https://doi.org/10.31891/2219-9365-2025-83-1
УДК 004.056.55:004.8

KOVALOVA Anna
Anbosoft LLC
https://orcid.org/0009-0008-1434-7298
e-mail: kovalova.ann@gmail.com

CONTROLLING SOFTWARE CODE VULNERABILITIES USING AI-ORIENTED STATIC ANALYSIS

This paper addresses the pressing issue of software security by exploring the integration of traditional static analysis techniques with advanced AI-based methods for source code vulnerability detection. The research proposes a hybrid architecture that combines rule-based engines, such as CodeQL with transformer-based neural networks like CodeBERT. While traditional static analyzers rely on manually crafted rules and patterns, they often fail to detect context-dependent or novel vulnerabilities. AI models, on the other hand, demonstrate a growing ability to learn latent semantic structures and security-relevant code patterns by leveraging abstract syntax trees (AST), data flow graphs (DFG), and language-model pretraining techniques. The presented architecture capitalizes on the strengths of both approaches by aggregating the results of a rule-based static analysis pipeline and an AI-assisted vulnerability classifier into a unified decision engine.

To assess the system's effectiveness, experiments were conducted on a labeled dataset of 15,000 code samples. The AI model, based on CodeBERT, was trained for 20 epochs using binary cross-entropy and evaluated by F1-score. Three approaches were compared: rule-based, standalone AI, and the hybrid model. Results showed that the AI-only model outperformed the rule-based analyzer (F1-score: 0.81 vs. 0.68), while the hybrid approach achieved the highest score of 0.86, balancing precision and recall.

Beyond classification accuracy, the research also considered the computational trade-offs and runtime implications of integrating AI into static analysis workflows. While the AI-enhanced pipeline incurs higher memory and processing time costs, its ability to identify critical vulnerabilities missed by traditional tools justifies its application in security-sensitive environments. Case studies highlighted examples such as heap buffer overflows and use-after-free vulnerabilities, which were correctly identified by the AI model but missed by pattern-matching rules.

The paper concludes that hybrid AI-assisted static analysis is a promising direction for enhancing secure software development practices, especially in the context of DevSecOps pipelines. Future work includes extending the architecture to support multiple programming languages, integrating explainable AI components for better result interpretability, and optimizing model performance for lightweight deployment scenarios. Overall, the findings emphasize the practical feasibility and advantages of embedding AI into traditional software assurance processes to improve code security in an automated and scalable manner.

Keywords: static code analysis, vulnerabilities, artificial intelligence, CodeBERT, secure development, hybrid model.

КОВАЛЬОВА Анна
Anbosoft LLC

КОНТРОЛЬ УРАЗЛИВОСТЕЙ ПРОГРАМНОГО КОДУ ШЛЯХОМ ВИКОРИСТАННЯ AI-ОРІЄНТОВАНОГО СТАТИЧНОГО АНАЛІЗУ

У статті досліджено підходи до виявлення уразливостей у програмному коді шляхом поєднання класичних методів статичного аналізу з сучасними AI-моделями. Розроблено гібридну архітектуру аналізатора, що об'єднує rule-based механізми з трансформерними нейронними мережами на основі CodeBERT, орієнтованими на глибоке семантичне розуміння коду. Проведено експериментальне порівняння ефективності трьох підходів до аналізу – класичного, AI-орієнтованого та комбінованого. Запропонований гібридний підхід продемонстрував найвищу точність виявлення уразливостей (F1-score = 0.86) порівняно з іншими моделями. Наведено приклади критичних уразливостей, які були успішно виявлені лише за допомогою AI-модуля, що підтверджує його здатність виявляти складні шаблони, недоступні для класичного rule-based аналізу.

Додатково проведено оцінку продуктивності та ресурсоспоживання кожного підходу, а також досліджено можливості інтеграції запропонованої системи в CI/CD-середовища для безперервного забезпечення безпеки коду. Застосування штучного інтелекту у поєднанні з класичними засобами дозволяє підвищити ефективність та надійність процесу аналізу, зменшуючи ймовірність пропуску критичних уразливостей. Отримані результати можуть бути використані як основа для впровадження інтелектуальних засобів контролю безпеки в сучасні середовища розробки програмного забезпечення.

Ключові слова: статичний аналіз коду, уразливості, штучний інтелект, CodeBERT, безпечна розробка, гібридна модель.

Стаття надійшла до редакції / Received 30.07.2025
Прийнята до друку / Accepted 22.08.2025

STATEMENT OF THE PROBLEM

In today's information technology environment, software complexity is rapidly increasing, which in turn is accompanied by an increase in the number of vulnerabilities discovered in the code [1]. Vulnerabilities left without proper control can lead to critical consequences – from the leakage of confidential information to the complete disruption of system functioning [2]. Traditional static analysis methods, although they remain the basis for detecting errors without the need to execute programs, demonstrate limitations in accuracy, scalability, and efficiency when

working with complex architectures and modern programming paradigms. With the growth of code volumes and the need to ensure continuous integration within CI/CD processes, the automation of the vulnerability control process is becoming particularly relevant. In this context, there is growing interest in the use of artificial intelligence, in particular machine learning and natural language processing methods, which allow analyzing program code not only from the standpoint of syntax, but also taking into account semantic dependencies and context [3-5]. AI-based tools are able to detect atypical or previously unknown vulnerabilities that remain beyond the scope of traditional analysis, which demonstrates the potential of combined approaches to improving the security of software systems [6, 7].

In this regard, it is relevant to develop an approach that combines the advantages of classical static analysis with the intelligent capabilities of AI models. Such an approach aims to provide more accurate, fast and scalable vulnerability detection, which can be effectively integrated into modern software development processes.

The aim of the study is to develop and experimentally evaluate an AI-oriented static analysis methodology aimed at detecting vulnerabilities in software code with an increased level of accuracy and minimizing false positives and false negatives.

THEORETICAL BACKGROUND AND RELATED WORKS

Over the past decades, static analysis has remained one of the key approaches to detecting errors in program code without executing it. Traditional methods include linting, type checking, data flow analysis, formal verification methods, and stylistic and structural compliance checks. The authors of [8] investigated the effectiveness of linting as a first-level check, which allows for quick detection of obvious errors, but does not provide deep semantic analysis. The paper [9] describes the use of data flow analysis to detect potential information leaks, in particular in web applications. At the same time, formal methods, as shown in [10], demonstrate high accuracy, but are limited in scalability and applicability to large systems with a high degree of variability. Against the background of the limitations of the traditional approach, attention is increasingly paid to methods based on artificial intelligence technologies. In particular, the paper [11] considers the use of Natural Language Processing to understand program code as a text with a certain structure, which allows detecting atypical patterns associated with vulnerability. The study [12] demonstrates the effectiveness of graph neural networks (GNN) for analyzing dependencies between code elements based on constructed flow control graphs or calls. The authors of [13] proposed the use of CodeBERT and GraphCodeBERT transformers trained on large corpora of program code, which are able to detect potential vulnerabilities at the function or method level. All these approaches emphasize the ability of deep learning models to process syntactic and semantic dependencies that are not always available during classical analysis. In parallel with academic research, applied code analysis tools are actively developing. For example, SonarQube offers a system for detecting bugs and code smells using rules focused on best programming practices. The paper [14] provides an example of using Fortify Static Code Analyzer for a corporate environment, where the emphasis is on compliance with security standards. The CodeQL tool, as noted in [15], combines the properties of database queries and classical static analysis, allowing for the formalization of vulnerability patterns. The study [16] demonstrates the integration of ML models into such tools, which allows for a reduction in the number of false positives.

A review of existing approaches suggests that none of the methods is universal in terms of accuracy and completeness of vulnerability detection. The most promising are hybrid approaches that combine the advantages of classical static analysis methods with the flexibility and learning ability of AI models. It is such approaches that allow for context-awareness, minimizing false positives, and ensuring adaptation to new types of threats.

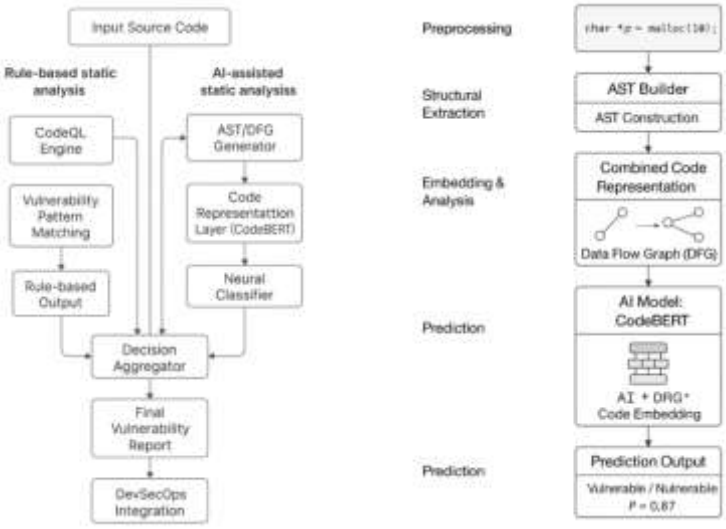
EXPERIMENTAL METHODOLOGY

To test the effectiveness of the AI-oriented approach to software code vulnerability control, an experimental architecture was designed that combines classical static analysis with deep learning based on a transformer model. The analysis was carried out on open sets of software code with vulnerability injections, as well as on real open-source repositories. The basis of the corpus was samples from the Juliet Test Suite (CWE) and modified examples from the SARD (Software Assurance Reference Dataset), which provide a wide range of monitored vulnerabilities according to the CWE (Common Weakness Enumeration) classification.

At the stage of data preprocessing, code tokenization was performed, an abstract syntax tree (AST) was built using the tree-sitter Python library, and a Data Flow Graph (DFG) was generated. The structured representations were used as input data to a CodeBERT-type model, which was pre-trained on code in C/C++ and Python.

Two branches of analysis are provided within the experimental architecture. The first branch implements traditional static analysis based on rules and pattern-based detection implemented via CodeQL. The second branch is AI-oriented, in which tokenized code is fed into a transformer architecture model that performs binary classification of code fragments (vulnerable/invulnerable). The final solution is formed by aggregating the results of both branches using a logical operator such as soft-voting ensemble.

This paper proposes a hybrid architecture of a static analyzer that combines classical rule-based verification with an AI model such as CodeBERT that works with graph representations of code. The system is built in two stages: the first stage is responsible for the analyzer architecture as a whole (Fig 1. a), the second is for detailed processing of the input code, construction of its abstract syntax tree and preparation of input tensors for the neural network (Fig 1. b).



a) **b)**

Fig. 1. a – Architecture of the hybrid static vulnerability analysis system; b – Pipeline of data preprocessing and feature extraction

Fig 1. a presents the logical structure of the components: a classic analyzer, a GNN module, a result aggregation module, and an interface for interacting with the CI/CD system. Fig 1. b details the process of generating input data for the model, including parsing, building a data flow graph, and tokenization for feeding to the transformer. Both parts of the scheme are key to implementing a full cycle of AI-oriented static analysis.

Formally, the classification process in the AI-oriented part of the model is described by function (1).

$$y = \sigma(W_2 \cdot ReLU(W_1 \cdot f(x) + b_1) + b_2) \quad (1)$$

where $f(x)$ is the vector representation of the code from the CodeBERT output, W_1, W_2 are the weight matrices of the fully connected layers, σ is the sigmoid activation function that returns the probability of a fragment belonging to the vulnerable class. The model was trained using binary cross-entropy as the loss function (2).

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}[y_1 \log(\widehat{y_i}) + (1 - y_i)\log(1 - \widehat{y_i})] \quad (2)$$

The training process included 20 epochs, batch size – 32, optimizer – Adam with a learning rate $\eta = 2 \cdot 10^{-5}$. An early stopping mechanism was implemented in the absence of F1-score improvement on the validation set.

For an objective comparison of the approaches, three scenarios were formed:

- S1 – only rule-based analyzer (CodeQL);
- S2 – only AI-model (CodeBERT+classifier);
- S3 – combined approach with aggregation (ensemble).

The performance metrics were calculated using classical formulas (3).

$$Precision = \frac{TP}{TP+FP}, Recall = \frac{TP}{TP+FN}, F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision+Recall} \quad (3)$$

where TP is true positive, FP is false positive, FN is false negative.

The proposed methodology combines the analytical power of formal methods with the flexibility of deep learning models, providing a qualitatively new level of accuracy in vulnerability control. The next section will be devoted to conducting the experiment and analyzing the results.

EXPERIMENT RESULTS

The experimental study was carried out on the basis of a combined dataset, which included 15,000 code fragments with notes on the presence or absence of vulnerabilities. Implementation environment: Google Colab Pro with GPU (Tesla T4), 16GB RAM, programming language – Python 3.10. 70% of the data was used for training, 15% for validation and 15% for testing. The CodeBERT AI model was trained for 20 epochs with a batch size of 32, the loss function – binary cross-entropy, the optimizer – Adam with parameters $\eta = 2 \cdot 10^{-5}, \beta_1 = 0.9, \beta_2 = 0.99$.

The stopping criteria were an improvement in the F1-score metric on the validation set of at least 0.001 for 5 consecutive epochs.

During the testing, the effectiveness of three approaches was analyzed:

- S1 – classic rule-based analysis using CodeQL;

- S2 – AI model without classic rules;
- S3 – hybrid model with aggregation of results.

The results are presented in Table 1, where the main accuracy metrics are listed.

Table 1

Comparison of the effectiveness of three approaches to vulnerability detection

Approach	Precision	Recall	F1-score
S1 (Rule-based)	0.72	0.64	0.68
S2 (AI-only)	0.84	0.79	0.81
S3 (Hybrid)	0.88	0.85	0.86

As can be seen from the table, the AI model (S2) demonstrates a significant improvement in metrics compared to the classic analyzer (S1), in particular, the F1-score is improved by 18%. The hybrid approach (S3) further increases the accuracy, achieving a balanced Precision/Recall ratio. This indicates the feasibility of combining classic and AI-oriented methods.

Examples of vulnerabilities that were missed by the classic analyzer, but detected by the AI model, include:

1. Heap buffer overflow, which occurs due to an incorrect size of allocated memory – the AI model recorded the danger despite the lack of a direct pattern.

2. Use-after-free, detected in a fragment where an object is reused after being freed – CodeQL did not have a corresponding rule, but the model learned to recognize this pattern from the context.

In addition to accuracy, the code processing speed (number of lines/second) and memory consumption were analyzed. Rule-based analysis (S1) is the fastest, but the AI model (S2) requires three times more RAM and analysis time, which is due to the complexity of the transformer architecture. The hybrid approach (S3) is in between, demonstrating an acceptable compromise between accuracy and performance.

Despite the high results, the AI model has a number of limitations. In particular, it demonstrates reduced accuracy when working with obfuscated code, and also requires pre-training on a representative corpus for each programming language. In addition, the complexity of the model makes it difficult to integrate it into a resource-constrained environment (for example, IDE plugins or CI servers with limited computing potential).

Development prospects lie in the use of lightweight models such as DistilCodeBERT, the implementation of attention mechanisms for visualizing explanations, as well as in the automatic generation of patches based on detected vulnerabilities. The results obtained indicate the feasibility of integrating AI mechanisms into secure software development processes, especially in the field of DevSecOps.

CONCLUSION

The study proved the effectiveness of using an AI-oriented approach to static analysis of software code to detect vulnerabilities. The proposed hybrid architecture, which combines traditional rule-based analysis with a transformative neural network, demonstrated improved accuracy in classifying vulnerable code fragments compared to each of the approaches separately. The model, trained on real examples using structural code representations in the form of AST and data flow graphs, allowed detecting complex and atypical patterns that remain beyond the capabilities of classical analysis tools. The hybrid approach demonstrated the highest F1-score, which indicates its ability to reduce both false positives and false negatives. A comparison of performance and resource consumption showed that although the AI model requires more computing resources, its integration into static analysis is justified due to a significant increase in accuracy. Of particular note is the potential of such a system for integration into continuous development processes within DevSecOps, where automated vulnerability detection is critical.

Of particular note are the limitations associated with the need for representative training samples and reduced accuracy in cases of obfuscated or compiled code. Promising areas of further research include optimizing the model for embedded systems, studying explainability approaches for visualizing AI module solutions, and expanding the architecture for multilingual support of code analysis. The results form the basis for creating a more secure and intelligent software quality control tool in the face of increasing security requirements.

References

1. Spring, J. M. (2023). An analysis of how many undiscovered vulnerabilities remain in information systems. Computers & Security, 131, 103191.
2. Duggineni, S. (2023). Impact of controls on data integrity and information systems. Science and technology, 13(2), 29-35.
3. Nakonechna, Y., Savchuk, B., & Kovalova, A. (2024). Fuzzy logic in risk assessment of multi-stage cyber attacks on critical infrastructure networks. Theoretical and Applied Cybersecurity, 6(2), 52-65. https://doi.org/10.20535/tacs.2664-29132024.2.318023
4. Jiang, E., Toh, E., Molina, A., Olson, K., Kayacik, C., Donsbach, A. & Terry, M. (2022, April). Discovering the syntax and strategies of natural language programming with generative language models. In Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (pp. 1-19).
5. Rozlomii, I., Faure, E., & Naumenko, S. (2025). Authentication methods in embedded systems with limited computing resources. Measuring and computing devices in technological processes, (1), 29-35. https://doi.org/10.31891/2219-9365-2025-81-4
6. Abdallah, M., Ngah, A., Al-Rahamneh, A., & Staegmann, D. (2025, May). AI-Based Program Slicing: Techniques, Tools, and

Comparative Analysis. In 2025 12th International Conference on Information Technology (ICIT) (pp. 334-338). IEEE.

7. Sergeyuk, A., Golubev, Y., Bryksin, T., & Ahmed, I. (2025). Using AI-based coding assistants in practice: State of affairs, perceptions, and ways forward. Information and Software Technology, 178, 107610.

8. Marotta, G., Lena, E., Brajnik, G., Novak, I., Colciago, M., Giani, E., & Baffa, C. (2024, July). Enhancing SKA software testing through data mining strategies. In Software and Cyberinfrastructure for Astronomy VIII (Vol. 13101, pp. 168-181). SPIE.

9. Zadereyko, O., Trofymenko, O., Prokop, Y., Loginova, N., Dyka, A., & Kukharenko, S. (2022). Research of potential data leaks in information and communication systems. Radioelectronic and computer systems, (4), 64-84.

10. Donnelly, J., Daneshkhah, A., & Abolfathi, S. (2024). Forecasting global climate drivers using Gaussian processes and convolutional autoencoders. Engineering Applications of Artificial Intelligence, 128, 107536.

11. Rozlomii, I., Yehorchenkova, N., Yarmilko, A., & Naumenko, S. (2023). Data Protection in the Utilization of Natural Language Processors for Trend Analysis and Public Opinion: ryptographic Aspect. In Proceedings of the 2nd International Workshop on Social Communication and Information Activity in Digital Humanities (SCIA-2023) (pp. 1-11).

12. Bloch, T., Borrmann, A., & Pauwels, P. (2023). Graph-based learning for automated code checking–Exploring the application of graph neural networks for design review. Advanced Engineering Informatics, 58, 102137.

13. Balami, B., & Shakya, J. (2024). Comparative Analysis of Transformer and CodeBERT for Program Translation. National College of Computer Studies Research Journal, 3(1), 19-32.

14. Kuszczyński, K., & Walkowski, M. (2023). Comparative analysis of open-source tools for conducting static code analysis. Sensors, 23(18), 7978.

15. Youn, D., Lee, S., & Ryu, S. (2023). Declarative static analysis for multilingual programs using CodeQL. Software: Practice and Experience, 53(7), 1472-1495.

16. Olateju, O., Okon, S. U., Igwenagu, U., Salami, A. A., Oladoyinbo, T. O., & Olaniyi, O. O. (2024). Combating the challenges of false positives in AI-driven anomaly detection systems and enhancing data security in the cloud. Available at SSRN 4859958.