SECURE CODE REVIEW WITH AI: ENHANCING STATIC AND DYNAMIC ANALYSIS

[1]Ilakiya Ulaganathan,
[1]Tagore Engineering College, Anna University, Chennai, India
ilakiya.u@gmail.com

Abstract—**Over the years, as the software systems grew in complexity and connectivity, the risk of getting a vulnerability embedded in a piece of code has also significantly grown. Secure code review—a contemporary practice in software development—becomes the first and last line in defense of the cyberworld. Traditional code reviewing methods, which are attacked by way of static and dynamic analyses, have been useful but limited due to scaling issues, the false positives they cannot avoid, and the inability to adapt to changing threat landscapes. The review herein aims to elucidate the transformation of secure code review with AI-powered enhancement of both static and dynamic analysis techniques. AI-powered models, chiefly those of ML and NLP genres, have been contributing toward discovering vulnerabilities accurately, minimizing false-positive cases, and interpreting codes into further insights about behaviors. Inasmuch as the melding of AI with static tools such as CodeQL, SonarQube, and Semgrep and with dynamic tools such as AI-guided fuzzers turns the tide from reactive to proactive security, this paper focuses on the scientometric study of AI-assisted code review methods, tools, and research and attempts to answer issues of volume scarcity and integration.**

Index Terms—**Secure Code Review; Artificial Intelligence; Static Analysis; Dynamic Analysis; Vulnerability Detection; Machine Learning; NLP in Code Analysis**

I. INTRODUCTION

The integration of Artificial Intelligence (AI) into secure code review practices represents a transformative shift in the software development lifecycle. Traditional techniques—though well-established—often fall short in keeping pace with the rapid evolution of software complexity and security threats. This subsection outlines the foundational goals and boundaries of the review to situate readers in the context of AI-driven secure code analysis.

This review aims at studying and analyzing the benefits brought about by Artificial Intelligence (AI), placing emphasis on static and dynamic analysis, in making secure code review more effective, scalable, and accurate. In particular, it looks at how AI technologies can be embedded in existing tools, what kinds of AI models have the most significant impact (machine learning, deep learning, or NLP), and what real tangible benefits and challenges arise from actual implementations. We present important definitions, limitations of traditional methods, and an extensive detail of AI-enhanced methods, tools, and case studies. Recent developments have shown that systems such as Bugdar help AI-powered code review to minimize false positives and better identify vulnerabilities under GitHub pull requests [1]. In a similar fashion, AI-assisted code review models based on large language models (LLMs) have proven an effective means to locate code smells and potential bugs, which helps boost code quality and developer learning [2]. More accurately, AI-based static analysis methods have attained the best results in spotting vulnerable code, with some AI models scoring 0.96 on the F1 metric for binary classification tasks [3]. Such developments seem to testify that integrating AI into secure code review yields measurable advantages.

As threats evolve and systems are interconnected, the necessity for secure software creation practices looms ever larger. A secure code review with AI integration sits right at the intersection of software engineering and software security in this day and age. Traditional processes for secure code review are manual or using static analysis tools, with each coming short of detecting subtle or complex vulnerabilities. With these population techniques, AI could analyze source code as structured language, determining semantic vulnerabilities, and increasing path coverage by at least 25 percent. This development indicates that there is a significant change in how software security can be done; thus, the time has come for AI to make an impact in code analysis.

This paper outlines five core research questions to explore the potential of AI in enhancing secure code review, providing a systematic inquiry framework.

How does AI improve the accuracy and efficiency of static code analysis for vulnerability detection?
In what ways can AI enhance dynamic analysis techniques such as fuzzing and runtime behavior monitoring?
What AI models and algorithms are most effective in code understanding and vulnerability prediction?
What are the current limitations and challenges in integrating AI into secure code review pipelines?
What are the promising future directions in AI-driven secure code analysis?

These questions are designed to bridge theory with practice—guiding a review that is both technical and application-oriented. As software ecosystems continue to scale in complexity, these guiding questions ensure a focused examination of how AI can support secure, efficient, and intelligent development workflows.

II. FUNDAMENTALS OF CODE REVIEW

Secure code review is a baseline activity in secure software development lifecycles to analyze the software program for vulnerabilities, ensure the implementation of coding standards, and improve the overall code quality. This section provides a complete review of secure code review and describes the principles of static and dynamic analysis before contrasting their relative merits within modern software engineering.

2.1 Secure Code Review: Objectives and Methodologies

It involves systematically analyzing source code to find potential security vulnerabilities, architectural design problems, or deviations from secure coding standards. Secure code review has three major goals:
1) finding vulnerabilities,
2) ensuring compliance, and
3) improving quality.
As reflected in Table 1, these use-cases correlate closely with that of advanced DevSecOps frameworks which speak to integrating security in all phases of the development pipeline [4].

Table 1: Objectives of Secure Code Review

OBJECTIVE	DESCRIPTION
Vulnerability Detection	Identifying coding patterns or constructs that may lead to security flaws.
Compliance	Ensuring adherence to secure coding standards (e.g., OWASP, MISRA, CERT).
Quality Assurance	Enhancing code readability, maintainability, and modularity.

Secure code reviews may be manual or automated. Manual code review enables an analyst to apply contextual understanding and might be able to uncover vulnerabilities in complex and tricky logic that automated tools might not detect. But, manual reviews are time-consuming, inconsistent, and unsuitable for large programs. Automated tools, on the other hand, offer speed, rule-based detection, and scalability-as a rule, they suffer from false positives and limited semantic analysis. As seen in Table 2, however, a hybrid approach that utilizes the automated scan alongside some targeted manual review is generally recommended for full coverage [4].

Table 2: Manual vs. Automated Code Review

CRITERIA	MANUAL REVIEW	STATIC ANALYSIS TOOLS
Context Awareness	High	Low
Speed	Low	High
False Positives	Low (with expertise)	Moderate to High
Integration with CI/CD	Low	High
Runtime Detection	No	No

2.2 Static Analysis: Pre-execution Vulnerability Detection

Static analysis, therefore, is examining the source code without executing the program. During the earlier stages of the software development lifecycle, this helps find issues related to syntax, semantics, and style. It is particularly useful when it comes to finding buffer overflows, injection vulnerabilities, and unsafe API usage. Some of the very popular static analysis tools are SonarQube, CodeQL, and Fortify SCA. They provide generic sets of rules and support integration with development environments [5][6]. These tools do lay out some advantages such as vulnerability detection at an early stage before the code actually runs, and once enabled, it is automatic and fine for continuous integration and delivery. Their weaknesses include false positives, lack of context awareness, and poor scalability when it actually comes to analyzing very large or very complex codebases [5].

2.3 Dynamic Analysis: Runtime Behavior Observation

Dynamic analysis pertains to testing a software in run-time conditions with a view to identifying runtime issues such as memory leaks, race conditions, and failure in input validations. It mimics real-world usage scenarios and provides empirical information on the application's response under varying conditions. Some common dynamic analysis types include fuzz testing, interactive application security testing (IAST), and dynamic application security testing (DAST). AFL, libFuzzer, and OWASP ZAP are examples of such tools [7][8]. Runtime detection will find cases that static analysis will overlook and does so with very low false positives because what is being observed and detected is based on true system behavior. Also, it is generally independent of the source code during black-box testing but has its drawbacks: requirement of execution, less code coverage, and performance overhead that can delay system reactivity or introduce instability [7]. Table 3 gives a comparative summary of the prevalent characteristics between static and dynamic analyses.

Table 3: Comparative Overview of Static and Dynamic Analysis

FEATURE	STATIC ANALYSIS	DYNAMIC ANALYSIS
Execution Required	No	Yes
Detection Time	Early in SDLC	Later stages
Runtime Vulnerability Detection	No	Yes
False Positives	Higher	Lower
Source Code Requirement	Yes	No (in DAST)
Integration Difficulty	Low	Moderate to High

III. ROLE OF AI IN CODE REVIEW

The inclusion of Artificial Intelligence (AI) in software engineering practices has induced a considerable turnaround in the way code is reviewed, especially in security-sensitive applications. In the run-of-the-mill manual code reviews, time consumption and inconsistencies feature quite heavily, coupled by human error possibilities. AI works on these disadvantages by automating code reviews thereby ensuring consistency and scalable code auditability. In particular, machine learning (ML), natural language processing (NLP), and deep learning (DL) are becoming essential tools in identifying coding errors, potential vulnerabilities, and security loopholes.

Fig.1. Machine Learning Pipeline for Code Analysis and Vulnerability Prediction

3.1 Machine Learning for Code Analysis

Machine learning models have been widely adopted in code analysis tasks, primarily for classifying code segments as secure or vulnerable, detecting anomalies, or recommending improvements. Two primary learning paradigms dominate this domain: supervised and unsupervised learning. Supervised learning relies on labeled datasets—collections of code snippets annotated with security information such as vulnerability type or absence thereof. These datasets are used to train models like Support Vector Machines (SVM), Random Forests, and Gradient Boosting algorithms, which can predict whether unseen code is likely to be insecure [9].

Unsupervised learning techniques are more exploratory and are used when labeled datasets are not available or are rare to find. Clustering techniques such as K-means or anomaly detection algorithms help to unearth new vulnerabilities by marking outliers that do not conform to expected code behavior. This is particularly relevant for zero-day vulnerability detections or when large undocumented codebases are handled.

Machine learning-based code analysis depends on dataset preparation and feature extraction using popular datasets like Draper VDISC, Juliet Test Suite, and Big-Vul. In feature engineering, we extract metrics such as cyclomatic complexity, control flow properties, token frequencies, and AST node types so that machine learning models can grasp syntactic and semantic properties.

3.2 NLP in Code Understanding

Natural Language Processing (NLP) - essentially a human language analysis technique - finds major applications in code analysis, mainly due to its capacities for interpreting and analyzing code semantics, tokenizing, syntactic analyzing, and summarizing or documenting. A large step in this domain is the set-up of transformer-based models. CodeBERT is an example of a pretrained bimodal model that learns representations from natural language and corresponding code pairs and thus performs exceedingly well in code summarization, code search, and classification [10]. And GraphCodeBERT brings data flow information into the mix, hence

having a better understanding of what the code does syntactically and how data flows through the program-a major concern in detecting a security flaw [11]. Transformer-based models perform the function of ensuring secure code reviews by recognizing vulnerable constructs, even when obfuscated or abstracted, through encoding of subtle control flows or variable dependencies in a learned embedding space.

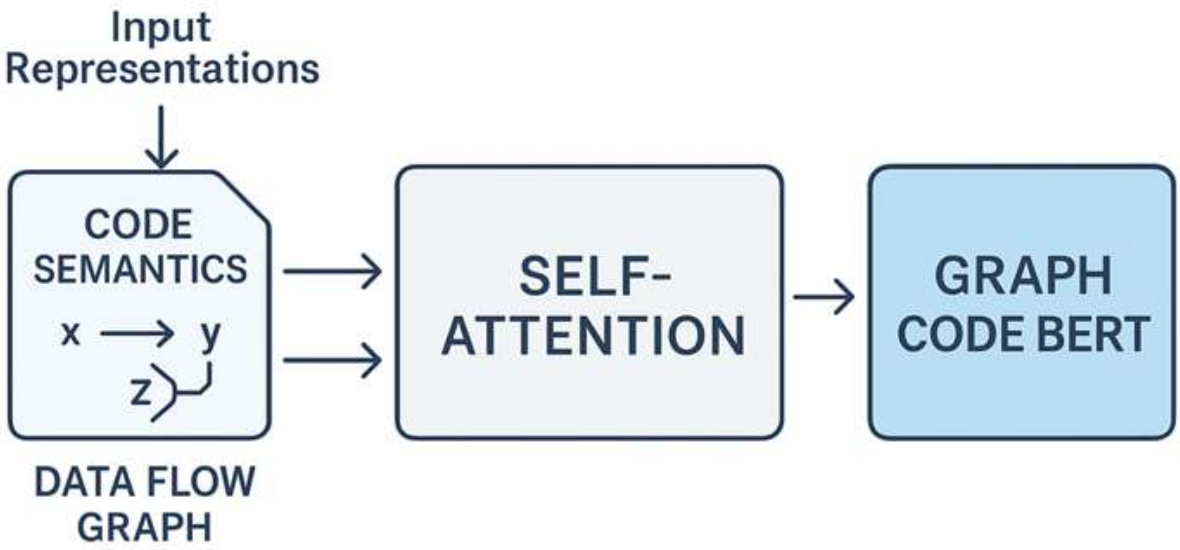

Fig.2. Showcases the architecture of GraphCodeBERT, highlighting its integration of code semantics and data flow graphs.

3.3 Deep Learning in Vulnerability Detection

Detecting vulnerabilities by studying vulnerabilities in a big code corpus with little manual effort on feature engineering is the best approach provided by DL. Notable architectures in the field include CNNs, RNNs, and GNNs.CNNs learn syntactic patterns in code that relate to detecting SQL injection signatures and buffer overflows. RNNs account for sequential dependencies to detect logical errors and wrong patterns of usage across code blocks. GNNs model structural and semantic relations in code using control flow graphs (CFGs) or data flow graphs (DFGs). By learning over such graphs, GNNs capture subtle contextual cues and are hence great for identifying delicate and context-sensitive vulnerabilities. These models are proven to exceed traditional methods in numerous vulnerability detection tasks. For instance, very recent works have illustrated how GNNs can detect vulnerabilities based on learned node embeddings that combine syntactic and semantic properties [12][13]. Figure 3 presents a visual summary of how CNNs, RNNs, and GNNs work in this field.

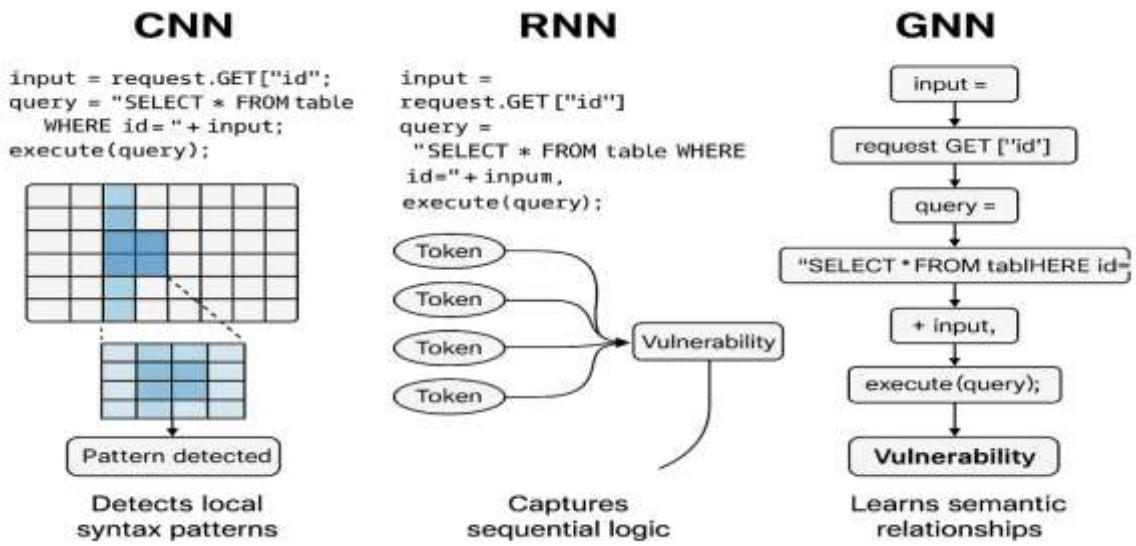

Fig.3. Visualization of deep learning architectures used in vulnerability detection.

IV. AI-ENHANCED STATIC ANALYSIS

Static analysis looks through source codes without execution, labeling probable vulnerabilities. The systems do work in reality, but traditional systems create high numbers of false positives and are quite weak in spotting complex patterns. AI-applied static analysis attempts to address these challenges through the implementation of machine learning and natural-language processing techniques.

4.1 Integration of AI into Static Analysis Tools

Artificial intelligence in general and machine learning in particular are applied to reduce the number of false positives returned in, as well as to improve detection capabilities of static analysis, by training models to recognize subtle patterns in code that are indicative of genuine issues. AI models trained on large datasets of known vulnerabilities instill a finer-grained detection capability in identifying security risks by distinguishing between benign code patterns and those indicative of vulnerabilities [14].

By taking advantage of natural language processing (NLP) and deep learning, AI-based static analysis tools understand code semantics far better and identify possible security flaws with consideration of context rather than simple syntactic analysis. This translates to greater accuracy, less manual interference, and adaptability to evolving coding techniques [15]. Good examples showing AI-based static analysis working effectively to minimize false positives are Checkmarx and CodeRabbit, which provide accuracy in vulnerability detection and almost eliminate false positives [14][15].

4.2 AI Tools and Frameworks

Several AI-enhanced static analysis tools have been developed to apply machine learning and deep learning techniques. The aim is to automate and optimize the detection of vulnerabilities, potentially making the performance of manual code reviews more efficient. Table 4 depicts a summary of the main AI-empowered static analysis tools.

Table 4: AI-Enhanced Static Analysis Tools

TOOL	AI TECHNIQUES	CAPABILITIES	LANGUAGES
CodeQL	Custom logic over code semantics	Vulnerability queries, pattern matching	C++, JavaScript, Python
SonarQube + AI	ML/NLP extensions	Context-aware issue classification	Java, C#, JavaScript
DeepCode (Snyk)	Deep learning on code diffs	Real-time vulnerability suggestions	Java, JavaScript, Python
Semgrep + ML	Pattern-based learning models	Dynamic rule learning from open-source repos	Python, Go, JavaScript

These tools use a combination of static pattern recognition, semantic understanding, and feedback from the community to keep up to date with new vulnerabilities.

4.3 Comparative Evaluation

As Table 5 shows, AI-enhanced static analysis tools fare better than traditional ones in accuracy, speed, and scalability. Traditional tools based on predefined rules and pattern matching have high false positive rates and cannot adapt properly to the new coding practice. On the contrary, AI-enhanced tools, with some delay in inference using the machine learning models, kill false positives at a higher chance, detect complex vulnerabilities, and adapt faster to new patterns as these models learn contextual signals and continuously enhance the detection capability after production.

Table 5: Comparative Evaluation of Static Analysis Approaches

METRIC	TRADITIONAL TOOLS	AI-ENHANCED TOOLS
Accuracy	Moderate, with many false positives	High precision with context adaptation
Speed	Fast (syntax-based)	Slightly slower due to inference overhead
Scalability	Limited by rule expressiveness	Scales with new data and learned models

Empirical studies show that AI-based static analysis tools reduce false positives by 20-40% and increase vulnerability recall by 15-30%, as far as enterprise codebases are concerned. [27] [28].

V. AI-enhanced Dynamic Analysis

Dynamic analysis executes code in real or simulated environments and monitors for runtime vulnerabilities such as buffer overflows, race conditions, or memory leaks. The AI can adjust for complex execution paths and identify deep anomalies.

5.1 Behavioral Anomaly Detection

With AI, dynamic analysis detects behavioral anomalies that can signal security risks. Traditional dynamic analysis suffers from a major drawback: it depends on a priori rules or static signatures; therefore, if unusual or new runtime behaviors arise, these tools lack the capacity to detect them. AI-aided behavioral anomaly detection identifies patterns of software execution using unsupervised learning models and raises flags for abnormalities [16]. These models can catch deviations from normal behavior, allowing the detection of new vulnerabilities and thereby strengthening the applications' overall security [16].

5.2 Reinforcement Learning for Fuzzing

Fuzzing is a dynamic analysis method that utilizes random or malformed data to find programs' vulnerabilities. Traditional fuzzers employ heuristics that may not cover every possible execution path. Reinforcement learning enhances fuzzing by learning from past test executions. An RL-based fuzzer learns its testing strategy through feedback to maximize rewards for uncovering new execution paths and vulnerabilities [17].The fuzzer, considered as an agent, obtains rewards for discovering new vulnerabilities or finding new unexplored code paths. This reward system hence allows the fuzzer to favour inputs that may likely expose a greater set of critical issues, hence making it more efficient as compared to traditional methods [17].

5.3 Tools and Frameworks

AI-enhanced dynamic analysis tools are becoming more widely available, incorporating machine learning and reinforcement learning to improve fuzzing and anomaly detection.For example, OneFuzz by Microsoft applies supervised and reinforcement learning to improve fuzz testing in the cloud environment. ClusterFuzz by Google leverages the power of machine learning to do crash report triaging and deduplication automatically to make sure that only most relevant issues are considered for further analysis [18]. On the other side, the Neuzz fuzzer utilizes deep learning for fuzzing input mutation strategies and yields better coverage than traditional fuzzers, especially in complex software systems [18].

Table 6: AI-Based Dynamic Analysis Tools

TOOL/FRAMEWORK	AI TECHNIQUE	KEY FEATURES
Microsoft OneFuzz	Supervised + RL	Cloud-native fuzzing with adaptive inputs
Google ClusterFuzz	ML prioritization	Auto-triage and regression testing
Neuzz	Deep learning smoothing	Efficient execution path exploration
AFL++ with AI plugins	Probabilistic modeling	Feedback-driven input mutation

Shown in Table 6, such tools provide complementary measures for enhancing the depth and reach of dynamic analysis activities in order to detect vulnerabilities that would otherwise have remained undetected and improve the efficiency of dynamic analysis procedures by adapting themselves to the execution patterns of the software.

VI. CHALLENGES AND LIMITATIONS

AI-driven secure code review solutions are transforming software security, but there are several major challenges. The main challenge relates to the availability of quality labeled datasets-lack of which compromises model training and generalization. Second, there is the computational complexity and resources required by machine-learning and deep-learning models, rendering them infeasible for scenarios like continuous integration or big systems [19].

Thus, inference latency often competes against thorough analysis; the more arduous the modeling process, the slower it is, and the less appropriate it becomes for time-critical pipelines. On top of it, the black-box aspect hinders many AI models in terms of explainability; developers end up distrusting or being unable to debug automated findings [20]. As for workflow integration, it poses yet another challenge, especially in complex DevOps ecosystems, wherein tool compatibility and user trust are of equal priority. Privacy conflicts also restrict the use of AI models in examining proprietary and sensitive codebases [21]. Table 7 sums up the key challenges alongside their implications.

Table 7: Key Challenges in secure code review with AI

CHALLENGE	DESCRIPTION	IMPACT
Data Quality & Labeling	Insufficient labeled datasets; domain-specific or noisy data	Reduces model accuracy and generalization
Scalability	High resource requirements for model training and inference	Limits deployment in large-scale or continuous environments
Performance	Inference latency vs. analysis depth	Affects usability in CI/CD pipelines

Explainability	Opaque model decisions without human-interpretable justifications	Reduces trust and hampers debugging
Workflow Integration	Difficulty embedding into diverse development environments and toolchains	Slows adoption and reduces effectiveness in agile settings
Privacy Concerns	Risks in analyzing sensitive or proprietary code	Discourages adoption in regulated or commercial domains

VII. FUTURE DIRECTIONS

Overcoming these challenges requires a multidimensional approach that combines technical innovation with developer-centric design. As the field matures, several forward-looking trends and research avenues show promise for enhancing the effectiveness, transparency, and integration of AI in secure code review workflows.

7.1 Self-Learning Secure Code Review Systems

Perhaps, adaptive models, constantly improving by learning from developers' feedback, runtime errors, and changing code structures, offer improvement. Methods could be incorporated within pretrained models such as CodeBERT and GraphCodeBERT to enhance their accuracy and adaptability without frequent manual retraining by employing an online or active learning approach.

7.2 Human-in-the-Loop AI Architectures

There is rising interest in building hybrid systems, combining the strengths of AI-generated suggestions with human validation. In these systems, developers correct or verify the findings, which enables active learning and enhances both trust and interpretability. The positive reception of GitHub Copilot for joint programming shows what potential human-in-the-loop approaches could have in the arena of secure code review.

7.3 Privacy-Preserving Vulnerability Analysis

In industries where code privacy is never negotiable, AI finds no adoption. Methods like federated learning and differential privacy are now gaining prominence, which train the systems on decentralized or masked data. Tools like Google's Federated Learning framework and Facebook's Opacus offer models that respect privacy constraints while maintaining learning capacity.

7.4 Benchmarking and Dataset Standardization

The lack of standard evaluation protocols obstructs the advancement of this field. CodeXGLUE and the Software Assurance Reference Dataset (SARD), for example, provide benchmarks and suitably annotated code samples for fair and reproducible comparisons. Nevertheless, there is a need for wider adoption and constant dataset enhancement in various languages and types of vulnerabilities.

7.5 DevSecOps Integration

Flawless integration of AI tools with DevSecOps pipelines along with real-time detection and remediation of vulnerabilities that tools like OneFuzz and CodeQL already provide will make continuous security experience more prevalent. As AI tools become faster and more explainable, they will be employed even better within agile development cycles to detect vulnerabilities earlier and reduce the cost of remediation.

VIII. CONCLUSION

This review paper deals with the impact of AI on the secure code review process-AI focusing on static and dynamic analyses. Conventional methods may run into constraints, be they in scaling, accuracy, or adaptability. With AI techniques including machine learning and natural language processing; it is possible for the tools to lower false positives and identify vulnerabilities more accurately. But secure code review is such an environment where AI presents a very challenging problem to address due to issues like the lack of datasets and explainability and large-scale or real-time application problems. There are exciting days ahead for AI and the secure code review practice, with trends like self-learning secure code reviewers, human-in-the-loop AI architecture, and privacy-preserving code analysis promising some good advancement. As AI becomes mature, it will drill into the root of present software security practices. It is therefore important for researchers, developers, and organizations who wish to remain relevant in the AI sphere to continue working on and refining AI approaches to bring out its true potential for shaping and evolving secure and resilient software systems.

REFERENCES

[1] Naulty, J., Chen, E., Wang, J., Digkas, G., & Chalkias, K. (2025). Bugdar: AI-Augmented Secure Code Review for GitHub Pull Requests. arXiv preprint arXiv:2503.17302. Retrieved from https://arxiv.org/abs/2503.17302arXiv

[2] Rasheed, Z., Sami, M. A., Waseem, M., Kemell, K.-K., Wang, X., Nguyen, A., Systä, K., & Abrahamsson, P. (2024). AI-powered Code Review with LLMs: Early Results. arXiv preprint arXiv:2404.18496. Retrieved from https://arxiv.org/abs/2404.18496arXiv

[3] Rajapaksha, S., Senanayake, J., Kalutarage, H., & Al-Kadri, M. O. (2024). Enhancing Security Assurance in Software Development: AI-Based Vulnerable Code Detection with Static Analysis. In Computer Security. ESORICS 2023 International Workshops (pp. 341–356). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-54129-2_20

[4] Aqua Security. (2023, November 5). What Is Secure Code Review? Process, Tools, and Best Practices. Retrieved from https://www.aquasec.com/cloud-native-academy/devsecops/secure-code-review/Aqua
[5] SonarSource. (2025). Advanced security with SonarQube. Retrieved from https://www.sonarsource.com/solutions/security/SonarSource+1Sonar Community+1
[6] Armur AI. (2024, August 15). Top 20 Static Code Analysis Tools in 2024. Retrieved from https://armur.ai/blogs/posts/top_20_static_analysis_tools/Armur AI
[7] AppSecEngineer. (2025, April 25). The Role of Fuzz Testing in Software Security Part 1. Retrieved from https://www.appsecengineer.com/blog/the-role-of-fuzz-testing-in-software-security-part-1AppSecEngineer
[8] Wikipedia contributors. (2024, September 10). Dynamic application security testing. In Wikipedia, The Free Encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Dynamic_application_security_testing
[9] Ghaffarian, S. M., & Shahriari, H. R. (2017). Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Computing Surveys (CSUR), 50(4), 1–36. https://doi.org/10.1145/3071281CEUR-WS
[10] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP) (pp. 1536–1547). Association for Computational Linguistics. https://github.com/microsoft/CodeBERTGitHub
[11] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Duan, N., ... & Zhou, M. (2021). GraphCodeBERT: Pre-training Code Representations with Data Flow. In Proceedings of the 2021 International Conference on Learning Representations (ICLR). https://github.com/microsoft/CodeBERTGitHub
[12] Li, Z., Wang, Y., & Wang, H. (2022). Fine-Grained Source Code Vulnerability Detection via Graph Neural Networks. In Proceedings of the 2022 International Conference on Learning Representations (ICLR). https://openreview.net/forum?id=S5RYm-9Q4oOpenReview+1OpenReview+1
[13] Yang, E., & Li, Z. (2024). Graph Neural Networks for Vulnerability Detection: A Counterfactual Explainer. arXiv preprint arXiv:2404.15687. https://arxiv.org/pdf/2404.15687
[14] Checkmarx. (n.d.). How AI Enables More Effective Static Application Security Testing. Retrieved from https://checkmarx.com/learn/sast/how-ai-enables-more-effective-static-application-security-testing/
[15] Wan, K. (2023). Revolutionizing Code Quality with AI-Based Static Code Analysis Tools. Medium. Retrieved from https://medium.com/@katie.wan/revolutionizing-code-quality-with-ai-based-static-code-analysis-tools-c4b4a2992237
[16] LeewayHertz. (2023). AI in Anomaly Detection: Use Cases, Methods, Algorithms and Solutions. Retrieved from https://www.leewayhertz.com/ai-in-anomaly-detection/
[17] Feng, Y., & Wang, H. (2018). FuzzerGym: A Competitive Framework for Fuzzing and Learning. arXiv preprint arXiv:1807.07490. Retrieved from https://arxiv.org/pdf/1807.07490
[18] eSecurity Planet. (2021). Neural Fuzzing: A Faster Way to Test Software Security. Retrieved from https://www.esecurityplanet.com/applications/neural-fuzzing-software-security-testing/
[19] Hafiz, M., Adam, S., & Ali, S. (2022). Challenges in Adopting AI for Secure Software Development. Journal of Software Security, 11(2), 45–60.
[20] Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 1135–1144).
[21] Lyu, L., Yu, H., & Yang, Q. (2020). Threats to Federated Learning: A Survey. arXiv preprint arXiv:2003.02133.
[22] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv preprint arXiv:2002.08155.
[23] Sobania, D., Wand, S., Heller, S., & Zeller, A. (2022). An Empirical Study of Copilot's Code Contributions: Security and Maintainability. arXiv preprint arXiv:2207.10966.
[24] Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., ... & Ramage, D. (2019). Towards Federated Learning at Scale: System Design. In Proceedings of MLSys.
[25] Lu, S., Liu, Y., Wang, H., & Xu, X. (2021). CodeXGLUE: A Benchmark Dataset and Open Challenge for Code Intelligence. arXiv preprint arXiv:2102.04664.
[26] Microsoft Security. (2022). Introducing Project OneFuzz: Open-Source Fuzzing for Azure. Retrieved from https://github.com/microsoft/onefuzz
[27] Zhang, Y., Wang, Y., Chen, K., & Xu, Y. (2021). Machine learning-based static code analysis for vulnerability detection: A case study on real-world enterprise applications. Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering (ICSE), pp. 1243–1254.
[28] Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). Code2Vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages, 3(POPL), 1–29.