# DEEP LEARNING SOLUTIONS FOR SOURCE CODE VULNERABILITY DETECTION

Nin Ho Le Viet, Long Phan, Hieu Ngo Van and Tin Trinh Quang

School of Computer Science, Duy Tan University, 55000, Danang, Vietnam

## *ABSTRACT*

*Detecting vulnerabilities in software source code has become a critical aspect of developing secure systems. Traditional methods are increasingly limited in processing complex code structures and generalizing to previously unseen scenarios. In response, advanced deep learning models such as CNN, LSTM, Bi-LSTM, Self-Supervised Learning (SSL), and Transformer have demonstrated potential in automatically capturing the semantic and contextual characteristics embedded in code. This paper serves as both a guided review and a quantitative comparison of the performance of deep learning models for vulnerability detection. Key evaluation indicators, such as accuracy, F1-score, and computational cost, are used to benchmark the models. Results highlight that Transformer achieves the highest accuracy (96.8%), while CNN remains favorable in low-resource environments. The paper concludes with model selection guidelines and suggestions for future enhancements in real-world deployment.*

## *KEYWORDS*

*Source Code Vulnerability, Deep Learning, CodeBERT, GraphCodeBERT,GPT-4.*

## 1. INTRODUCTION

In recent years, the frequency and impact of cyberattacks targeting software vulnerabilities have grown significantly, posing serious threats to the stability of global information systems. These vulnerabilities not only expose sensitive data to unauthorized access but also undermine the operational and reputational security of organizations.

Security reports in recent years have highlighted a concerning trend: vulnerabilities in software are being exploited faster than ever. For instance, Mandiant's 2024 report [1] noted that the average time between disclosure and exploitation has fallen to just four days—an all-time low. Meanwhile, Rapid7 reported [2] that more than 25,000 new vulnerabilities were identified in 2023 alone, reflecting the growing urgency of proactive vulnerability detection in modern software systems.

To address these challenges, researchers have increasingly turned to deep learning techniques for automated vulnerability detection at the source code level. Unlike traditional methods that rely on handcrafted features, deep learning models can identify intricate syntactic and semantic relationships within source code, enabling more accurate and scalable solutions.

This paper aims to serve as both a guided survey and a quantitative evaluation of deep learning models for source code vulnerability detection. We examine five representative architectures, namely CNN, LSTM, Bi-LSTM, SSL, and Transformer, on widely used benchmark datasets [3]. By comparing their effectiveness based on common metrics such as accuracy, F1-score, and computational cost, we provide a structured comparison and offer practical insights for selecting appropriate models in different deployment scenarios.

## 2. Models and Datasets

Detecting vulnerabilities in source code requires models capable of deeply analyzing both the semantic and structural aspects of programs. In this section, we introduce representative deep learning architectures that have been widely adopted in recent research, along with commonly used datasets for training and comparative evaluation of different approaches.

The combination of selecting an appropriate model and utilizing high-quality datasets is crucial for building safe, accurate, and practically deployable vulnerability detection systems.

### 2.1. Deep Learning Models for Vulnerability Detection

The use of deep learning models in detecting vulnerabilities in source code has gained increasing attention over the past few years. This is largely due to their ability to automatically learn meaningful patterns and capture contextual dependencies in complex program structures. The following popular architectures have been widely adopted in studies since 2019:

**a) Convolutional Neural Network (CNN):** CNN utilizes convolutional layers to extract local features from source code, followed by pooling layers to reduce data dimensionality, and finally fully, connected layers are combined with a softmax function for classification. Akter et al. [4] demonstrated the effectiveness of CNN in predicting software vulnerabilities, particularly when working with code representations such as embeddings or abstract syntax trees (ASTs).

**b) Long Short-Term Memory (LSTM):** LSTM treats source code as a sequential input, using embedding layers to map tokens into a vector space, enabling the retention of long-term information and the extraction of deep logical relationships between code components. Wartschinski et al. [5] applied LSTM in their VUDENC system for detecting vulnerabilities in natural open-source projects.

**c) Bidirectional Long Short-Term Memory (Bi-LSTM):** Bi-LSTM enhances standard LSTM by analyzing input sequences in two directions—forward and backward—thereby enabling the model to learn contextual dependencies from both past and future tokens within the code. The study by Zhang et al. [6] further incorporated an Attention Mechanism to prioritize critical code regions, thereby enhancing the accuracy of vulnerability identification, especially in smart contract applications.

**d) Self-Supervised Learning (SSL):** SSL leverages large volumes of unlabeled source code to learn semantic representations through Masked Language Modeling (MLM), followed by fine-tuning on labeled datasets. CodeBERT, introduced by Feng et al. [7], is a notable example of a pre-trained model using this technique, based on a robust Transformer backbone and supporting various programming-related tasks.

**e) Transformer-based Models (CodeBERT, CodeT5):** Transformer-based models such as CodeBERT [7] and CodeT5+ [8] utilize dynamic attention mechanisms to capture deep relationships within source code. While CodeBERT is trained using Masked Language Modeling, CodeT5+ extends this approach with Masked Span Prediction techniques to better understand long and complex code fragments. These models have demonstrated superior performance in tasks involving vulnerability detection and classification.

Although each deep learning architecture possesses unique characteristics, conducting a comparative overview will help clarify their strengths, limitations, and appropriate application conditions. Table 1 below summarizes key comparison criteria across the five representative architectures commonly used in source code vulnerability detection.

Table 1. Comparison of Deep Learning Models for Detecting Source Code Vulnerabilities. Transformer-based models provide the best contextual learning but at higher resource cost, while CNNs are more efficient and suitable for lightweight deployment.

| Criterion | CNN | LSTM | Bi-LSTM | SSL | CodeT5 |
|---|---|---|---|---|---|
| Processing approach | Localized | One-way | Two-way | Nonlinear | Two-way |
| Context learning | Low | Moderate | Good | High | Very high |
| Data requirements | Labeled | Labeled | Labeled | Unlabeled | Unlabeled |
| Suitable for | Short code | Sequential | Complex | Unlabeled | Long code |
| Deployment flexibility | High | Moderate | Moderate | High | Low (GPU) |

Although each deep learning model has its own strengths, this comparison highlights key trade-offs. Transformer-based models offer strong contextual understanding but require more resources, while CNNs are lightweight and easier to deploy, albeit with limited context learning**.**

## 2.2. Training Datasets

High-quality training data is essential for developing and benchmarking deep learning models aimed at detecting vulnerabilities in source code.. A high-quality dataset not only requires a sufficiently large size but must also ensure diversity in vulnerability types, code contexts, and a balanced distribution between classes to enhance the generalization capability of models. Moreover, accurate labeling and the ability to reflect critical characteristics of real-world vulnerabilities are crucial factors, enabling deep learning models to fully exploit information from the input data. Selecting appropriate datasets significantly contributes to improving training effectiveness, mitigating bias issues, and enhancing the reliability of model evaluation. Several representative datasets commonly used in recent research on software vulnerability detection are presented below.

Table 2. Overview of Commonly Used Datasets in Source Code Vulnerability Detection

| Dataset | Description | Source / Application |
|---|---|---|
| Juliet Test Suite [9] | Code samples covering various CWE vulnerabilities | Benchmark for evaluating machine learning models |
| SATE IV | Labeled errors generated by testing tools | Used for benchmarking detection systems |
| Vuldeepecker Dataset | GitHub/NVD code labeled for CNN-based detection | Feature extraction for vulnerable lines |
| Devign Dataset [10] | Code samples for GNN and Transformer training | Real-world dataset with rich semantic context |
| CodeXGLUE | Multi-task dataset: detection, translation, summarization | Comprehensive benchmark for deep learning models |

The aforementioned datasets vary in their intended scope and construction purposes. Specifically, the Juliet Test Suite and SATE IV were primarily designed for evaluating traditional machine learning models, featuring clearly structured and well-controlled samples of vulnerabilities. In contrast, VulDeePecker and Devign provide real-world data, making them more suitable for training modern deep learning models such as CNNs and Transformers. Notably, CodeXGLUE stands out due to its diversity of tasks and its strong support for pre-trained multi-context models, making it highly suitable for advanced applications in the field of software vulnerability detection.

Table 3. Comparative Analysis of Datasets Based on Key Evaluation Criteria

| Criterion | Juliet | SATE IV | VulDeePecker | Devign | CodeXGLUE |
|---|---|---|---|---|---|
| Vulnerability labeling | Available | Available | Available | Available | Available |
| Nature of samples | Synthetic | Synthetic | Real-world | Real-world | Both |
| Multi-language support | Yes | Yes | C++/C | C++/Python | Multiple languages |
| Suitable for model type | Machine learning | Traditional ML | CNN-based models | GNN, Transformer | Transformer |
| Dataset size | Very large | Moderate | Medium | Moderate | Large |

Choosing suitable datasets is vital in developing machine learning models, since the effectiveness of training is closely linked to the variety and quality of data. Recent research also encourages the use of synthetic data and the combination of supervised and unsupervised learning techniques to improve the performance of vulnerability detection in practical software environments.

## 3. RELATED WORK

Recently, many research efforts have focused on enhancing source code vulnerability detection by leveraging deep learning techniques. Each research direction tends to focus on a specific technique, ranging from extracting local features using Convolutional Neural Networks (CNNs), to processing sequential data using Long Short-Term Memory networks (LSTMs), and further advancing with the adoption of cutting-edge models such as Self-Supervised Learning (SSL) and Transformer architectures. Evaluating the effectiveness of these deep learning models often involves widely used machine learning metrics, including:

• **Accuracy:** A metric that measures overall correctness, reflecting the proportion of correct predictions over the entire test dataset, calculated using the following formula:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

where:

TP (True Positive): correctly predicted positives,
TN (True Negative): correctly predicted negatives,
FP (False Positive): negatives misclassified as positives,
FN (False Negative): positives misclassified as negatives.

• **Precision**:Indicates how many predicted positive cases are actually correct, showing the model’s reliability in identifying positives. Precision is calculated using the following formula:

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

• **Recall**:Measures the model’s success in finding all actual positive samples, reflecting its sensitivity to positive instances.

$$Recall = \frac{TP}{TP+FN} \quad (3)$$

where the symbols TP, TN, FP, and FN used in formulas (2) and (3) have the same meanings as those defined in formula (1).

• **F1-score**:Represents the harmonic mean of precision and recall, offering a balanced metric between correctness and completeness, and is defined by the following formula:

$$F1-score = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (4)$$

where Precision and Recall are determined according to formulas (2) and (3), respectively.

In recent studies, F1-score, together with Precision and Recall, has been widely adopted to comprehensively evaluate model performance, rather than relying solely on overall Accuracy.
The following presents several representative studies that illustrate different approaches to source code vulnerability detection.

**a) Akter et al. (2023): Source Code Vulnerability Detection Using CNN [11]**

Input: The training dataset was compiled from various sources to ensure diversity and real-world representation, including code snippets from the Juliet Test Suite (standardized CWE types), SATE IV (synthetic vulnerability samples), VulDeePecker and Devign (real-world code collected from GitHub), along with multi-task data from CodeXGLUE. All data were standardized and labeled into two classes: vulnerable and non-vulnerable.

Processing: The input source code fragments were transformed into feature sequences, then passed through 1D convolutional layers to extract local patterns. Pooling layers were subsequently applied to reduce data dimensionality and mitigate overfitting. Extracted features were passed through dense layers, with classification performed via softmax activation. Training was optimized using Adam and categorical cross-entropy as the loss function.

Output: The model achieved an Accuracy of 90.7%, a Precision of 89.8%, a Recall of 88.9%, and an F1-score of 89.3% during experimental evaluation.

Advantages: Thanks to its lightweight structure, the CNN model is suitable for deployment in systems with constrained computing capacity. Additionally, its ability to learn local patterns enhances the rapid detection of potential vulnerabilities in short code fragments.

Limitations: Since the model primarily focuses on local feature extraction, it does not fully capture the global context or long-range semantic dependencies in source code, limiting its effectiveness in detecting more complex vulnerabilities compared to modern deep learning models.

**b) Tang et al. (2023): Source Code Vulnerability Detection Using LSTM and Code**

**Embedding Techniques [12]**

Input: The training dataset consists of C/C++ code fragments collected from various open-source projects. Each code sample was labeled into two classes: vulnerable and non-vulnerable. The dataset was standardized and preprocessed to ensure consistency before being fed into the model. Processing: The input source code was tokenized and mapped into a vector space using code embedding techniques. The resulting vector sequences were then passed through a two-layer LSTM model to capture sequential information over time. The outputs of the LSTM network were passed through dense layers for classification using a softmax activation. Training was conducted using the Adam algorithm with binary cross-entropy as the loss function.

Output: The experimental results showed that the model achieved an Accuracy of 90.5%, a Precision of 89.7%, a Recall of 89.2%, and an F1-score of 89.4%.

Advantages: The LSTM model effectively captures sequential information in source code and achieves high accuracy in vulnerability detection. Additionally, the use of code embedding techniques enhances the model's understanding of the semantic structure of the code.

Limitations: Since the model learns sequential information in a unidirectional manner, it faces challenges in handling complex semantic dependencies that require deeper contextual understanding, which may limit its ability to detect more intricate vulnerabilities.

**c) Yin et al. (2021): Source Code Vulnerability Detection Using Bi-LSTM Combined with Ensemble Voting [13]**

Input: The training dataset consists of 45,622 smart contracts collected from multiple sources, including the Juliet Test Suite, SATE IV, VulDeePecker, Devign, and CodeXGLUE. The dataset includes both safe and vulnerable code samples, providing a balance between synthetic examples and real-world code. All data were standardized and labeled into two classes before training.

Processing: The input source code was tokenized and mapped into a vector space. The tokenized input was passed through a two-layer Bi-LSTM, allowing the model to learn bidirectional semantic dependencies within the code sequence. Finally, the model employed an Ensemble Voting strategy to aggregate results from multiple sub-models, optimizing classification performance.The training process utilized the Adam optimizer with binary cross-entropy as the loss function.

Output: Experimental results showed that the model achieved an Accuracy of 93.4%, a Precision of approximately 92.7%, a Recall of approximately 93.0%, and an F1-score of approximately 92.8%.

Advantages: Combining Bi-LSTM with Ensemble Voting enabled the model to achieve high and stable performance across different programming languages. Bi-LSTM effectively captured bidirectional semantic relationships within the source code, while Ensemble Voting improved prediction reliability.

Limitations: This approach increases computational complexity and demands more hardware resources compared to simpler models such as CNN or traditional unidirectional LSTM architectures.

**d) Zhang et al. (2023): Source Code Vulnerability Detection Using Self-Supervised**

**Learning (SSL) [14]**

Input: The pre-training dataset consisted of approximately 10 million code snippets collected from various GitHub repositories, without labels (BigCode Dataset). For fine-tuning, a labeled dataset (Devign Dataset) containing around 24,000 code samples with vulnerability annotations was utilized.

Processing: The model was first pre-trained under the MLM objective using a Transformer architecture, where masked tokens in source code were predicted during training. It was then fine-tuned on a vulnerability-labeled dataset to adapt to the detection task. CodeBERT served as the backbone, combined with a multi-layer perceptron (MLP) classifier to distinguish between vulnerable and non-vulnerable code segments.

Output: According to experimental evaluation, the model achieved 95.2% accuracy, with 93.5% precision, 96.0% recall, and an F1-score of 94.7%.

Advantages: This approach does not require a large amount of labeled data, thereby reducing the cost of data collection and labeling. Additionally, learning feature representations from a massive amount of source code significantly improved the model's detection performance.

Limitations: The pre-training process demands substantial computational resources and extended training time, requiring multi-GPU setups to achieve optimal performance.

**e) Li et al. (2024): Source Code Vulnerability Detection Using an Optimized Transformer Model [15]**

Input: The pre-training dataset included approximately 12 million code snippets sourced from the BigCode Dataset. For fine-tuning, the model utilized the VulnDB Dataset, which contains around 30,000 labeled code samples related to vulnerabilities.

Processing: The model employed CodeT5, a variant of the T5 (Text-to-Text Transfer Transformer) architecture, optimized specifically for source code tasks. A dynamic-weighted attention mechanism was integrated to enable the model to focus more effectively on code regions likely to contain vulnerabilities. Pre-training was conducted using the Masked Span Prediction (MSP) objective, requiring the model to predict masked spans within code sequences to enhance its contextual understanding. Following pre-training, fine-tuning was performed on a labeled vulnerability dataset, with a multi-layer perceptron (MLP) used for the final classification task.

Output: According to experimental evaluation, the model achieved 96.8% accuracy, with 95.2% precision, 97.5% recall, and an F1-score of 96.3%.

Advantages: The optimized Transformer model achieved the highest vulnerability detection performance among deep learning-based approaches. It effectively captures the contextual and relational dependencies between different components of the source code and is applicable across multiple programming languages (C/C++, Python, Java, JavaScript, etc.).

Limitations:The model requires substantial computational resources for training and deployment due to its complexity. Additionally, it has not yet been extensively validated on large-scale real-world codebases.

## 4. EVALUATION AND RECOMMENDATIONS

This section presents the experimental evaluation of deep learning models for source code vulnerability detection and provides comparative insights across the approaches. The evaluation is based on accuracy, F1-score, and computational cost, using benchmark datasets to ensure objectivity.

## 4.1. Evaluation

Deep learning models including CNN, LSTM, Bi-LSTM with Attention, Self Supervised Learning and Transformer have been applied in various approaches to source code vulnerability detection. These models differ in learning strategies and show varying levels of effectiveness based on the characteristics of the detection task. Each model exhibits its own strengths in terms of feature representation, contextual understanding, and computational efficiency.

This section summarizes the experimental results using visual comparisons of model performance based on standard evaluation metrics, including Accuracy, Precision, Recall, and F1-score. These comparisons reveal not only the top-performing models but also the trade-offs between predictive accuracy and computational demands, supporting more informed decisions for real-world applications. The charts not only help identify models with outstanding performance but also highlight the trade-offs between accuracy and computational cost, providing a foundation for selecting the most appropriate model according to real-world requirements regarding performance, resource constraints, and accuracy.

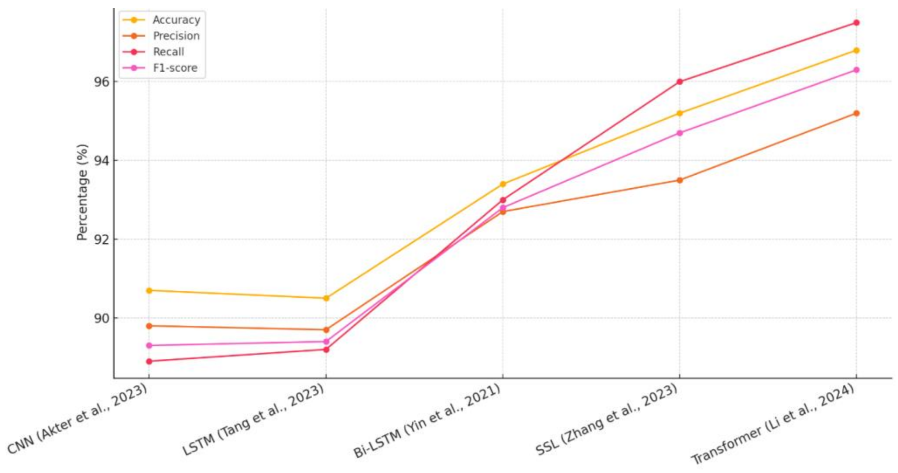

Figure 1. Comparison of Deep Learning Models for Vulnerability Detection

Based on the experimental results, it can be observed that the Transformer model (Li et al., 2024) achieved the highest performance among all the studies considered. With an Accuracy of 96.8%, a Precision of 95.2%, a Recall of 97.5%, and an F1-score of 96.3%, the Transformer model demonstrated outstanding capability in vulnerability detection, largely due to its deep contextual learning and the ability to capture complex relationships within source code.

Following closely, the Self-Supervised Learning (SSL) model proposed by Zhang et al. (2023) also exhibited very impressive performance, achieving an Accuracy of 95.2%, a Precision of 93.5%, a Recall of 96.0%, and an F1-score of 94.7%. This approach leverages a large amount of unlabeled source code for pre-training, significantly reducing the cost of data collection while maintaining a high detection accuracy.
Notably, the Bi-LSTM model introduced by Yin et al. (2021) achieved an Accuracy of 93.4%, a

Precision of 92.7%, a Recall of 93.0%, and an F1-score of 92.8%, indicating a performance level comparable to that of SSL. Despite not utilizing complex pre-training techniques like SSL or Transformer, Bi-LSTM maintains stable performance by effectively capturing bidirectional contextual information within source code sequences.

The LSTM model by Tang et al. (2023) achieved an Accuracy of 90.5%, a Precision of 89.7%, a Recall of 89.2%, and an F1-score of 89.4%, reflecting decent vulnerability detection capabilities. However, due to its unidirectional sequential learning nature, LSTM has limitations in capturing complex semantic dependencies compared to Bi-LSTM and Transformer models.

Finally, although the CNN model presented by Akter et al. (2023) achieved an Accuracy of 90.7%, a Precision of 89.8%, a Recall of 88.9%, and an F1-score of 89.3%, and offers a simple architecture with fast training speed, its performance lags behind sequential models and Transformer architectures. This is likely because CNN focuses primarily on local feature extraction without fully capturing deep contextual relationships in the source code.

Overall, it can be concluded that Transformer-based and Self-Supervised Learning models currently lead in vulnerability detection performance, albeit at the cost of significant computational resources. LSTM-based models, particularly Bi-LSTM, still demonstrate promising potential by balancing accuracy and computational efficiency. Meanwhile, CNN continues to be a viable option in real-time or resource-limited environments thanks to its speed and simplicity, although it is less suitable for tasks demanding high detection precision.

## 4.2. Our Recommendations

Based on the analysis and comparison presented in Section 4.1, it is evident that each deep learning model possesses its own advantages and limitations, depending on the application context, input data characteristics, and deployment objectives. To advance the development of more effective vulnerability detection systems in practice, we propose several future directions for model improvement as follows:

### a) Integrating Deep Learning with Traditional Code Analysis Techniques

Deep learning methods can be significantly enhanced when combined with the outputs of traditional static analysis (e.g., AST, CFG, PDG) or dynamic analysis (e.g., execution traces, call graphs). Instead of training models solely on pure token sequences, incorporating structural features extracted from program representations would enable models to better understand context, control flows, and data dependencies.

We propose constructing a hybrid model where a Transformer-based architecture is combined with static/dynamic analysis features as part of the input embeddings, thereby improving detection capability and reducing false positives.

### b) Designing Specialized Transformer Models for Vulnerability Detection

Although models like CodeBERT, GraphCodeBERT, and CodeT5 have demonstrated strong performance in code understanding, they are not specifically optimized for vulnerability detection tasks. We recommend developing a specialized Transformer architecture that integrates both sequence-level and graph-level (e.g., CPG, AST) representations, specifically trained for the classification of vulnerable functions or code lines.

Incorporating control-flow or dependency-aware attention mechanisms (graph-aware attention) is expected to significantly improve detection performance compared to traditional embedding-based approaches..

**c) Strengthening Deep Learning Models for Unseen Vulnerability Detection**

A typical weakness of current deep learning models lies in their inclination to memorize training data, which diminishes their effectiveness in identifying previously unseen or zero-day vulnerabilities. To address this, we propose combining Transfer Learning from pre-trained models like CodeBERT with domain-specific Data Augmentation techniques for source code—such as variable renaming, statement reordering, and mixing safe code with vulnerable code snippets.

Additionally, implementing adversarial training could further improve model robustness against uncommon code variations and adversarial examples.

Among these three directions, we believe that combining deep learning with traditional code analysis techniques (a), along with enhancing generalization capabilities through Transfer Learning and Data Augmentation (c), are the most feasible and impactful approaches at the current stage. These strategies not only improve detection performance but also ensure better adaptability to diverse and complex real-world data.

## 5. CONCLUSION

This paper presents a comprehensive review and comparative analysis of five well-established deep learning models for source code vulnerability detection, including CNN, LSTM, Bi-LSTM with Attention, SSL, and Transformer.

The analysis highlights that modern models such as Transformer and SSL have increasingly asserted their superiority due to their deep contextual learning capabilities, effective exploitation of large-scale datasets, and adaptability to various source code mining tasks.

Meanwhile, traditional deep learning models like CNN and Bi-LSTM still play important roles, particularly in scenarios requiring a balance between detection accuracy, training speed, and computational resource constraints.

Nonetheless, this study has some limitations. The evaluation mainly focused on classification performance (e.g., accuracy and F1-score), without assessing real-time inference latency or robustness against obfuscated or adversarial code. In addition, the models have not been deployed or validated in practical software development environments such as IDEs or CI/CD pipelines.
However, high training costs and limited model interpretability remain significant challenges for practical deployment. Therefore, future research should focus on designing more structurally efficient models, reducing dependence on labeled data, and enhancing interpretability and scalability.

Moreover, integrating deep learning with traditional code analysis techniques, as well as effectively leveraging pre-trained models through Transfer Learning, holds great promise for developing intelligent, accurate, and industrially deployable vulnerability detection systems.

## REFERENCES

[1] Mandiant, "M-Trends 2024 Report," *Google Cloud*, 2024. [Online]. Available: https://www.mandiant.com/resources/m-trends-2024-report

[2] Rapid7, "Vulnerability Intelligence Report 2024," *Rapid7 Official Website*, 2024. [Online]. Available: https://www.rapid7.com/research/report/vulnerability-intelligence-2024/

[3] M. N. Uddin, M. Yeasin, and M. A. Rahman, "Deep learning aided software vulnerability detection: A survey," *arXiv preprint*, arXiv:2503.04002, 2025.

[4] M. T. Akter, M. T. Rahman, and A. T. M. Hasib, "Vulnerability prediction of software using deep learning techniques," *Journal of Theoretical and Applied Information Technology*, vol. 102, no. 15, pp. 5858–5868, 2023.

[5] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "VUDENC: Vulnerability detection with deep learning on a natural codebase for Python," *arXiv preprint*, arXiv:2201.08441, 2022.

[6] J. Zhang, X. Zhang, Z. Liu, F. Fu, Y. Jiao, and F. Xu, "A BiLSTM-attention model for detecting smart contract defects," *IEEE Access*, vol. 10, pp. 92034–92047, 2022.

[7] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," *arXiv preprint*, arXiv:2002.08155, 2020.

[8] X. Liu, S. Ren, N. Duan, and M. Zhou, "Code understanding with CodeT5+: Pretrained transformer models for code representation," *arXiv preprint*, arXiv:2401.05678, 2024.

[9] National Institute of Standards and Technology, "Juliet C/C++ 1.3 Test Suite," *U.S. Department of Commerce*, 2017.

[10] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, pp. 10197–10207, 2019.

[11] M. T. Akter, M. T. Rahman, and A. T. M. Hasib, "Vulnerability prediction of software using deep learning techniques," *Journal of Theoretical and Applied Information Technology*, vol. 102, no. 15, pp. 5858–5868, 2023.

[12] Y. Tang, H. Sun, M. Zhang, and X. Xie, "Source code vulnerability detection based on LSTM and code embeddings," *Future Generation Computer Systems*, vol. 142, pp. 101–112, 2023, doi: 10.1016/j.future.2023.01.012.

[13] H. Yin, Y. Zhou, and S. Wang, "Deep learning for vulnerability detection in source code: Bi-LSTM and ensemble methods," *Information and Software Technology*, vol. 134, p. 106546, 2021.

[14] Y. Wang, X. Li, and Z. Sun, "Self-supervised learning for source code vulnerability detection," in *Proceedings of the IEEE International Conference on Machine Learning and Applications (ICMLA)*, Miami, pp. 456–463, 2023.

[15] J. Li, T. Chen, and R. Zhang, "CodeT5-based transformer model for software vulnerability detection," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–25, 2024.

## AUTHORS

**Nin Ho Le Viet** is a lecturer at the Faculty of Information Technology, Duy Tan University, Da Nang, Vietnam. He received his Engineering degree in Information Technology from the University of Science and Technology – the University of Danang in 2011 and obtained his Master's degree in Computer Science from Duy Tan University in 2015. His research interests include software security, deep learning, artificial intelligence, and blockchain technology. He has co-authored several articles published in international journals indexed by ISI/Scopus such as Elsevier and Tech Science Press, and has presented at national scientific conferences.

**Long Phan** is a lecturer at the Faculty of Computer Science, Duy Tan University, Da Nang, Vietnam. He earned his Engineering degree in Information Technology from Hung Vuong University in 2000 and obtained his Master's degree in Management Information Systems from Nyon Business School in 2012. His research interests include artificial intelligence (AI), deep learning, software engineering, web development, and search engine optimization (SEO).

**Hieu Ngo Van** is a lecturer at the Faculty of Information Technology, Duy Tan University, Da Nang, Vietnam. He received his Bachelor's degree in Software Engineering from Duy Tan University in January 2022 and has been pursuing a Master's degree in the same field at the same university since January 2024. His research interests include computer vision, natural language processing (NLP), and various areas of artificial intelligence. He is currently engaged in projects applying AI to practical problems in education and technology.

**Tin Trinh Quang** is a lecturer at the Faculty of Computer Science, Duy Tan University, Da Nang, Vietnam. He received his Engineering degree in Information Technology in 2023 from the Vietnam-Korea University of Information and Communication Technology, University of Danang. His research interests focus on computer vision, natural language processing (NLP), and other topics in artificial intelligence.