

A Survey on Software Vulnerability Detection Techniques Using Static and Dynamic Analysis

Dr.Jvalant Kumar Kanaiyalal Patel

Assistant Professor

Shri Manilal Kadakia College of Commerce, Management, Science And Computer Studies

Ankleshwar

jvalant007@gmail.com

Corresponding Author* Jvalant Kumar Kanaiyalal Patel (Gmeil @ com)

Received: 12 August 2026 | **Revised:** 16 August 2026 | **Accepted:** 25 August 2026

Abstract—Software vulnerabilities pose significant security risks to modern applications, particularly as software systems become increasingly complex, interconnected, and widely deployed across critical domains. Effective vulnerability detection is therefore essential for identifying security weaknesses before they can be exploited. This survey provides a comprehensive review of software vulnerability detection techniques, with particular emphasis on static analysis, dynamic analysis, and hybrid approaches. Static analysis enables early detection and broad code coverage, while dynamic analysis evaluates runtime behavior and supports vulnerability validation. Hybrid approaches combine these complementary capabilities to improve detection coverage and reduce false-positive findings. The survey further examines recent AI-driven approaches, including machine learning, deep learning, graph neural networks, transformer-based models, Large Language Models (LLMs), and multimodal learning. The reviewed studies indicate that AI-assisted and hybrid approaches can improve vulnerability detection capability, semantic understanding, vulnerability coverage, and automation compared with isolated traditional techniques. However, challenges involving false positives, dataset quality, computational cost, explainability, and model generalization remain. Overall, the findings highlight the potential of integrating traditional program analysis with AI-driven methods to develop more accurate, scalable, and reliable software vulnerability detection systems.

Keywords—Software Vulnerability Detection, Static Analysis, Dynamic Analysis, Secure Software Development, Vulnerability Detection Tools.

I. INTRODUCTION

Software systems are essential for automating processes, managing information, supporting communication, and delivering digital services across modern sectors, making their reliability, security, and efficient operation increasingly important [1]. They play a vital role in modern applications, including cloud computing, the Internet of Things (IoT), healthcare, finance, and critical infrastructure [2]. As software complexity increases, security vulnerabilities such as buffer overflows, injection attacks, memory corruption, and access control flaws have become major cybersecurity concerns. Exploiting these vulnerabilities can lead to data breaches,

financial losses, and service disruption, making software vulnerability detection an essential component of secure software development.

Software vulnerability detection techniques are primarily classified into static analysis and dynamic analysis. Static analysis examines source code or binaries without executing the program, enabling early detection of security flaws during development. Dynamic analysis evaluates software during execution to identify runtime vulnerabilities under different execution scenarios [3]. While static analysis provides broad code coverage, it may generate false positives, whereas dynamic analysis offers more realistic detection but is limited by execution path coverage. Consequently, hybrid approaches that integrate both techniques have been developed to improve detection accuracy and reliability.

Recent advances in artificial intelligence have significantly improved software vulnerability detection. Deep learning, graph neural networks, transformer models, and Large Language Models (LLMs) have demonstrated strong capabilities in learning code representations and identifying complex vulnerability patterns. Furthermore, integrating AI with traditional techniques such as taint analysis, symbolic execution, and fuzz testing has enhanced automated vulnerability detection and reduced false-positive rates [4]. This survey provides a comprehensive overview of software vulnerability detection techniques based on static and dynamic analysis, including their fundamental approaches, advantages, limitations, and recent advancements. It also examines hybrid and AI-driven techniques to highlight their potential for improving the effectiveness, accuracy, and reliability of software vulnerability detection.

The remainder of this paper is organized as follows: Section II presents software vulnerability detection and classification. Section III discusses static, dynamic, and hybrid techniques. Section IV reviews AI-driven approaches. Section V presents related literature, while Section VI concludes the survey and discusses future research directions.

II. SOFTWARE VULNERABILITY DETECTION AND CLASSIFICATION

Software vulnerability detection is the process of identifying weaknesses in software that could be exploited to compromise the confidentiality, integrity, or availability of a system. Vulnerabilities may arise from programming errors [5], insecure coding practices, improper input validation, memory management issues, or software misconfigurations. Detecting these vulnerabilities during the software development lifecycle helps reduce security risks, improve software quality, and prevent cyber-attacks [6]. Modern vulnerability detection combines traditional program analysis techniques with AI-driven approaches to improve detection accuracy and automation. A software vulnerability is a weakness or flaw in software that can be exploited by attackers to perform unauthorized actions or disrupt system operations. Vulnerabilities may originate during software design, implementation, testing, or deployment and often result from insecure coding practices or inadequate validation mechanisms. The most common software vulnerabilities include:

- **Buffer Overflow:** Occurs when a program writes data beyond the allocated memory boundary, potentially allowing arbitrary code execution or system crashes [7].
- **SQL Injection (SQLi):** Enables attackers to inject malicious SQL statements through user inputs to manipulate backend databases and gain unauthorized access to sensitive information [8].
- **Cross-Site Scripting (XSS):** Allows malicious scripts to be injected into web applications, leading to session hijacking, credential theft [9], or unauthorized actions on behalf of users.
- **Cross-Site Request Forgery (CSRF):** Tricks authenticated users into executing unintended requests on trusted web applications [10].

A. Vulnerability Identification and Classification Frameworks

To standardize vulnerability identification and reporting, the cybersecurity community relies on two widely adopted frameworks developed by MITRE:

1) Common Weakness Enumeration (CWE)

A standardized catalog of software weakness types that describes common programming and design flaws, such as improper input validation [11], buffer overflows, and memory management errors. Developers and researchers use CWE to classify weaknesses and improve secure coding practices.

2) Common Vulnerabilities and Exposures (CVE)

A publicly available database that assigns unique identifiers to disclosed software vulnerabilities [12]. Each CVE entry contains information about the affected software, vulnerability description, severity, and references, enabling consistent vulnerability tracking and management across security tools and organizations.

B. Software Vulnerability Detection Process

The software vulnerability detection process consists of systematically identifying, validating, and mitigating security weaknesses throughout the Software Development Life Cycle (SDLC). Early vulnerability detection minimizes remediation costs and reduces the likelihood of successful cyber-attacks

[13]. The typical vulnerability detection process includes the following stages:

- **Software Analysis:** Source code, binaries, or running applications are analyzed to identify potential security weaknesses.
- **Vulnerability Identification:** Static analysis, dynamic analysis, or hybrid techniques are used to detect potential vulnerabilities.
- **Validation:** Detected vulnerabilities are verified to distinguish genuine security flaws from false positives.
- **Risk Assessment:** Vulnerabilities are prioritized based on severity, exploitability, and their impact on confidentiality, integrity, and availability.
- **Remediation:** Developers apply security patches, code modifications, or configuration changes to eliminate identified vulnerabilities.
- **Verification and Monitoring:** The software is retested to ensure vulnerabilities have been resolved, followed by continuous monitoring to detect newly introduced security issues [14][15].

III. SOFTWARE VULNERABILITY DETECTION TECHNIQUES

Software vulnerability detection techniques are designed to identify security weaknesses during different phases of the software development lifecycle. These techniques are broadly categorized into static analysis, dynamic analysis, and hybrid analysis, each employing different strategies to discover vulnerabilities. Static analysis focuses on examining software artifacts without execution, dynamic analysis evaluates software behavior during execution, and hybrid analysis combines the strengths of both approaches to achieve higher detection accuracy and broader vulnerability coverage. The selection of an appropriate technique depends on factors such as software complexity, required analysis depth, execution environment, and performance constraints.

A. Static Analysis

Static analysis examines source code, bytecode, or executable binaries without executing the software. It enables early vulnerability detection during development and can analyze reachable code paths and integrate with continuous integration workflows. However, it may generate false positives and cannot always determine exploitability. No single static analysis tool can detect every vulnerability type, highlighting the need for complementary techniques [16]. The major static analysis techniques include:

- **Source Code Analysis:** Examines source code to identify insecure programming practices, coding errors, and potential vulnerabilities before software execution [17].
- **Binary Analysis:** Analyzes compiled executables when source code is unavailable, making it suitable for third-party software and malware assessment.
- **Taint Analysis:** Tracks the flow of untrusted input throughout a program to detect vulnerabilities such as SQL injection, command injection, and cross-site scripting.
- **Symbolic Execution:** Uses symbolic inputs instead of concrete values to explore multiple execution paths and identify conditions that may trigger vulnerabilities.
- **Static Application Security Testing (SAST):** Employs automated tools to scan source code,

bytecode, or binaries for known security weaknesses and coding rule violations during development.

B. Dynamic Analysis

Dynamic analysis evaluates software while it is executing to identify vulnerabilities that become apparent only during runtime. Unlike static analysis, it observes actual program behavior under different inputs and execution environments, making it effective for validating exploitability and detecting runtime security issues. Because vulnerabilities are detected through execution, dynamic analysis generally produces fewer false positives than static analysis. However, its effectiveness depends on execution path coverage, and vulnerabilities located in unexecuted code paths may remain undiscovered [3]. The principal dynamic analysis techniques include:

- **Runtime Monitoring:** Continuously observes memory usage, program execution, and system interactions to identify abnormal or malicious behavior.
- **Dynamic Application Security Testing (DAST):** Evaluates running applications by interacting with them externally to discover security flaws without requiring source code access.

- **Fuzz Testing (Fuzzing):** Automatically generates large numbers of malformed or unexpected inputs to trigger crashes, exceptions, or security vulnerabilities.
- **Concolic Testing:** Combines concrete execution with symbolic execution to improve execution path coverage and generate effective test cases for vulnerability discovery.

C. Hybrid Analysis

Hybrid analysis integrates static and dynamic analysis to exploit their complementary strengths. Static analysis identifies potential vulnerabilities, while dynamic analysis validates them through program execution. This combination improves vulnerability coverage, reduces false positives, and enhances detection accuracy. Modern hybrid frameworks combine static analysis with symbolic execution, taint analysis, fuzz testing, or artificial intelligence to identify complex vulnerability patterns [18]. Although hybrid analysis provides more comprehensive detection, it requires greater computational resources, implementation complexity, and analysis time. Despite these challenges, it is a promising approach for developing intelligent and automated software vulnerability detection systems. Table I compares static, dynamic, and hybrid software vulnerability detection techniques based on their working principles, major methods, advantages, and limitations.

TABLE I. COMPARATIVE ANALYSIS OF STATIC, DYNAMIC, AND HYBRID SOFTWARE VULNERABILITY DETECTION TECHNIQUES

Technique	Working Principle	Major Methods	Advantages	Limitations
Static Analysis	Examines source code, bytecode, or executable binaries without executing the software to identify potential security flaws during development.	Source Code Analysis, Binary Analysis, Taint Analysis, Symbolic Execution, Static Application Security Testing (SAST)	Early vulnerability detection, broad code coverage, scalable, integrates well with Secure SDLC and CI/CD pipelines.	High false-positive rate, cannot validate runtime exploitability, limited understanding of dynamic program behavior.
Dynamic Analysis	Evaluates software during execution to identify vulnerabilities based on actual runtime behavior under different inputs and environments.	Runtime Monitoring, Dynamic Application Security Testing (DAST), Fuzz Testing, Concolic Testing	Lower false positives, detects runtime vulnerabilities, validates exploitability, identifies memory and logic errors.	Limited execution-path coverage, runtime overhead, may miss vulnerabilities in unexecuted code.
Hybrid Analysis	Combines static and dynamic analysis to improve vulnerability coverage and detection accuracy through complementary analysis.	Static Analysis + Dynamic Analysis, AI-Assisted Hybrid Analysis, Symbolic Execution with Fuzz Testing, Taint-Guided Testing	Higher detection accuracy, reduced false positives, comprehensive vulnerability assessment, improved test-case generation.	Increased computational cost, higher implementation complexity, longer analysis time, greater resource requirements.

IV. AI-DRIVEN AND EMERGING VULNERABILITY DETECTION APPROACHES

Recent advances in artificial intelligence (AI) have significantly transformed software vulnerability detection by enabling automated learning of complex code patterns and semantic relationships. Unlike traditional rule-based techniques, AI-driven approaches can identify hidden vulnerability patterns from large software repositories, improving detection accuracy and scalability. Machine learning (ML), deep learning (DL), large language models (LLMs), and multimodal learning have become prominent research directions because they reduce manual feature engineering and support intelligent vulnerability assessment. However, their effectiveness largely depends on the availability of high-quality datasets, representative code embeddings, and robust model architectures [19].

A. Machine Learning and Deep Learning Methods

Machine learning and deep learning techniques automatically learn vulnerability-related features from source code, binary code, abstract syntax trees (ASTs), control flow

graphs (CFGs), and program dependency graphs (PDGs). Unlike traditional static analysis, which relies on predefined rules, these methods learn complex semantic and structural relationships directly from training data, enabling the detection of previously unseen vulnerability patterns [20]. The Commonly used learning models include:

- **Traditional Machine Learning:** Algorithms such as Support Vector Machine (SVM), Random Forest (RF), Decision Tree (DT), and XGBoost use handcrafted software metrics and code features for vulnerability classification.
- **Convolutional Neural Networks (CNNs):** Capture local code patterns and syntactic features to identify vulnerability signatures.
- **Recurrent Neural Networks (RNNs) and LSTM:** Model sequential dependencies in source code, making them suitable for analyzing execution logic and code context.
- **Graph Neural Networks (GNNs):** Learn structural relationships from Abstract Syntax Trees (ASTs),

Control Flow Graphs (CFGs), and Program Dependency Graphs (PDGs), enabling more accurate representation of software semantics.

- **Transformer-Based Models:** Utilize self-attention mechanisms to capture long-range dependencies within source code, improving vulnerability classification and localization.
- **Code Representation Learning:** Converts source code into vector embeddings using token-, graph-, or tree-based representations, allowing learning models to effectively capture semantic and syntactic information [21].

B. Large Language Models (LLMs) and Multimodal Approaches

Large Language Models (LLMs) have recently emerged as a promising paradigm for software vulnerability detection by leveraging knowledge acquired from large-scale source code and natural language corpora [22]. Models such as ChatGPT, GPT-4, and code-oriented language models can understand code semantics, generate vulnerability explanations, detect security flaws, and even recommend remediation strategies. Their ability to reason over source code using natural language prompts has introduced new possibilities for automated security analysis [23]. Recent research has further extended LLM-based vulnerability detection through several emerging approaches:

- **Prompt Engineering:** Enhances detection performance by designing structured prompts that incorporate contextual, syntactic, and semantic information.
- **Code-Specific LLMs:** Models trained primarily on programming languages provide more accurate vulnerability identification and code understanding than general-purpose language models.
- **Neuro-Symbolic Methods:** Combine LLM reasoning with traditional program analysis techniques, such as taint analysis and symbolic execution, to improve precision while reducing false positives.
- **Multimodal Vulnerability Detection:** Integrates multiple information sources, including source code, commit messages, software metrics, and syntactic structures, to learn richer vulnerability representations.
- **Knowledge Distillation:** Transfers knowledge from large, computationally expensive models to lightweight models suitable for practical deployment while preserving detection performance [24][25].

V. LITERATURE REVIEW

This section reviews recent advancements in software vulnerability detection using static analysis, dynamic analysis, large language models, multimodal learning, and hybrid approaches.

Rim Mahouachi (2026) Investigates how heterogeneous modalities contribute to vulnerability detection and proposes a multimodal deep learning framework integrating code semantics, commit messages, static metrics, and syntactic

structures through an attention-based fusion mechanism. To support real-world deployment where commit information may be unavailable, a teacher–student knowledge distillation scheme transfers knowledge from a multimodal teacher to a lightweight code-only student model. Experiments on multiple vulnerability datasets demonstrate that multimodal fusion improves detection performance, while knowledge distillation retains most of these gains for practical deployment [26].

Ziyang Li, Saikat Dutta and Mayur Naik (2025) IRIS proposes a neuro-symbolic approach that combines LLMs with static analysis for whole-repository security vulnerability detection. It uses LLMs to infer taint specifications and perform contextual analysis, reducing the need for manual specifications and inspection. Evaluated on the CWE-Bench-Java dataset containing 120 validated vulnerabilities, IRIS with GPT-4 detects 55 vulnerabilities compared to Cadel’s 27, reduces the average false discovery rate by 5 percentage points, and identifies 4 previously unknown vulnerabilities [27].

Huiwen Yang et al. (2024) Implemented Cross Fuzz, an extension of Confuses, and evaluated it on 396 real-world smart contracts. The results show that Cross Fuzz outperforms fuzz with a 10.58% increase in bytecode coverage and detects 1.82 times more security vulnerabilities than Confuses. The method utilizes data flow information to optimize mutation strategies, significantly improving the efficiency of cross-contract vulnerability detection [28].

Chenyuan Zhang et al. (2024) investigates software vulnerability detection using ChatGPT with different prompt designs. It enhances basic prompts by incorporating structural, sequential, and multi-round dialogue information. Experiments on two vulnerability datasets demonstrate the effectiveness of prompt-enhanced ChatGPT for improving software vulnerability detection [29].

Moumita Das Purba et al. (2023) evaluates four well-known LLMs on two widely used public datasets for software vulnerability detection. Results reveal a performance gap compared to popular static analysis tools, mainly due to high false positive rates. However, LLMs demonstrate strong potential in identifying subtle vulnerability patterns, highlighting the benefits of combining LLMs with program analysis techniques for improved detection [30].

Kanchan Singh, Sakshi S Grover and Ranjini Kishen Kumar (2022) aim to develop a software vulnerability detection system by treating source code as text using Natural Language Processing (NLP) and recent deep learning models. Various models were developed and compared, with the best-performing model achieving 95% accuracy. Additionally, the study predicts the vulnerability class of source code and develops a robust dashboard using Fast API and ReactJS [31].

Table II summarizes the objectives, techniques, detection focus, key contributions of representative studies, providing a comparative overview of existing research trends.

TABLE II. COMPARATIVE ANALYSIS OF SOFTWARE VULNERABILITY DETECTION TECHNIQUES USING STATIC AND DYNAMIC ANALYSIS

Authors	Objective	Techniques / Approach	Detection Focus	Key Findings / Contributions
Rim Mahouachi (2026)	To improve software vulnerability detection	Multimodal Deep Learning, Attention-Based Fusion,	Multimodal Vulnerability Detection	Proposed a multimodal framework integrating code semantics, commit messages, static metrics, and syntactic structures. Knowledge distillation enabled

	using multimodal learning.	Teacher–Student Knowledge Distillation		a lightweight code-only model while preserving most of the detection performance.
Ziyang Li, Saikat Dutta & Mayur Naik (2025)	To improve repository-level software vulnerability detection by integrating LLMs with static analysis.	IRIS Neuro-Symbolic Framework, GPT-4, Static Analysis, Taint Specification Inference	Static Analysis + LLM-Assisted Detection	IRIS detected 55 vulnerabilities compared to 27 by CodeQL, reduced false discovery rates, and identified 4 previously unknown vulnerabilities, demonstrating the effectiveness of neuro-symbolic analysis.
Huiwen Yang et al. (2024)	To enhance cross-contract vulnerability detection in blockchain smart contracts.	CrossFuzz, Data Flow Analysis, Fuzz Testing, Mutation Optimization	Dynamic Analysis (Fuzz Testing)	CrossFuzz achieved 10.58% higher bytecode coverage than xFuzz and detected 1.82× more vulnerabilities than ConFuzzius through data-flow-guided mutation strategies.
Chenyuan Zhang et al. (2024)	To investigate the effectiveness of ChatGPT for software vulnerability detection using different prompt designs.	ChatGPT, Prompt Engineering, Structural Context, Sequential Context, Multi-Round Dialogue	LLM-Based Vulnerability Detection	Enhanced prompting strategies significantly improved ChatGPT's vulnerability detection capability across multiple benchmark datasets.
Moumita Das Purba et al. (2023)	To evaluate the capability of large language models for software vulnerability detection.	Large Language Models (LLMs), Comparative Evaluation, Static Analysis Benchmarking	AI-Based Vulnerability Detection	LLMs successfully identified subtle vulnerability patterns but generated higher false positive rates than conventional static analysis tools, indicating the need for hybrid detection methods.
Kanchan Singh, Sakshi S. Grover & Ranjini Kishen Kumar (2022)	To automate software vulnerability detection using NLP and deep learning techniques.	Natural Language Processing (NLP), Deep Learning, Source Code Classification, FastAPI, ReactJS	Source Code Vulnerability Detection	Achieved 95% detection accuracy, predicted vulnerability classes effectively, and developed a dashboard for vulnerability classification and visualization.

VI. CONCLUSION AND FUTURE WORK

Software vulnerability detection is essential for maintaining the security, reliability, and resilience of modern software systems. This survey reviewed major vulnerability detection techniques, focusing on static analysis, dynamic analysis, and hybrid approaches, along with recent AI-driven developments. Static analysis provides early detection and broad code coverage, while dynamic analysis validates vulnerabilities through runtime behavior and improves exploitability assessment. Hybrid analysis combines these complementary strengths to achieve more comprehensive detection and reduce false positives. The review further highlighted the growing contribution of machine learning, deep learning, graph neural networks, transformer-based models, LLMs, and multimodal learning in automating vulnerability detection and capturing complex code semantics. Recent studies demonstrate that integrating AI with traditional program analysis can improve detection capability, vulnerability coverage, and practical scalability. Overall, the survey highlights hybrid and AI-assisted approaches as promising directions for developing accurate, automated, and reliable software vulnerability detection systems.

Future research should focus on integrated AI-driven hybrid frameworks that combine static analysis, dynamic testing, LLM reasoning, and multimodal code representations. Emphasis should also be placed on reducing false positives, improving explainability, handling evolving vulnerabilities, strengthening generalization across datasets, and developing lightweight models suitable for real-time and large-scale software security applications.

- **Funding:** This research received no external funding.
 - **Institutional Review Board Statement:** Not applicable.
 - **Informed Consent Statement:** Not applicable.
 - **Data Availability Statement:** The data supporting the findings of this study are available from the corresponding author upon reasonable request.
 - **Conflicts of Interest:** The author declares no conflict of interest.

How to cite this article:

J. K. K. Patel, “A Survey on Software Vulnerability Detection Techniques Using Static and Dynamic Analysis,” *Journal of Global Research in Electronics and Communication*, vol. 2, no. 8, pp. 16–21, Aug. 2026, Doi,XXX

REFERENCES

[1] R. Vishvakarma and S. Jacob, “Risks and Challenges of Using Agentic AI in Enterprise Software Systems,” *Int. J. Comput. Appl.*, vol. 187, no. 128, pp. 48–57, Jul. 2026, doi: 10.5120/ijcaaf6b11ab4d7e.

[2] Y. Yang *et al.*, “Dlap: A Deep Learning Augmented Large Language Model Prompting Framework for Software Vulnerability Detection,” 2024, *Elsevier*. doi: 10.2139/ssm.4831591.

[3] N. T. Cong, L. H. Toan, and T. M. Thanh, “Tổng quan về phân tích tĩnh và phân tích động trong kiểm tra bảo mật ứng dụng,” *J. Mil. Sci. Technol.*, vol. 99, no. 99, pp. 1–11, Nov. 2024, doi: 10.54939/1859-1043.j.mst.99.2024.1-11.

[4] S. Bin Hulayyil, S. Li, and L. Xu, “Machine-Learning-Based Vulnerability Detection and Classification in Internet of Things Device Security,” *Electronics*, vol. 12, no. 18, p. 3927, Sep. 2023, doi: 10.3390/electronics12183927.

[5] R. Azmeera, “The Effects of Increased Software Errors on Software Product Functionality,” University of the Cumberland, 2025. doi: 10.13140/RG.2.2.18820.80009.

[6] K. Filus and J. Domańska, “Software vulnerabilities in TensorFlow-based deep learning applications,” *Comput. Secur.*, vol. 124, p. 102948, Jan. 2023, doi: 10.1016/j.cose.2022.102948.

[7] B. Barchuk, “Fundamentals of buffer overflow attacks and detection techniques,” *World J. Adv. Res. Rev.*, vol. 26, no. 2, pp. 2895–2908, May 2025, doi: 10.30574/wjarr.2025.26.2.1967.

[8] K. Meshram, J. Augustine, K. Pund, K. Kanojiya, and S. Manwatkar, “SQL Injection Attack Detection and Prevention System,” *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 13, no. 9, pp. 1632–1639, 2025, doi: 10.22214/ijraset.2025.74282.

[9] S. J. Y. Weamie, “Cross-Site Scripting Attacks and Defensive Techniques: A Comprehensive Survey*,” *Int. J. Commun. Netw. Syst. Sci.*, vol. 15, no. 08, pp. 126–148, 2022, doi: 10.4236/ijens.2022.158010.

[10] C. N. Siahhaan, M. Rufisanto, R. Nolasco, S. Achmad, and C. R. P. Siahhaan, “Study of Cross-Site Request Forgery on Web-Based Application: Exploitations and Preventions,” *Procedia Comput. Sci.*, vol. 227, pp. 92–100, 2023, doi: 10.1016/j.procs.2023.10.506.

[11] S. Al Attiq, C. Gehrmann, K. Dahlén, and K. Khalil, “From Generalist to Specialist: Exploring CWE-Specific Vulnerability Detection,” *arXiv*, pp. 1–12, 2024, doi: 10.48550/arXiv.2408.02329.

[12] E. Aghaei, E. Al-shaer, W. Shadid, and X. Niu, “Automated Cve

- Analysis For Threat Prioritization and Impact Prediction,” 2023, *arXiv preprint*. doi: 10.48550/arXiv.2309.03040.
- [13] J. Hauroné, N. Basheer, and S. Islam, “Vulnerability detection using BERT based LLM model with transparency obligation practice towards trustworthy AI,” *Mach. Learn. with Appl.*, vol. 18, no. October, p. 100598, 2024, doi: 10.1016/j.mlwa.2024.100598.
- [14] K. Bennouk, N. Ait Aali, Y. El Bouzekri El Idrissi, B. Sebai, A. Z. Faroukhi, and D. Mahouachi, “A Comprehensive Review and Assessment of Cybersecurity Vulnerability Detection Methodologies,” *J. Cybersecurity Priv.*, vol. 4, no. 4, pp. 853–908, 2024, doi: 10.3390/jcp4040040.
- [15] U. Ravindran and R. V. Potukuchi, “A Review on Web Application Vulnerability Assessment and Penetration Testing,” *Rev. Comput. Eng. Stud.*, vol. 9, no. 1, pp. 1–22, Mar. 2022, doi: 10.18280/rces.090101.
- [16] M. Alqaradaghi and T. Kozsik, “Comprehensive Evaluation of Static Analysis Tools for Their Performance in Finding Vulnerabilities in Java Code,” *IEEE Access*, vol. 12, no. February, pp. 55824–55842, 2024, doi: 10.1109/ACCESS.2024.3389955.
- [17] R. Azmeera and J. Hyatt, “<p>Software Errors, Product Functionality, and Organizational Cascades: Evidence from Developer-Lived Experience</p>,” 2026. doi: 10.2139/ssrn.6924439.
- [18] F. A. Aboaja, A. Zainal, F. A. Ghaleb, B. A. S. Al-rimy, T. A. E. Eisa, and A. A. H. Elnour, “Malware Detection Issues, Challenges, and Future Directions: A Survey,” *Appl. Sci.*, vol. 12, no. 17, p. 8482, Aug. 2022, doi: 10.3390/app12178482.
- [19] M. Malkawi and R. Alhajj, “AI-Powered Vulnerability Detection and Patch Management in Cybersecurity: A Systematic Review of Techniques, Challenges, and Emerging Trends,” *Mach. Learn. Knowl. Extr.*, vol. 8, no. 1, p. 19, Jan. 2026, doi: 10.3390/make8010019.
- [20] N. Shiri Harzevili, A. Boaye Belle, J. Wang, S. Wang, Z. M. (Jack) Jiang, and N. Nagappan, “A Systematic Literature Review on Automated Software Vulnerability Detection Using Machine Learning,” *ACM Comput. Surv.*, vol. 57, no. 3, pp. 1–36, Mar. 2025, doi: 10.1145/3699711.
- [21] N. S. Harzevili *et al.*, “A Survey on Automated Software Vulnerability Detection Using Machine Learning and Deep Learning,” *arXiv Prepr. arXiv2306.11673*, vol. 37, no. 4, pp. 1–37, 2023, doi: 10.48550/arXiv.2306.11673.
- [22] S. R. Somula, “Mitigating Estimation Challenges in Large-Scale Projects: An Integrated Framework for Enhanced Accuracy and Adaptability,” *IPHO-Journal Adv. Res. Sci. Eng.*, vol. 3, no. 10, 2025, doi: 10.5281/zenodo.17462211.
- [23] S. Kaniewski, F. Schmidt, M. Enzweiler, M. Menth, and T. Heer, “A Systematic Literature Review on Detecting Software Vulnerabilities with Large Language Models,” *ACM Trans. Softw. Eng. Methodol.*, pp. 1–61, May 2026, doi: 10.1145/3815425.
- [24] Z. Gao, H. Wang, Y. Zhou, W. Zhu, and C. Zhang, “How Far Have We Gone in Vulnerability Detection Using Large Language Models,” 2023, *arXiv preprint*. doi: 10.48550/arXiv.2311.12420.
- [25] T. Banerjee, Piyush, M. V., and G. Singh, “Multimodal and Large Language Model Approaches in Cybersecurity: A Systematic Review,” Apr. 27, 2026. doi: 10.21203/rs.3.rs-9527568/v1.
- [26] R. Mahouachi, “Knowledge distillation-driven commit-aware multimodal learning for software vulnerability detection,” *Autom. Softw. Eng.*, vol. 33, no. 2, p. 48, 2026, doi: 10.1007/s10515-026-00595-z.
- [27] Z. Li, S. Dutta, and M. Naik, “IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities,” 2025, *ICLR 2025*. doi: 10.48550/arXiv.2405.17238.
- [28] H. Yang, X. Gu, X. Chen, L. Zheng, and Z. Cui, “CrossFuzz: Cross-contract fuzzing for smart contract vulnerability detection,” *Sci. Comput. Program.*, vol. 234, no. December 2023, p. 103076, 2024, doi: 10.1016/j.scico.2023.103076.
- [29] C. Zhang, H. Liu, J. Zeng, K. Yang, Y. Li, and H. Li, “Prompt-Enhanced Software Vulnerability Detection Using ChatGPT,” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, New York, NY, USA: ACM, Apr. 2024, pp. 276–277. doi: 10.1145/3639478.3643065.
- [30] M. Das Purba, A. Ghosh, B. J. Radford, and B. Chu, “Software Vulnerability Detection using Large Language Models,” in *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Florence, Italy: IEEE, 2023, pp. 112–119. doi: 10.1109/ISSREW60843.2023.00058.
- [31] K. Singh, S. S. Grover, and R. K. Kumar, “Cyber Security Vulnerability Detection Using Natural Language Processing,” in *2022 IEEE World AI IoT Congress (AIoT)*, Seattle, WA, USA: IEEE, 2022, pp. 174–178. doi: 10.1109/AIIoT54504.2022.9817336.