ДОДАТОК А

Графічний матеріал кваліфікаційної роботи

кваліфікаційна робота

Метод предметно-орієнтовного аналізу вихідних текстів програм

Виконав:
студент гр. СПм-23-5
Литвинов А.П.

Керівник:
доц. каф. ЕОМ, к.т.н.
Ткачов В.М.

Мета роботи та завдання

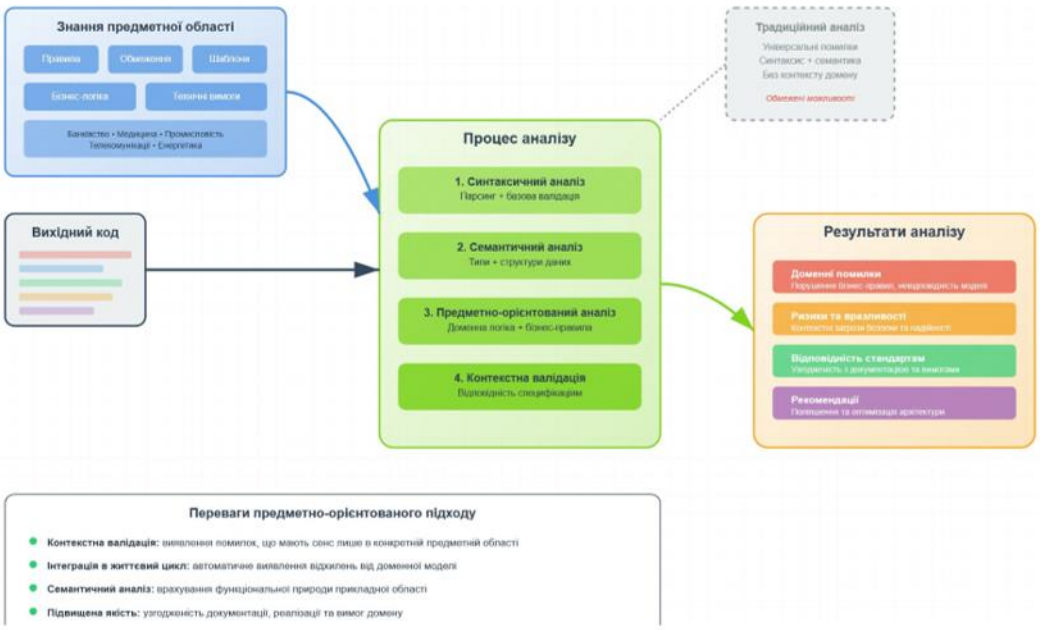
Метою роботи є розробка та реалізація комплексного методу аналізу вихідного коду програмного забезпечення, який поєднує класичні та інноваційні підходи.

Задачі:

- Провести аналіз сучасних підходів до аналізу вихідного коду програмного забезпечення, зокрема статичних, динамічних та інтелектуальних методів.
- Сформулювати вимоги до предметно-орієнтованого методу аналізу вихідного коду, який враховує як синтаксичні, так і семантичні характеристики програм.
- Розробити алгоритм реалізації обраного методу з урахуванням можливості його практичного використання у середовищі Google Colab.
- Реалізувати програмний засіб аналізу коду на мові програмування Python із застосуванням бібліотек для обробки, верифікації та візуалізації вихідного тексту.
- Провести експериментальне тестування розробленого методу на прикладах коду з помилками різного типу, у тому числі з реалістичними складними сценаріями.
- Забезпечити побудову візуалізацій за результатами аналізу, включно з графіками розподілу елементів, частоти помилок та особливостей структури.
- Оцінити ефективність запропонованого методу, проаналізувавши точність виявлення помилок, зручність інтерпретації результатів та можливість інтеграції у практичні процеси розробки.

Об’єкт дослідження: вихідний код програмного забезпечення, який розглядається як структурована текстова форма, що містить формальні інструкції для виконання програмою, і піддається систематичному аналізу з метою виявлення помилок, відхилень від стандартів, потенційних вразливостей та загальних характеристик якості програмного продукту.

Переваги предметно-орієнтованого підходу 3



Основні завдання аналізу вихідних текстів програм 4

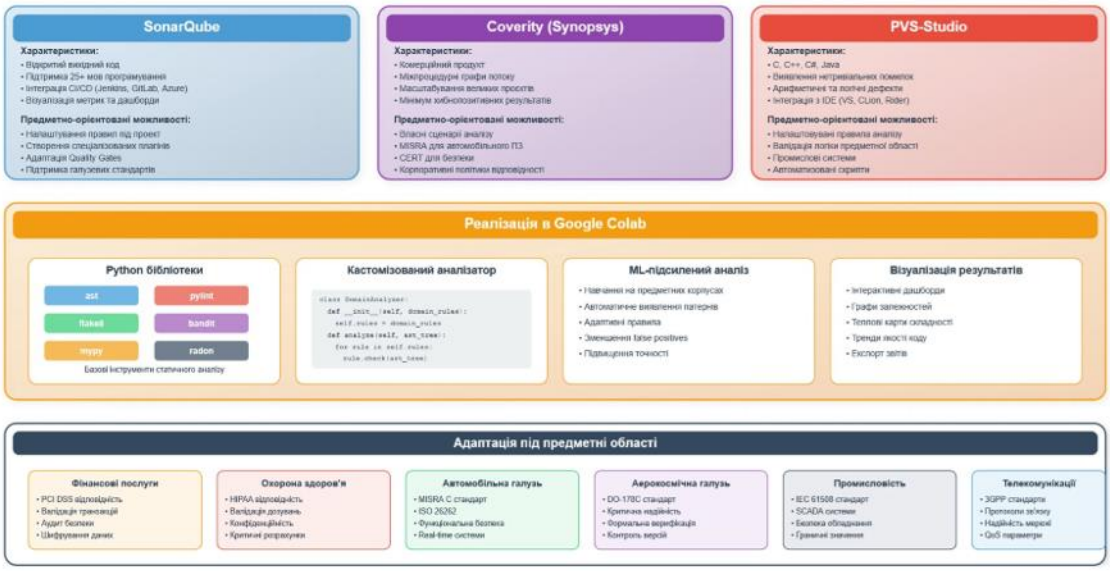
Порівняльний аналіз методів 5

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

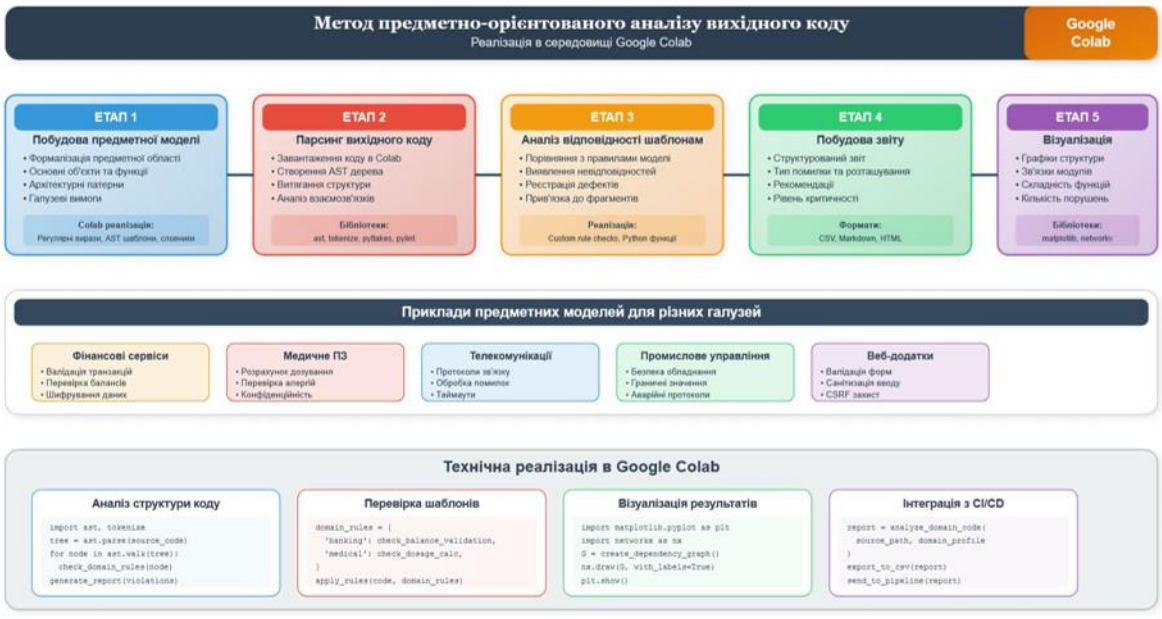
Типові проблеми при динамічному аналізі 6

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

7 Огляд інструментів статичного аналізу



8 Метод предметно-орієнтованого аналізу вихідного коду



Реалізація в Google Colab 9

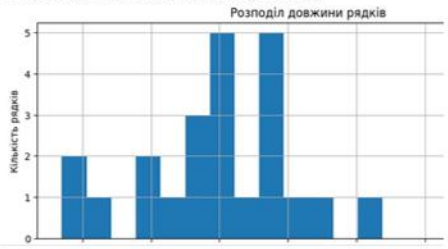
```
# Завантаження користувацького файлу
from google.colab import files
uploaded = files.upload()

# Зчитування коду з файлу
filename = next(iter(uploaded))
with open(filename, 'r') as file:
    lines = file.readlines()

# Виведення перших рядків
print("Перші 10 рядків коду:")
print(''.join(lines[:10]))
```

```
Browse... updated_code_analysis_notebook.ipynb
updated_code_analysis_notebook.ipynb(n/a) - 3079 bytes, last modified: n/a - 100% done
Saving updated_code_analysis_notebook.ipynb to updated_code_analysis_notebook.ipynb
Перші 10 рядків коду:
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "# Аналіз вихідного коду з візуалізацією\n",
    "Цей ноутбук реалізує метод аналізу вихідного коду з підтримкою візуалізації та можливістю
   ]
  },
```

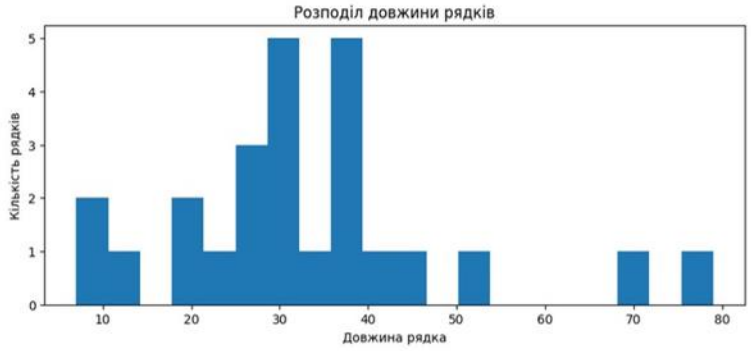
```
[2] # Аналіз коду
    import re
    import matplotlib.pyplot as plt
    import seaborn as sns
    import numpy as np

    # Розрахунок довжини рядків
    line_lengths = [len(line) for line in lines if line.strip()]

    # Ідентифікатори
    identifiers = re.findall(r'\b[_a-zA-Z][_a-zA-Z0-9]*\b', ''.join(lines))
```

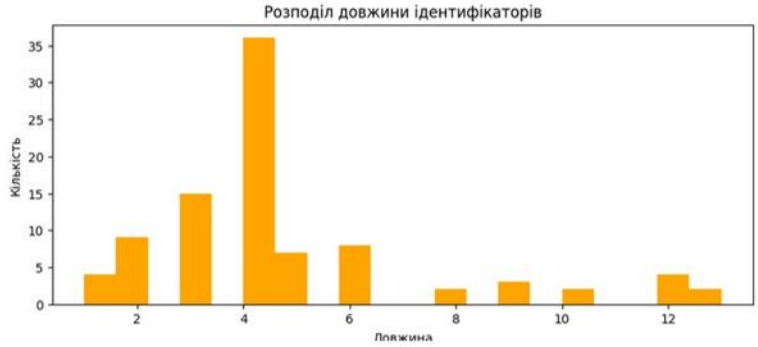
```
plt.ylabel("Кількість")
plt.xticks(rotation=45)
plt.grid(True)
plt.show()

# Простий пошук помилок
errors = []
for i, line in enumerate(lines):
    if "==" in line and "if" not in line and "#" not in line:
        errors.append((i + 1, "Ймовірне порівняння поза умовою"))
    if "except" in line and "as" not in line:
        errors.append((i + 1, "Недостатньо обробки винятку"))
    if re.match(r'^\s*print\s+[^(]', line):
        errors.append((i + 1, "print без дужок (сумісність з Python 2)"))

# Виведення знайдених помилок
if errors:
    print("Виявлено помилки:")
    for err in errors:
        print(f"Рядок {err[0]}: {err[1]}")
else:
    print("Помилок не виявлено.")
```

```
Browse... complex_faulty_code.py
complex_faulty_code.py(n/a) - 836 bytes, last modified: n/a - 100% done
Saving complex_faulty_code.py to complex_faulty_code (1).py
```



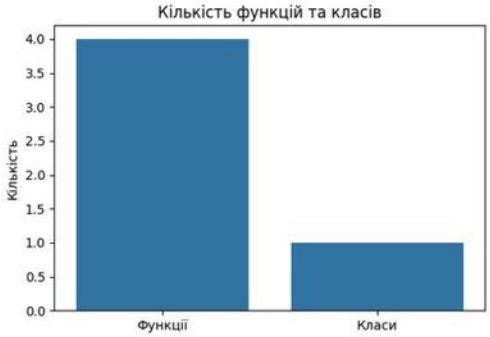
Аналіз результатів 10

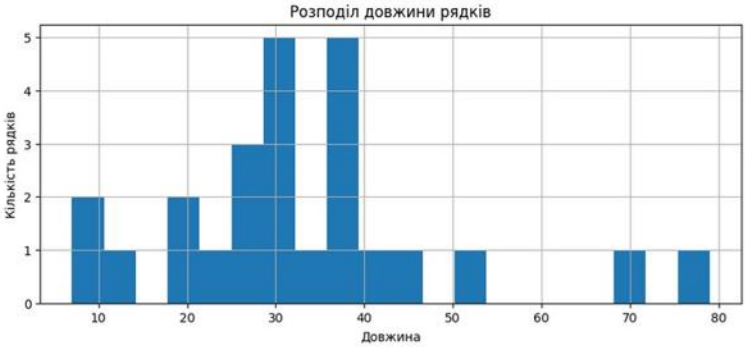




Результати роботи

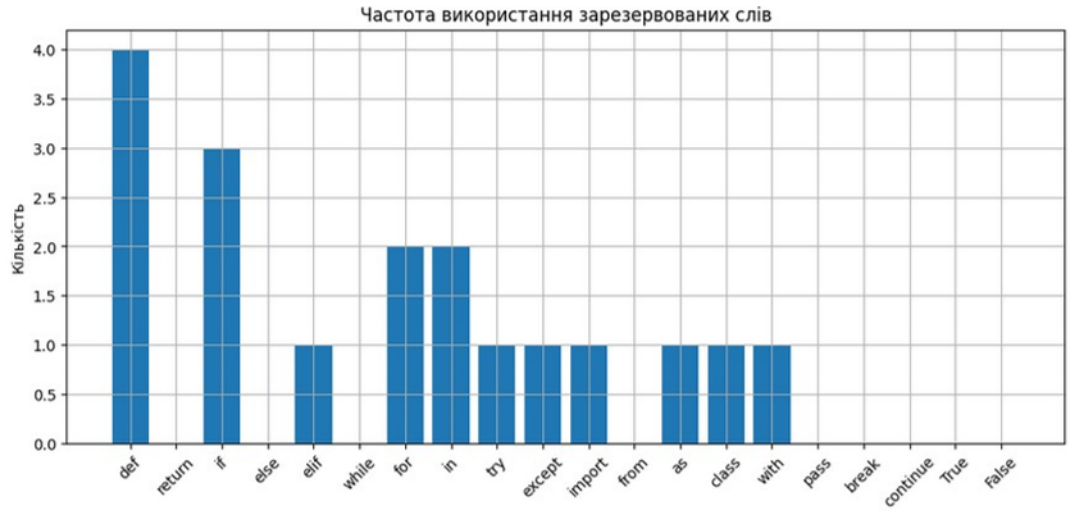
11





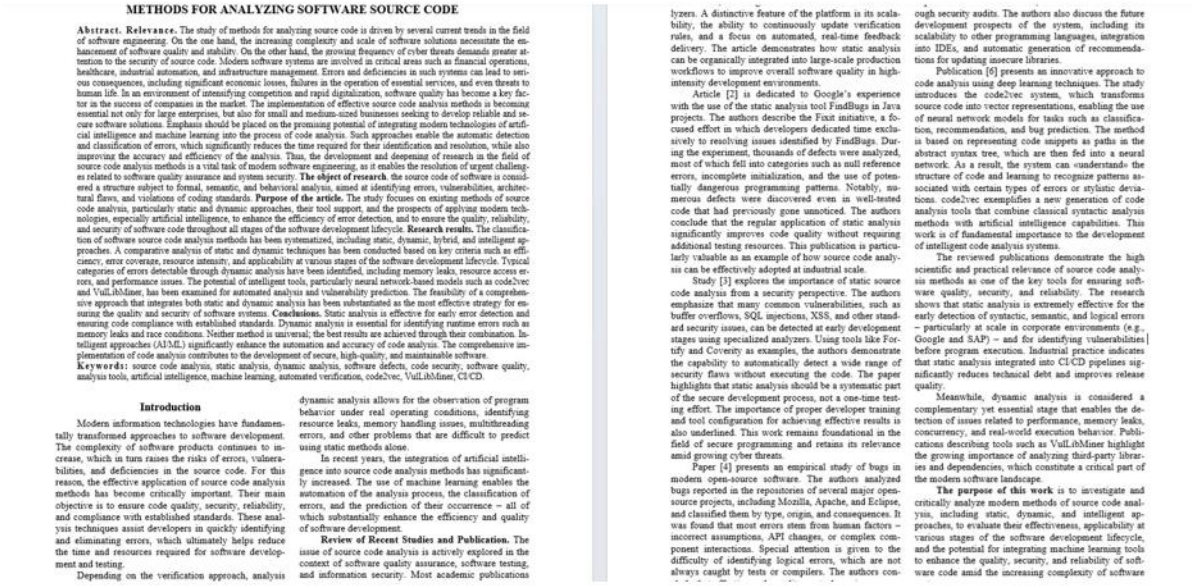
Результати роботи

12




```
Виявлено помилки:
Рядок 16: Недостатньо обробки винятку
```

Апробація результатів 13



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 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

lyzers. 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 con-

ough 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

S. Kuzhel, A. Lytvynov, O. Pliekhov. METHODS FOR ANALYZING SOFTWARE SOURCE CODE. Системи управління, навігації та зв'язку, вип.3. Полтава, 2025. С. 81-86.

Висновки 14

У результаті виконаної роботи було розроблено, реалізовано та протестовано метод предметно-орієнтовного аналізу вихідного коду програм з використанням інструментальних засобів Python у середовищі Google Colab. Запропонований підхід поєднує елементи класичного статичного аналізу з можливостями семантичного розбору, простежування структури коду та візуалізації його характеристик, що дозволяє ефективно виявляти низку поширених помилок, невідповідностей та стилістичних відхилень.

Аналіз проведено як на рівні синтаксичних елементів (ідентифікатори, ключові слова, довжина рядків), так і на рівні логіки структурування (класи, функції, конструкції контролю потоку), що забезпечило глибше розуміння внутрішньої організації програмного коду. Виявлені помилки класифікувались і супроводжувалися поясненням контексту, в якому вони виникають, що полегшує процес корекції та підвищує якість зворотного зв'язку.

Окрема увага приділена візуалізації результатів аналізу: побудовано гістограми розподілу довжини рядків, довжини ідентифікаторів, кількості функцій і класів, частоти вживання зарезервованих слів. Ці графічні представлення сприяють інтуїтивному розумінню загального стилю програмування, виявленню аномалій та можливих зон для рефакторингу.

Практична реалізація методу в Google Colab продемонструвала його доступність, відтворюваність та інтерактивність, що дозволяє використовувати розроблений інструмент як освітній, дослідницький або допоміжний засіб у командній розробці програмного забезпечення. За рахунок підтримки автоматичного завантаження користувацького коду та подальшого аналізу, система є гнучкою і масштабованою.

15

АПРОБАЦІЯ


Є. Ю. Глоба[1], В. Р. Смірнов[1], М.С. Нараєвський[1]

[1] Харківський національний технічний університет радіоелектроніки, Харків, Україна

МЕТОД ВИЯВЛЕННЯ АНОМАЛІЙ В КОРПОРАТИВНІЙ МЕРЕЖІ

Глоба Є. Ю., Смірнов В. Р., Нараєвський М. С. Метод виявлення аномалій в корпоративній мережі. Системи управління, навігації та зв'язку, вип.3. Полтава, 2025. С. 154-158.

16

ВИСНОВКИ

На основі виконаної роботи, яка включала створення, симуляцію, збереження, обробку та візуальний аналіз мережевого трафіку в умовах корпоративної мережі, можна сформулювати низку важливих висновків.

Розроблений метод моніторингу трафіку виявився ефективним для структурованого поділу трафіку на логічні потоки, аналізу вкладених протоколів та обробки корисного навантаження з обрахунком ентропії. Це дозволило не лише ідентифікувати джерела підвищеної мережевої активності, а й оцінити потенційні аномалії без необхідності глибокого інспектування вмісту пакетів, що особливо важливо у випадках використання зашифрованого трафіку.

Аналіз довжин мережевих пакетів засвідчив характерний для корпоративного середовища профіль – переважання коротких службових повідомлень з рідкісними передачами великих пакетів. Це дозволяє зробити висновок про типову інфраструктуру з великою кількістю клієнтських звернень до обмеженої кількості серверів.

17

ВИСНОВКИ

Ентропійний аналіз корисного навантаження вказав на змішаний характер трафіку – одночасну присутність як відкритих, так і потенційно зашифрованих протоколів. Значення ентропії, що коливаються в межах 5.5–6.5 біт, вказують на високу насиченість інформацією, притаманну потокам мультимедіа, VPN або інших зашифрованих сервісів.

Графи логічних з'єднань та розподіл активності за IP-адресами дозволили виокремити як стандартні з'єднання клієнт-сервер, так і потенційно підозрілі концентрації трафіку. Особливо корисною виявилась візуалізація на основі протоколів у часі, яка дозволила простежити інтенсивність і періодичність застосування ICMP та UDP, а також виявити моменти різкого зростання активності – потенційні атаки або автоматизовані скрипти.

ДОДАТОК Б

Програмний код

```
# Завантаження користувацького файлу
from google.colab import files
uploaded = files.upload()

# Зчитування коду з файлу
filename = next(iter(uploaded))
with open(filename, 'r') as file:
    lines = file.readlines()

# Виведення перших рядків
print("Перші 10 рядків коду:")
print(''.join(lines[:10]))
# Аналіз коду
import re
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# Розрахунок довжини рядків
line_lengths = [len(line) for line in lines if line.strip()]

# Ідентифікатори
identifiers = re.findall(r'\b[_a-zA-Z][_a-zA-Z0-9]*\b',
''.join(lines))
identifier_lengths = [len(id_) for id_ in identifiers]

# Пошук кількості функцій і класів
function_count = sum(1 for line in lines if
line.strip().startswith('def '))
class_count = sum(1 for line in lines if
line.strip().startswith('class '))

# Візуалізації
plt.figure(figsize=(10, 4))
plt.hist(line_lengths, bins=20)
plt.title('Розподіл довжини рядків')
plt.xlabel('Довжина рядка')
plt.ylabel('Кількість рядків')
plt.show()

plt.figure(figsize=(10, 4))
plt.hist(identifier_lengths, bins=20, color='orange')
plt.title('Розподіл довжини ідентифікаторів')
plt.xlabel('Довжина')
plt.ylabel('Кількість')
plt.show()

# Візуалізація кількості функцій і класів
```

```
plt.figure(figsize=(6, 4))
sns.barplot(x=['Функції', 'Класи'], y=[function_count,
class_count])
plt.title('Кількість функцій та класів')
plt.ylabel('Кількість')
plt.show()

import matplotlib.pyplot as plt
import re
from collections import Counter
from google.colab import files

# Завантаження файлу користувачем
uploaded = files.upload()
filename = list(uploaded.keys())[0]

with open(filename, 'r') as file:
    lines = file.readlines()

# Перевірка довжини рядків
line_lengths = [len(line) for line in lines if line.strip()]
plt.figure(figsize=(10, 4))
plt.hist(line_lengths, bins=20)
plt.title('Розподіл довжини рядків')
plt.xlabel('Довжина')
plt.ylabel('Кількість рядків')
plt.grid(True)
plt.show()

# Аналіз зарезервованих слів
keywords = [
    'def', 'return', 'if', 'else', 'elif', 'while', 'for', 'in',
'try', 'except',
    'import', 'from', 'as', 'class', 'with', 'pass', 'break',
'continue', 'True', 'False'
]
keyword_counter = Counter()
for line in lines:
    for kw in keywords:
        keyword_counter[kw] += len(re.findall(rf'\b{kw}\b',
line))

# Візуалізація частоти ключових слів
plt.figure(figsize=(12, 5))
plt.bar(keyword_counter.keys(), keyword_counter.values())
plt.title("Частота використання зарезервованих слів")
plt.ylabel("Кількість")
plt.xticks(rotation=45)
plt.grid(True)
plt.show()

# Простий пошук помилок
errors = []
```

```
for i, line in enumerate(lines):
    if "==" in line and "if" not in line and "#" not in line:
        errors.append((i + 1, "Ймовірне порівняння поза
умовою"))
    if "except" in line and "as" not in line:
        errors.append((i + 1, "Недостатньо обробки винятку"))
    if re.match(r'^\s*print\s+[^("]', line):
        errors.append((i + 1, "print без дужок (сумісність з
Python 2?)"))

# Виведення знайдених помилок
if errors:
    print("Виявлено помилки:")
    for err in errors:
        print(f"Рядок {err[0]}: {err[1]}")
else:
    print("Помилок не виявлено.")
```