IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, October 2026
Impact Factor: 8.2

Machine Learning-Based Vulnerability Detection in AI-Generated Source Code: A Systematic Literature Review and Research Directions

Prof. Rana Afreen Sheikh[1], Shrutika S. Mehare[2], Vishakha K. Topare[3]
Professor, MCA Department[1]
Students, MCA Department[2-3]
Vidya Bharati Mahavidyalaya, Amravati, Maharashtra, India

Abstract: *The rapid adoption of Generative AI and Large Language Models (LLMs) has transformed software development, but AI-generated code may contain security vulnerabilities such as injection flaws, weak authentication, and unsafe API usage. Traditional rule-based and static-analysis techniques struggle with semantically diverse generated code. This paper presents a systematic literature review of Machine Learning (ML), Deep Learning (DL), Graph Neural Network, Transformer, and LLM-based vulnerability detection, with emphasis on AI-generated source code. It compares approaches, datasets, and code representations, and examines challenges including data imbalance, false positives, limited AI-generated datasets, and explainability. A conceptual framework combining preprocessing, structural and semantic representation, ML/DL classification, CWE categorization, explainability, and validation is proposed. Future directions include hybrid ML-LLM models, real-time IDE integration, and automated vulnerability repair..*

Keywords: Machine Learning, Deep Learning, Generative AI, Large Language Models, AI-Generated Code, Vulnerability Detection, Cybersecurity, Source Code Analysis

I. INTRODUCTION

Generative Artificial Intelligence has become an important technology in modern software development. Large Language Models can generate functions, classes, scripts, database queries, test cases, and complete application components from natural-language instructions. AI-assisted programming tools can therefore reduce development effort and support developers in many programming tasks.

Despite these advantages, AI-generated code cannot automatically be considered secure. AI models learn programming patterns from large collections of existing source code, which may contain insecure coding practices. As a result, generated programs can reproduce vulnerabilities such as SQL injection, cross-site scripting, command injection, improper authentication, insecure cryptographic practices, and inadequate input validation.

Traditional vulnerability-detection approaches include static analysis, dynamic analysis, rule-based systems, and manual code review. These approaches remain important but may have difficulties identifying complex vulnerabilities that depend on program context and semantic relationships.

Machine Learning provides a data-driven alternative. ML models can learn patterns from vulnerable and non-vulnerable programs and classify previously unseen code. Deep Learning further improves this process by learning representations automatically from source code. Recent research has investigated CNN, LSTM, GNN, Transformer, and LLM-based approaches for vulnerability detection.

A recent systematic review of LLM-based vulnerability detection and repair analyzed 58 studies and highlighted the increasing role of LLMs in software-security research [1]. Similarly, recent surveys of deep-learning-based

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, October 2026
Impact Factor: 8.2

vulnerability detection demonstrate the transition from manually engineered features toward token, graph, and semantic representations [2].

This paper therefore reviews existing ML-based vulnerability-detection techniques and discusses their applicability to AI-generated source code.

The major objectives are:

- To review ML and DL techniques used for vulnerability detection.
- To analyze vulnerability detection approaches applicable to AI-generated code.
- To identify commonly used datasets and source-code representations.
- To analyze major research challenges.
- To propose future research directions for AI-generated code security.

II. LITERATURE REVIEW

A. Traditional Vulnerability Detection

Traditional vulnerability detection primarily depends on predefined rules, signatures, static analysis, dynamic analysis, and manual inspection.

Static Application Security Testing (SAST) analyzes source code without executing the program. It can efficiently identify known insecure programming patterns. However, rule-based approaches may generate false positives and may not detect vulnerabilities requiring complex semantic understanding.

Dynamic analysis executes software and examines its behavior. Although it can identify runtime vulnerabilities, it can be computationally expensive and may not cover all execution paths.

These limitations motivated researchers to investigate Machine Learning approaches capable of learning vulnerability characteristics from historical source-code examples.

B. Machine Learning-Based Approaches

ML-based vulnerability detection generally follows the pipeline:

Source Code → Feature Extraction → ML Model → Vulnerability Classification

Common algorithms include:

- Support Vector Machine
- Random Forest
- Decision Tree
- Logistic Regression
- Naïve Bayes
- Gradient Boosting

Early approaches extracted features such as token frequencies, API calls, code complexity, function characteristics, and syntactic patterns.

Traditional ML models are comparatively efficient and easier to train than large neural models. However, their performance depends heavily on feature engineering. A manually designed feature set may not capture all forms of a vulnerability.

C. Deep Learning-Based Detection

Deep Learning enables models to learn representations directly from source code. CNN, LSTM, GNN, and Transformer architectures have been widely investigated.

CNN models can identify local patterns in source-code sequences, while LSTM models can capture sequential dependencies. GNN-based methods represent programs as graphs and learn relationships among program elements.

Recent research surveys indicate that deep-learning vulnerability detection commonly uses token-based, syntax-based, and graph-based representations [2].

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, October 2026
Impact Factor: 8.2

The general deep-learning process is:

Source Code → Representation → Deep Learning Model → Vulnerability Prediction

Deep learning reduces dependence on manually engineered features but generally requires larger datasets and greater computational resources.

D. Graph-Based Vulnerability Detection

Source code contains multiple structural relationships. These can be represented using:

- Abstract Syntax Trees (AST)
- Control Flow Graphs (CFG)
- Data Flow Graphs (DFG)
- Program Dependency Graphs (PDG)
- Code Property Graphs (CPG)

Graph-based methods are particularly useful because vulnerabilities often depend on relationships between variables, statements, functions, and data flows.

The Devign dataset is an important example of a real-world vulnerability dataset used in research. It contains function-level code samples collected from open-source projects and labeled according to vulnerability status [6].

Graph Neural Networks can learn relationships between nodes and edges, but graph construction can be computationally expensive.

E. Transformer and LLM-Based Approaches

Transformer models have significantly changed source-code analysis. Their attention mechanism allows them to model relationships between distant tokens.

A simplified Transformer-based detection process is:

Source Code → Tokenization → Transformer → Contextual Representation → Classifier

Large Language Models extend this capability by processing both natural-language instructions and source code. LLMs can potentially identify vulnerabilities, classify vulnerability types, explain suspicious code, and suggest remediation.

However, LLM-based detection also presents challenges including hallucination, prompt sensitivity, inconsistent results, computational requirements, and limited explainability [1].

F. Vulnerability Detection in AI-Generated Code

AI-generated code differs from traditional human-written code because it is produced from prompts and learned programming patterns. Generated code may contain:

- Missing input validation
- Insecure API usage
- Hard-coded credentials
- Improper authentication
- Unsafe database queries
- Incorrect security assumptions
- Vulnerable dependencies
- Incomplete security mechanisms

Research specifically addressing AI-generated code has identified vulnerability detection and repair as important open problems [3].

DeVAIC was proposed as a tool for security assessment of AI-generated Python code and addresses challenges associated with incomplete generated code [4]. Other work has investigated LLM-based vulnerability detection in Python codebases, demonstrating the potential of LLMs for automated security analysis [5].

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, October 2026
Impact Factor: 8.2

These studies suggest that conventional vulnerability datasets alone may not be sufficient. Dedicated AI-generated code datasets are required to understand security patterns introduced by generative models.

G. Summary of Key Reviewed Studies

Besides the individual approaches discussed above, several surveys and tools provide the foundation for this review. Broader surveys of large language models in software security [7] and of modern machine learning, deep learning and LLM-based vulnerability detection [10] give additional overviews of the field. Table I summarises the main studies used in this paper and their relevance to AI-generated code.

TABLE I: Summary of key studies reviewed in this paper.

Ref.	Study	Focus	Relevance to this work
[1]	Zhou et al.	Literature review of LLMs for vulnerability detection and repair	Shows the growing role of LLMs; supports the hybrid ML-LLM direction
[2]	Liang et al.	Survey of deep-learning-based source code vulnerability analysis	Guides the choice of token, graph and semantic representations
[3]	Kaniewski et al.	Vulnerability handling of AI-generated code	Identifies detection and repair of AI-generated code as open problems
[4]	DeVAIC	Security assessment tool for AI-generated Python code	Addresses incomplete generated code
[5]	SecureQwen	LLM-based vulnerability detection in Python codebases	Shows the potential of LLMs for automated security analysis
[6]	Devign	Graph-neural-network vulnerability identification using function-level data	Baseline real-world dataset and graph-based method
[7]	Sheng et al.	Survey of LLMs in software security	Wider view of LLM-based detection techniques
[10]	Electronics survey	Survey of ML, DL and LLM-based vulnerability detection	Supports the comparison of approaches in Section III

H. Common Vulnerability Categories in AI-Generated Code

Security guidance such as the OWASP Top 10 [9] and the OWASP Top 10 for Large Language Model Applications [8] provides widely used descriptions of software security risks. Table II maps vulnerability categories that are frequently discussed for generated code to their Common Weakness Enumeration (CWE) identifiers and shows how each may appear. The detection difficulty column is an illustrative judgement based on whether a pattern is mostly lexical or depends on program semantics. Fig. 1 groups the same categories by how much their detection depends on surface patterns, data flow or program semantics.

TABLE II: Illustrative vulnerability categories in AI-generated code.

Vulnerability	CWE	How it may appear in generated code	Detection difficulty
SQL injection	CWE-89	Queries built by concatenating user input	Medium (needs data flow)
Cross-site scripting	CWE-79	User input written to HTML without escaping	Medium
OS command injection	CWE-78	User input passed to shell commands	Medium
Hard-coded credentials	CWE-798	Keys or passwords placed directly in sample code	Low (pattern-based)
Improper authentication	CWE-287	Missing or weak login and session checks	High (semantic)

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, October 2026
Impact Factor: 8.2

Missing input validation	CWE-20	No type, length or range checks	High (semantic)
Weak cryptography	CWE-327	Outdated hash or cipher algorithms	Low to medium
Path traversal	CWE-22	File paths taken from user input unchecked	Medium

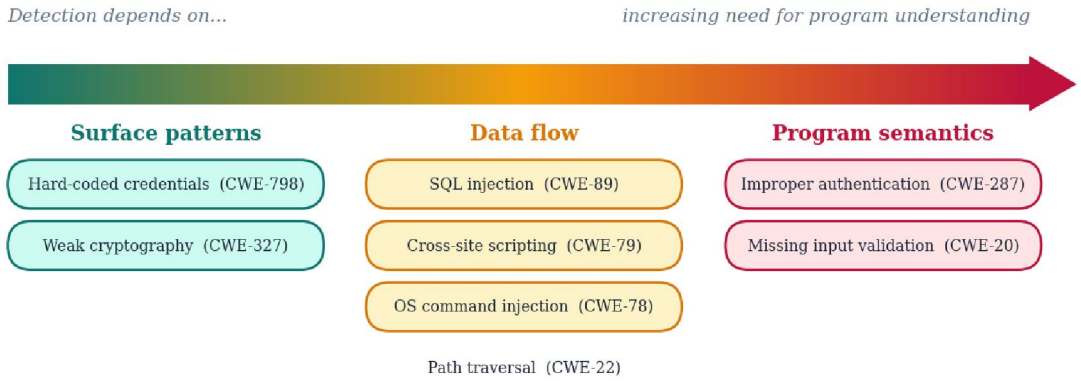


Fig. 1 Vulnerability categories arranged by how much their detection depends on program understanding (illustrative).

III. COMPARATIVE ANALYSIS

Fig. 2 shows how detection approaches have developed from fixed rules to models that learn from code structure and context. Table III summarises the main approaches reviewed in this paper, with their typical code representation, advantages and limitations.

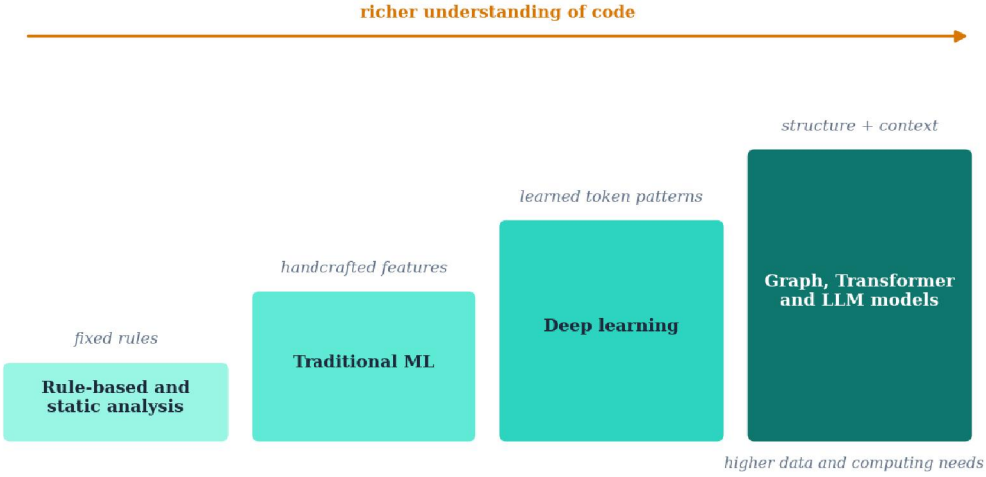


Fig. 2 Development of vulnerability detection approaches (illustrative).

TABLE III: Comparison of vulnerability detection approaches.

Approach	Representation	Advantages	Limitations
Rule-Based	Security rules	Simple and explainable	Limited adaptability
Traditional ML	Handcrafted features	Efficient	Feature engineering required
CNN	Token sequence	Detects local patterns	Limited global context
LSTM	Sequential code	Captures dependencies	Computational complexity

Copyright to IJARSCT
www.ijarsct.co.in
DOI: 10.48175/IJARSCT-38851

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, October 2026
Impact Factor: 8.2

GNN	AST/CFG/DFG	Captures program structure	Graph construction cost
Transformer	Token/context	Captures long-range relationships	High resource requirements
LLM	Code + context	Strong semantic understanding	Hallucination and reliability issues
Hybrid Models	Multiple representations	Combines complementary information	Increased complexity

The literature demonstrates a clear progression from rule-based approaches toward ML, DL, graph-based learning, and LLM-based approaches.

The comparison leads to the following observations.

• **Representation matters.** Methods that use only token sequences miss program structure, whereas graph-based and Transformer-based representations capture structure and long-range context at a higher computational cost.

• **LLMs need validation.** LLMs offer strong semantic understanding, but hallucination and inconsistent output mean that their results need validation before being trusted.

• **Datasets lag behind the problem.** Most available datasets contain human-written code, and very few studies target code produced by AI models [3].

• **Explanations are missing.** Detection results are rarely accompanied by reasons, although developers need to know why a piece of code was flagged before they change it.

• **Hybrid designs are promising.** Hybrid designs combine complementary strengths but add complexity, which favours a modular framework in which individual components can be replaced.

IV. PROPOSED METHODOLOGY

Based on the literature review, this paper proposes a conceptual framework specifically designed for vulnerability detection in AI-generated source code.

A. Proposed Framework

The framework consists of the following stages (Fig. 3):

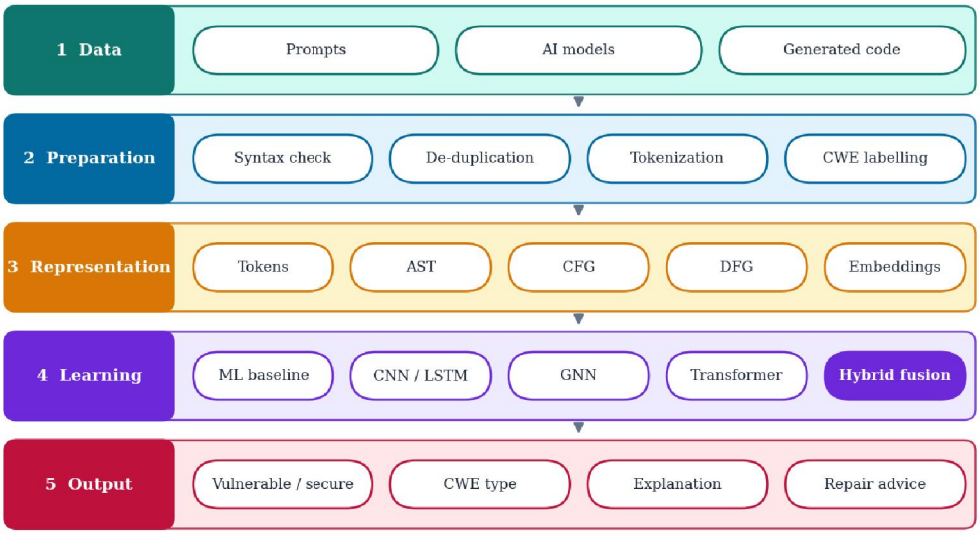


Fig. 3 Layered architecture of the proposed framework.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, October 2026
Impact Factor: 8.2

B. Dataset Collection

AI-generated code can be collected using different AI coding models and different prompts. Each record should contain:

- Programming language
- Prompt
- Generated code
- AI model used
- Vulnerability label
- CWE category
- Static-analysis result
- Human verification

Both vulnerable and secure code should be included.

Existing datasets such as Devign and Juliet can be used as baseline datasets, but additional AI-generated samples are necessary for domain-specific evaluation.

C. Preprocessing

The preprocessing stage includes:

- Syntax validation.
- Duplicate removal.
- Tokenization.
- AST generation.
- Control-flow extraction.
- Data-flow extraction.
- Vulnerability labeling.

AI-generated code may sometimes be incomplete. Therefore, syntax correction or controlled completion may be necessary before analysis.

D. Feature Representation

A hybrid representation is recommended.

- **Token Features:** Capture lexical patterns.
- **AST Features:** Capture syntactic structure.
- **CFG Features:** Capture execution paths.
- **DFG Features:** Capture relationships between variables.
- **Transformer Embeddings:** Capture semantic relationships.

Combining structural and semantic information may improve detection compared with relying on a single representation.

E. Model Training and Classification

After representations are extracted, the framework uses classifiers at two levels. A binary classifier first decides whether a code sample is vulnerable or secure, and a multi-class classifier then assigns the CWE category. Traditional models such as Random Forest and Support Vector Machine trained on handcrafted features serve as efficient baselines. CNN, LSTM, GNN and Transformer models are trained on token, graph and contextual representations, and a hybrid model fuses the learned embeddings before the final classification layer.

The data should be split into training, validation and test sets so that code from the same project, prompt or AI model does not appear in more than one set. This prevents inflated results caused by near-duplicate samples. Class imbalance is handled by class weighting or oversampling, and hyperparameters are tuned on the validation set only.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, October 2026
Impact Factor: 8.2

F. Explainability and Security Validation

A detection result is more useful when the developer can see why it was produced. Explainable AI techniques can highlight suspicious statements, APIs, variables and data-flow paths. In the validation stage, static-analysis tools and manual review by security-aware developers confirm the detected vulnerabilities and reduce false positives. Confirmed results are stored as new labelled data, so that the system can be retrained as new patterns appear. Fig. 4 shows the complete detection-to-repair workflow.

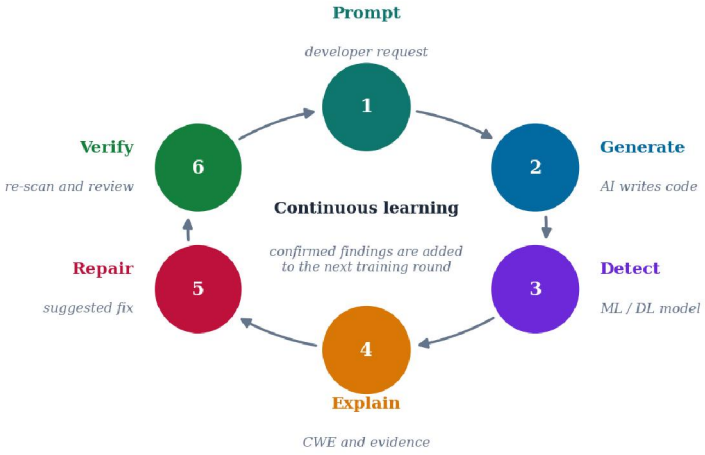


Fig. 4 Detection-to-repair cycle with continuous learning.

G. Datasets

No single public resource covers vulnerabilities in AI-generated code, so training combines several sources (Table IV). Real-world datasets provide realistic samples, controlled test suites support evaluation of specific weakness categories, and a purpose-built AI-generated dataset addresses the gap discussed in Section VI.

TABLE IV: Data sources for training and evaluation.

Source	Type of data	Role in the framework
Devign [6]	Function-level code from open-source projects with vulnerability labels	Baseline training and comparison
Juliet Test Suite	Test cases covering known weakness categories	Controlled evaluation of CWE classification
AI-generated dataset (proposed)	Code generated by several AI models from varied prompts, labelled and verified by humans	Domain-specific training and evaluation

H. Evaluation Metrics

Accuracy alone is not enough for vulnerability detection, because secure samples usually outnumber vulnerable ones. Table V lists the measures recommended for evaluating the framework, where TP, FP, TN and FN denote true positives, false positives, true negatives and false negatives.

TABLE V: Metrics for evaluating the framework.

Metric	Definition	Why it matters
Precision	TP / (TP + FP)	Shows how many alerts are real vulnerabilities
Recall	TP / (TP + FN)	Shows how many vulnerabilities are found
F1-score	2 x Precision x Recall / (Precision + Recall)	Balances precision and recall

Copyright to IJARSCT
www.ijarsct.co.in
DOI: 10.48175/IJARSCT-38851

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 2, October 2026 Impact Factor: 8.2

False positive rate	FP / (FP + TN)	Indicates alert noise and developer trust
ROC-AUC	Area under the ROC curve	Compares models across decision thresholds
PR-AUC	Area under the precision-recall curve	More informative under class imbalance

V. APPLICATIONS

The framework can support several stages of AI-assisted software development. Table VI summarises the main settings in which ML-based detection of vulnerabilities in generated code can be used.

TABLE VI: Application areas of vulnerability detection in AI-generated code.

Setting	Use of vulnerability detection	Benefit
IDE integration	Scan AI suggestions before they are accepted	Problems are caught at the earliest stage
Code review	Flag risky AI-generated changes in pull requests	Reduces the workload of reviewers
CI/CD (DevSecOps)	Act as an automated security gate in build pipelines	Consistent checks on every change
Education	Give feedback to students who use AI coding assistants	Builds secure coding habits
Security auditing	Rank files and functions by likelihood of vulnerability	Makes audits of large codebases efficient

A. Responsible use of detection tools

- **Developer responsibility.** The tool should assist developers, who remain responsible for the final decision to accept or reject code.
- **Trust and false positives.** Excessive false alarms reduce trust, so alerts should include confidence information and explanations.
- **Confidentiality of code.** Source code may contain proprietary logic or secrets, so analysis should use access controls or on-premise deployment where needed.
- **Over-reliance.** A clean result is not a guarantee of security; detection should be combined with testing and code review.
- **Evolving threats.** Code generators and attack techniques change over time, which calls for regular retraining and evaluation on new samples.

VI. RESEARCH CHALLENGES

A. Limited AI-Generated Datasets

Most existing vulnerability datasets consist primarily of human-written code. Large, diverse, accurately labeled AI-generated vulnerability datasets are still required.

B. Class Imbalance

Some vulnerability types occur much less frequently than secure code. This can bias classification models toward the majority class.

Possible approaches include class weighting, oversampling, undersampling, and synthetic data generation.

C. False Positives

Security systems generating excessive false alarms may reduce developer trust. Therefore, practical systems should provide confidence information and explanations.

D. Explainability

Developers need to understand why a model identifies particular code as vulnerable. Explainable AI can highlight suspicious statements, APIs, variables, and data-flow paths.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, October 2026
Impact Factor: 8.2

E. Programming-Language Dependency

Vulnerability patterns differ across programming languages. A model trained on one language may not perform equally well on another.

F. Computational Cost

Large Transformer and LLM models require considerable computational resources. Developing lightweight models suitable for real-time development environments remains an important research problem.

Table VII relates each challenge to the component of the proposed framework that responds to it.

TABLE VII: Research challenges and the corresponding framework response.

Challenge	Effect on detection	Response in the proposed framework
Limited AI-generated datasets	Models trained on human-written code may not transfer	Purpose-built AI-generated dataset with human verification (Section IV-G)
Class imbalance	Bias towards the secure class	Class weighting or oversampling; reporting of recall and PR-AUC
False positives	Loss of developer trust	Security validation stage and confidence-aware reporting
Explainability	Opaque alerts cannot be acted on	Explanation output that highlights statements, APIs and data flows
Programming-language dependency	Poor transfer across languages	Structural representations (AST, CFG, DFG) with per-language fine-tuning
Computational cost	Large models are hard to run inside editors	Lightweight models for real-time use, larger models for batch analysis

VII. CONCLUSION

Generative AI and Large Language Models are rapidly changing software development, but their adoption also introduces new software-security challenges. AI-generated code can contain vulnerabilities caused by insecure programming practices, inadequate validation, unsafe APIs, and incomplete security mechanisms.

This literature review examined the development of vulnerability detection from traditional rule-based methods to Machine Learning, Deep Learning, graph-based methods, Transformers, and LLM-based approaches. The literature indicates that ML and DL methods can identify vulnerability patterns, while graph-based and Transformer models provide additional structural and contextual information. Recent research has also demonstrated the potential of LLMs for detecting and repairing vulnerabilities in source code.

However, significant challenges remain, particularly the lack of representative AI-generated datasets, class imbalance, false positives, explainability, language dependency, and computational cost.

A future vulnerability-detection system should therefore integrate multiple sources of information, including lexical features, program structure, data flow, semantic representations, and LLM-based reasoning. Such a system should not only detect vulnerabilities but also classify their CWE category, explain the cause, recommend a secure repair, and verify the repaired code.

The development of reliable ML-based security mechanisms will be important for enabling safer and more trustworthy adoption of AI-assisted software development.

VIII. FUTURE SCOPE

Several research directions can extend current vulnerability-detection systems.

• **Hybrid ML–LLM Models.** Future systems can combine static analysis, GNNs, Transformers, and LLMs (Static Analysis + GNN + Transformer + LLM Explanation). This may combine structural analysis with semantic understanding.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 2, October 2026 Impact Factor: 8.2

• **Real-Time IDE Integration.** Security analysis can be integrated directly into AI-assisted programming environments. Generated code could be analyzed before being accepted into a software project.
• **Automated Vulnerability Repair.** Future systems should progress from detection to classification, explanation, repair and verification. The generated patch should be automatically analyzed again to verify that the repair does not introduce another vulnerability.
• **Security-Aware Code Generation.** Security requirements could be incorporated during code generation instead of relying exclusively on post-generation analysis.
• **Continual Learning.** Vulnerability detection systems should continuously learn from newly discovered CVEs, CWEs, secure coding practices, and emerging AI-generated vulnerability patterns.
• **Multilingual Detection.** Future models should support multiple programming languages including Python, Java, C/C++, JavaScript, PHP, and others.

ACKNOWLEDGMENT

The authors sincerely thank their guide, Prof. Rana Afreen Sheikh, for her valuable guidance and support during the preparation of this paper.

REFERENCES

1. X. Zhou, S. Cao, X. Sun, and D. Lo, "Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead," *ACM Transactions on Software Engineering and Methodology*, 2025.
2. C. Liang, Q. Wei, J. Du, Y. Wang, and Z. Jiang, "Survey of Source Code Vulnerability Analysis Based on Deep Learning," *Computers & Security*, vol. 148, 2025, Art. 104098.
3. S. Kaniewski, D. Holstein, F. Schmidt, and T. Heer, "Vulnerability Handling of AI-Generated Code—Existing Solutions and Open Challenges," arXiv:2408.08549, 2024.
4. "DeVAIC: A Tool for Security Assessment of AI-Generated Code," *Information and Software Technology*, vol. 177, 2025, Art. 107572.
5. "SecureQwen: Leveraging LLMs for Vulnerability Detection in Python Codebases," *Computers & Security*, 2025.
6. Y. Zhou et al., "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," *Advances in Neural Information Processing Systems*, 2019.
7. Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, "LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights," *ACM Computing Surveys*, 2025.
8. OWASP Foundation, "OWASP Top 10 for Large Language Model Applications," 2025.
9. OWASP Foundation, "OWASP Top 10:2025," 2025.
10. "Modern Approaches to Software Vulnerability Detection: A Survey of Machine Learning, Deep Learning, and Large Language Models," *Electronics*, vol. 14, no. 22, 2025, Art. 4449.