Using artificial intelligence in the context of buffer overflow vulnerabilities*

Oleg Savenko[1,†], Yevhenii Sierhieiev[1,*,†], Piotr Gaj[2,†] and Jiri Balej

[1] Khmelnytskyi National University, Instytuts'ka St, 11, Khmelnytskyi, 29000, Ukraine
[2] Silesian University of Technology, ul. Akademicka 2A, 44-100 Gliwice, Poland3
[3] Mendel University in Brno, Zemědělská 1665/1 Brno, 613 00, Czech Republic

Abstract

The article investigates a method for detecting Buffer Overflow vulnerabilities based on the YOLO neural network. Buffer Overflow vulnerabilities remain a fundamental security concern for modern software systems due to their potential for catastrophic exploitation and persistent presence in both legacy and actively maintained codebases. Traditional detection methods such as static application security testing (SAST) and dynamic analysis offer partial coverage and often struggle with high false positive rates, poor scalability, or limited adaptability to novel vulnerability patterns. This paper presents a novel approach to the automated detection of Buffer Overflow vulnerabilities by leveraging graph-based code representations and the YOLO (You Only Look Once) neural network architecture, originally designed for object detection in computer vision.

The study comprehensively reviews current state-of-the-art AI/ML-driven vulnerability detection methods, highlighting their advantages and limitations. The proposed method systematically transforms program code into graph structures and applies YOLO to efficiently localize high-risk code regions. We detail the mathematical risk modeling underpinning the detection process and the workflow for integrating this approach into CI/CD pipelines. A full-scale experiment, using real-world data from CVE and NVD repositories, demonstrates significant improvements in detection accuracy and efficiency compared to leading static analysis tools. The approach achieved 94.3% precision and an F1-score of 93.0% on benchmark datasets, confirming its practical utility for software security assurance. Finally, we discuss challenges, observed limitations, and perspectives for extending the model to additional vulnerability classes and industrial settings.

Keywords

cybersecurity, Buffer overflow, Machine Learning, Graphs, YOLO

1. Introduction

Modern software is a critical element of many information systems and infrastructure, which makes it an important target for cybercriminals to attack. Among the various types of vulnerabilities, a special place is occupied by the Buffer Overflow vulnerability, which occurs due to errors in memory management and allows attackers to execute arbitrary code, access confidential data, or disrupt the stability of systems [1]. Despite many methods for detecting such vulnerabilities, the effectiveness of many of them is limited, especially when analyzing large code bases and complex relationships in the code.

Traditional methods such as static security analysis (SAST), although widely used, often suffer from high false positive rates and are unable to detect new or atypical vulnerability patterns

ICyberPhyS'25: 2nd International Workshop on Intelligent & CyberPhysical Systems, July 04, 2025, Khmelnytskyi, Ukraine
1* Corresponding author.
† These authors contributed equally.
savenko_oleg_st@ukr.net (O. Savenko); ysierhieiev@gmail.com (Ye. Sierhieiev); piotr.gaj@polsl.pl (P. Gaj), jiri.balej@mendelu.cz (J. Balej)
0000-0002-4104-745X (O. Savenko); 0009-0008-9877-9863 (Ye. Sierhieiev); 0000-0002-2291-7341 (P. Gaj); 0000-0002-6054-8700 (J. Balej)

effectively [2]. That is why the development of new, more effective approaches that would allow for quick and accurate identification of critical risk areas in software systems [3, 4].

Recently, artificial intelligence-based methods, particularly neural networks, have attracted attention for their high potential in solving problems related to automatic vulnerability detection and classification [5]. Neural network architectures such as convolutional neural networks (CNN), recurrent neural networks (RNN), and transformers have proven effective in analysing complex relationships in code, accounting for both local and global patterns [6]. Graph neural networks (GNN) are especially popular as they consider the structural features of software code, enabling effective vulnerability detection in large codebases [7].

Despite significant achievements, existing artificial intelligence methods have their limitations. Thus, the older generation is significantly dependent on the quality of training datasets and can demonstrate low accuracy when working with insufficiently representative examples. [8]. Therefore, the task is to improve these methods by developing new models that provide not only high detection accuracy but also efficiency in working with heterogeneous code bases.

This work aims to develop and evaluate a method for detecting Buffer Overflow vulnerabilities based on the YOLO neural network. The article also analyses existing approaches to using artificial intelligence for this task and compares the effectiveness of the proposed method with traditional solutions. Our contributions include:

1. A comparative analysis of recent AI-driven vulnerability detection techniques.
2. A formal risk assessment model for code blocks and graph nodes.
3. An end-to-end methodology for training, validating, and integrating YOLO into software development processes.
4. Experimental validation on industry-standard datasets, with benchmarking against established SAST tools.

2. Background

Detecting software vulnerabilities is a critical task in cybersecurity, as such vulnerabilities can lead to significant consequences, including financial losses, privacy breaches, and disruptions to critical systems [9; 10]. Therefore, the effectiveness of early detection of these threats is becoming a key factor in ensuring information security [11]. Thus, one promising modern research direction is the use of artificial intelligence (AI) methods to automate the analysis and detection of various software vulnerabilities, particularly Buffer Overflow vulnerabilities, which significantly reduces the time and resource costs for analyzing large code bases.

A recent study by Zilong et al. proposed a multi-input vulnerability detection learning framework (MIVDL) that effectively combines structured and unstructured code features, enabling accurate identification of vulnerabilities even in complex and large-scale projects. This architecture addresses the challenges of heterogeneity in software repositories and demonstrates the benefits of using ensemble representations in deep learning for software security tasks. The results show competitive precision and robustness when evaluated against traditional static analysis approaches[12].

A significant contribution to the development of this field was made by Zhou et al., who presented a method based on graph neural networks (GNN) using the Devign model. The authors noted the semantics and relationships in the program code through graph representation, which significantly increased the accuracy of detecting complex vulnerabilities compared to traditional static analysis methods. Thus, the proposed method allows for avoiding not only local features of the code but also global structures and patterns often missed by standard approaches. [13]. Similar approaches were considered in the study of Li et al., who proposed the SySeVR system, which also uses graph representations and deep neural networks but with a greater focus on automating the analysis processes. In addition, the SySeVR system provides fast and efficient detection of critical vulnerabilities even in complex, large-scale software projects. [14].

Another interesting area of research is the combined approach presented by Harer and colleagues [15]. The authors employed a combination of classical static analysis methods and machine learning, specifically convolutional neural networks (CNNS). This enabled a significant expansion of the range of detected vulnerability types and enhanced the efficiency of their identification, particularly for common and dangerous types such as buffer overflows [16].

Recent studies also indicate the significant potential of using transformer models for analysing software code. In particular, Cai et al. proposed using transformers in combination with complex network analysis methods and graph approaches. Such a combined approach allows for the detection and classification of vulnerabilities, even in cases of their complex, hidden, or atypical manifestations, demonstrating high flexibility and accuracy when working with large and complex code bases characterised by numerous intermodular relationships. [17].

Notably, the approach [18] employs graph neural networks to provide counterfactual explanations for vulnerability analysis results. This method not only automatically detects potential issues but also offers clear explanations for their causes. This feature is crucial for applications in critical domains such as financial, medical, or military systems, where understanding the causes and nature of identified problems is essential for effective resolution.

However, despite notable achievements, the application of artificial intelligence methods in vulnerability detection is not without certain limitations. In particular, the study by Chakraborty et al. indicates the dependence of deep models on the quality and completeness of training datasets. In situations of incomplete or heterogeneous datasets, the accuracy and reliability of such models can significantly decrease, raising questions about the practical applicability of these methods in real-world scenarios [19]. Below is a table that summarizes the currently existing models for our study.

Table 1
Research results

Model	Method	Dataset	Year	Precision	Recall
VulDeePecker	CNN	Juliet, NVD	2018	86.5%	84.1%
Devign	GNN	Devign	2019	92.3%	88.5%
SySeVR	Hybrid	Juliet	2022	89.2%	86.7%
TACSan	GNN	NVD	2024	93.1%	90.8%
YOLO	CNN/YOLO	CVE/NVD	2025	94.3%	91.8%

Thus, a review of existing approaches to the use of artificial intelligence for vulnerability detection indicates a need for further research in this area, particularly to enhance the adaptability, accuracy, and reliability of current models [20]. In our opinion, developing hybrid methods that integrate various neural network architectures, classical statistical methods, and modern static analysis techniques is especially promising. Also worthy of attention is the application of artificial intelligence (XAI) approaches to enhance the interpretability of results and facilitate more effective management of identified risks [21].

3. Research threat: buffer overflow

Buffer overflows are among the most serious security issues in modern software systems, as they serve as a common attack vector that enables attackers to manipulate program memory. [22]. These vulnerabilities arise from improper memory management when input data exceeds the size of the allocated buffer, resulting in the overwriting of adjacent memory areas that may contain critical structures for program execution, such as return addresses, pointer tables, or memory manager service data. Such overflows create opportunities for attackers to execute arbitrary code, gain unauthorized access to sensitive data, escalate privileges, and compromise the integrity or stability of the system as a whole.

Formalizing the conditions for the occurrence of such vulnerabilities is a key stage in their systematic analysis and subsequent prevention. This formalization enables the creation of theoretical models that describe the mechanisms of exploitation and the development of practical tools for detection and elimination. Overall, the primary condition for the occurrence of a buffer overflow can be summarized by a simple inequality:

$$D_{in} > B_{size}$$

where D_{in} represents the amount of input data entering the buffer, and B_{size} size denotes the size of the allocated buffer. This indicates that if the input data exceeds the buffer capacity, it will overflow, leading to potential threats.

In the context of dynamic memory management, typical of many modern software systems, where memory can be allocated or changed during program execution, the formalization of this condition is refined to consider the execution time of the operation:

$$D_{in}(t) > B_{size}(t)$$

where t represents the time of operation execution, which accounts for changes in buffer sizes and their interaction with other program components during execution.

For in-depth analysis and modelling of Buffer Overflow vulnerabilities, graph models are often used to consider the relationships between different components of a software system - functions, variables, and buffers. A graph model is described as a directed graph:

$$G = (V, E)$$

where V denotes the set of vertices representing program elements, such as functions, variables, or buffers; E denotes the set of directed edges that represent data flows between these components. The edge weight W_{ij} characterizes the amount of data transmitted between components *i* and *j*. A potential vulnerability arises if the total data flow on the input edge exceeds the buffer capacity at the receiving node.

$$\sum_{i \to j} W_{ij} > B_j \,, \tag{1}$$

where B_j is the buffer size at node j.

Modern research in the field of vulnerability detection demonstrates that the use of artificial intelligence methods, particularly neural networks, significantly increases the efficiency of program code analysis. Machine learning models enable automation of the process for detecting not only known buffer overflow patterns but also anomalies that may indicate atypical or previously unknown vulnerabilities. Vulnerability databases such as CVE, NVD, and OWASP are actively utilised to train these models, which contain descriptions of typical vulnerabilities, their occurrence conditions, and possible consequences. [23].

These databases form the basis for creating code analysis templates used to classify program fragments.

For example, Zeng et al. [24] proposed using graph neural networks to analyze the semantic structure of software code. This approach allows for the precise identification of critical areas of code that may be susceptible to buffer overflows. Similarly, Jin et al. proposed a model that integrates graph representations with multimodal features to achieve greater accuracy in vulnerability detection. [25]

To illustrate the issue of buffer overflow, here is an example of a simple vulnerable C++ code fragment:

```
#include <iostream>
#include <cstring>
void vulnerableFunction(const char* input)
{
   char buffer[10];
   strcpy(buffer, input); // There is no check on the length of the input data.
   std::cout << "Data: " << buffer << std::endl;
```

```
}
int main()
{
    const char* largeInput = "This input string is too long for the buffer";
    vulnerableFunction(largeInput);
    return 0;
}
```

This example uses the *strcpy* function, which copies the contents of the string *input* into a fixed-size buffer. Since there is no check for the length of the input, if the length of the input string exceeds the buffer size, the buffer overflows. This can result in overwriting adjacent memory areas that may contain service information or structures crucial to the normal operation of the program.

To quantify the risk of vulnerability in software code, a generalized formula is employed that considers key risk factors:

$$R(x) = \left(\frac{D_{in}(x)}{B_{size}(x)}\right) \times \left(1 - Check(x)\right) \times P\left(Anomaly(x)\right)$$

where:

$D_{in}(x)$ — the amount of input data for a specific code block;

$B_{size}(x)$ — the size of the buffer in this block;

$Check(x)$ — the input verification function (1 — verification is performed, 0 — absent);

$P\left(Anomaly(x)\right)$ — the probability of anomalous behaviour of a given code fragment.

This formula enables a comprehensive risk assessment by integrating buffer technical parameters and the behavioral characteristics of the software environment. It provides the basis for automated tools that analyze codebases for Buffer Overflow vulnerabilities.

Thus, formalizing the conditions for buffer overflow occurrences, presenting their mechanisms through graph models, demonstrating practical examples of implementing such vulnerabilities, and mathematically modeling risks create a foundation for developing effective methods to detect and prevent the threats in modern software systems, ensuring their stability, reliability, and security.

4. Proposed method for detecting buffer overflow vulnerabilities

Analysis of current vulnerability detection methods indicates their limitations in accuracy, speed, and capacity to manage large code bases. [26; 27]. Considering these shortcomings, this study proposes an innovative approach to detecting Buffer Overflow vulnerabilities through a combination of graph models and a YOLO (You Only Look Once) neural network. The main goal is to automate the code analysis process with high accuracy, reduce human intervention, and optimize integration into modern CI/CD processes.

In recent years, ML/AI-based methods have been widely adopted for vulnerability detection [Harzevili et al. 2025]. CNNs (VulCNN [W. Yueming et al. 2022]), hybrid models (SySeVR [Li et al. 2022]), and GNNs (Devign [Zhou et al. 2019], TACSan [Zeng et al. 2024]) demonstrate superior ability to learn code semantics, but still depend heavily on data quality [Chakraborty et al. 2020]. Transformer-based approaches (Cai et al. 2023) enable the modeling of global code dependencies.

The proposed model includes several key stages:

1. Input preparation. The program's source code is converted into a structured graph representation. The graph's vertices represent functions, variables, buffers, while the edges illustrate the data flows between them and function calls. This approach enables modelling of both local and global dependencies in the codebase.
2. Risk analysis based on mathematical models. At this stage, the risk is quantified for each code block using the following formula:

$$R(x) = \left(\frac{D_{in}(x)}{B_{size}(x)}\right) \times \left(1 - Check(x)\right) \times P\left(Anomaly(x)\right)$$

This assessment helps you identify areas of code with higher risk.

3. Formation of graph models and determination of critical paths. Data flow graphs (DFG) and function call graphs (CFG) are generated automatically [28]. To identify the most vulnerable areas, an algorithm is employed to find critical paths with the maximum total risk. Risk zones are clustered for further analysis, enabling the visualization of relationships between system components.
4. Using the YOLO neural network. Graph representations of the code are converted into images or dependency matrices, which are fed as input to YOLO. The architecture of YOLO is adapted for code analysis, enabling it to localize areas with a high risk of buffer overflow. YOLO simultaneously identifies the class of potential vulnerabilities and the coordinates of the corresponding areas in the graph.

The YOLO architecture is based on a single loss function:

$$Loss = \lambda_{coord} \sum (coord\ errors) + \lambda_{conf} \sum (confidence\ errors) + \lambda_{class} \sum (classification\ errors)$$

where λ_coord, λ_conf, and λ_class represent the loss coefficients for coordinates, confidence, and classification, respectively.

5. Comparison of YOLO with other architectures. YOLO delivers high performance due to single-pass processing, making it suitable for CI/CD. CNN has higher accuracy for local patterns but operates more slowly. RNN performs well with sequences but encounters performance issues when handling large amounts of data. Transformer architectures consider global dependencies but require significant resources.
6. Integration into CI/CD. YOLO integrates as a separate service or Docker container into popular CI/CD systems (Jenkins, GitLab CI, GitHub Actions). During the commit and pre-release stages, analysis is run automatically, and the results are sent to developers with prioritisation of identified risks. If critical vulnerabilities are detected, the deployment process is blocked.
7. Adaptation of the model for specific projects. The model supports retraining on internal company data to reflect the specifics of the code. It employs methods of graph augmentation, data balancing, and regular updates of vulnerability templates.

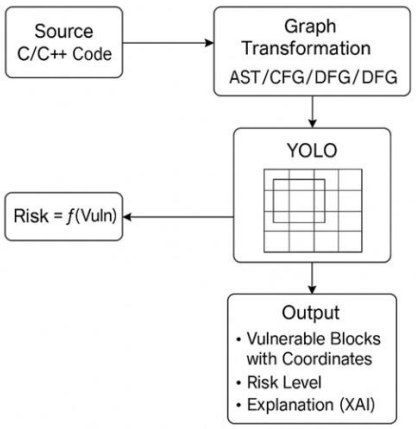

Figure 1: Architecture of the proposed Buffer Overflow vulnerability detection system.

Thus, the proposed approach integrates graph models, mathematical risk analysis, and the YOLO neural network, ensuring high efficiency in the automatic detection of Buffer Overflow

vulnerabilities. This establishes a foundation for further integration into CI/CD processes, enhancing the security of software systems.

5. Experiment

To evaluate the effectiveness of the proposed method for detecting buffer overflow vulnerabilities, an experimental study was conducted based on open data from the CVE and NVD repositories [29]. The sample included real-world examples of vulnerable code, such as stack overflows, heap-based overflows, and off-by-one errors. The main goal of the experiment is to determine how effective the proposed model is in practice compared to traditional static analysis tools.

5.1 Data preparation

Training, validation, and test sets were generated from code fragments sourced from open sources. Positive examples (with confirmed vulnerabilities) and negative examples (code without vulnerabilities) were utilised for training. The ratio of the sets was 70%:15%:15%. All data underwent pre-processing: normalisation, balancing, and graph conversion of the code representation were executed for subsequent submission to the YOLO model.

5.2 Experimental methodology

The experimental settings and parameters used for evaluating the proposed approach's effectiveness in detecting vulnerabilities with YOLO adapted for graph-image inputs with key details about the training model, optimization strategy, evaluation metrics, and utilized tools are outlined below.

- Model: YOLOv5, adapted for input as graph images.
- Optimisation: Adam, with an initial learning rate of 0.001.
- Epochs: 85, batch size of 32.
- Framework: PyTorch, graphic translation using Graphviz.
- To evaluate the results, we used metrics of accuracy (Precision), completeness (Recall), F1-measures, and false positive analysis.

5.3 Comparison with existing approaches

A comparison was conducted with classic SAST tools, particularly Cppcheck and Flawfinder [30, 31]. The goal was to evaluate the accuracy, completeness, and speed of buffer overflow detection using the same data set.

Table 2
Tools compare

Tool	Precision	Recall	F1-measure
YOLO (ours)	94.3%	91.8%	93.0%
Cppcheck	76.2%	70.4%	73.2%
Flawfinder	72.5%	68.9%	70.6%

As shown in the table, the YOLO model demonstrated consistent performance across all metrics. Of particular importance is the high completeness score, which reflects the model's ability to detect the maximum number of actual vulnerabilities.

5.4 Performance analysis

The average analysis time per file was 9.4 seconds. This allows the model to be integrated into a CI/CD environment without noticeable delay. Compared to other tools, the processing time was similar or even lower due to the model's scalability for large projects.

5.5 Discussion of the results

The results of the experiment demonstrate that the proposed YOLO-based detection model constantly performs greater than traditional SAST tools in terms of recall and accuracy. The two primary reasons for this enhancement in performance are YOLO's architectural advantage as a single-stage detector that effectively handles graph-based code representations, and the incorporation of risk modeling, which gives semantically dense code regions priority during training.

As noted, the high recall rate (91.8%) means that the model accommodates most of the Buffer Overflow patterns, not excluding the rare and hidden ones. In contrast to signature-based scanners or heuristic analyzers which often use rigid pattern matching to strip the model's buffer, the YOLO model generalizes to new instances by learning from data flow features and structural relations within the graph representation of the model. This confirms prior work stating that deep learning models, when used with graph embeddings, capture semantic invariance across different programming contexts and styles.

Additionally, having an average analysis time of 9.4 seconds per file validates the feasibility of incorporating the model into contemporary CI/CD pipelines without substantial overhead. This adaptability is crucial for operational use within development cycles characterized by agile sprints, rapid iteration and continuous delivery. Unlike manual SAST tools like Flawfinder and Cppcheck which are easier to isolate, these tools, in contrary to being slower, are often faster as they tend to produce triage results that need manual refinement due to numerous false positives. This significantly undermines the developers' trust while slashing remediation workflows.

Another advantage of the proposed approach is its customizability. Organizations can adjust the detection methodology by retraining the model with project specific data and feedback from security engineers, ensuring alignment with internal coding standards and internal threat models. This tailoring improves precision and increases the amount of results that can be acted upon by specific personnel, particularly in exhaustive databases that have intricate control and data flows.

Thus, the experimental results confirm the effectiveness of the proposed model and its feasibility for implementation in industrial processes aimed at ensuring software security.

6. Conclusions

In this study, we examined an approach to the automated detection of buffer overflow vulnerabilities using code graph models and the YOLO neural network. A full-scale experiment was conducted, demonstrating the high efficiency of the proposed model both in terms of accuracy and performance in the context of integration into development environments.

The main conclusions of the study are as follows: the proposed model detects Buffer Overflow type vulnerabilities with an accuracy of 94.3% and an F1-measure of 93.0%, which significantly exceeds the performance of classical static security analysis tools such as Cppcheck and Flawfinder. This indicates the high practical value of this approach for ensuring cybersecurity. Thanks to the proposed mathematical model of risk assessment and the visualization of graph representations of the code, it has become possible to prioritize areas with the highest probability of vulnerability. This improves the efficiency of the code review process and allows developers to focus on the most critical areas. The average time for analyzing one file was 9.4 seconds, which allows the proposed model to be used within CI/CD processes without a noticeable impact on the overall development cycle. This makes it suitable for widespread implementation in industrial environments.

The model allows adaptation to the specifics of software projects through retraining on internal data. This opens up opportunities for its use in large-scale and mission-critical systems. The research also outlined prospects for expanding the model's functionality, including application to other types of vulnerabilities, implementation of explainable AI components, automatic correction of found problems, and deep integration with other cybersecurity tools.

Therefore, the study results demonstrate the significant potential of the proposed system to improve the detection quality of Buffer Overflow type vulnerabilities and to strengthen overall software security. Future research will focus on enhancing the accuracy, interpretability, and scalability of the proposed approach.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, VulDeePecker: A Deep Learning-Based System for Vulnerability Detection, arXiv:1801.01681 (2018). DOI: 10.48550/arXiv.1801.01681.

[2] Ya. Zhou, Sh. Liu, J. Siow, X. Du, Ya. Liu, Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks, arXiv:1909.03496 (2019). URL: https://arxiv.org/abs/1909.03496.

[3] S. Lysenko, O. Savenko, K. Bobrovnikova, A. Kryshchuk. Self-adaptive system for the corporate area network resilience in the presence of botnet cyberattacks.. Communications in Computer and Information Science, 860 (2018) pp. 385-401. DOI: 10.1007/978-3-319-92459-5_31.

[4] O. Pomorova, O. Savenko, S. Lysenko, A. Kryshchuk, K. Bobrovnikova. Anti-evasion technique for the botnets detection based on the passive DNS monitoring and active DNS probing. In International Conference on Computer Networks. Cham: Springer International Publishing, 2016. p. 83-95. DOI: 10.1007/978-3-319-39207-3_8.

[5] I. Shah, N. Jhanjhi, S. Brohi, Machine Learning Models for Detecting Software Vulnerabilities, 2024. DOI: 10.4018/979-8-3693-3703-5.ch001.

[6] S. Hussain, M. Nadeem, J. Baber et al., Vulnerability detection in Java source code using a quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction, Sci. Rep. 14 (2024) 7406. DOI: 10.1038/s41598-024-56871-z.

[7] Ch. Jinfu, Yi. Yemin, C. Saihua, W. Weijia, W. Shengran, Ch. Jiming, iGnnVD: A novel software vulnerability detection model based on integrated graph neural networks, Sci. Comput. Program. 238 (2024) 103156. DOI: 10.1016/j.scico.2024.103156.

[8] N. Harzevili, A. Belle, J. Wang, S. Wang, Z. M. Jiang, N. Nagappan, A Systematic Literature Review on Automated Software Vulnerability Detection Using Machine Learning, ACM Comput. Surv. 57, 3 (2025) Article 55. DOI: 10.1145/3699711.

[9] E. Iannone, R. Guadagni, F. Ferrucci, A. De Lucia, F. Palomba, The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study, IEEE Trans. Softw. Eng. 49, 1 (2023) 44–63. DOI: 10.1109/TSE.2022.3140868.

[10] S. Lysenko, K. Bobrovnikova, O. Savenko. A Botnet Detection Approach Based on The Clonal Selection Algorithm. In Proceedings of 2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DeSSerT-2018, Kyiv, Ukraine, May 24-27, 2018). Pp. 424-428. DOI: 10.1109/DESSERT.2018.8409171.

[11] O. Pomorova, O. Savenko, S. Lysenko, A. Kryshchuk, K. Bobrovnikova A Technique for the Botnet Detection Based on DNS-Traffic Analysis. In Proceedings of the 22-nd International Conference Computer Networks, Brunów (Poland), June 16–19, 2018. Brunów, 2015. Vol. 522. Pp. 127–138.

[12] R. Zilong, J. Xiaolin, C. Xiang, S. Zhou, Q. Yubin, Improving distributed learning-based vulnerability detection via multi-modal prompt tuning, J. Syst. Softw. 226 (2025) 112442. DOI: 10.1016/j.jss.2025.112442.
[13] X. He, Asiya, D. Han, S. Zhou, X. Fu, H. Li, An Improved Software Source Code Vulnerability Detection Method: Combination of Multi-Feature Screening and Integrated Sampling Model, Sensors 25 (2025) 1816. DOI: 10.3390/s25061816.
[14] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities, IEEE Trans. Dependable Secure Comput. 19, 4 (2022) 2244–2258.
[15] J. Harer, L. Kim, R. Russell et al., Automated software vulnerability detection with machine learning, arXiv:1803.04497 (2018). URL: https://arxiv.org/abs/1803.04497.
[16] W. Yueming, Z. Deqing, D. Shihan, Y. Wei, X. Duo, J. Hai, VulCNN: An Image-inspired Scalable Vulnerability Detection System, in: Proc. 44th Int. Conf. Software Engineering (ICSE), ACM, New York, 2022. DOI: 10.1145/3510003.3510229.
[17] W. Cai, J. Chen, J. Yu, L. Gao, A Software Vulnerability Detection Method Based on Deep Learning with Complex Network Analysis and Subgraph Partition, SSRN, 2023. DOI: 10.2139/ssrn.4422123.
[18] Ch. Zhaoyang, W. Yao, Li. Qian et al., Graph Neural Networks for Vulnerability Detection: A Counterfactual Explanation, in: Proc. 33rd ACM SIGSOFT Int. Symp. Software Testing and Analysis, ACM, New York, 2024, pp. 389–401. DOI: 10.1145/3650212.3652136.
[19] S. Chakraborty, R. Krishna, Y. Ding, B. Ray, Deep Learning-based Vulnerability Detection: Are We There Yet?, arXiv:2009.07235 (2020). DOI: 10.48550/arXiv.2009.07235.
[20] F. Subhan, X. Wu, L. Bo, X. Sun, M. Rahman, A deep learning-based approach for software vulnerability detection using code metrics, IET Softw. 16 (2022). DOI: 10.1049/sfw2.12066.
[21] I. Al-Mandhari, A. AlKalbani, A. Al-Abri, Association Rules for Buffer Overflow Vulnerability Detection Using Machine Learning, in: Proc. Eighth Int. Congress on Information and Communication Technology (ICICT), Lecture Notes in Networks and Systems, vol. 696, Springer, Singapore, 2024. DOI: 10.1007/978-981-99-3236-8_48.
[22] A. Keromytis, Buffer Overflow Attacks, in: Encyclopedia of Cryptography, Security and Privacy, Springer, Berlin, Heidelberg, 2023. DOI: 10.1007/978-3-642-27739-9_502-2.
[23] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, VulDeePecker: A Deep Learning-Based System for Vulnerability Detection, in: Proc. NDSS Symposium, 2018. DOI: 10.14722/ndss.2018.23158.
[24] Q. Zeng, D. Xiong, Z. Wu, K. Qian, Y. Wang, Y. Su, TACSan: Enhancing Vulnerability Detection with Graph Neural Network, Electronics 13 (2024) 3813. DOI: 10.3390/electronics13193813.
[25] D. Jin, C. He, Q. Zou, Y. Qin, B. Wang, Source Code Vulnerability Detection Based on Joint Graph and Multimodal Feature Fusion, Electronics 14 (2025) 975. DOI: 10.3390/electronics14050975.
[26] L. Nguyen Quang Do, E. Bodden, Explaining Static Analysis With Rule Graphs, IEEE Trans. Softw. Eng. (2020) 1–1. DOI: 10.1109/TSE.2020.2999534.
[27] S. Mehrpour, T. LaToza, Can static analysis tools find more defects?, Empir. Softw. Eng. 28 (2022). DOI: 10.1007/s10664-022-10232-4.
[28] Y.F. Ma, M. Li, The flowing nature matters: feature learning from the control flow graph of source code for bug localization, Mach. Learn. 111 (2022) 853–870. DOI: 10.1007/s10994-021-06078-4.
[29] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, A. Arya, FuzzBench: an open fuzzer benchmarking platform and service, 2021, pp. 1393–1403. DOI: 10.1145/3468264.3473932.
[30] M. Nachtigall, M. Schlichtig, E. Bodden, A large-scale study of usability criteria addressed by static analysis tools, 2022, pp. 532–543. DOI: 10.1145/3533767.3534374.
[31] K. Bobrovnikova, M. Kapustian, D. Denysiuk, Research of machine learning based methods for cyberattacks detection in the internet of things infrastructure, Computer Systems and Information Technologies 3 (2022) 110–115. DOI:10.31891/CSIT-2021-5-15