Policy Research Journal
ISSN (E): 3006-7030 ISSN (P) : 3006-7022 Volume 2, Issue 4, 2024

AUTOMATED CODE VULNERABILITY DETECTION USING STATIC ANALYSIS AND TRANSFORMER-BASED SOURCE CODE EMBEDDINGS

Syed Muhammad Junaid Hassan[*1], Muhammad Naseem[2], Ghulam Yasin[3]

[*1]*Assistant Professor, Department of Information Technology, Faculty of ICT, Balochistan University of Information Technology, Engineering and Management Sciences (BUITEMS)*
[2,3]*University of Makran*

[*1]smjunaid.it@gmail.com [2]naseem.ahmed@uomp.edu.pk [3]ghulam.yasin@uomp.edu.pk

DOI: https://doi.org/10.5281/zenodo.19876398

Keywords
Automated vulnerability detection, transformer models, static analysis, source code embeddings, Graph Code BERT, Code BERT, vulnerability localization, software security

Article History
Received: 04 November 2024
Accepted: 11 December 2024
Published: 27 December 2024

Corresponding Author: *
Syed Muhammad Junaid Hassan

Abstract

This paper provides a comprehensive review of automated code vulnerability detection methodologies integrating static analysis techniques with transformer-based source code embeddings, examining how hybrid systems are redefining software security assurance in an era of escalating software complexity and rapid open-source proliferation. The vulnerability lifecycle comprising black risk (discovery to disclosure), gray risk (disclosure to patching), and white risk (post-countermeasure) phases establishes the temporal urgency for automated detection, with transformer-based analysis achieving approximately twice the accuracy of statistical keyword approaches in identifying emerging threats before official notification. The transformer architecture's self-attention mechanism enables parallel processing and weighting of distant tokens, proving uniquely suited for code where variable definitions and subsequent misuse may span hundreds of lines. Specialized models examined include CodeBERT for bimodal natural language and programming language understanding, GraphCodeBERT integrating data flow graphs through graph-guided masked attention, and UniXcoder incorporating abstract syntax trees. Hybrid methodologies demonstrate substantial improvements: the SIExVulTS framework combining transformer-based source-sink identification, CodeQL propagation analysis, and GraphCodeBERT semantic verification increased precision from 22.61% to 87.23%, leading to discovery of six new CVEs in Apache projects. Granularity advances from function-level to statement-level localization through node classification over program dependency graphs and attention-based methods achieve Top-10 Accuracy of 0.65, significantly outperforming graph-based baselines. Explainable AI frameworks provide faithful justifications measured by mean statement precision and recall, while large language models generate natural language descriptions of root causes and mitigations. Empirical comparisons show transformer models achieving F1 scores of 0.75-0.79 on real-world projects, substantially outperforming industrial static analyzers scoring 0.26-0.55.

1. Introduction

The escalating complexity of modern software ecosystems, coupled with the rapid proliferation of open-source components, has rendered the identification of security vulnerabilities a critical priority in the software development lifecycle (Hassan et al., 2025). Traditional methodologies, while foundational, increasingly struggle to address the sheer volume and intricacy of code produced in contemporary environments (Khankhoje, 2022). As software systems become more interconnected, the window of exploitation particularly during the period between vulnerability discovery and public disclosure poses a profound threat to digital infrastructure (Gudepu & Jaladi, 2022). This phase, often characterized as the "black risk period," represents a critical interval where vulnerabilities remain undocumented and unpatched, yet potentially exploitable by sophisticated actors (Perez & Livshits, 2021). Consequently, the research community has pivoted toward automated detection frameworks that leverage the dual strengths of rigorous static analysis and the nuanced semantic understanding provided by transformer-based source code embeddings (Alshomrani et al., 2024).

Static analysis, long the cornerstone of software assurance, provides a structured approach to examining code without execution. However, its reliance on predefined rules and symbolic logic frequently results in high false-positive rates and an inability to generalize to novel or zero-day threats (Kader & Ahmed, 2025). In contrast, the emergence of deep learning, specifically transformer architectures, has introduced a paradigm shift in how code is represented and interpreted. By training on massive corpora of source code, these models learn to capture the subtle linguistic and structural patterns that define vulnerable code (Radanliev, 2025). This report provides an exhaustive review of the methodologies, architectures, and empirical outcomes associated with the fusion of these technologies, delineating how hybrid systems are redefining the boundaries of automated vulnerability detection and localization (Ali et al., 2025).

2. Theoretical Foundations and the Vulnerability Lifecycle

To understand the necessity of automated detection, one must examine the temporal dynamics of software flaws. The vulnerability lifecycle is segmented into three distinct stages: black risk, gray risk, and white risk (Algarni, 2022). The black risk stage is the most hazardous, occurring from the moment a vulnerability is discovered until it is disclosed on an information channel. Pinpointing this initial discovery is challenging; it is often defined as the first date information becomes publicly available through a trusted, independent channel (Yaacoub et al., 2022). Timely identification during this phase is essential for reducing the window of exploitation (Pureti, 2022). Once a vulnerability enters the gray risk stage post-disclosure but prior to widespread patching it remains a significant threat, though defensive mechanisms begin to coalesce. The white risk stage follows the deployment of countermeasures (Malik, 2024).

Early identification efforts traditionally focused on communication channels, such as GitHub issues or developer forums. Research indicates that transformer-based models can analyze these channels with an accuracy approximately twice that of statistical keyword-based approaches (Imran, 2024). While statistical baselines often rely on regular expressions to detect CVE identifiers or specific keywords like "vulnerability" and "NVD," transformer models utilize contextual embeddings to interpret the semantic intent behind developer discussions, enabling the prevention of compromised software usage before official notifications are issued (Taghavi Far, & Feyzi, 2025).

Table 1: Software Vulnerability Lifecycle Phases and Detection Focus

Lifecycle Phase	Description	Security Impact	Detection Focus
Black Risk	Discovery to Disclosure	Maximum; unknown exploit window	Communication channels, GitHub issues
Gray Risk	Disclosure to Countermeasure	High; known but unpatched	Static scanners, vulnerability reports
White Risk	Post-Countermeasure	Minimal; remediation active	Patch verification, regression testing

3. Architectural Evolution of Code Representation

The efficacy of any automated detection system hinges on its internal representation of source code. This representation must bridge the gap between human-readable text and the numerical formats required by machine learning algorithms (Khan, 2025). The evolution of these representations has moved from simple tokenization to complex, structure-aware embeddings that incorporate the logical flow of a program (Biparva et al., 2024).

3.1. Static Analysis and Symbolic Representations

Static Application Security Testing (SAST) tools, such as SonarQube, CodeQL, and Snyk Code, operate by examining the source code for predefined patterns or rules indicative of undesirable behavior (Mosulet, 2021). These tools typically utilize control flow analysis, data flow analysis, and pattern matching against known error signatures (Wu et al., 2018). For example, taint analysis is a primary mechanism for detecting SQL injection or cross-site scripting (XSS) by tracing the flow of untrusted user input to sensitive sinks (Feukoun, 2021). While SAST tools are integrated into continuous integration pipelines and are trusted by millions of developers, they are often limited by the rigidity of their rule sets (Singh, 2025). If a vulnerability does not match a predefined pattern, it remains undetected (Dong et al., 2019).

3.2. The Shift to Deep Learning and Transformers

Deep learning (DL) architectures have emerged to address the limitations of rule-based systems by automatically learning hierarchical code representations (Kierner et al., 2023). Unlike traditional machine learning, which requires manual feature engineering such as calculating cyclomatic complexity or nesting depth deep learning models learn directly from raw data (Ravishankar & Battineni, 2025). Early DL approaches utilized Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks to capture sequential information in code (Bouktif et al., 2020). However, these models often suffer from the vanishing gradient problem when dealing with long sequences and struggle to capture the non-sequential dependencies inherent in code (Zhang et al., 2026).

The introduction of the transformer architecture, which relies on self-attention mechanisms, has revolutionized the field (Soydaner, 2022). Transformers can process sequences in parallel and weigh the significance of different parts of the input regardless of their distance from one another (Yun et al., 2019). This is particularly advantageous for code, where the definition of a variable and its subsequent misuse may be separated by hundreds of lines (Allamanis et al., 2018). The standard scaled dot-product attention is defined as:

Attention(Q,K,V)=softmax(QKTdk)VAttention(*Q*,*K*,*V*)=softmax(*dk*QK*T*)*V*

where QQ, KK, and VV represent the query, key, and value matrices, respectively, and dk*dk* is the dimensionality of the keys (Soydaner, 2022). This mechanism allows the model to "attend" to

security-critical tokens, such as API calls or memory management functions, while maintaining the broader context of the function's logic (Mandana, 2025). As shown in Figure 1, the self-attention mechanism enables the model to capture dependencies between distant code tokens. This capability is essential for identifying vulnerabilities where the source and sink are separated by multiple lines.

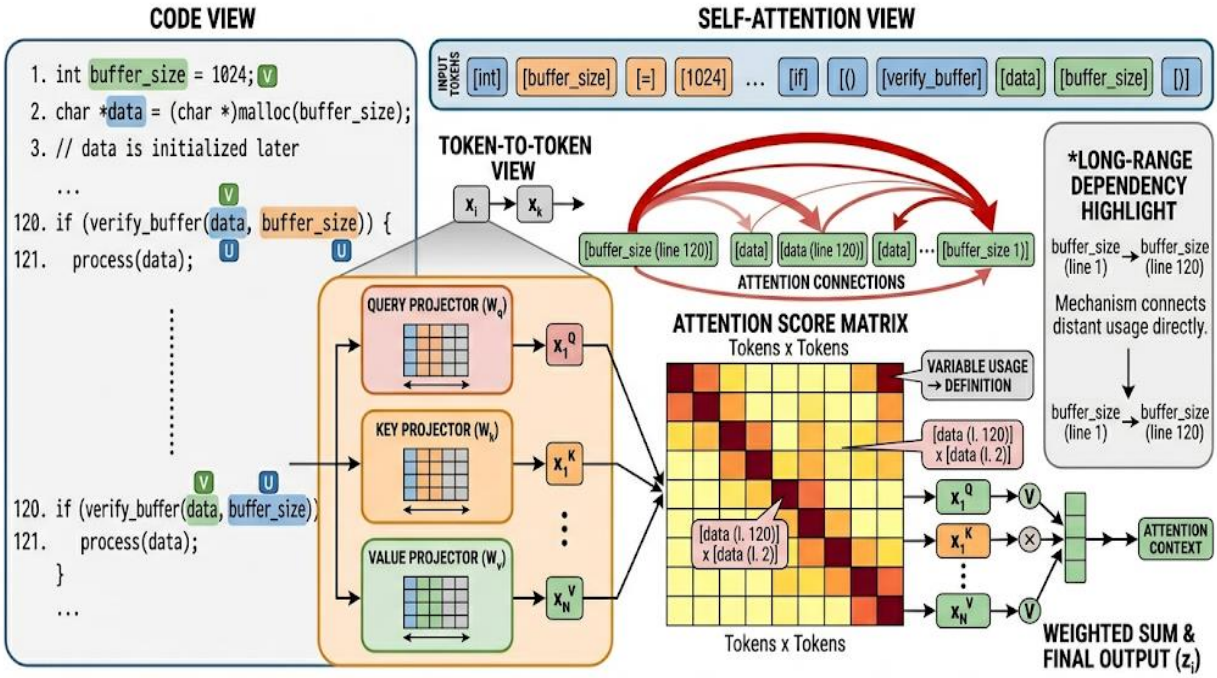

Figure 1: Self-Attention Mechanism in Transformer-Based Code Analysis

4. State-of-the-Art Transformer Models for Source Code

Several specialized transformer models have been developed to handle the unique syntax and semantics of programming languages. These models are generally pre-trained on large-scale datasets from public repositories and then fine-tuned for downstream tasks like vulnerability detection (Kovács et al., 2023).

4.1. CodeBERT: A Bimodal Foundation

CodeBERT is a leading pre-trained model that processes both natural language (NL) and programming language (PL). It is built on the RoBERTa architecture and uses a multilayer transformer encoder (Console, 2024). CodeBERT is pre-trained using two objectives: Masked Language Modeling (MLM), where tokens are randomly masked and predicted from context, and Replaced Token Detection (RTD), where the model identifies tokens that have been altered by a generator (Zou et al., 2025). This bimodal training allows CodeBERT to understand the relationship between code and its documentation, making it highly effective for identifying vulnerabilities that may be described in comments or commit messages (Tang et al., 2024).

4.2. GraphCodeBERT: Incorporating Data Flow

While early transformer-based models such as CodeBERT treat source code primarily as a sequential text representation, GraphCodeBERT extends this paradigm by explicitly incorporating semantic program structures. In particular, it leverages data flow graphs (DFGs), which encode dependency relationships describing where variable values originate and how they propagate through the program (Yang et al., 2024).

Compared with the deeply nested hierarchy of an abstract syntax tree (AST), data flow graphs provide a comparatively flatter and computationally efficient representation of program semantics, enabling more direct modeling of variable interactions and execution logic. To integrate this structural information, GraphCodeBERT employs a graph-guided masked attention mechanism that constrains the transformer's attention patterns according to graph connectivity (Antonios, 2022).

The modified attention mechanism can be expressed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V,$$

where Q denotes the query matrix, K represents the key matrix, and V is the value matrix derived from input embeddings; d_k is the dimensionality of the key vectors used for scaling; and M is the mask matrix that encodes structural constraints derived from the data flow graph. Specifically, the matrix M permits attention between a graph node v_i and a code token c_j only when the variable associated with node v_i is directly identified from token c_j. This structural alignment ensures that semantic relationships captured in the abstract graph representation are consistently preserved within the token-level attention mechanism, thereby enhancing the model's ability to learn meaningful program semantics (Su et al., 2024).

4.3. UniXcoder and Unified Architectures

UniXcoder is designed to support both code understanding and generation by utilizing mask attention matrices with prefix adapters. It incorporates ASTs and code comments to enhance representation (Gong et al., 2024). To handle the tree structure of the AST within the transformer's sequential input, UniXcoder employs a one-to-one mapping function (Chirkova & Troshin, 2021). The algorithm recursively traverses the AST:

1. For a root node root, let name be its identifier.

(Moon & Cohen, 2026).

This transformation ensures that all structural information is retained while allowing parallel encoding with code tokens and comments (Gao et al., 2023).

Table 2: Comparison of Pre-trained Transformer Models for Code Security

Model	Primary Input Modalities	Structural Encoding	Best Use Case
CodeBERT	NL + PL (Sequence)	Token-level context	Bimodal tasks, general classification
GraphCodeBERT	PL + DFG	Graph-guided attention	Data-flow sensitive vulnerabilities
UniXcoder	PL + AST + Comments	AST-to-sequence mapping	Multimodal understanding & generation
CodeT5	PL + Identifier Tags	Identifier tagging objective	Software intelligence, bug fixing
PLBART	PL + NL (Denoising)	Encoder-Decoder	Code summarization, translation

5. Hybrid Methodologies: Fusing Static Analysis with Transformers

The most effective modern detection systems are those that integrate the interpretability and structural grounding of static analysis with the predictive power of transformer-based embeddings (Mowbray, 2025). These hybrid systems address the high false-positive rates of traditional tools by using neural models as a verification or filtering layer (Mohammed et al., 2025).

5.1. The SIExVulTS Framework for Information Exposure

The SIExVulTS system is a representative example of this fusion, specifically targeting Sensitive Information Exposure (CWE-200) in Java (Franco, 2019). The system operates in three stages:

1. **Attack Surface Detection:** A transformer-based engine (using SentBERT) identifies sensitive "sources" (e.g., variables containing passwords or financial data) and potential "sinks" (e.g., log files or network outputs). This context-aware engine achieves an average F1 score of over 93% (Wei et al., 2021).
2. **Exposure Analysis:** The system uses CodeQL to instantiate queries that trace propagation paths between the detected sources and sinks. This stage maps potential vulnerabilities to the CWE-200 hierarchy (Ghebremichael et al., 2026).
3. **Flow Verification:** To mitigate the false positives common in static analysis, a third engine uses GraphCodeBERT to semantically validate the identified dataflows (Shin & Woo, 2025). This stage identifies whether a path is actually exploitable or just structurally connected. This hybrid approach increased precision from 22.61% to 87.23% and led to the discovery of six new CVEs in Apache projects (de Sá Freire, 2022).

Figure 2 presents the multi-stage architecture of the SIExVulTS framework integrating transformer-based detection with static analysis. This hybrid pipeline demonstrates how semantic validation significantly reduces false positives.

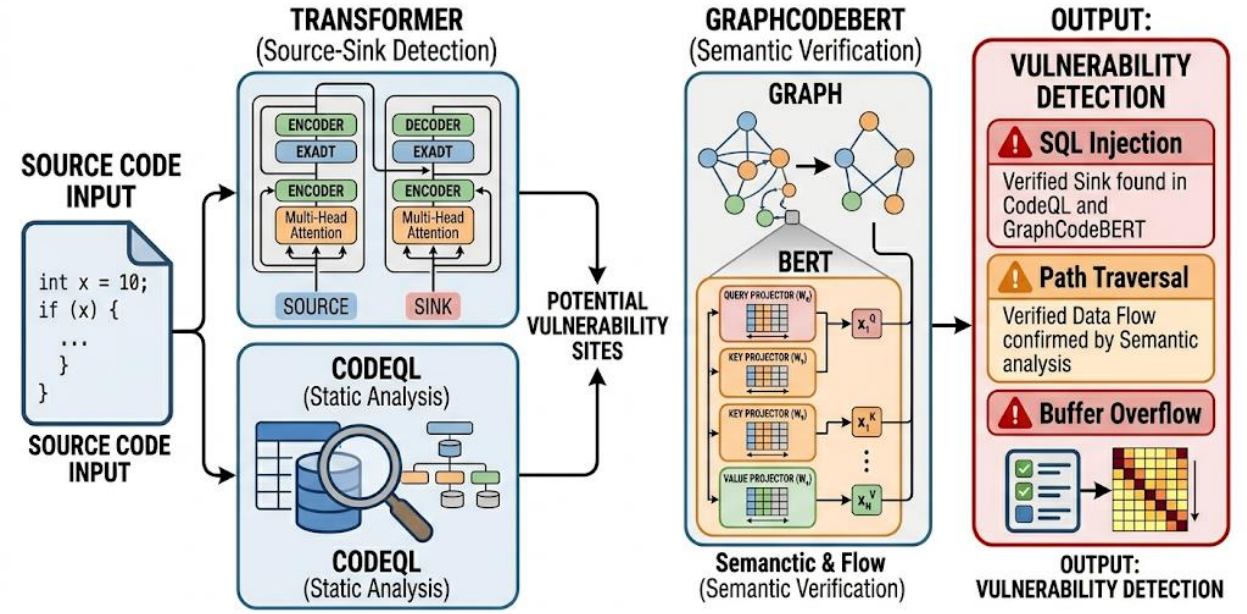

Figure 2: Hybrid Detection Framework Combining Static Analysis and Transformer Models

5.2. Heuristic and Neural Fusion in HYDRA

The HYDRA architecture explores a different integration strategy by combining symbolic heuristic rules with deep GraphCodeBERT embeddings and a Variational Autoencoder (VAE) (Kaiser et al., 2025). HYDRA is designed to uncover "residual risky code" patterns that may persist even after a vulnerability patch has been applied (Xiao et al., 2020). The system extracts five high-impact binary vulnerability signatures using regular expressions, targeting root causes such as missing null pointer checks (CWE-476) and race conditions (CWE-362) (Muhtadi et al., 2026). These handcrafted features are concatenated with neural embeddings to provide an auxiliary signal that improves recall on semantically simple yet security-critical cases often missed by black-box deep learning models (Shrivastwa, 2023).

6. Granularity and Localization in Vulnerability Detection

A significant limitation of early deep learning models was their coarse granularity, typically providing predictions at the function or file level (Emmert-Streib et al., 2020). For developers, a function-level alert is only marginally useful, as it requires manual inspection to find the specific buggy line (Ruan et al., 2025). This has led to the development of line-level Automated Vulnerability Localization (AVL) (Kalouptsoglou et al., 2026).

6.1. Node Classification and Graph Neural Networks

Frameworks like LineVD and LineVul address the localization problem from different perspectives. LineVD formulates statement-level detection as a node classification task over program dependency graphs (Weng et al., 2024). By combining Graph Neural Networks (GNNs) which capture control and data dependencies with CodeBERT for raw token processing, LineVD achieved a 105% improvement in F1-score on the Big-Vul dataset (Tran Dinh et al., 2025). This approach allows the model to learn which specific nodes (statements) are most likely to be vulnerable based on their local semantic features and their broader relationships within the graph (Chami et al., 2022).

6.2. Attention-Based Localization

LineVul utilizes the internal self-attention mechanism of the transformer to identify vulnerable lines. By analyzing which tokens the model focuses on when predicting a vulnerability, LineVul can pinpoint the contributing statements without requiring explicit node-level labels during training (Lingxiang et al., 2025). Empirical evaluations on real-world C/C++ datasets showed that LineVul achieved a Top-10 Accuracy of 0.65, significantly outperforming graph-based baselines like IVDetect (Fu & Tantithamthavorn, 2022).

Table 3: Performance Metrics for Statement-Level Vulnerability Localization

Localization Metric	LineVD Performance	LineVul Performance	Baseline (IVDetect)
F1 Score	0.638 (Fine-tuned)	0.91 (Function-level)	0.35
Precision	High stability	0.97	-
Top-10 Accuracy	-	0.65	0.52 - 0.58
Recall	Significant gain	0.86	-

7. Explainability and Interpretability Frameworks

The "black-box" nature of deep learning models is a major barrier to their adoption in security-critical environments. Developers and reverse engineers require an explanation for why a model flagged a specific code segment (Arikan et al., 2026). Explainable AI (XAI) models, such as VulExplainer and EXVUL, aim to provide high-fidelity justifications (Akinsiku, 2025).

7.1. Faithful and Stable Explanations

The EXVUL framework addresses the interpretability gap by providing understandable explanations in the form of code statements that positively contribute to the detection result. It builds upon GNNExplainer but incorporates a deviation-aware alignment strategy to ensure stability consistency across different runs for the same input (Mao et al.,). To measure the quality of these explanations, research utilizes specific metrics:

- **Mean Statement Precision (MSP):** The fraction of correctly identified vulnerability-related statements (Li et al., 2024).
- **Mean Statement Recall (MSR):** The proportion of the actual triggering path covered by the explanation (Le et al., 2025).
- **Stability (Stb):** The degree of overlap between explanatory outputs across multiple runs (Asfew & Bedemo, 2022).

EXVUL demonstrated a 22.97% improvement in MSP and a 49.55% improvement in MSR over the PGExplainer baseline, highlighting its

effectiveness in pointing out actual triggering paths for vulnerabilities (Butun et al., 2019).

7.2. Natural Language Explanations with LLMs

Beyond highlighting code lines, the latest generation of systems uses Large Language Models (LLMs) to generate detailed textual explanations (Jiang et al., 2026). Frameworks like LLMVulExp leverage a "teacher-student" paradigm where a high-capacity model (e.g., GPT-4o) generates explanatory data from ground-truth vulnerability datasets (Egea et al., 2025). This data is then used to fine-tune a more specialized "student" model (e.g., CodeLlama) via instruction tuning with Low-Rank Adaptation (LoRA) (Rajabzadeh, 2026). This specialization allows the model to not only detect the vulnerability but also describe its root cause and suggest mitigation strategies in natural language (Zhou et al., 2025).

8. Comparative Analysis of Model Performance

The evaluation of automated detection models is conducted across diverse datasets, ranging from synthetic benchmarks to real-world code repositories (Zhu et al., 2025).

8.1. Benchmark Datasets and Their Limitations

Datasets like SARD and the Juliet Test Suite provide hundreds of thousands of programs with labeled weaknesses, but their synthetic nature often leads to biased results, as they cannot capture the full range of variability in real-world code (Reiner Benaim et al., 2020). Conversely, datasets like Big-Vul and CVEfixes are derived from real-world commits but often suffer from data imbalance, where non-vulnerable samples vastly outnumber vulnerable ones (Taghavi Far & Feyzi, 2025). This imbalance can degrade the performance of standard cross-entropy loss functions, necessitating the use of weighted loss or focal loss to focus the model's attention on rare vulnerability patterns (Dina et al., 2023).

Table 4: Common Datasets for Training and Evaluating Vulnerability Detectors

Dataset	Type	Languages	Entry Count	Use Case
SARD/Juliet	Synthetic	C/C++, Java	450,000+	Initial training, rule testing
Big-Vul	Real-world	C/C++	188,000+	Statement-level localization
Devign	Real-world	C/C++	27,318	Function-level classification
DiverseVul	Real-world	Multilingual	Large scale	LLM evaluation
SC-LOC	Real-world	Solidity	1,369	Smart contract security

8.2. Empirical Performance Results

Recent studies show that transformer-based models consistently outperform traditional ML and rule-based tools (Greco & Tagarelli, 2024). In one head-to-head evaluation, LLMs like GPT-4 and Mistral-Large achieved F1 scores around 0.75-0.79 on C# projects, while industrial static analyzers like SonarQube and CodeQL scored between 0.26 and 0.55 (Gnieciak & Szandala, 2025). This discrepancy is largely due to the LLMs' superior recall, as they are better at reasoning across broader code contexts (Wei & Xin, 2025).

In another study focusing on sequence analysis tasks, large-scale transformer models (with billions of parameters) outperformed traditional support vector machines by nearly 40% and standard BERT models by over 5% (Jamshidian, 2023). This suggests that the increased parameter count and massive pre-training of LLMs provide a fundamental advantage in capturing complex patterns across various domains (Hadi et al., 2024).

9. Challenges in Industrial Deployment

While transformer-based models show great promise in research settings, their practical application in industrial DevOps environments faces several challenges (Rane, 2023).

9.1. Scalability and Resource Constraints

The computational overhead of running large transformer models can be significant (Ganesh et al., 2021). Static analyzers integrated into CI/CD pipelines must operate within tight time

constraints to avoid disrupting developer workflows (Wadhams et al., 2024). To address this, tools like AI-DO have been developed, which embed a fine-tuned CodeBERT model specifically optimized for detecting and localizing vulnerabilities during code review (Mock et al., 2025). This approach uses undersampling and efficient fine-tuning to maintain high accuracy without the latency of general-purpose LLMs (Srinivasan et al., 2024).

9.2. Context Window and Long-Range Dependencies

Standard transformer models are limited by a fixed input length (the context window), often restricted to 512 or 1024 tokens (Huang et al., 2023). This is problematic for detecting vulnerabilities that involve cross-function interactions or deep call stacks (Hossain & Shapna, 2025). Approaches to mitigate this include hierarchical graph refinement, which filters redundant information to reduce graph size, and context-aware learning that combines local GNNs with global graph transformers to capture long-distance dependencies (Motamedi, 2025). Additionally, API abstraction techniques have been shown to help LLMs focus on security-relevant logic while ignoring boilerplate code (Bagheri, 2025).

9.3. Robustness to Adversarial Attacks

Vulnerability detection models are increasingly targeted by adversarial attacks, where subtle, semantic-preserving changes are made to code to bypass detection (Sun, 2025). Furthermore, neural autocompleters and detection models are vulnerable to data-poisoning attacks, where malicious files added to the training corpus can influence the model's future suggestions or classification of specific code contexts (Schuster et al., 2021). Ensuring the robustness of these models through adversarial training and rigorous data validation is a priority for future research (Silva & Najafirad, 2020).

10. Future Directions and Emerging Technologies

The field of automated vulnerability detection is rapidly evolving toward integrated frameworks that not only identify flaws but also provide automated remediation (Goswami, 2019).

10.1. Multi-Agent and RAG Frameworks

The integration of Retrieval-Augmented Generation (RAG) with multi-agent systems is an emerging trend (Xu et al., 2024). By referencing an external knowledge base of security documentation and known patches, RAG-enabled LLMs can mitigate the issues of hallucination and outdated information (Alakulju Dkhili, 2025). These systems can autonomously detect, analyze, and generate fixes for vulnerabilities, creating a closed-loop security environment (Al-Hawawreh et al., 2024).

10.2. Reinforcement Learning and Human-in-the-Loop

The use of reinforcement learning (RL) to refine model predictions is also gaining traction (Khurana et al., 2018). Systems like ProRLearn use an RL agent to dynamically adjust a pre-trained model's output based on rewards for correct classifications (Schmied et al., 2023). Furthermore, incorporating human feedback and semantic rewards into the training process helps align the model's security objectives with practical coding standards (McIntosh et al., 2024).

Table 5: Emerging Research Frontiers in Automated Vulnerability Management

Research Frontier	Key Technology	Objective
Automated Repair	Encoder-Decoder + RAG	Generate patches for detected flaws
Interpretability	XAI + Subgraph Masking	Provide verifiable evidence for alerts
Scalability	API Abstraction + LoRA	Faster detection in CI/CD pipelines
Robustness	Adversarial Training	Defend against semantic-preserving attacks
Multi-language	Cross-modal Embeddings	Consistent detection across PLs

Conclusion

The synthesis of static analysis methodologies, transformer-based code representations, hybrid detection frameworks, localization techniques, explainability approaches, and empirical evaluations presented in this paper demonstrates that automated code vulnerability detection has undergone a fundamental paradigm shift from rule-based pattern matching toward deep learning architectures that learn the subtle linguistic and structural signatures of vulnerable code, with the transformer's self-attention mechanism proving uniquely suited to capture the long-range dependencies and non-sequential relationships that characterize security-critical software flaws across distributed codebases. The vulnerability lifecycle framework distinguishing the black risk phase from discovery to disclosure where vulnerabilities remain undocumented yet potentially exploitable, the gray risk phase post-disclosure but pre-patching, and the white risk phase after countermeasure deployment establishes the temporal urgency that motivates automated detection, with transformer-based analysis of developer communication channels achieving approximately twice the accuracy of statistical keyword approaches in identifying emerging threats before official notifications are issued. The architectural evolution from recurrent neural networks and long short-term memory networks, which suffer from vanishing gradient problems with long code sequences, to transformer architectures that process sequences in parallel through scaled dot-product attention has proven particularly consequential for code analysis, where the semantic relationship between a variable definition and its unsafe usage may span hundreds of lines and where the self-attention mechanism's ability to weigh the significance of distant tokens regardless of positional distance enables models to "attend" to security-critical tokens such as API calls and memory management functions while maintaining broader functional context. Among specialized transformer models, CodeBERT's bimodal pre-training using masked language modeling and replaced token detection establishes foundational understanding of the relationship between code and natural language documentation; GraphCodeBERT's integration of data flow graphs through graph-guided masked attention, where the mask matrix M allows attention between a node v_i and a code token c_j only when the variable represented by the node originates from that specific token, preserves the semantic "where-the-value-comes-from" relationships that are essential for identifying dataflow-sensitive vulnerabilities including injection flaws and information exposure; and UniXcoder's AST-to-sequence mapping through recursive traversal transforming hierarchical tree structures into linear sequences enables unified processing of multiple structural modalities. The most significant performance gains, however, arise from hybrid methodologies that explicitly fuse the complementary strengths of static analysis and transformers: the SIExVulTS framework's three-stage architecture transformer-based attack surface detection of sensitive sources and sinks (exceeding 93% F1), CodeQL-based propagation path instantiation mapping potential vulnerabilities to CWE hierarchies, and GraphCodeBERT-based semantic flow verification to distinguish exploitable paths from merely structurally connected ones transformed precision from 22.61% to 87.23% and led to the discovery of six new CVEs in Apache projects, demonstrating that static analysis provides the structural grounding and interpretability that pure neural models lack, while transformers supply the semantic nuance and generalization that rule-based systems cannot achieve. The HYDRA architecture's concatenation of five handcrafted binary vulnerability signatures from regular expressions with GraphCodeBERT embeddings and variational autoencoder representations further illustrates how heuristic features can serve as auxiliary signals that improve recall on semantically simple yet security-critical cases often missed by black-box deep learning models. The advancement from function-level to statement-level vulnerability localization represents a critical maturation for practical deployment, as function-level alerts requiring manual inspection provide limited developer utility; LineVD's formulation of

statement-level detection as node classification over program dependency graphs combined with Graph Neural Networks capturing control and data dependencies achieved a 105% improvement in F1-score on the Big-Vul dataset, while LineVul's exploitation of transformer self-attention to pinpoint contributing statements without explicit node-level labels during training achieved Top-10 Accuracy of 0.65, substantially outperforming graph-based baselines including IVDetect's 0.52-0.58 range. The interpretability imperative for security-critical applications has driven development of explainable AI frameworks including EXVUL, which provides faithful explanations measured by mean statement precision, mean statement recall, and stability metrics, demonstrating 22.97% and 49.55% improvements over the PGExplainer baseline, while large language model-based systems now generate natural language explanations describing root causes and mitigation strategies through teacher-student paradigms where high-capacity models generate explanatory data used to fine-tune specialized student models via instruction tuning with Low-Rank Adaptation. Empirical benchmarking reveals that transformer-based models consistently outperform both traditional machine learning and industrial static analyzers: in head-to-head evaluations, large language models including GPT-4 and Mistral-Large achieved F1 scores of 0.75-0.79 on real-world C# projects, while static analyzers including SonarQube and CodeQL scored between 0.26 and 0.55, with the discrepancy primarily attributable to LLMs' superior recall enabled by broader context reasoning across hundreds of lines rather than pattern matching against isolated rules. However, industrial deployment faces persistent challenges: scalability constraints in CI/CD pipelines require optimized models such as AI-DO using undersampling and efficient fine-tuning of CodeBERT to maintain high accuracy without latency unacceptable for developer workflows; context window limitations (typically 512-1024 tokens) impede detection of cross-function vulnerabilities and deep call stacks, mitigated through hierarchical graph refinement that filters redundant information to reduce graph size and API abstraction techniques that help models focus on security-relevant logic while ignoring boilerplate code; and robustness against adversarial attacks where semantic-preserving code modifications evade detection demands adversarial training and rigorous data validation to prevent poisoning of training corpora. Looking toward the future, emerging technologies including retrieval-augmented generation with multi-agent systems that reference external security documentation and known patches to mitigate hallucination and outdated information, reinforcement learning for dynamic adjustment of pre-trained model outputs based on classification rewards, and automated patch generation through encoder-decoder architectures will drive the evolution from vulnerability detection toward closed-loop automated remediation. Ultimately, the most effective vulnerability detection systems will be those that embrace hybrid architectures combining static analysis's structural verifiability and interpretability with transformer-based embeddings' semantic generalization and contextual reasoning, deployed within CI/CD pipelines optimized for scalability, augmented with explainability frameworks that build developer trust, and continuously updated through adversarial training to maintain robustness against evolving threats, thereby shifting software security from reactive patching toward proactive identification and remediation throughout the development lifecycle.

REFERENCES

Hassan, S., Shafi, I., Ahmad, J., Khan, A., Khurshaid, T., & Ashraf, I. (2025). Review of techniques for integrating security in software development lifecycle. *Computers, Materials, & Continua*, *82*(1), 139.

Khankhoje, R. (2022). Beyond coding: A comprehensive study of Low-Code, No-Code and traditional automation. *Journal of Artificial Intelligence & Cloud Computing. SRC/JAICC-160. DOI: doi. org/10.47363/JAICC/2022 (1)*, *148*, 2-5.

Gudepu, B. K., & Jaladi, D. S. (2022). Data Discovery and Security: Protecting Sensitive Information. *International Journal of Acta Informatica*, *1*(1), 176-187.

Perez, D., & Livshits, B. (2021). Smart contract vulnerabilities: Vulnerable does not imply exploited. In *30th USENIX security symposium (USENIX Security 21)* (pp. 1325-1341).

Alshomrani, M., Albeshri, A., Alturki, B., Alallah, F. S., & Alsulami, A. A. (2024). Survey of transformer-based malicious software detection systems. *Electronics*, *13*(23), 4677.

Kader, M. A., & Ahmed, R. (2025). A Comprehensive Review of Recent Advances in Program Analysis Techniques for Software Reliability and Security. *Scientia. Technology, Science and Society*, *2*(10), 72-79.

Radanliev, P. (2025). Artificial intelligence: reflecting on the past and looking towards the next paradigm shift. *Journal of Experimental & Theoretical Artificial Intelligence*, *37*(7), 1045-1062.

Ali, H., Safdar, R., Liu, J., Binti Abd Manan, T. S., Hu, G., Rasool, M. H., ... & Gao, F. (2025). Hybrid fusion paradigm in advanced process monitoring: a panoramic review and future perspectives. *Industrial & Engineering Chemistry Research*, *64*(47), 22465-22514.

Algarni, A. M. (2022). The historical relationship between the software vulnerability lifecycle and vulnerability markets: Security and economic risks. *Computers*, *11*(9), 137.

Yaacoub, J. P. A., Noura, H. N., Salman, O., & Chehab, A. (2022). Robotics cyber security: Vulnerabilities, attacks, countermeasures, and recommendations. *International Journal of Information Security*, *21*(1), 115-158.

Pureti, N. (2022). Zero-day exploits: Understanding the most dangerous cyber threats. *International Journal of Advanced Engineering Technologies and Innovations*, *1*(2), 70-97.

Malik, G. (2024). From Discovery to Disclosure: A Policy Analysis of Coordinated Vulnerability Disclosure Models. *Frontiers in Emerging Computer Science and Information Technology*, *1*(2), 01-28.

Imran, M. M. (2024). Towards Effective Developer Communication in Open Source Software via Emotional Awareness.

Taghavi Far, S. M., & Feyzi, F. (2025). Large language models for software vulnerability detection: a guide for researchers on models, methods, techniques, datasets, and metrics. *International Journal of Information Security*, *24*(2), 78.

Khan, A. (2025). An Empirical Evaluation of Model Design and Code Representation for Deep Learning-Based Source Code Vulnerability Detection.

Biparva, M., Karimi, R., Faez, F., & Zhang, Y. (2024). TodyFormer: Towards holistic dynamic graph transformers with structure-aware tokenization. *arXiv preprint arXiv:2402.05944*.

Mosulet, P. P. (2021). An analysis of the usage and impact of static code analysis tools.

Wu, J., Dong, M., Ota, K., Li, J., & Guan, Z. (2018). Big data analysis-based secure cluster management for optimized control plane in software-defined networks. *IEEE Transactions on Network and Service Management*, *15*(1), 27-38.

Feukoun, J. P. N. (2021). *Mitigate SQL Injection and Cross-Site Scripting Attacks on Web Applications* (Master's thesis, Utica College).

Singh, B. (2025). Integrating security seamlessly into DevOps development pipelines through DevSecOps: A holistic approach to secure software delivery. *Available at SSRN*, *5267955*.

Dong, Y., Guo, W., Chen, Y., Xing, X., Zhang, Y., & Wang, G. (2019). Towards the detection of inconsistencies in public security vulnerability reports. In *28th USENIX security symposium (USENIX Security 19)* (pp. 869-885).

Kierner, S., Kucharski, J., & Kierner, Z. (2023). Taxonomy of hybrid architectures involving rule-based reasoning and machine learning in clinical decision systems: A scoping review. *Journal of biomedical informatics*, *144*, 104428.

Ravishankar, S., & Battineni, G. (2025). A survey on recent advancements in auto-machine learning with a focus on feature engineering. *Journal of Computational and Cognitive Engineering*, *4*(1), 56-63.

Bouktif, S., Fiaz, A., Ouni, A., & Serhani, M. A. (2020). Multi-sequence LSTM-RNN deep learning and metaheuristics for electric load forecasting. *Energies*, *13*(2), 391.

Zhang, R., Zhang, R., Lu, Y., Chen, W., Ai, B., & Niyato, D. (2026). Mamba for wireless communications and networking: Principles and opportunities. *IEEE Communications Magazine*.

Soydaner, D. (2022). Attention mechanism in neural networks: where it comes and where it goes. *Neural Computing and Applications*, *34*(16), 13371-13385.

Yun, C., Bhojanapalli, S., Rawat, A. S., Reddi, S. J., & Kumar, S. (2019). Are transformers universal approximators of sequence-to-sequence functions?. *arXiv preprint arXiv:1912.10077*.

Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, *51*(4), 1-37.

Soydaner, D. (2022). Attention mechanism in neural networks: where it comes and where it goes. *Neural Computing and Applications*, *34*(16), 13371-13385.

Mandana, E. (2025). *Vulnerability Detection in Smart Contracts using Large Language Models* (Doctoral dissertation, ARISTOTLE UNIVERSITY OF THESSALONIKI).

Kovács, L., Csépányi-Fürjes, L., & Tewabe, W. (2023, October). Transformer models in natural language processing. In *International Conference Interdisciplinarity in Engineering* (pp. 180-193). Cham: Springer Nature Switzerland.

Console, F. (2024). Application of language models on code analysis.

Zou, W., Li, Q., Li, C., Ge, J., Chen, X., Huang, L., & Luo, B. (2025). Improving Source Code Pre-Training via Type-Specific Masking. *ACM Transactions on Software Engineering and Methodology*, *34*(3), 1-34.

Tang, X., Wang, L., Liu, Y., Chai, L., Yang, J., Li, Z., ... & Bissyande, T. F. (2024). In-Context Code-Text Learning for Bimodal Software Engineering. *arXiv preprint arXiv:2410.18107*.

Yang, J., Ruan, O., & Zhang, J. (2024). Tensor-based gated graph neural network for automatic vulnerability detection in source code. *Software Testing, Verification and Reliability*, *34*(2), e1867.

Antonios, M. (2022). Transformer-based Source Code Description Generation: An ensemble learning-based approach.

Su, Z., Xu, X., Huang, Z., Zhang, Z., Ye, Y., Huang, J., & Zhang, X. (2024). Codeart: Better code models by attention regularization when symbols are lacking. *Proceedings of the ACM on Software Engineering*, *1*(FSE), 562-585.

Gong, L., Elhoushi, M., & Cheung, A. (2024). Ast-t5: Structure-aware pretraining for code generation and understanding. *arXiv preprint arXiv:2401.03003*.

Chirkova, N., & Troshin, S. (2021, August). Empirical study of transformers for source code. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering* (pp. 703-715).

Moon, H., & Cohen, E. (2026). Lightweight and Faithful Visual Condition Checking in Behavior Trees via Expert-Regularized Reinforcement Learning.

Gao, S., Gao, C., He, Y., Zeng, J., Nie, L., Xia, X., & Lyu, M. (2023). Code structure–guided transformer for source code summarization. *ACM Transactions on Software Engineering and Methodology*, *32*(1), 1-32.

Mowbray, T. (2025). A survey of deep learning architectures in modern machine learning systems: From CNNs to transformers. *Journal of Computer Technology and Software*, *4*(8).

Mohammed, S. H., Singh, M. S. J., Al-Jumaily, A., Islam, M. T., Islam, M. S., Alenezi, A. M., & Soliman, M. S. (2025). Dual-hybrid intrusion detection system to detect False Data Injection in smart grids. *PLoS One*, *20*(1), e0316536.

Franco, N. D. (2019). Suspicious to Whom: Reforming the Suspicious Activity Reporting Program to Better Protect Privacy and Prevent Discrimination. *NYU Rev. L. & Soc. Change*, *43*, 611.

Wei, H., Lin, G., Li, L., & Jia, H. (2021). A context-aware neural embedding for function-level vulnerability detection. *Algorithms*, *14*(11), 335.

Ghebremichael, J., Vasan, S., Ullah, S., Tystahl, G., Adei, D., Kruegel, C., ... & Kapravelos, A. (2026). Multi-Agent Taint Specification Extraction for Vulnerability Detection. *arXiv preprint arXiv:2601.10865*.

Shin, H., & Woo, J. (2025). Semantic and structural fusion for malware detection: Leveraging CodeBERT, GraphCodeBERT, and AST-GCN: H. Shin, JY Woo. *International Journal of Information Security*, *24*(6), 239.

de Sá Freire, I. F. (2022). *Data Collection and Processing Pipeline for Cyber Vulnerability Intelligence* (Doctoral dissertation, Master's thesis, University of Brasılia).

Kaiser, A., Leoveanu-Condrei, C., Gold, R., Dinu, M. C., & Hofmarcher, M. (2025). HyDRA: A Hybrid-Driven Reasoning Architecture for Verifiable Knowledge Graphs. *arXiv preprint arXiv:2507.15917*.

Xiao, Y., Chen, B., Yu, C., Xu, Z., Yuan, Z., Li, F., ... & Shi, W. (2020). {MVP}: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)* (pp. 1165-1182).

Muhtadi, M., Mahmoud, Q. H., & Azim, A. (2026). Adaptive Self-Prompting in Agentic LLM Frameworks for Code Fault Detection. *Software*, *5*(2), 16.

Shrivastwa, R. R. (2023). *Enhancements in Embedded Systems Security using Machine Learning* (Doctoral dissertation, Institut Polytechnique de Paris).

Emmert-Streib, F., Yang, Z., Feng, H., Tripathi, S., & Dehmer, M. (2020). An introductory review of deep learning for prediction models with big data. *Frontiers in artificial intelligence*, *3*, 4.

Ruan, H., Zhang, Y., & Roychoudhury, A. (2025, April). Specrover: Code intent extraction via llms. In *2025 IEEE/ACM 47th International Conference on Software En*

Kalouptsoglou, I., Siavvas, M., Ampatzoglou, A., Kehagias, D., & Chatzigeorgiou, A. (2026). LocVul: Line-level vulnerability localization based on a Sequence-to-Sequence approach. *Information and Software Technology*, *189*, 107940.

Weng, C., Qin, Y., Lin, B., Liu, P., & Chen, L. (2024, July). Matsvd: Boosting statement-level vulnerability detection via dependency-based attention. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware* (pp. 115-124).

Tran Dinh, K., Bui Vuong Tam, A., Nguyen Vo Tien, L., Nguyen Phan Quoc, D., To, T. N., The Duy, P., & Pham, V. H. (2025, April). An empirical review of the effectiveness of different language processing approaches in software code vulnerability detection. In *Asian Conference on Intelligent Information and Database Systems* (pp. 301-313). Singapore: Springer Nature Singapore.

Chami, I., Abu-El-Haija, S., Perozzi, B., Ré, C., & Murphy, K. (2022). Machine learning on graphs: A model and comprehensive taxonomy. *Journal of Machine Learning Research*, *23*(89), 1-64.

Lingxiang, W., Fu, Q., Song, W., Deng, G., Liu, Y., Williams, D., & Zhang, Y. (2025). SAVANT: Vulnerability Detection in Application Dependencies through Semantic-Guided Reachability Analysis. *arXiv preprint arXiv:2506.17798*.

Fu, M., & Tantithamthavorn, C. (2022, May). Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th international conference on mining software repositories* (pp. 608-620).

Arikan, K. E., Doğan, S. M., Yilmaz, E. N., & Gönen, S. (2026). From code to security: machine learning approaches in android vulnerability detection. *International Journal of Information Security*, *25*(1), 17.

Akinsiku, A. M. (2025). Literature Review on Explainable Artificial Intelligence (XAI): Techniques, Tools, and Applications. *Tech-Sphere Journal for Pure and Applied Sciences*, *2*(1).

Mao, Q., Li, Z., Hu, X., Liu, K., Xia, X., & Sun, J. (2025). Towards explainable vulnerability detection with large language models. *IEEE Transactions on Software Engineering*.

Li, G., Zhao, W., Zhao, L., Li, H., Guo, S., & Hao, L. (2024, December). Identifying meaningful vulnerability report in common weakness enumeration. In *2024 14th International Conference on Information Science and Technology (ICIST)* (pp. 599-608). IEEE.

Le, T. M., Le, V., Do, K., Gupta, S., Venkatesh, S., & Tran, T. (2025). Finding the Trigger: Causal Abductive Reasoning on Video Events. *arXiv preprint arXiv:2501.09304*.

Asfew, M., & Bedemo, A. (2022). Impact of climate change on cereal crops production in Ethiopia. *Advances in Agriculture*, *2022*(1), 2208694.

Cao, S., Sun, X., Liu, W., Wu, D., Zhang, J., Li, Y., ... & Gao, L. (2024). EXVul: Toward Effective and Explainable Vulnerability Detection for IoT Devices. *IEEE Internet of Things Journal*, *11*(12), 22385-22398.

Butun, I., Österberg, P., & Song, H. (2019). Security of the Internet of Things: Vulnerabilities, attacks, and countermeasures. *IEEE Communications Surveys & Tutorials*, *22*(1), 616-644.

Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. (2026). A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, *35*(2), 1-72.

Egea, D., Halder, B., & Dutta, S. (2025, October). Vision: Robust and interpretable code vulnerability detection leveraging counterfactual augmentation. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society* (Vol. 8, No. 1, pp. 812-823).

Rajabzadeh, H. (2026). Efficient Learning for Large Language Models.

Zhou, X., Cao, S., Sun, X., & Lo, D. (2025). Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology*, *34*(5), 1-31.

Zhu, H. N., Furth, R. M., Pradel, M., & Rubio-González, C. (2025). From Bugs to Benchmarks: A Comprehensive Survey of Software Defect Datasets. *arXiv preprint arXiv:2504.17977*.

Reiner Benaim, A., Almog, R., Gorelik, Y., Hochberg, I., Nassar, L., Mashiach, T., ... & Beyar, R. (2020). Analyzing medical research results based on synthetic data and their relation to real data results: systematic comparison from five observational studies. *JMIR medical informatics*, *8*(2), e16492.

Taghavi Far, S. M., & Feyzi, F. (2025). Large language models for software vulnerability detection: a guide for researchers on models, methods, techniques, datasets, and metrics. *International Journal of Information Security*, *24*(2), 78.

Dina, A. S., Siddique, A. B., & Manivannan, D. (2023). A deep learning approach for intrusion detection in Internet of Things using focal loss function. *Internet of Things*, *22*, 100699.

Greco, C. M., & Tagarelli, A. (2024). Bringing order into the realm of Transformer-based language models for artificial intelligence and law. *AI & L.*, *32*, 863.

Gnieciak, D., & Szandala, T. (2025). Large language models versus static code analysis tools: A systematic benchmark for vulnerability detection. *IEEE Access*, *13*, 198410-198422.

Wei, X., & Xin, J. (2025, November). Evaluate LLMs on Code Review Understanding: A Multi-step Reasoning Benchmark. In *International Conference on Neural Information Processing* (pp. 500-514). Singapore: Springer Nature Singapore.

Jamshidian, M. (2023). Evaluation of text transformers for classifying sentiment of reviews by using tf-idf, bert (word embedding), sbert (sentence embedding) with support vector machine evaluation.

Hadi, M. U., Al-Tashi, Q., Qureshi, R., Shah, A., Muneer, A., Irfan, M., ... & Wu, J. (2024). LLMs: A comprehensive survey of applications, challenges, datasets, models, limitations, and future prospects. *TechRxiv*, 439-464.

Rane, N. (2023). Transformers in industry 4.0, industry 5.0, and Society 5.0: roles and challenges.

Ganesh, P., Chen, Y., Lou, X., Khan, M. A., Yang, Y., Sajjad, H., ... & Winslett, M. (2021). Compressing large-scale transformer-based models: A case study on bert. *Transactions of the Association for Computational Linguistics*, *9*, 1061-1080.

Wadhams, Z., Reinhold, A. M., & Izurieta, C. (2024, March). Automating static code analysis through CI/CD pipeline integration. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)* (pp. 119-125). IEEE.

Mock, M., Forrer, T., & Russo[1], B. (2025, November). Detection on Open and Industry Data. In *Product-Focused Software Process Improvement. Industry, Doctoral-Symposium, Tutorial, and Workshop Papers: 26th International Conference, PROFES 2025, Salerno, Italy, December 1–3, 2025, Proceedings* (p. 36). Springer Nature.

Srinivasan, K. P. V., Gumpena, P., Yattapu, M., & Brahmbhatt, V. H. (2024). Comparative analysis of different efficient fine tuning methods of large language models (llms) in low-resource setting. *arXiv preprint arXiv:2405.13181*.

Huang, Y., Xu, J., Lai, J., Jiang, Z., Chen, T., Li, Z., ... & Zhao, P. (2023). Advancing transformer architecture in long-context large language models: A comprehensive survey. *arXiv preprint arXiv:2311.12351*.

Hossain, S. M. S., & Shapna, A. M. (2025). Modern Approaches to Software Vulnerability Detection: A Survey of Machine Learning, Deep Learning, and Large Language Models. *Electronics*, *14*(22), 4449.

Motamedi, A. (2025). Exploring Embedding Strategies for Heterogeneous Graph Representation Learning with GNNs.

Bagheri, A. (2025). *Application of Advanced AI Methods for Precise Vulnerability Detection* (Doctoral dissertation, University of Szeged).

Sun, S. (2025). *Investigating Adversarial Robustness in Language Models: Adversarial Attacks, Certification, and Defense* (Doctoral dissertation, University of Liverpool).

Schuster, R., Song, C., Tromer, E., & Shmatikov, V. (2021). You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th USENIX Security Symposium (USENIX Security 21)* (pp. 1559-1575).

Silva, S. H., & Najafirad, P. (2020). Opportunities and challenges in deep learning adversarial robustness: A survey. *arXiv preprint arXiv:2007.00753*.

Goswami, M. (2019). Utilizing AI for automated vulnerability assessment and patch management. *Eduzone*.

Xu, H., Yuan, J., Zhou, A., Xu, G., Li, W., Ban, X., & Ye, X. (2024). Genai-powered multi-agent paradigm for smart urban mobility: Opportunities and challenges for integrating large language models (llms) and retrieval-augmented generation (rag) with intelligent transportation systems. *arXiv preprint arXiv:2409.00494*.

Alakulju Dkhili, M. (2025). Beyond Native LLMs Outputs: Exploring Retrieval-Augmented Generation and Evaluation Methods.

Al-Hawawreh, M., Baig, Z., & Zeadally, S. (2024). Resilient intrusion detection models for closed control-loop in cyber-physical systems: combating adversarial examples. *IEEE Internet of Things Magazine*, *8*(1), 73-80.

Khurana, U., Samulowitz, H., & Turaga, D. (2018, April). Feature engineering for predictive modeling using reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 32, No. 1).

Schmied, T., Hofmarcher, M., Paischer, F., Pascanu, R., & Hochreiter, S. (2023). Learning to modulate pre-trained models in rl. *Advances in Neural Information Processing Systems*, *36*, 38231-38265.

McIntosh, T. R., Susnjak, T., Liu, T., Watters, P., & Halgamuge, M. N. (2024). The inadequacy of reinforcement learning from human feedback radicalizing large language models via semantic vulnerabilities. *IEEE Transactions on Cognitive and Developmental Systems*, *16*(4), 1561-1574.