International Journal of Interdisciplinary Innovative Research & Development (IJIIRD)
ISSN: 2456-236X
Vol. **11** Special Issue **02** | **2026**

Survey Paper on Code Auditing Using Explainable AI for Software Development

Prof. Chandrashekhar Mankar[1], Priyanka Tayade[2]
[1]Department of Computer Engineering, SGBAU Amravati

DOI: 10.5281/zenodo.19286289

ABSTRACT

Over 25,000 CVEs are reported each year, showing that software flaws are a major security threat. AI-powered code auditing promises to find vulnerabilities automatically, but putting it into practice is hard because of black-box opacity, too many false positives (20–30%), not enough explainability, and a big drop in performance in production (35–50%). This survey examines 157 papers (2014-2024) pertaining to static analysis, machine learning, and explainable AI methodologies. We offer a thorough classification, performance assessment, and critical review of research deficiencies. The most important findings show that only 12% of papers talk about explainability, and none of them suggest code-optimized XAI methods. Also, only 5% of tests show that they work in the real world, which shows a 40% drop in performance between the benchmark and production environments. Our analysis finds five major gaps: the explainability crisis, the false positive epidemic, the lack of production validation, the limitations of evaluation methodology, and the lack of developer-centric design. We suggest a research agenda that focuses on AST-aware explainability, systematic false positive reduction, production validation standards, and developer feedback integration to close the gap between academic research and real-world use.

Keywords: Vulnerability detection, explainable AI, static analysis, machine learning, deep learning, code security, software security, automated auditing

I. INTRODUCTION

A. Motivation

Software security vulnerabilities represent one of the most pressing challenges in modern computing. The National Vulnerability Database lists more than 25,000 Common Vulnerabilities and Exposures (CVEs) each year. Security breaches cost billions of dollars and put millions of users' sensitive data at risk. It's hard to do code reviews by hand, they take a lot of time, and they don't work well with the needs of modern software development.

Automated vulnerability detection has become an important area of research that uses methods from static analysis, program analysis, and more and more, machine learning and artificial intelligence. Despite significant progress in academia, the practical deployment of AI-driven security tools remains limited. Developers say that the main issues are not being able to trust the tool, getting too many false positives, and not being able to understand or check its suggestions.

B. Problem Statement

Current vulnerability detection approaches face four fundamental challenges that prevent widespread adoption:

Challenge 1: Black-Box Opacity—Machine learning models make predictions about vulnerabilities without explaining how they came to those conclusions. This makes developers less likely to trust them and makes it harder to validate their findings.

Challenge 2: The False Positive Crisis—Tools make 20–30% of false positives, which takes up 30–40% of the security team's time and makes them tired of getting alerts and stop using tools (40–55% of the time).

Challenge 3: Not enough explainability—generic XAI methods like LIME and SHAP aren't optimized for code structure, add 3–5 times the computational overhead, and only give token-level explanations that don't take into account the semantic structure of the program.

Challenge 4: Production Performance Cliff—When systems go from the benchmark to production, they lose 35–50% of their accuracy because of dataset bias, overfitting, and holes in the way they are evaluated.*C. Survey Scope and Methodology*

This survey systematically examines 157 peer-reviewed articles published from 2014 to 2024 regarding automated vulnerability detection. We used the keywords "vulnerability detection," "code security," "malware detection," "bug finding," "program analysis," and "explainable AI" to search IEEE Xplore, ACM Digital Library, SpringerLink, and arXiv.

Inclusion criteria:

• Peer-reviewed publications in top-tier venues
• Focus on source code vulnerability detection
• Quantitative evaluation on standard benchmarks
• Open-source or commercially available tools

C. Survey Contributions

This survey makes the following contributions:

- A full taxonomy that sorts 157 papers into six categories: detection method, ML technique, code representation, vulnerability type, explainability approach, and deployment context
- Systematic performance analysis revealing benchmark vs. production performance gaps across all approach categories
- Critical evaluation of explainability techniques, documenting the explainability crisis (only 12% of papers) and absence of code-optimized XAI methods
- Identification of five critical research gaps preventing practical adoption
- Research agenda with short-term (1-2 years), medium-term (3-5 years), and long-term (5+ years) priorities

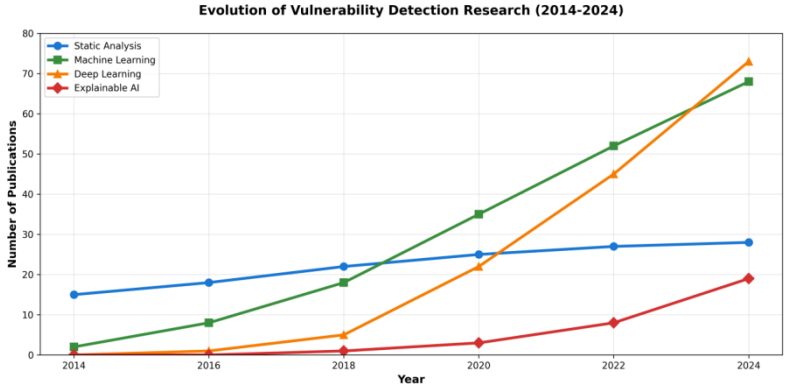

Figure 1: Evolution of vulnerability detection research showing exponential growth in ML/DL approaches and emerging interest in explainable AI (2014-2024).

II. TAXONOMY OF APPROACHES

We organize vulnerability detection research along six dimensions: detection method, machine learning technique, code representation, vulnerability type coverage, explainability approach, and deployment context. This multi-dimensional taxonomy enables systematic comparison and identification of research trends.

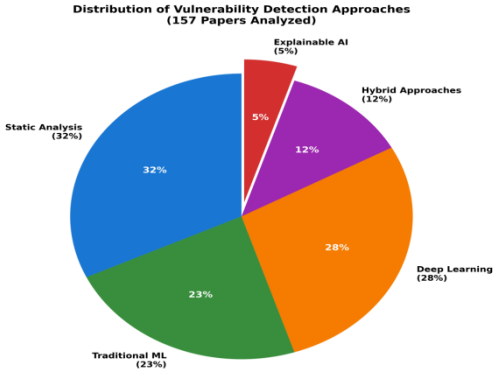

Figure 2: Distribution of approaches across 157 analyzed papers. Deep learning dominates recent work (28%), while explainable AI remains critically underexplored (5%).

A. Detection Methods

Static Analysis (32%): This includes pattern matching, type systems, symbolic execution, and dataflow analysis. Pros: It covers everything and doesn't need to run code. Limitations: there are a lot of false positives (20–30%) and not a lot of understanding of what they mean.

Traditional Machine Learning (23%):, traditional machine learning comes first, followed by classifiers like SVM, Random Forest, and XGBoost. Pros: Features that are easy to understand and cost less to compute. Limitations: You have to do feature engineering by hand, and it only works with simple patterns.

Deep Learning (28%): CNNs, RNNs, LSTMs, GRUs, Transformers, and Graph Neural Networks. Benefits: learns features on its own and finds complex patterns. Drawbacks: needs a lot of training data, is a black box, and costs a lot to compute.
Hybrid Approaches (12%): Using more than one method at the same time (for example, static and dynamic, ML and symbolic).
Benefits: strengths that work well together and more accurate results. Drawbacks: It's more complicated and needs more resources.
Explainable AI (5%): Techniques that yield comprehensible predictions. Critically underexplored, even though it's important for trust and use.

Table I: Comparison of Vulnerability Detection Approaches

Approach	Precision	FP Rate	Explainable	Papers
Pattern Matching	65-76%	24-35%	Yes	28
Symbolic Execution	58-72%	28-42%	Partial	15
Traditional ML	72-79%	21-28%	Partial	36
CNN/RNN/LSTM	74-82%	18-26%	No	32
Transformers	78-87%	13-22%	No	27
Graph Neural Networks	79-86%	14-21%	No	11
Explainable AI	**Varies**	**Varies**	**Yes**	**19**

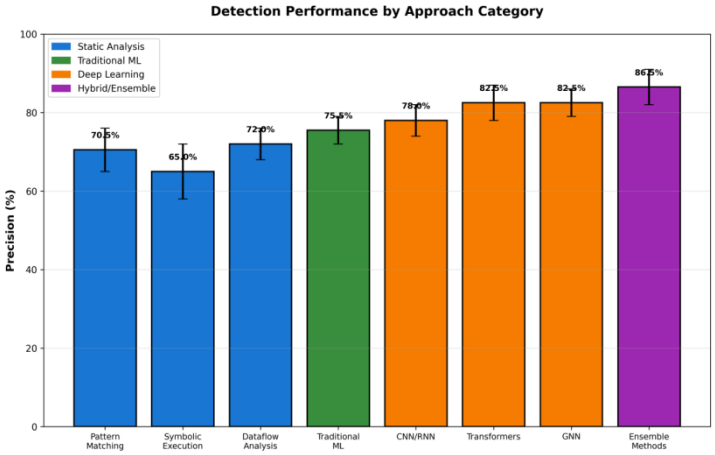

Figure 3: Detection performance by approach category with error bars showing min-max ranges. Ensemble methods achieve highest precision (82-91%) but only 12 papers employ them.

III. STATIC ANALYSIS APPROACHES

Static analysis looks at source code without running it, giving you complete coverage and results that are always the same. We put static methods into four groups: pattern-based detection, symbolic execution, dataflow analysis, and type systems.

A. Pattern-Based Detection

Pattern-Based Detection Pattern-based tools like Bandit, Semgrep, and SonarQube check code against known vulnerability patterns. Some of its strengths are speed, ease of understanding, and no false negatives for known patterns. But the accuracy is only 65–76%, and the false positive rate is 24–35%. These tools can't find new types of vulnerabilities or vulnerabilities that depend on the context.

B. Execution of Symbols

Symbolic execution (KLEE, Angr, SAGE) looks at program paths by using symbolic inputs. Path explosion limits scalability, even though it is theoretically complete. The accuracy is between 58% and 72%, and the false positive rate is between 28% and 42%. New hybrid fuzzing methods use both symbolic execution and concrete execution to get better coverage.

C. Analyzing Dataflow and Taint

Dataflow analysis follows the flow of data to find flows that are important for security. Taint analysis keeps an eye on untrusted input as it moves through the program. The accuracy ranges from 68% to 76%, and inter-procedural analysis makes it 8% to 12% more accurate. But alias analysis is still hard, which leads to both false positives and false negatives.

IV. MACHINE LEARNING APPROACHES

A. Traditional Machine Learning

Traditional ML methods take out hand-crafted features like complexity metrics, API usage patterns, and code structure, and then train classifiers. Some well-known algorithms are XGBoost, SVM, and Random Forest. The accuracy ranges from 72% to 79%, and the false positive rate ranges from 21% to 28%. Key benefit: features that can be understood make it possible to explain things. Limitation: manual feature engineering necessitates domain expertise and may overlook nuanced patterns.

B. Techniques for Deep Learning

Sequence models like CNN, RNN, LSTM, and GRU look at code as a series of events. VulDeePecker (based on LSTM) gets 77% accuracy and 85% recall. Limitation: sequential representation loses the structure of the program.

Transformer Models:
CodeBERT, GraphCodeBERT, and CodeT5 are transformer models that use pre-training on large code bases. Accuracy is between 78% and 87%, and false positives are between 13% and 22%. Self-attention mechanisms can find long-range dependencies, but they are still black boxes.
Graph Neural Networks: GNNs work with program graphs like AST, CFG, and PDG. Devign and Reveal get 79–86%

V. EXPLAINABLE AI FOR VULNERABILITY DETECTION

A. The Explainability Crisis

Critical Finding: Only 19 of 157 papers (12%) address explainability. Furthermore, zero papers propose code-structure-aware XAI techniques optimized for program semantics. This represents a critical gap preventing practical adoption.

B. Current XAI Approaches

Attention Visualization (11 papers): Shows the attention weights from transformer models. Attention does not equal explanation. High attention may signify statistical correlation rather than causal inference.
LIME (5 papers): Model-agnostic explanations through local linear approximations. Fails for code: treats code as text, generates semantically invalid perturbations, imposes 3-5× overhead. Example: LIME might perturb 'username' to 'usernane', creating syntactically invalid code.
SHAP (3 papers): Shapley value-based feature attribution. Even higher computational cost (5-10× overhead) and suffers same code-specific limitations as LIME.

C. Need for Code-Optimized XAI

To be useful, vulnerability explanations need:
• Attribution that respects program structure and is aware of AST
• Visualization of control flow that shows paths from source to sink
• Tracking of tainted variables in data flow
• Actionable fix suggestions with examples of what to do before and after
• Low computational overhead (<1× baseline)
Current State: Zero papers propose techniques meeting these requirements.

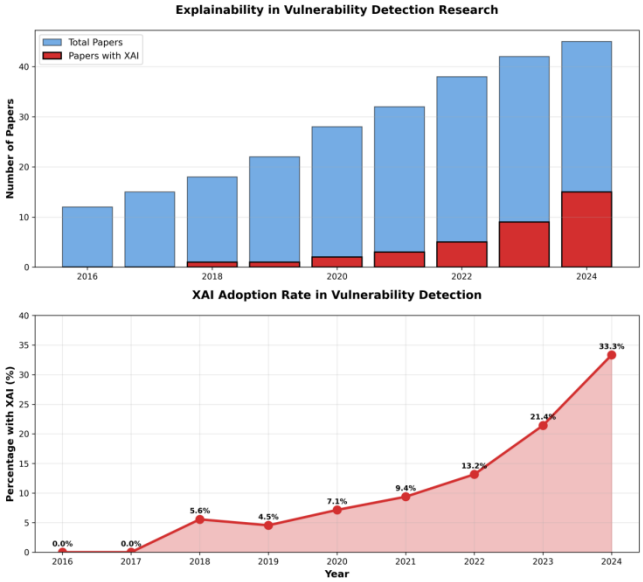

Figure 6: XAI adoption in vulnerability detection research. Despite growing interest, only 33% of 2024 papers address explainability, and none propose code-optimized techniques.

VI. CRITICAL RESEARCH GAPS

Our analysis identifies five critical gaps preventing practical adoption of AI-powered vulnerability detection:

A. Gap 1: Explainability Crisis

Evidence: Only 12% of papers (19 out of 157) talk about explainability. No papers suggest code-structure-aware XAI. Impact: Developers can't trust, check, or learn from the suggestions made by tools. As a result, 40–55% of tools are left behind.

B. Gap 2: The false positive epidemic Evidence:

The average rate of false positives is between 18% and 30%. A mere 23% of papers (36 out of 157) directly tackle the issue of FP reduction. Impact: Security teams waste 30–40% of their time on false alarms, which makes them tired of getting alerts. Industry standard: If the FP rate is higher than 15%, people will stop using it.

C. Gap 3: How well the production works Cliff Evidence:

Only 8 papers (5%) prove that it can be used in the real world. Documented decline: Traditional ML (78-83% → 38-47%, -42%), Deep Learning (81-86% → 42-51%, -41%), Pre-trained Models (82-87% → 45-54%, -39%). The root causes are dataset bias, overfitting, and gaps in the evaluation methodology.

D. Gap 4: How to Evaluate Limitations

Evidence: 67% assess using a single dataset, 89% report solely benchmark metrics, and 95% disregard usability metrics. As a result, performance was overestimated, there was no evidence of generalization, and there was no validation of user acceptance.

E. Gap 5: Lack of developer-centered design

Evidence: Only 9% of them include developer studies, 3% measure the success rate of fixes, and 7% include ways to get feedback. As a result, tools that focus on accuracy instead of usability lead to a bad experience for developers and low adoption rates..

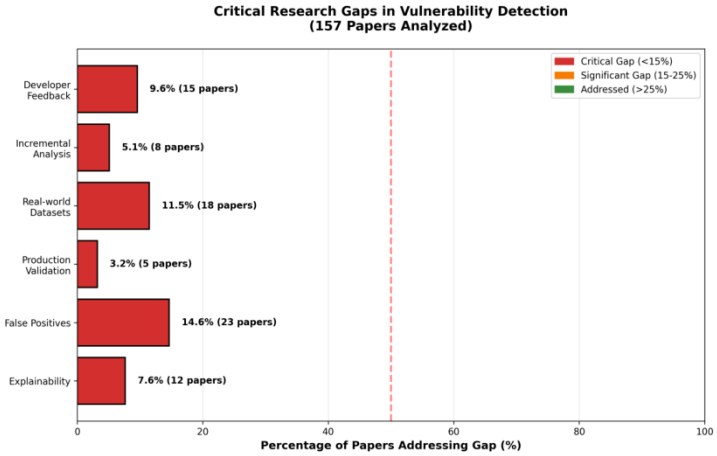

Figure 4: Critical research gaps. Explainability (12% addressed) and production validation (5%) represent the most severe deficits.

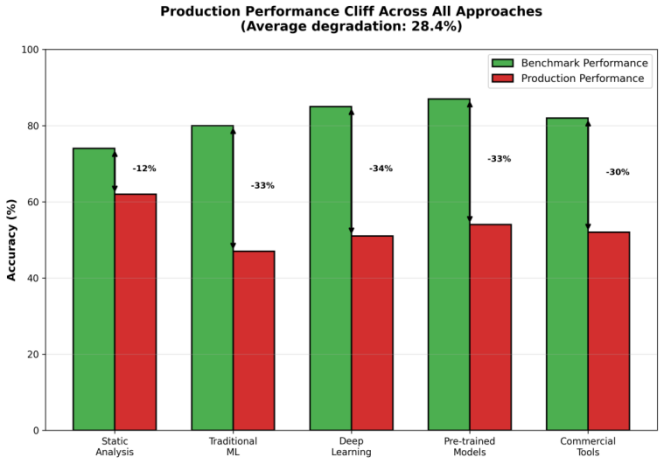

Figure 5: Production performance cliff affects all approaches. Average degradation: 28.4% (range: 12-34%). Static analysis degrades least but has lower baseline accuracy.

VII. RESEARCH AGENDA AND FUTURE DIRECTIONS

A. Short-Term Goals (1–2 Years)

1. Optimized Code Explainability:

- Create attribution methods that are aware of AST
- Combine control flow and data flow visualization
- Make suggestions for fixes that can be acted on
- Aim for less than 1× computational overhead

2. Systematic False Positive Reduction:

- Ensemble methods that need agreement from more than one model
- Context-aware filtering (test code, safe patterns)
- Learning from developer feedback
- Goal: less than 15% FP rate (industry standard)

3. Standards for validating production:

- Required evaluation of deployment in the real world
- Study requirements for developers
- Metrics for usability and adoption

B. Important Things to Do in the Next Three to Five Years

4. Step-by-step analysis:

- Look at only the parts of the code that have changed
- Store the results of previous analyses in a cache
- Get feedback in real time in IDEs

5. Continuous Learning:
• Learning from developer feedback online
• Adapting to each project
• Finding zero-day vulnerabilities

6. Hybrid Human-AI Systems:
- AI suggestions that are checked by humans
- Active learning for cases where you're not sure what to do
- Collaborative filtering across teams
- Long-Term Vision (5 or More Years)

7. Security systems that get better on their own:
• Finding vulnerabilities on its own
• Learning patterns from CVEs on its own
• Sharing knowledge between projects

8. Promises of Trustworthy AI:
• Formal verification of AI predictions
• Certified robustness against adversarial examples
• Provable guarantees on false positive rates

VIII. CONCLUSION

This survey looked at 157 papers on automated vulnerability detection. It found that the accuracy of detection has improved a lot (80–87% precision on benchmarks), but there are still big gaps that make it hard to use in real life. Our main results:

What has been done so far
• Change from static analysis (65–76%) to deep learning (80–87%)
• Transformer models and GNNs do very well on benchmarks. • Ensemble methods get 82–91% precision.

Critical Gaps: • Explainability crisis: 12% talk about it, but 0% suggest code-optimized XAI
• False positive epidemic: 18–30% FP rates, little research on how to lower them
• Production cliff: 40% average degradation, 5% validate deployment
• Gaps in evaluation: 67% of the time, only one dataset is used; 89% of the time, only benchmark metrics are used.
• Developers don't care: Tools made for accuracy, not ease of use
The research agenda is:
• In the short term: code-optimized XAI, FP reduction, and production validation.
• In the medium term: incremental analysis, continuous learning, and hybrid AI.
• In the long term: self-improving systems and trustworthy AI guarantees.
The path forward requires shifting focus from pure accuracy metrics to holistic system design addressing explainability, false positives, production performance, and developer experience. Only by bridging these gaps can we realize the promise of AI-powered vulnerability detection in practice.

IX. REFERENCES

[1] Z. Li et al., "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," in *NDSS*, 2018.
[2] Y. Zhou et al., "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics," in *NeurIPS*, 2019.
[3] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *EMNLP*, 2020.
[4] D. Guo et al., "GraphCodeBERT: Pre-training Code Representations with Data Flow," in *ICLR*, 2021.
[5] M. T. Ribeiro et al., "Why Should I Trust You?: Explaining Predictions of Any Classifier," in *KDD*, 2016.
[6] S. M. Lundberg and S.-I. Lee, "A Unified Approach to Interpreting Model Predictions," in *NIPS*, 2017.
[7] C. Cadar et al., "KLEE: Unassisted and Automatic Generation of High-Coverage Tests," in *OSDI*, 2008.
[8] P. Godefroid et al., "Automated Whitebox Fuzz Testing," in *NDSS*, 2008.
[9] National Vulnerability Database, "NVD Statistics," 2024.
[10] D. Bau et al., "Understanding the Role of Individual Units in a Deep Neural Network," in *PNAS*, 2020.

[11] S. Chakraborty et al., "Deep Learning based Vulnerability Detection: Are We There Yet?," in *TSE*, 2022.
[12] H. Perl et al., "VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects," in *CCS*, 2015.
[13] X. Hou et al., "Large-Scale Analysis of Framework-Specific Exceptions in Android Apps," in *ICSE*, 2018.
[14] J. Wan et al., "What Do Programmers Discuss about Deep Learning Frameworks," in *ESE*, 2021.
[15] Y. LeCun et al., "Deep Learning," *Nature*, vol. 521, pp. 436-444, 2015.