UDC 004. 056.55:004.312.2 **doi: 10.26906/SUNZ.2025.3.106-111**

S. Kuzhel[1], A. Lytvynov[1], O. Pliekhov[2]

[1] Kharkiv National University of Radio Electronics, Kharkiv, Ukraine
[2] V.N. Karazin Kharkiv National University, Kharkiv, Ukraine

# METHODS FOR ANALYZING SOFTWARE SOURCE CODE

**Abstract. Relevance.** The study of methods for analyzing source code is driven by several current trends in the field of software engineering. On the one hand, the increasing complexity and scale of software solutions necessitate the enhancement of software quality and stability. On the other hand, the growing frequency of cyber threats demands greater attention to the security of source code. Modern software systems are involved in critical areas such as financial operations, healthcare, industrial automation, and infrastructure management. Errors and deficiencies in such systems can lead to serious consequences, including significant economic losses, failures in the operation of essential services, and even threats to human life. In an environment of intensifying competition and rapid digitalization, software quality has become a key factor in the success of companies in the market. The implementation of effective source code analysis methods is becoming essential not only for large enterprises, but also for small and medium-sized businesses seeking to develop reliable and secure software solutions. Emphasis should be placed on the promising potential of integrating modern technologies of artificial intelligence and machine learning into the process of code analysis. Such approaches enable the automatic detection and classification of errors, which significantly reduces the time required for their identification and resolution, while also improving the accuracy and efficiency of the analysis. Thus, the development and deepening of research in the field of source code analysis methods is a vital task of modern software engineering, as it enables the resolution of urgent challenges related to software quality assurance and system security. **The object of research**. the source code of software is considered a structure subject to formal, semantic, and behavioral analysis, aimed at identifying errors, vulnerabilities, architectural flaws, and violations of coding standards. **Purpose of the article.** The study focuses on existing methods of source code analysis, particularly static and dynamic approaches, their tool support, and the prospects of applying modern technologies, especially artificial intelligence, to enhance the efficiency of error detection, and to ensure the quality, reliability, and security of software code throughout all stages of the software development lifecycle. **Research results.** The classification of software source code analysis methods has been systematized, including static, dynamic, hybrid, and intelligent approaches. A comparative analysis of static and dynamic techniques has been conducted based on key criteria such as efficiency, error coverage, resource intensity, and applicability at various stages of the software development lifecycle. Typical categories of errors detectable through dynamic analysis have been identified, including memory leaks, resource access errors, and performance issues. The potential of intelligent tools, particularly neural network-based models such as code2vec and VulLibMiner, has been examined for automated analysis and vulnerability prediction. The feasibility of a comprehensive approach that integrates both static and dynamic analysis has been substantiated as the most effective strategy for ensuring the quality and security of software systems. **Conclusions.** Static analysis is effective for early error detection and ensuring code compliance with established standards. Dynamic analysis is essential for identifying runtime errors such as memory leaks and race conditions. Neither method is universal; the best results are achieved through their combination. Intelligent approaches (AI/ML) significantly enhance the automation and accuracy of code analysis. The comprehensive implementation of code analysis contributes to the development of secure, high-quality, and maintainable software.

**Keywords:** source code analysis, static analysis, dynamic analysis, software defects, code security, software quality, analysis tools, artificial intelligence, machine learning, automated verification, code2vec, VulLibMiner, CI/CD.

## Introduction

Modern information technologies have fundamentally transformed approaches to software development. The complexity of software products continues to increase, which in turn raises the risks of errors, vulnerabilities, and deficiencies in the source code. For this reason, the effective application of source code analysis methods has become critically important. Their main objective is to ensure code quality, security, reliability, and compliance with established standards. These analysis techniques assist developers in quickly identifying and eliminating errors, which ultimately helps reduce the time and resources required for software development and testing.

Depending on the verification approach, analysis methods are generally categorized into static (performed without executing the code) and dynamic (conducted during program execution). Static analysis enables the detection of syntactic and semantic errors, logical flaws, and vulnerabilities at early development stages, while dynamic analysis allows for the observation of program behavior under real operating conditions, identifying resource leaks, memory handling issues, multithreading errors, and other problems that are difficult to predict using static methods alone.

In recent years, the integration of artificial intelligence into source code analysis methods has significantly increased. The use of machine learning enables the automation of the analysis process, the classification of errors, and the prediction of their occurrence – all of which substantially enhance the efficiency and quality of software development.

**Review of Recent Studies and Publication.** The issue of source code analysis is actively explored in the context of software quality assurance, software testing, and information security. Most academic publications classify analysis methods into static, dynamic, and hybrid categories, while also highlighting the growing importance of automated tools and machine learning algorithms. In article [1], the authors describe the development and deployment of Google’s internal tool called

Tricorder. It is a scalable ecosystem for static source code analysis. The primary goal of the project was to create a system capable of detecting bugs and vulnerabilities in code before execution, fully integrated into developers' daily workflows. Tricorder supports multiple programming languages and operates on a modular architecture, allowing for the inclusion of various analyzers. A distinctive feature of the platform is its scalability, the ability to continuously update verification rules, and a focus on automated, real-time feedback delivery. The article demonstrates how static analysis can be organically integrated into large-scale production workflows to improve overall software quality in high-intensity development environments.

Article [2] is dedicated to Google's experience with the use of the static analysis tool FindBugs in Java projects. The authors describe the Fixit initiative, a focused effort in which developers dedicated time exclusively to resolving issues identified by FindBugs. During the experiment, thousands of defects were analyzed, most of which fell into categories such as null reference errors, incomplete initialization, and the use of potentially dangerous programming patterns. Notably, numerous defects were discovered even in well-tested code that had previously gone unnoticed. The authors conclude that the regular application of static analysis significantly improves code quality without requiring additional testing resources. This publication is particularly valuable as an example of how source code analysis can be effectively adopted at industrial scale.

Study [3] explores the importance of static source code analysis from a security perspective. The authors emphasize that many common vulnerabilities, such as buffer overflows, SQL injections, XSS, and other standard security issues, can be detected at early development stages using specialized analyzers. Using tools like Fortify and Coverity as examples, the authors demonstrate the capability to automatically detect a wide range of security flaws without executing the code. The paper highlights that static analysis should be a systematic part of the secure development process, not a one-time testing effort. The importance of proper developer training and tool configuration for achieving effective results is also underlined. This work remains foundational in the field of secure programming and retains its relevance amid growing cyber threats.

Paper [4] presents an empirical study of bugs in modern open-source software. The authors analyzed bugs reported in the repositories of several major open-source projects, including Mozilla, Apache, and Eclipse, and classified them by type, origin, and consequences. It was found that most errors stem from human factors – incorrect assumptions, API changes, or complex component interactions. Special attention is given to the difficulty of identifying logical errors, which are not always caught by tests or compilers. The authors conclude that effective code quality control requires a combined strategy that includes both static and dynamic analysis, as well as active user feedback. This study is valuable for gaining a realistic understanding of the nature of software defects and identifying the most error-prone areas of code.

As a result of study [5], the VulLibMiner framework was developed, introducing a novel approach to the automated detection of vulnerable third-party Java libraries based solely on textual vulnerability descriptions. This is highly relevant in modern software development practice, where a significant portion of code depends on external libraries, often used without thorough security audits. The authors also discuss the future development prospects of the system, including its scalability to other programming languages, integration into IDEs, and automatic generation of recommendations for updating insecure libraries.

Publication [6] presents an innovative approach to code analysis using deep learning techniques. The study introduces the code2vec system, which transforms source code into vector representations, enabling the use of neural network models for tasks such as classification, recommendation, and bug prediction. The method is based on representing code snippets as paths in the abstract syntax tree, which are then fed into a neural network. As a result, the system can «understand» the structure of code and learning to recognize patterns associated with certain types of errors or stylistic deviations. code2vec exemplifies a new generation of code analysis tools that combine classical syntactic analysis methods with artificial intelligence capabilities. This work is of fundamental importance to the development of intelligent code analysis systems.

The reviewed publications demonstrate the high scientific and practical relevance of source code analysis methods as one of the key tools for ensuring software quality, security, and reliability. The research shows that static analysis is extremely effective for the early detection of syntactic, semantic, and logical errors – particularly at scale in corporate environments (e.g., Google and SAP) – and for identifying vulnerabilities before program execution. Industrial practice indicates that static analysis integrated into CI/CD pipelines significantly reduces technical debt and improves release quality.

Meanwhile, dynamic analysis is considered a complementary yet essential stage that enables the detection of issues related to performance, memory leaks, concurrency, and real-world execution behavior. Publications describing tools such as VulLibMiner highlight the growing importance of analyzing third-party libraries and dependencies, which constitute a critical part of the modern software landscape.

**The purpose of this work** is to investigate and critically analyze modern methods of source code analysis, including static, dynamic, and intelligent approaches, to evaluate their effectiveness, applicability at various stages of the software development lifecycle, and the potential for integrating machine learning tools to enhance the quality, security, and reliability of software code amid the increasing complexity of software systems.

## Main part

Source code analysis is a systematic process of examining the program text to identify syntactic, semantic, logical, and architectural errors, as well as to verify compliance with programming standards, security requirements, and performance. This analysis is one of the fundamental stages of the software development lifecycle, as it allows minimizing risks associated with errors at later stages such as testing, deployment, or operation.

The source code, or source text of programs, serves as the main object of research. Unlike binary analysis, which works with executable files, source code analysis provides a deeper and more contextual understanding of the program's logic, structural features, component interconnections, and potential risk areas.

In modern software engineering, code analysis performs several important functions: quality assurance, security enhancement, standards compliance auditing, maintenance, and support. There are various methods of source code analysis, conventionally divided into static (performed without executing the program) and dynamic (requiring execution of the code or its parts in a controlled environment). Hybrid approaches that combine features of both types and intelligent methods based on machine learning algorithms [7] are also distinguished. Depending on the objective, analysis can be carried out at different levels: syntactic, semantic, structural, functional, and behavioral. Source code analysis is not only a technical process but also an integral part of development culture – a practice that allows creating reliable, maintainable, and secure software.

Methods of source code analysis are classified according to different criteria, particularly by processing method, environment type, software lifecycle phase, and degree of automation. The most fundamental and common classification is the distinction between static and dynamic analysis methods. Both approaches have their advantages, limitations, and application specifics.

Static analysis involves analyzing code without executing it. It is performed at the development or compilation stage and allows detecting syntactic, semantic, and structural errors. These methods are based on building syntax trees, function call graphs, and analyzing data and control flow. They are used to check compliance with coding standards, detect code duplication, incorrect constructions, "dead" code, or potentially unsafe operations. Examples of tools include SonarQube, ESLint, PVS-Studio, Coverity, and others. The advantages of static analysis include its ability to work without program execution and identify issues at early stages.

However, despite many advantages, static analysis has several limitations. Firstly, it cannot provide a complete check of the program's dynamic behavior. For example, it cannot detect memory leaks, race conditions, failures due to lack of resources, or other errors that manifest only during execution. In such cases, dynamic analysis methods are necessary.

Another limitation is the high level of false positives – situations where the analyzer reports an error that does not actually exist. This may reduce trust in the tool and lead to unnecessary time spent by developers. Some tools also have limited support for certain programming languages or do not consider the specifics of non-standard architectures, which reduces analysis accuracy.

It should also be noted that the effective use of static analyzers requires appropriate skills and configuration, as different projects may have their own stylistic, architectural, or domain-specific characteristics that need to be considered when setting up the analysis system.

Dynamic analysis, unlike static analysis, is performed during program execution. It allows examining the software's behavior in a real or simulated execution environment. Dynamic analysis includes profiling, tracing, memory usage monitoring, detection of memory leaks, incorrect variable access, performance checking, thread analysis, etc. This approach is necessary for detecting errors that cannot be identified statically, especially in multithreaded or event-driven systems. Examples include Valgrind, Intel VTune, AddressSanitizer, and IDE profilers. Its main advantage is analysis accuracy, but it requires time, a configured test environment, and coverage of real scenarios.

In addition to the basic classification, the following are also distinguished: hybrid methods combining the advantages of static and dynamic approaches to achieve higher accuracy; semantic analysis that examines the meaning and context of instructions and helps detect logic inconsistencies; formal analysis based on mathematical models to prove program correctness; intelligent methods using machine learning or deep learning algorithms for automated error pattern detection, code style analysis, or vulnerability prediction; dependency analysis that studies the use of third-party libraries and packages, particularly for known vulnerabilities or licensing incompatibility.

The disadvantages of dynamic source code analysis lie in a range of technical, methodological, and practical limitations that complicate its application or reduce its effectiveness under certain conditions (see Fig. 2). Dynamic analysis depends on which code fragment is executed during testing. If a specific branch of logic is not activated in the selected scenario, the related errors will remain undetected. That is, it does not guarantee complete verification unless all execution paths are covered.

Running a program with monitoring tools significantly increases CPU load, memory usage, and execution time. Some tools slow down program performance by tens of times (e.g., Valgrind), making them unsuitable for large or production-level systems. Performing dynamic analysis requires creating a test environment that simulates the system's operation under conditions close to real ones. This demands additional resources, configurations, interaction scenarios, and input/output data.

In large distributed or microservice systems, dynamic analysis may require simultaneous execution of many components, complicating automation –especially in cases of limited access to full infrastructure (e.g., cloud services or container clusters).

Dynamic tools do not always provide complete information about the source of an error, especially if it arises in a compiled module without access to the source code. This complicates further localization and elimination of the defect.

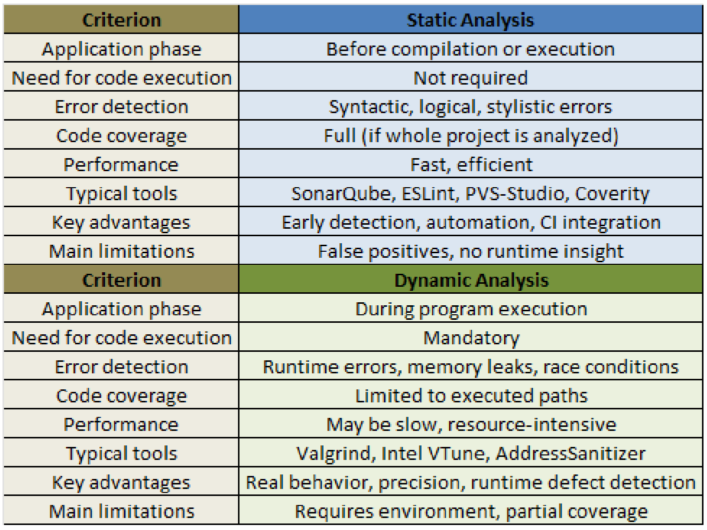
Additionally, effective dynamic analysis requires a set of realistic and well-designed test cases. If such tests are absent or of low quality, a significant portion of errors will remain unnoticed.

Dynamic analysis is an indispensable means of detecting behavioral, multithreading, and resource-related problems. However, it should not be used in isolation. Its limitations are compensated by combining it with static analysis methods, integration into CI/CD pipelines, and careful preparation of the test environment.

The classification of source code analysis methods reflects the diversity of technical tasks faced by developers and analysts and allows for selecting appropriate approaches depending on goals, development stage, and software characteristics. Fig. 1 visualizes the key features of both methods according to main criteria and demonstrates that neither static nor dynamic analysis is self-sufficient; however, together, they form a powerful toolkit for deep diagnostics, quality control, and ensuring the security of software code.

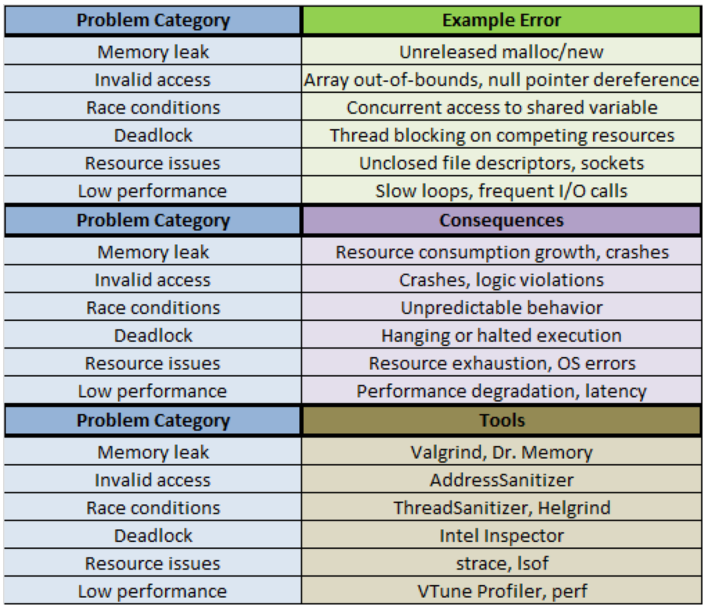
Their integration into the software development lifecycle is critically important for modern software engineering.

Fig. 2 systematizes the typical errors identified by dynamic methods of software analysis, classifying them into three key aspects: the category of the issue, an example of the error and its consequences, and provides appropriate tools for detecting each type of defect.



| Criterion | Static Analysis |
|---|---|
| Application phase | Before compilation or execution |
| Need for code execution | Not required |
| Error detection | Syntactic, logical, stylistic errors |
| Code coverage | Full (if whole project is analyzed) |
| Performance | Fast, efficient |
| Typical tools | SonarQube, ESLint, PVS-Studio, Coverity |
| Key advantages | Early detection, automation, CI integration |
| Main limitations | False positives, no runtime insight |
| **Criterion** | **Dynamic Analysis** |
| Application phase | During program execution |
| Need for code execution | Mandatory |
| Error detection | Runtime errors, memory leaks, race conditions |
| Code coverage | Limited to executed paths |
| Performance | May be slow, resource-intensive |
| Typical tools | Valgrind, Intel VTune, AddressSanitizer |
| Key advantages | Real behavior, precision, runtime defect detection |
| Main limitations | Requires environment, partial coverage |

**Fig. 1.** Comparative analysis of static and dynamic methods of source code analysis



| Problem Category | Example Error |
|---|---|
| Memory leak | Unreleased malloc/new |
| Invalid access | Array out-of-bounds, null pointer dereference |
| Race conditions | Concurrent access to shared variable |
| Deadlock | Thread blocking on competing resources |
| Resource issues | Unclosed file descriptors, sockets |
| Low performance | Slow loops, frequent I/O calls |
| **Problem Category** | **Consequences** |
| Memory leak | Resource consumption growth, crashes |
| Invalid access | Crashes, logic violations |
| Race conditions | Unpredictable behavior |
| Deadlock | Hanging or halted execution |
| Resource issues | Resource exhaustion, OS errors |
| Low performance | Performance degradation, latency |
| **Problem Category** | **Tools** |
| Memory leak | Valgrind, Dr. Memory |
| Invalid access | AddressSanitizer |
| Race conditions | ThreadSanitizer, Helgrind |
| Deadlock | Intel Inspector |
| Resource issues | strace, lsof |
| Low performance | VTune Profiler, perf |

**Fig. 2.** Typical Issues in Dynamic Analysis

Dynamic analysis is a critically important stage in ensuring software quality, particularly for detecting errors that cannot be identified through static analysis. Without its implementation, it is impossible to guarantee the performance, stability, or security of complex systems. Therefore, the competent use of dynamic tools, aligned with the types of potential errors, is an essential component of the modern development process.

The ideal strategy involves the sequential application of both approaches. In the early stages of the software life cycle, static analysis is employed to quickly and cost-effectively identify syntactic, stylistic, and logical errors, violations of coding standards, and potential vulnerabilities. This stage is particularly effective when integrated into continuous integration (CI) systems, enabling code quality control during development. After development and compilation are completed – or in parallel – dynamic analysis is introduced. It focuses on testing the software's behavior in real-world conditions, identifying resource interaction issues, analyzing multithreading, performance, and overall system stability.

A comparative analysis of the effectiveness of these approaches demonstrates that their combination leads to significantly better results in reducing software defects than relying on either approach alone. For instance, static analyzers can successfully detect up to 70–80% of defects at early development stages, whereas dynamic methods are effective in identifying issues that are not reflected in the code's structure but cause failures during execution. Thus, the integrated use of both methods provides more reliable protection against critical errors and substantially reduces risks during software operation.

Innovative techniques are also used in formal validation analysis. For example, systems trained on thousands of known vulnerabilities can predict the appearance of similar defects in new projects based on architectural or API-call similarities. These models can identify zero-day vulnerabilities even before they are exploited.

Intelligent code analysis offers several advantages: it operates in real-time, learns from new examples, adapts to a team's coding style, and can be integrated into IDEs or CI/CD systems. However, these methods require high-quality training datasets, substantial computational resources, and proper model configuration – factors that may limit their use in small teams or constrained environments.

Despite these limitations, the rapid development of AI-based tools indicates that intelligent methods are poised to become the foundation of a new generation of code analysis systems – adaptive, context-aware, and self-learning. This evolution will not only improve the efficiency of error detection but also make software development more secure, stable, and predictable. Therefore, the integration of AI into source code analysis is not merely a technical innovation but a strategic necessity for the future of software engineering.

## Conclusions

As a result of the conducted research, a systematic analysis was carried out on the main methods used for examining the source code of software, aimed at detecting errors, vulnerabilities, violations of coding standards, and ensuring the overall quality and security of the software product. The key characteristics, advantages, and limitations of both static and dynamic analysis were reviewed, along with the prospects for the development of intelligent methods based on machine learning algorithms.

The comparative study showed that static analysis is effective during the early stages of development when the program has not yet been executed. It enables the detection of syntactic and logical errors, coding style violations, code duplication, and potential security threats that are independent of the execution context. At the same time, dynamic analysis demonstrates its effectiveness in scenarios where it is necessary to evaluate the program's behavior in a real runtime environment. It can detect memory leaks, race conditions, thread blocking, resource issues, and performance problems – i.e., aspects that static analysis does not cover.

It is worth noting that the combined use of static and dynamic analysis represents the most effective approach, as it ensures comprehensive coverage of potential issues in the code and allows for enhanced software stability and quality. The integration of both approaches into the development process, particularly within CI/CD pipelines, is a practice that is increasingly being adopted across the industry.

Special attention was given to innovative source code analysis methods involving artificial intelligence. Modern deep learning-based systems such as code2vec, VulLibMiner, and others demonstrate significant potential in improving defect detection and vulnerability prediction before they occur. These approaches enable substantial automation of the analysis process, adaptation to team-specific coding styles, and reduced workload for developers.

Therefore, the research confirms that the use of source code analysis methods is a necessary condition for the development of high-quality, secure, and maintainable software. The implementation of analytical tools within the development process promotes a higher coding culture, reduces the number of critical errors, and lays the foundation for a long software product lifecycle. The further development of the field will undoubtedly be linked to the growing role of artificial intelligence, the improvement of automated verification tools, and the adoption of hybrid analysis strategies that encompass both syntactic and behavioral aspects of code.

## References

1. C. Sadowski, J. van Gogh, Jaspan C., E. Söderberg, C. Winter. Tricorder: Building a program analysis ecosystem. ICSE '15: Proceedings of the 37th International Conference on Software Engineering, vol., 2015. P. 598-608. https://doi.org/10.1109/ICSE.2015.76 .
2. Ayewah N., Pugh W. The Google FindBugs fixit. ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis, 2010. P. 241-252. https://doi.org/10.1145/1831708.1831738 .
3. B. Chess, G. McGraw. Static Analysis for Security. IEEE Security & Privacy, vol. 2, No. 6, 2004. P. 76-79. https://doi.org/10.1109/MSP.2004.111 .
4. Z. Li, L. Tan, Y. Wang, S. Lu, Y. Zhou, C. Zhai. Have Things Changed Now? An Empirical Study of Bug Characteristics in Modern Open Source Software. Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, ASID 2006, San Jose, California, USA, October 21, 2006. 9 p. https://doi.org/10.1145/1181309.1181314.

5. T. Chen, L. Li, B. Shan, G. Liang, D. Li, Q. Wang, T. Xie. Identifying Vulnerable Third-Party Java Libraries from Textual Descriptions of Vulnerabilities and Libraries. Cornell University. Computer Science. Cryptography and Security, 2023. 23 p. https://doi.org/10.48550/arXiv.2307.08206 .
6. Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav. code2vec: Learning Distributed Representations of Code. Cornell University. Computer Science. Machine Learning, 2018. 23 p. https://doi.org/10.48550/arXiv.1803.09473 .
7. Flach P. A. Machine Learning: The Art and Science of Algoritms that Makes Sense of Data. Cambridge: Cambridge University Press, 2012. 291 p. https://doi.org/10.1017/CBO9780511973000 .

Received (Надійшла) 02.06.2025
Accepted for publication (Прийнята до друку) 27.08.2025

ВІДОМОСТІ ПРО АВТОРІВ / ABOUT THE AUTHORS

**Кужель Сергій Ігорович** – студент кафедри електронних обчислювальних машин, Харківський національний університет радіоелектроніки, Харків, Україна;
**Serhii Kuzhel** – student, Department of Electronic Computers, Kharkiv National University of Radio Electronics, Kharkiv, Ukraine;
e-mail: serhii.kuzhel@nure.ua; ORCID Author ID: http://orcid.org/0009-0001-7778-4881.

**Литвинов Андрій Павлович** – студент кафедри електронних обчислювальних машин, Харківський національний університет радіоелектроніки, Харків, Україна;
**Andrii Lytvynov** – student, Department of Electronic Computers, Kharkiv National University of Radio Electronics Kharkiv, Ukraine;
e-mail: andrii.lytvynov1@nure.ua; ORCID Author ID: http://orcid.org/0009-0008-1254-1513.

**Плєхов Олександр Вікторович** – магістрант, кафедра органічного сінтезу, Харківський національний університет імені В.Н. Каразіна, Харків, Україна;
**Oleksandr Pliekhov** – master, Department of Organic Synthesis, V.N. Karazin Kharkiv National University, Kharkiv, Ukraine;
e-mail: aleks.plekhov@gmail.com; ORCID Author ID: http://orcid.org/0009-0000-1343-4308.

## Методи аналізу вихідних текстів програм

С. І. Кужель, А. П. Литвинов, О. В. Плєхов

**Анотація. Актуальність.** Актуальність дослідження методів аналізу вихідних текстів програм зумовлена низкою сучасних тенденцій у сфері програмної інженерії. З одного боку, зростає складність і масштаби програмних рішень, що обумовлює необхідність підвищення якості та стабільності програмного забезпечення. З іншого боку, зростає частота кіберзагроз, що змушує особливу увагу приділяти безпеці програмного коду. Сучасні програмні системи задіяні у критичних сферах: фінансових операціях, охороні здоров'я, автоматизації виробництва, управлінні інфраструктурою тощо. Помилки і недоліки в таких системах можуть мати серйозні наслідки, що призводять до великих економічних втрат, збоїв у роботі критичних служб і навіть створюють загрозу життю людей. В умовах посилення конкуренції та стрімкої цифровізації якість програмного забезпечення стає визначальним фактором успіху компаній на ринку. Впровадження ефективних методів аналізу вихідних текстів програм стає необхідністю не лише для великих підприємств, але й для малих і середніх компаній, що прагнуть створювати надійне та безпечне програмне забезпечення. Окремо варто підкреслити перспективність інтеграції сучасних технологій штучного інтелекту та машинного навчання у процес аналізу коду. Такі підходи дозволяють здійснювати автоматичний пошук і класифікацію помилок, що значно скорочує витрати часу на їх виявлення і усунення, а також підвищує точність та ефективність аналізу. Таким чином, розвиток та поглиблення досліджень у сфері методів аналізу вихідних текстів програм є важливим завданням сучасної програмної інженерії, що дозволяє вирішувати актуальні проблеми забезпечення якості та безпеки програмних систем. **Об'єкт дослідження**: вихідні тексти програмного забезпечення як структура, що підлягає формальному, семантичному та поведінковому аналізу з метою виявлення помилок, вразливостей, недоліків архітектури та порушень стандартів кодування. **Мета статті**: дослідження існуючих методів аналізу вихідних текстів програм, зокрема статичних і динамічних підходів, їх інструментального забезпечення та перспектив застосування сучасних технологій, зокрема штучного інтелекту, для підвищення ефективності виявлення помилок, забезпечення якості, надійності та безпеки програмного коду на всіх етапах життєвого циклу розробки програмного забезпечення. **Результати дослідження**. Систематизовано класифікацію методів аналізу вихідного коду програмного забезпечення, зокрема статичних, динамічних, гібридних та інтелектуальних. Проведено порівняльний аналіз статичного та динамічного підходів за ключовими критеріями: ефективність, охоплення помилок, ресурсомісткість та застосовність на різних етапах життєвого циклу ПЗ. Визначено типові категорії помилок, які виявляються динамічними методами: витоки пам’яті, помилки доступу до ресурсів та проблеми продуктивності. Проаналізовано можливості використання інтелектуальних інструментів, зокрема нейромережевих моделей (code2vec, VulLibMiner), для автоматизованого аналізу та прогнозування вразливостей. Обґрунтовано доцільність комплексного підходу, що поєднує статичний і динамічний аналіз, як найбільш ефективну стратегію забезпечення якості та безпеки програмного забезпечення. **Висновки**. Статичний аналіз ефективний для раннього виявлення помилок і відповідності коду стандартам. Динамічний аналіз необхідний для виявлення помилок виконання, таких як витоки пам’яті та умови гонки. Жоден метод не є універсальним. Найкращі результати дає їх поєднання. Інтелектуальні підходи (AI/ML) значно підвищують автоматизацію та точність аналізу. Комплексне впровадження аналізу коду сприяє створенню безпечного, якісного та підтримуваного ПЗ.

**Ключові слова:** аналіз вихідного коду, статичний аналіз, динамічний аналіз, помилки в програмному забезпеченні, безпека коду, якість програмного забезпечення, інструменти аналізу, штучний інтелект, машинне навчання, автоматизована перевірка, code2vec, VulLibMiner, CI/CD.