUR Workshop Proc., (2025), 4042, pp. 80-89.

[8] N. Petliak, Y. Klots, M. Karpinski, V. Titova, D. Tymoshchuk, Hybrid system for detecting abnormal traffic in IoT. CEUR Workshop Proc., (2025), 4057, pp. 21-36.

[9] D. Tymoshchuk, N. Zagorodna, Y. Klots, V. Yatskiv, N. Petliak, AutoML and explainable AI-based approach to enhance the efficiency and interpretability of IDS. CEUR Workshop Proc., 2025, 4163, pp. 231-246.

[10] Y. Ding, et al., Vulnerability detection with code language models: how far are we?, In: 2025 IEEE/ACM 47th international conference on software engineering (ICSE), IEEE, 2025, c. 1729-1741. doi:10.1109/icse55347.2025.00038.

[11] C. Minisini, N. Antol´ın, Cryptoformaleval: Integrating llms and formal verification for automated cryptographic protocol vulnerability detection, 2024.

[12] X. Du, et al., Generalization-Enhanced code vulnerability detection via multi-task instruction fine-tuning, In: Findings of the association for computational linguistics ACL, Association for Computational Linguistics, Stroudsburg, PA, USA, 2024, p. 10507-10521. doi:10.18653/v1/2024.findings-acl.625.

[13] J. He, M. Vechev, Large language models for code: security hardening and adversarial testing, In: CCS '23: ACM SIGSAC conference on computer and communications security, ACM, New York, NY, USA, 2023. doi:10.1145/3576915.3623175.

[14] Y. Boltov, I. Skarga-Bandurova, M. Derkach. A comparative analysis of deep learning-based object detectors for embedded systems. In: 2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), vol. 1, pp. 1156-1160).

[15] E. Beljulji, I. Matta, Large language models in security code review and testing, J. Syst. Res. 5.1 (2026). doi:10.5070/sr3.62177.

[16] V. Shvyrov, D. Kapustin, R. Sentyay, T. Shulika, Analysis of datasets and large language models for vulnerability detection in imperative programming language code, Program. Ingeneria 15.11 (2024) 555-569. doi:10.17587/prin.15.555-569.

[17] V. Kouliaridis, G. Karopoulos, G. Kambourakis, Assessing the effectiveness of llms in android application vulnerability analysis, In: Lecture notes in computer science, Springer Nature Switzerland, Cham, 2025, c. 139-154. doi:10.1007/978-3-031-85593-1_9.

[18] D. Ajiga, P. Okeleke, S. Folorunsho, Ch. Ezeigweneme, The role of software automation in improving industrial operations and efficiency, Int. J. Eng. Res. Updates 7.1 (2024), 022-035. doi:10.53430/ijeru.2024.7.1.0031.

[19] R. Venkatasubramanyam, S. Gupta, U. Uppili, Assessing the effectiveness of static analysis through defect correlation analysis, In: 2015 IEEE 10th international conference on global software engineering (ICGSE), IEEE, 2015. doi:10.1109/icgse.2015.18.

[20] D. Singh, V. Sekar, K. Stolee, B. Johnson, Evaluating how static analysis tools can reduce code review effort, In: 2017 IEEE symposium on visual languages and human-centric computing (VL/HCC), IEEE, 2017. doi:10.1109/vlhcc.2017.8103456.

[21] Y. Chen, Z. Ding, L. Alowain, X. Chen, D. Wagner, DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection, In: RAID 2023: the 26th international symposium on research in attacks, intrusions and defenses, ACM, New York, NY, USA, 2023. doi:10.1145/3607199.3607242.

[22] I. Simões, E. Venson, Evaluating source code quality with large languagem models: a comparative study, In: SBQS 2024: XXIII brazilian symposium on software quality, ACM, New York, NY, USA, 2024, p. 103-113. doi:10.1145/3701625.3701650.

[23] F. Cheirdari, G. Karabatis, Analyzing false positive source code vulnerabilities using static analysis tools, In 2018 IEEE international conference on big data (big data), IEEE, 2018. doi:10.1109/bigdata.2018.8622456.

[24] H. Khater, M. Khayat, S. Alrabaee, M. Serhani, E. Barka, F. Sallabi, AI techniques for software vulnerability detection and mitigation, In: 2023 IEEE conference on dependable and secure computing (DSC), IEEE, 2023. doi:10.1109/dsc61021.2023.10354233.

[25] Ayobami Olwadamilola Adebayo, Automating security compliance in devsecops through ai-driven policy enforcement, Int. J. Sci. Res. Arch. 15.2 (2025) 670–675. doi:10.30574/ijsra.2025.15.2.1457.

[26] A. Mohammed, Elevating cybersecurity audits: how AI is shaping compliance and threat detection, Elev. Cybersecur. Audit. 2.1 (2023) 1-9. doi:10.5281/zenodo.14760068.

[27] L. Babatunde, E. Etim, I. Essien, E. Cadet, J. Ajayi, E. Erigha, E. Obuse, Adversarial machine learning in cybersecurity: vulnerabilities and defense strategies, J. Front. Multidiscip. Res. 1.2 (2020) 31-45. doi:10.54660/.jfmr.2020.1.2.31-45.

[28] GitHub models.

[29] S. Ponta, H. Plate, A. Sabetta, Detection, assessment and mitigation of vulnerabilities in open source dependencies, Empir. Softw. Eng. 25.5, 2020, 3175-3215. doi:10.1007/s10664-020-09830-x.

[30] U. Mittal, D. Panchal, AI-based evaluation system for supply chain vulnerabilities and resilience amidst external shocks: An empirical approach, Rep. Mech. Eng. 4.1 (2023) 276-289. doi:10.31181/rme040122112023m.

[31] G. Bhandari, A. Naseer, L. Moonen, CVEfixes: automated collection of vulnerabilities and their fixes from open-source software, In: PROMISE '21: 17th International Conference On Predictive Models and Data Analytics in Software Engineering, ACM, New York, NY, USA, 2021. doi:10.1145/3475960.3475985.

[32] O. Deineka, O. Harasymchuk, A. Partyka, A. Obshta, Application of LLM for Assessing the Effectiveness and Potential Risks of the Information Classification System According to SOC 2 Type II, CEUR Workshop Proceedings 3991 (2025) 215-232.

[33] O. Vakhula, I. Opirskyy, AI-driven Security as Code for software development using multi-agent systems, CEUR Workshop Proc. 4024 (2025) 170-185.