Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів статичного аналізу коду
(тема)

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-2

Рахматі М.А.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник доц. Лещинська І.О.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Рахматі Мохаммад Алі
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження методів статичного аналізу програмного коду»
Затверджена наказом по університету від 29.03.2024р. № 250 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 25.06.2024
3. Вихідні дані до роботи опис досліджуваних методів статичного аналізу коду, вимоги до розробки методів статичного аналізу коду за обраною предметною областю, теоретичні відомості про статичний аналіз даних, інструменти що реалізують методи статичного аналізу коду.
4. Перелік питань, що потрібно опрацювати в роботі
аналіз та порівняння методів статичного аналізу коду, вибір підходящих методів аналізу для дослідження, написання програмних рішень, проведення експериментів та аналіз отриманих результатів

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі та постановка задачі	23.01 – 14.02.24	*виконано*
2	Аналіз та вибір методів для дослідження	15.02 – 23.02.24	*виконано*
3	Аналіз та моделювання предметної області	23.02 – 28.02.24	*виконано*
4	Планування експериментів	28.02 – 29.02.24	*виконано*
5	Програмна реалізація кожного з обраних методів для дослідження	29.02 – 01.04.24	*виконано*
6	Експериментальні дослідження	02.04 – 22.04.24	*виконано*
7	Аналіз результатів експериментальних досліджень та розробка рекомендацій	23.04 – 27.04.24	*виконано*
8	Написання та оформлення статті та тез доповіді	27.04 – 30.04.24	*виконано*
9	Підготовка пояснювальної записки	14.04 – 12.06.24	*виконано*
10	Підготовка презентації та доповіді	07.06 – 12.06.24	*виконано*
11	Нормоконтроль	12.06– 17.06.24	*виконано*
12	Рецензування	17.06 – 19.06.24	*виконано*
13	Занесення диплома в електронний архів	19.06 – 22.06.24	*виконано*
14	Попередній захист	20.06 – 23.06.24	*виконано*
15	Допуск до захисту у зав. кафедри	23.06.24	*виконано*

Дата видачі завдання 29 лютого 2024р

Студент (ка) _____ Рахматі М.А.
(підпис)

Керівник роботи _____ доц. Лещинська І.О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Кваліфікаційна робота магістра містить: 54 сторінок, 18 рисунків, 1 таблиця, 9 джерел.

СИНТАКСИС КОДУ, МЕТОДИ АНАЛІЗУ, АРХІТЕКТУРА СПРИЙНЯТТЯ, НЕЙРОНА МЕРЕЖА, МЕТОД ПРОГНОЗУВАННЯ.

Об’єктом дослідження є процес статичного аналізу вихідного коду програм.

Метою роботи є покращення ефективності процесу статичного аналізу вихідного коду за критеріями: кількість оброблених вузлів за секунду та кількість отриманих звітів; шляхом розроблення нового методу статичного аналізу.

У роботі розглядаються методи аналізу статичного програмного коду, способи їх реалізації та кінцеві результати для порівняльної характеристики.

У результаті був проведений аналіз предметної галузі, поставлені завдання дослідження, аналіз існуючих методів та алгоритмів, планування та проведення дослідження та виявлено найбільш влучний метод для аналізу статичного програмного коду.

CODE SYNTAX, ANALYSIS METHODS, SPRING ARCHITECTURE, MERAGE NEURON, PREDICTION METHOD.

The object of research is the process of static analysis of the source code of programs.

The purpose of the work is to improve the efficiency of the process of static analysis of the source code according to the following criteria: the number of processed nodes per second and the number of received reports; by developing a new method of static analysis.

The work considers the methods of analysis of static software code, methods of their implementation and final results for comparative characteristics.

As a result, an analysis of the subject area was carried out, research tasks were set, analysis of existing methods and algorithms, planning and conducting of research, and the most appropriate method for analyzing static software code was identified.

Я, Рахматі Мохаммад Алі, студент гр. ІПЗм-22-2, здобувач вищої освіти на другому(магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів статичного аналізу програмного коду», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

Щорічний розвиток IT-технологій є невід'ємною частиною нашого інформаційного суспільства, що виявляється у зростанні кількості фахівців, що входять у сферу розробки програмного забезпечення, а також у збільшенні складності самого програмного коду. Сучасні розробники все частіше використовують сторонні бібліотеки та фреймворки, щоб полегшити свою роботу. Однак, незважаючи на ці переваги, неможливо гарантувати відсутність дефектів у використовуваному програмному забезпеченні, що робить систему вразливою.

Інструменти статичного, семантичного та динамічного аналізу є ефективними засобами для виявлення проблем у програмному коді, зокрема щодо якості. Динамічний аналіз ідеально підходить для виявлення помилок, пов'язаних із багатопотоковістю та керуванням пам'яттю, хоча й може бути витратним для великих проектів. У той час як статичний аналіз може ефективно використовуватися для виявлення помилок під час процедури перевірки коду, полегшуючи роботу рецензента, він також має свої обмеження, такі як недоліки у точності та можливість видачі помилкових повідомлень.

Однак існує постійне прагнення до удосконалення та оптимізації методів статичного аналізу для підвищення їхньої ефективності, зокрема шляхом прискорення часу виконання. Такі інструменти стають невід'ємними помічниками розробників, забезпечуючи не лише швидке виявлення помилок, але й загальне підвищення продуктивності та надійності розробки програмного забезпечення.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз предметної галузі

Статичний аналіз програмного коду визначається як ключовий елемент процесу розробки програмного забезпечення, спрямований на покращення його якості та забезпечення безпеки[1]. Основною метою статичного аналізу є виявлення помилок та вразливостей на ранніх етапах розробки, забезпечуючи швидке та ефективне виправлення.

Однією з головних переваг статичного аналізу є можливість виявлення помилок та вразливостей у програмному коді за лічені секунди, що дозволяє зберегти час розробників. Цей процес може бути інтегрований у процес розробки, відповідаючи застосунком Continuous Integration та Continuous Delivery (CI/CD)[2].

Використання статичного аналізу включає в себе виявлення вразливостей, недоліків у контролі потоку, а також проблем у використанні пам'яті. Важливо враховувати, що завдання статичного аналізу може бути алгоритмічно нерозв'язним, але в реальних програмах доводиться приймати компроміси, наприклад, обмежуючи умови завдання чи допускаючи неточності в результатах.

Інструментарій для статичного аналізу включає різні програми та бібліотеки, такі як ESLint, PyLint, FindBugs, SonarQube, Coverity та інші. Вони дозволяють автоматизувати процес аналізу, забезпечуючи високий стандарт коду та виявлення вразливостей.

Дослідження та порівняння різних методів статичного аналізу включає вивчення їх ефективності, точності та можливостей виявлення різних категорій помилок. Це дозволяє визначити оптимальні стратегії використання в конкретних умовах розробки.

Загальною висновок з аналізу предметної галузі є те, що статичний аналіз програмного коду є важливим інструментом для покращення його якості, безпеки та ефективності, сприяючи створенню надійних та безпечних програмних продуктів у сучасних умовах розробки.

1.2 Актуальність роботи

Сучасний ландшафт розробки програмного забезпечення характеризується великим обсягом коду, розподіленими командами розробників та постійним ростом складності проектів. У такому контексті актуальність дослідження методів статичного аналізу програмного коду стає неоспоримою.

Сучасні програмні проекти, включаючи ті, що використовують штучний інтелект, мають величезний обсяг коду. Статичний аналіз дозволяє виявляти потенційні помилки та вразливості на ранніх етапах розробки, сприяючи покращенню структури та якості програм.

Розподілена та різноманітна команда розробників стає стандартом. Статичний аналіз є необхідним інструментом для уніфікації стандартів коду та забезпечення єдиної методології в розробці.

У світлі постійно зростаючих загроз кібербезпеки, статичний аналіз дозволяє виявляти та усувати потенційні вразливості, забезпечуючи безпеку програмного забезпечення вже на етапі розробки.

У сфері штучного інтелекту, де точність та надійність є пріоритетами, статичний аналіз допомагає забезпечити стійкість та ефективність роботи алгоритмів, виявляючи потенційні проблеми ще на етапі розробки.

Таким чином, дослідження методів статичного аналізу коду виявляється актуальним та ключовим для забезпечення якості, безпеки та ефективності програмного забезпечення в умовах сучасного інформаційного суспільства.

1.3 Постановка задачі

Сучасне середовище розробки програмного забезпечення вимагає вдосконалення методів аналізу коду для забезпечення високої якості, безпеки та ефективності програмних продуктів. У цьому контексті проведення дослідження методів статичного аналізу програмного коду стає критично важливим завданням[3].

Провести глибокий аналіз різноманітних методів статичного аналізу, включаючи статичний аналіз коду, семантичний аналіз, аналіз взаємодії модулів та інші підходи. Зосередитися на їхніх теоретичних основах та практичних застосуваннях.

Визначити переваги та обмеження кожного методу статичного аналізу. Розглянути їхню ефективність в знаходженні помилок, швидкість виконання, витрати ресурсів та інші аспекти.

Провести порівняльний аналіз популярних інструментів статичного аналізу, оцінити їхні можливості та обмеження. Врахувати фактори, такі як підтримка мов програмування, гнучкість конфігурації та можливість інтеграції.

Розглянути, як застосування різних методів статичного аналізу впливає на якість та безпеку програмного коду. Визначити їхні можливості виявлення та усунення вразливостей та помилок.

Дослідити оптимальні підходи до інтеграції обраних методів статичного аналізу в розробчий процес. Врахувати взаємодію з іншими етапами розробки та вплив на продуктивність розробників.

Очікується, що результати дослідження нададуть вичерпні відомості про ефективність та застосування різних методів статичного аналізу, що відобразиться у порадах та рекомендаціях щодо їхнього використання для покращення якості та безпеки програмного коду в реальних умовах розробки.

2 АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ, РІШЕНЬ І АЛГОРИТМІВ

2.1 Опис існуючих алгоритмів

Порівняльний аналіз різних методів та інструментів статичного аналізу програмного коду включає в себе різні аспекти, такі як емпіричне тестування, аналіз впливу на продуктивність розробників, а також сам порівняльний аналіз. Давайте розглянемо кожен з них і їх взаємозв'язок.

Емпіричне тестування - це практика проведення конкретних експериментів або тестів для оцінки ефективності інструментів статичного аналізу[4]. Це може включати запуск тестових сценаріїв на реальних програмах з використанням різних інструментів, аналіз результатів та порівняння їх для визначення кращого підходу.

Аналіз впливу на продуктивність розробників - це оцінка того, як використання певного інструменту статичного аналізу впливає на продуктивність розробників. Це може включати оцінку часу, необхідного для виконання аналізу, а також складність у використанні інструменту. Аналіз цього впливу може допомогти визначити, які інструменти є найбільш придатними для реальних проектів та розробників.

Порівняльний аналіз інструментів - це процес порівняння різних інструментів статичного аналізу за різними критеріями, такими як точність виявлення помилок, швидкодія аналізу, обсяг пам'яті, необхідний для виконання, зручність у використанні та інші. Цей аналіз може включати як кількісні метрики, такі як час виконання або обсяг пам'яті, так і якісні характеристики, такі як зручність використання або рівень деталізації результатів.

Порівняльний аналіз інструментів може включати емпіричні тестування, щоб отримати конкретні дані про їхню ефективність, а також аналіз впливу на продуктивність розробників, оскільки різні інструменти можуть мати різний вплив на швидкодію та зручність використання. Таким чином, ці три аспекти взаємодоповнюються, допомагаючи зробити обґрунтований вибір між різними інструментами статичного аналізу програмного коду.

2.2 Опис існуючих аналогів

Перший аналог - SonarQube є популярним інструментом для статичного аналізу коду. Він підтримує багато мов програмування і надає зручний інтерфейс для відстеження якості коду. SonarQube виявляє потенційні помилки, вразливості та стилістичні недоліки, забезпечуючи широкий огляд кодової бази (див. рис. 2.1).

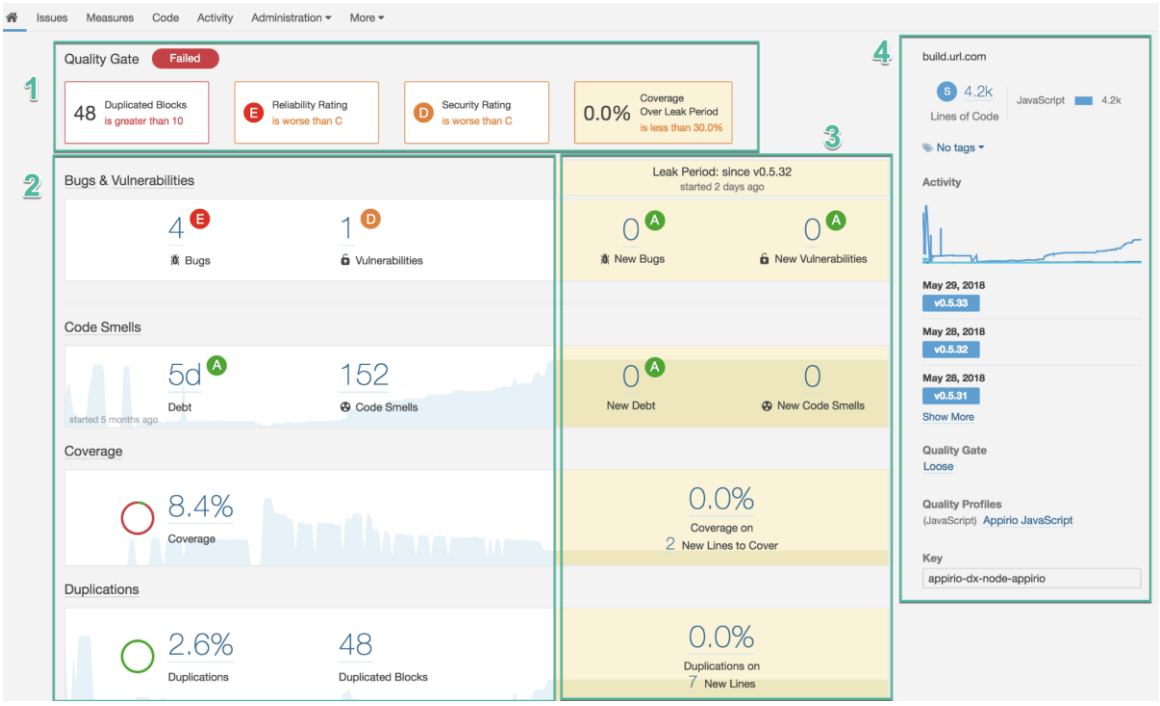

Рисунок 2.1 – Приклад роботи SonarQube

SonarQube використовує статичний аналіз коду для перевірки вихідного коду на наявність проблем. Це означає, що він аналізує код без його виконання. Інструмент збирає дані про різні аспекти коду, такі як складність, дублювання, покриття тестами, вразливості та інші. SonarQube має набір вбудованих правил для кожної підтримуваної мови програмування. Також можна створювати власні правила. Результати аналізу представлені у вигляді звітів і дашбордів, які показують загальний стан проекту, виявлені проблеми та пропозиції щодо їх вирішення.

Розглянемо приклад інтегрування SonarQube до проекту на Java та його роботу. Встановлюємо SonarQube на сервер, створюємо проект у веб-інтерфейсі та

додаємо плагін до build-інструмента (наприклад Maven чи Gradle). На рисунку 2.2 зображено приклад, як додавати плагін SonarQube на Maven.

```
<plugin>
  <groupId>org.sonarsource.scanner.maven</groupId>
  <artifactId>sonar-maven-plugin</artifactId>
  <version>3.7.0.1746</version>
</plugin>
```

Рисунок 2.2 – SonarQube на Maven

Виконуємо команду для запуску аналізу на Maven (див. рис. 2.3)

```
mvn sonar:sonar -Dsonar.projectKey=my_project_key -Dsonar.host.url=http://localhost:9000 -Dsonar.login=my_token
```

Рисунок 2.3 – Команда для запуску аналізу кода

Після завершення аналізу, результати будуть доступні у веб-інтерфейсі SonarQube. У інтерфейсі можемо побачити виявленні помилки, проблеми та пропозиції щодо покращення роботи коду.

Далі роздивимось PVS-Studio, як попередній аналог він також аналізує код, но вже має підтримку мови програмування C/C++. Але на відміну від попереднього аналога, його аналіз базується на патентах для виявлення труднощів у логіці програм та оптимізації. Тобто, PVS-Studio виконує аналіз коду без його виконання, він аналізує вихідний код на основі правил і алгоритмів. Інструмент має великий набір вбудованих правил для різних мов програмування, ці правила допомагають виявляти помилки, які можуть бути пропущені під час рецензування коду або його тестування. PVS-Studio легко інтегрується з популярними середовищами розробки, такими як Visual Studio, IntelliJ IDEA, Eclipse ті інші, що дозволяє розробникам швидко отримувати результати аналізу безпосередньо в їхньому робочому середовище. Інструмент генерує докладні звіти з виявленими помилками, які можуть бути збереженні у різних форматах (HTML, XML, текстові файли) для подальшого аналізу і спільної роботи в команді. PVS-Studio може бути інтегрований у процес CI/CD, що дозволяє автоматично виконувати аналіз коду під

час кожного коміту або збірки проекту. Приклад роботи з PVS-Studio у додатку Visual Studio зображений на рисунку 2.4.

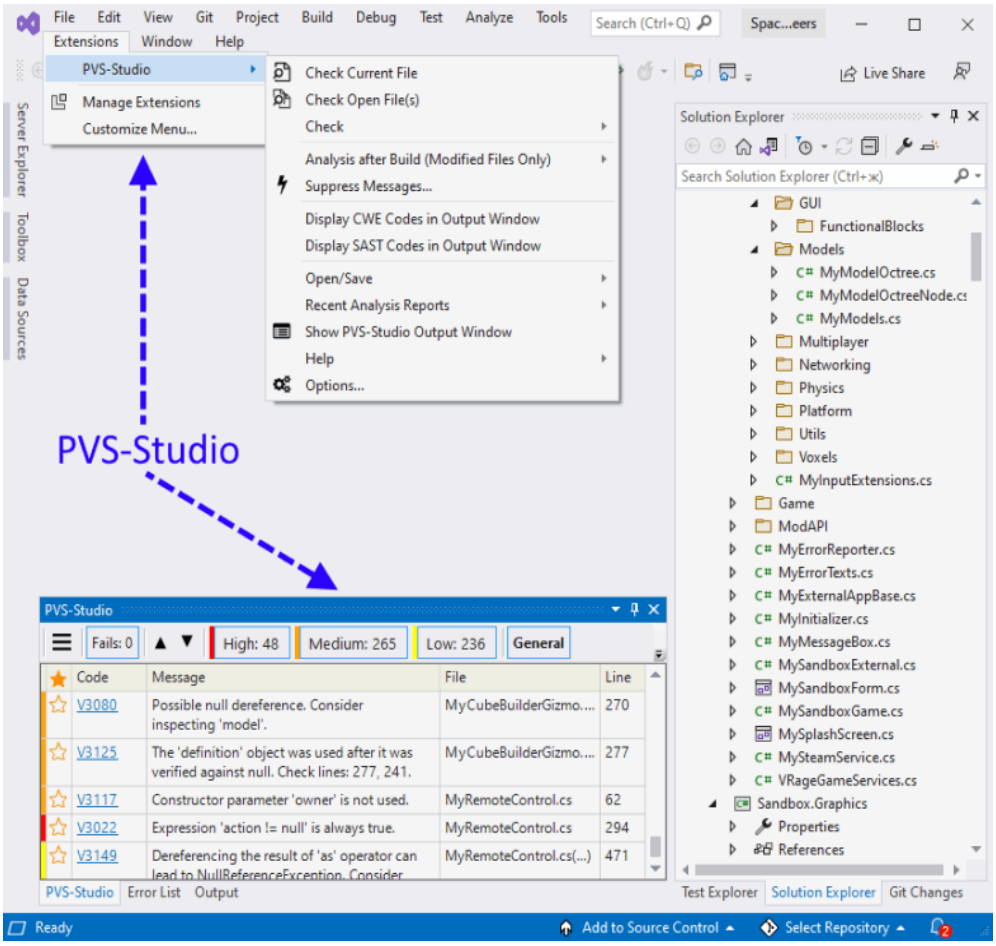

Рисунок 2.4 – Приклад роботи PVS-Studio

Для використання PVS-Studio у проекту C++, треба встановити PVS-Studio з офіційного сайту, інтегрувати його з IDE (в нашому випадку це Visual Studio), відкрити проект та виконати аналіз, вибравши в меню PVS-Studio і натиснувши «Check Current Project». Після завершення аналізу, результати з’являться у вікні «PVS-Studio Output». Можемо побачити список виявлених помилок з детальним описом і рекомендаціями щодо їх виправлення (див. рис. 2.5).

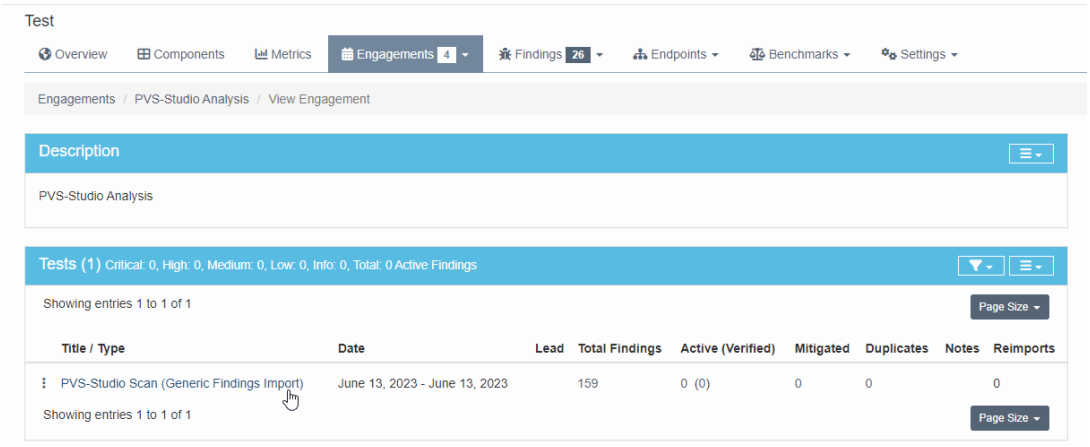

Рисунок 2.5 – Результати аналізу PVS-Studio на C++

Розглянемо приклад роботи цього інструмента на мові програмування Java, а саме на платформі IntelliJ IDEA. Треба так само встановити та додати PVS-Studio до нашого проекту, на цей раз приклад буде з Grandle (див. рис. 2.6) та виконуємо команду Grandle для запуску аналізу (див. рис. 2.7)

```
49 plugins {
50     id "com.viva64.pvs-studio" version "1.0.0"
51 }
52
53 pvsStudio {
54     files = fileTree(dir: 'src/main/java', include: '**/*.java'
55 }
```

Рисунок 2.6 – Встановлення PVS-Studio на проект Java

```
./gradlew pvsStudio
```

Рисунок 2.7 – Команда запуску аналізатора

Результати аналізу будуть доступні у вигляді звітів у зазначеній директорії.

2.3 Порівняння ESLint та TSLint між собою

ESLint та TSLint - це два інструменти, які використовуються для аналізу та підтримки якості програмного коду, зокрема у мовах програмування JavaScript та

TypeScript відповідно. Обидва інструменти мають свої схожі та відмінні характеристики, які слід врахувати при виборі між ними.

ESLint призначений для аналізу JavaScript-коду, тоді як TSLint розроблений спеціально для аналізу коду на TypeScript. Така спеціалізація робить TSLint більш ефективним і спрямованим на проекти, які використовують TypeScript, у порівнянні з ESLint, який може використовуватися для будь-яких JavaScript-проектів, навіть тих, де немає TypeScript.

ESLint відзначається дуже гнучкою системою налаштувань, що дозволяє розробникам вибирати та налаштовувати правила для аналізу коду, а також використовувати плагіни для розширення його функціональності. Такий підхід робить ESLint більш гнучким та привабливим для широкого спектру проектів. З іншого боку, TSLint, хоч і має можливості налаштування, може бути менш гнучким у порівнянні з ESLint, оскільки він спеціалізується на TypeScript. Тим не менше, для проектів, де використовується виключно TypeScript, TSLint може бути більш підходящим вибором, оскільки він зорієнтований саме на цю мову програмування.

Зараз, з урахуванням офіційної рекомендації від Microsoft про припинення підтримки TSLint на користь ESLint, останній став більш привабливим вибором для проектів на TypeScript. Ця рекомендація підкреслює перевагу ESLint з плагінами для TypeScript-проектів. Обидва інструменти можуть легко інтегруватися з різними інструментами розробки, такими як редактори коду, системи контролю версій та засоби CI/CD. Однак, з огляду на ширший розповсюдження та активний розвиток, ESLint, ймовірно, має більш широку підтримку та інтеграцію з іншими інструментами.

У підсумку, вибір між ESLint та TSLint залежить від конкретних потреб вашого проекту. ESLint може бути кращим вибором для проектів, які використовують як JavaScript, так і TypeScript, а також для тих, хто шукає більш гнучку систему налаштувань. Однак, якщо ваш проект використовує виключно TypeScript, ви можете розглянути використання TSLint.

3 ОГЛЯД ОСНОВНИХ МЕТОДІВ СТАТИЧНОГО АНАЛІЗУ КОДУ

3.1 Аналіз вихідного коду

Аналіз вихідного коду — це метод статичного аналізу, який використовується для оцінки програмного коду без його фактичного виконання. Цей підхід дозволяє виявити потенційні помилки, проблеми та аномалії в коді на ранніх етапах розробки, що сприяє підвищенню якості програмного забезпечення та зменшенню ризику виникнення помилок під час роботи програми. Далі буде розглянуто основні етапи аналізу вихідного коду та їх значення.

Аналізатори якості коду, які використовуються у статичному аналізі, являють собою інструменти, призначені для оцінки та визначення якості програмного коду без необхідності його фактичного виконання. Ці інструменти зосереджуються на дотриманні коду певним стандартам, виявленні потенційних проблем і наданні рекомендацій щодо покращення читабельності, ефективності та підтримуваності коду. Для цього аналізатори використовують визначені стандарти коду та метрики, що дозволяє оцінювати якість програмного забезпечення.

Основні функції аналізаторів включають перевірку правильності форматування коду, дотримання стилістичних вимог і відповідність визначеним критеріям якості. Ці інструменти допомагають виявляти «забруднення» або погані практики в програмному коді, які можуть призводити до серйозних проблем у майбутньому. Наприклад, аналізатори можуть знаходити дублікати коду, надмірно довгі методи, антипатерни та інші аномалії.

Деякі аналізатори також обчислюють цикломатичну складність коду, яка вказує на кількість незалежних шляхів у коді. Висока цикломатична складність може свідчити про необхідність рефакторингу для поліпшення читабельності та обслуговуваності програмного забезпечення.

Аналізатори якості коду також можуть виявляти потенційні проблеми та недоліки, такі як невикористані змінні, недоступний код та інші конструкції, які можуть викликати проблеми під час виконання програми. Це допомагає

розробникам знаходити й виправляти помилки ще на етапі написання коду, що підвищує загальну якість програмного забезпечення.

Аналізатори типів і анотації типів є важливими інструментами для перевірки відповідності типів даних у програмному коді. Вони допомагають виявляти помилки, пов’язані з неправильним використанням типів, що може призвести до потенційно небезпечних ситуацій або помилок під час виконання програми. Такі аналізатори використовують інформацію про типи, яку може надавати сам код або спеціальні анотації типів. Багато мов програмування дозволяють використовувати анотації типів для зазначення типів параметрів функцій, змінних та інших елементів коду.

Деякі аналізатори типів дозволяють визначати межі для типів даних. Наприклад, можна вказати, що певна змінна повинна бути підтипом конкретного типу. Це сприяє підвищенню безпеки та надійності програмного коду, оскільки неправильне використання типів може призвести до серйозних помилок під час виконання програми.

Окрім лінтерів, існує ще один вид інструментів для статичного аналізу коду – аналізатори вразливостей безпеки. Вони призначені для виявлення потенційних вразливостей, які можуть бути використані зловмисниками для атаки на систему. Їх основне завдання полягає у виявленні вразливостей, пов’язаних з недостатньою перевіркою введених даних, невірним керуванням доступом, буферними переповненнями та іншими небезпеками. Такі аналізатори можуть виявляти можливості для виконання SQL-ін’єкцій, введення шелл-команд, атак на основі XSS, CSRF та інших вразливостей.

Відомими інструментами для виявлення вразливостей безпеки є SonarQube, Checkmarx та Fortify. Деякі з них можуть перевіряти потік даних у застосунку, виявляючи шляхи атак та точки введення, які можуть бути використані хакерами. Наприклад, Fortify та Veracode мають такий функціонал. Аналізатори можуть застосовувати набір правил для перевірки дотримання кодом стандартів безпеки, включаючи правила для уникнення використання небезпечних функцій та

неналежного використання криптографії. Прикладами таких інструментів є ESLint для JavaScript та Bandit для Python.

Деякі інструменти можуть інтегруватися з базами даних відомих загроз і використовувати їх для виявлення вразливостей. Це дозволяє швидше знаходити вразливості, які вже були описані та класифіковані. Основні типи вразливостей безпеки, такі як SQL-ін’єкції, буферні переповнення та XSS-атаки, можуть бути виявлені та усунені за допомогою таких аналізаторів.

3.1.1 Лексичний аналіз

На першому етапі аналізу вихідного коду текст програми перетворюється на послідовність токенів (лексем). Токени — це мінімальні одиниці коду, такі як ключові слова, ідентифікатори, оператори, літерали та інші елементи мови програмування. Лексичний аналіз полягає в розбитті вихідного коду на ці компоненти, що дозволяє спростити подальший аналіз.

Лексичний аналізатор (лексер) проходить через текст програми і виділяє токени на основі правил мови програмування. Наприклад, у мові JavaScript ключовими словами є «if», «else», «for», «while» тощо. Лексер розпізнає ці ключові слова, а також ідентифікатори змінних, оператори (наприклад «+», «-«, «*», «/»), і визначає їх тип.

3.1.2 Синтаксичний аналіз

Після лексичного аналізу програма піддається синтаксичному аналізу, або парсингу. На цьому етапі перевіряється, чи відповідає код синтаксичним правилам мови програмування. Синтаксичний аналізатор (парсер) будує дерево синтаксичного розбору (abstract syntax tree, AST), яке представляє структуру програми. Парсер перевіряє, чи правильно використовуються конструкції мови програмування. Наприклад, він перевіряє, чи правильно відкриті і закриті дужки,

чи коректно побудовані вирази та чи відповідають вони синтаксичним правилам. Якщо виявляються синтаксичні помилки, програма не може бути скомпільована або виконана.

Приклад синтаксичної помилки: відсутня закриваюча дужка в умовному операторі «if» зображено на рисунку 3.1. Цей код не пройде синтаксичну перевірку через відсутність закриваючої дужки «)» після «a > b».

```
72 if (a > b {
73     console.log("a більше b");
74 }
75
```

Рисунок 3.1 – Приклад синтаксичної помилки

3.1.3 Семантичний аналіз

Після синтаксичного аналізу виконується семантичний аналіз, який перевіряє сенс і коректність використання конструкцій у коді. На цьому етапі визначається, чи правильно використовуються змінні, чи відповідають типи змінних та виразів очікуваним, чи правильно викликаються функції тощо. Семантичний аналізатор може виявити такі помилки, як використання змінних, які не були оголошені, або спроби виконати операції над змінними несумісних типів.

Приклад семантичної помилки: використання змінної перед її оголошенням зображено на рисунку 3.2. Цей код викличе помилку, оскільки змінна x використовується до її оголошення.

```
84 console.log(x);
85 let x = 10;
```

Рисунок 3.2 – Приклад семантичної помилки

3.1.4 Аналіз потоку даних

Аналіз потоку даних визначає, як дані передаються через програму і як вони використовуються. Цей аналіз допомагає виявляти можливі витоки пам'яті, невизначені значення змінних, використання змінних без ініціалізації та інші проблеми, пов’язані з обробкою даних. Аналіз потоку даних може виявити такі помилки, як використання змінних, які не були ініціалізовані, або неправильне управління пам’яттю (наприклад, в мовах C/C++).

Приклад помилки потоку даних: використання змінної без ініціалізації зображено на рисунку 3.3. У цьому коді змінна «x» використовується без попередньої ініціалізації, що може призвести до непередбачуваних результатів.

```
135 int main() {
136     int x;
137     printf("%d", x);
138     return 0;
139 }
```

Рисунок 3.3 – Приклад помилки ініціалізації

3.2 Автоматичні засоби статичного аналізу

Автоматизовані засоби статичного аналізу, також відомі як статичні аналізатори, — це програми та інструменти, призначені для проведення аналізу вихідного коду програми без її фактичного виконання. Вони дозволяють розробникам виявляти та виправляти помилки, вразливості та інші проблеми на ранніх етапах розробки, забезпечуючи високу якість та надійність програмного забезпечення.

Про деякі вже було сказано у попередньому розділі, тому розглянемо інші, такі як аналіз потоку даних. Він аналізує передачу і використання даних у програмі, виявляючи можливі проблеми, такі як витоки пам’яті та невизначені значення змінних. Аналіз контролю потоку визначає шлях виконання програми,

допомагаючи виявляти недосяжний код, циклічні залежності та інші помилки. Існує ще аналіз безпеки, він перевіряє код на вразливостей таких як SQL-ін’єкції, міжсайтові сценарії (XSS) та інші проблеми безпеки.

3.3 Типи помилок, які можуть виявляти інструменти статичного аналізу

Розглянемо найчастіше помилки, які можна виявити за допомогою інструментів статичного аналізу коду:

- Дублювання коду – це виявлення повторюваних фрагментів коду, що може призвести до зниження якості та підтримуваності коду;
- Недосяжний код – це визначення ділянок коду, які ніколи не виконуються, що може свідчити про логічні помилки;
- Невизначені змінні – це виявлення змінних, які використовуються до їх ініціалізації, що може призвести до непередбачуваної поведінки програми;
- Вразливості безпеки – це виявлення потенційних вразливостей, таких як SQL-ін'єкції, XSS, CSRF та інші проблеми, що можуть бути використані зловмисниками;
- Проблеми з витоками пам’яті – це виявлення місць у коді, де пам'ять виділяється, але не звільняється належним чином, що може призвести до витоків пам'яті;
- Неправильне використання API – це виявлення випадків, коли API використовуються неправильно або недоцільно, що може призвести до помилок виконання;
- Недотримання стандартів коду – це перевірка коду на відповідність встановленим стандартам та рекомендаціям щодо стилю програмування.

3.4 Технології для аналізу безпеки

Існує кілька технологій для аналізу безпеки, серед яких найбільш уживаними є:

1. SAST (Static Application Security Testing). Статичне тестування безпеки додатків, розроблене для знаходження вразливостей безпеки в вихідному коді на ранніх етапах розробки. SAST забезпечує відповідність коду стандартам безпеки без його виконання.

2. DAST (Dynamic Application Security Testing). Динамічне тестування безпеки додатків, що виявляє потенційні вразливі місця у коді під час його виконання, зазвичай у вебзастосунках. DAST виявляє поширені вразливості, такі як SQL-ін'єкції та XSS, за допомогою впровадження помилок.

3. IAST (Interactive Application Security Testing). Інтерактивне тестування безпеки додатків, яке поєднує можливості SAST і DAST, проводячи аналіз у реальному часі на будь-якому етапі процесу розробки. IAST аналізує код, потоки даних, конфігурації, HTTP-запити та відповіді, бібліотеки, фреймворки та інші компоненти.

4. RASP (Run-time Application Security Protection). Захист додатків у реальному часі, який не є засобом тестування, а забезпечує захист системи, відображаючи атаки та повідомляючи про них захисників. RASP превентивно завершує сесії зловмисників.

Окремо треба виділити лінтери. Лінтери — це інструменти статичного аналізу, які перевіряють код на відповідність певним специфікаціям та стандартам. Вони допомагають забезпечити дотримання стилістичних правил та виявляють потенційні проблеми у коді.

4 ПЛАНУВАННЯ ЕКСПЕРИМЕНТАЛЬНОЇ ЧАСТИНИ ДОСЛІДЖЕННЯ

Методи та алгоритми дозволяють проводити аналіз програмних систем з максимальною кількістю рядків коду в автоматизованому режимі. Дані методи та алгоритми використано для створення універсального аналізатора кодів програм. Перевіряючі модулі з підтримкою запропонованого комбінованого програмного методу статичного аналізу, мають високу та достатню для практичного застосування точність та швидкодію аналізу.

Методи аналізу програм, застосовні для проектів масштабу операційних систем та їх наборів додатків, реалізовані в практично використовуваному аналізаторі програмного коду.

4.1 Запропонований комбінований метод статичного аналізу

Запропонований метод базується на підході Data-flow аналізу. Цей метод володіє рядом особливостей, зокрема, він не проводить перевірку коду на існуючі вразливості, на відміну від Taint аналізу. Припускаючи, що об'єднавши обидва ці формальні методи статичного аналізу, можна отримати комбінований метод з покращеною ефективністю через деталізацію результатів аналізатора вихідного коду.

Зазначені методи мають спільні етапи роботи, такі як читання вхідного коду, розбиття на лексеми та побудова графу зв'язку між вузлами. Щоб уникнути повторюваних операцій, ці етапи можна винести в спільний модуль, яким можуть користуватися обидва методи.

Ще однією недолікою Data-flow аналізу є використання тільки типів, визначених у коді програми. Для виправлення цього, використовується гіпотеза використання методу виводу типів Хіндлі-Мілнера[5]. Цей метод дозволяє уточнити деякі поліморфні класи та методи, зменшуючи кількість ітерацій алгоритму основного методу.

Алгоритм виводу типів може використовуватися паралельно з методами Data-flow та Taint аналізу, проте не передбачається його застосування з мовами програмування C та C++. Це може вивести більш загальний тип даних, ніж вказано у вихідному коді, але цей дефект можна легко виправити, додавши етап перевірки отриманих значень до аналізу кожного класу[6].

Отже, розроблено комбінований метод статичного аналізу вихідного коду користувача, заснований на методі Data-flow аналізу. Теоретично цей метод повинен виявити покращену ефективність, що вимірюється кількістю проаналізованих вузлів та кількістю звітів, отриманих в результаті його виконання[7]. Ключові етапи цього методу включають:

1) Розбиття коду користувача на лексеми.

2) Побудова графу потоку управління (CFG).

3) Застосування алгоритму Хіндлі-Мілнера.

4) Виконання аналізу Data-flow методу.

5) Виконання аналізу методу перевірки на чистоту (Taint).

Схема роботи цього методу зображена на рисунку 4.1. Основна ідея модифікації методу полягає в об'єднанні двох методів та введенні додаткових прапорців з типізацією для змінних в інструкціях тексту програми. Змінна може мати різні типи, такі як примітив, об'єкт з наслідуванням. У всіх випадках змінні розглядаються як об'єкти зі станом, де примітивні типи мають лише одне поле - значення цієї змінної.

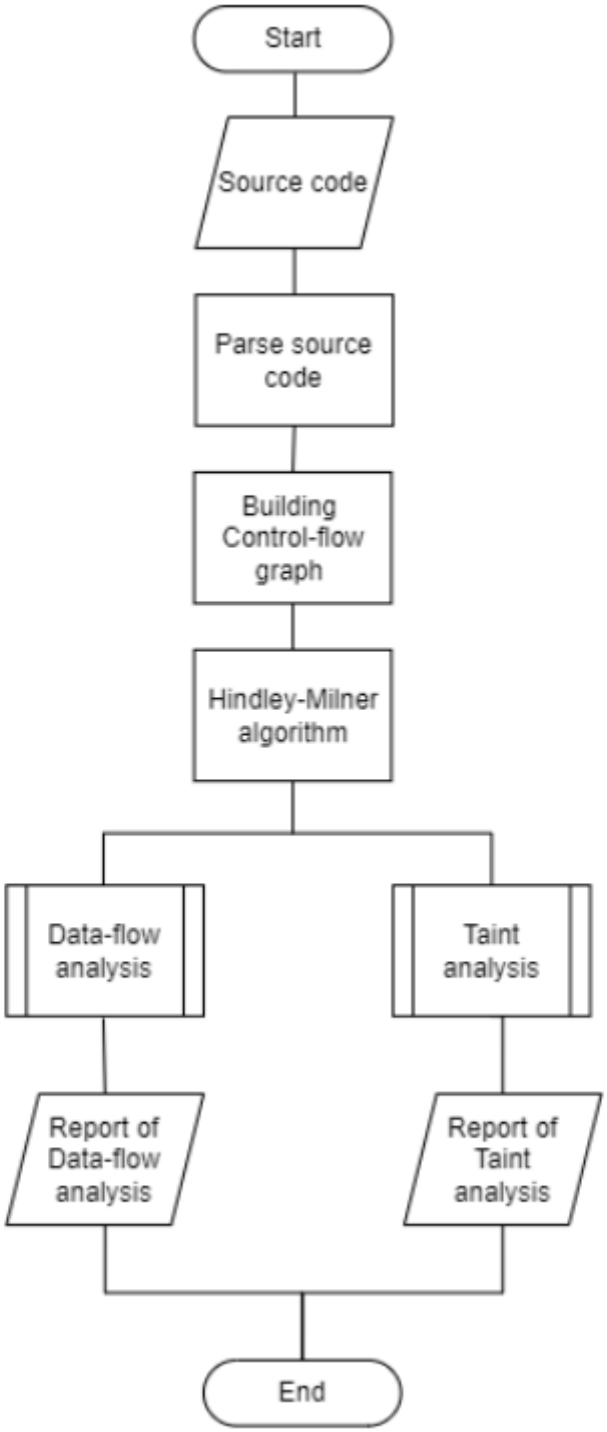

Рисунок 4.1 – Алгоритм реалізації комбінованого методу статичного аналізу

4.2 Опис набору даних

У цій частині дослідження розглянемо використання різних інструментів статичного аналізу коду для одного і того ж набору даних. Для цього використовуватимемо наступні інструменти:

1. ESLint для JavaScript
2. Муру для Python
3. PVS-Studio для C++

Створимо приклади коду для кожного інструменту, запустимо аналіз і порівняємо результати.

Для консистентності, використовуватимемо аналогічні приклади коду для кожної мови програмування. Нижче наведено вихідний код з помилками для кожної мови.

5 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТАЛЬНОЇ ЧАСТИНИ ДОСЛІДЖЕННЯ

5.1 Аналіз та підготовка набору даних

Напишемо код, який будемо використовувати для роботи зі всіма інструментами, щоб побачити якість роботи кожного інструмента в однаковому середовищі. Так як код буде відрізнятися на мовах програмування, буде наведено приклад на мові Java (див. рис. 5.1).

```
// example.js
function addNumbers(a, b) {
    return a + b;
}

function getEvenNumbers(numbers) {
    return numbers.filter(num => num % 2 == 0);
}

let x = 10;
let y = "20";
console.log(addNumbers(x, y));

let numbers = [1, 2, 3, 4, 5, 6];
let evenNumbers = getEvenNumbers(numbers);
console.log("Even numbers: " + evenNumbers);
```

Рисунок 5.1 – Приклад коду для перевірки інструментів

Розпочнемо огляд з ESLint для JavaScript (див. рис. 5.2). Результати аналізу програмного коду зображено на рисунку 5.3.

```
// example.js
function addNumbers(a, b) {
    return a + b;
}

function getEvenNumbers(numbers) {
    return numbers.filter(num => num % 2 == 0);
}

let x = 10;
let y = "20";
console.log(addNumbers(x, y));

let numbers = [1, 2, 3, 4, 5, 6];
let evenNumbers = getEvenNumbers(numbers);
console.log("Even numbers: " + evenNumbers);
```

Рисунок 5.2 – Приклад коду для перевірки ESLint для JavaScript

```
example.js
  9:17  error  '20' is assigned a value but never used  no-unused-vars
  9:23  error  Strings must use singlequote             quotes
  10:28 error  Expected '===' and instead saw '=='      eqeqeq
  14:19 error  Missing semicolon                        semi

✖ 4 problems (4 errors, 0 warnings)
```

Рисунок 5.3 – Результати перевірки ESLint для JavaScript

Наступним розглянемо аналіз коду за допомогою інструменту Python (Mypy) (див. рис. 5.4). Результати перевірки програмного коду можна побачити на зображені 5.5.

```python
from typing import List

def add_numbers(a: int, b: int) -> int:
    return a + b

def get_even_numbers(numbers: List[int]) -> List[int]:
    return [num for num in numbers if num % 2 == 0]

def main() -> None:
    x: int = 10
    y: str = "20"  сла
    result: int = add_numbers(x, y)
    print(f"The sum of {x} and {y} is {result}")

    numbers: List[int] = [1, 2, 3, 4, 5, 6]
    even_numbers: List[int] = get_even_numbers(numbers)
    print(f"Even numbers: {even_numbers}")

if __name__ == "__main__":
    main()
```

Рисунок 5.4 – Приклад коду для перевірки Python (Mypy)

```
example.py:10: error: Argument 2 to "add_numbers" has incompatible type "str"; expected "int"
example.py:17: error: Incompatible types in assignment (expression has type "List[int]", variable has
    type "List[str]")
Found 2 errors in 1 file (checked 1 source file)
```

Рисунок 5.5 – Результат аналізу коду для перевірки Python (Mypy)

Останнім розглянемо аналіз коду за допомогою інструменту C++ (PVS-Studio) (див. рис. 5.6). Результати перевірки програмного коду можна побачити на зображені 5.7.

```cpp
#include <iostream>
#include <vector>

int addNumbers(int a, int b) {
    return a + b;
}

std::vector<int> getEvenNumbers(const std::vector<int>& numbers) {
    std::vector<int> evens;
    for (int num : numbers) {
        if (num % 2 == 0) {
            evens.push_back(num);
        }
    }
    return evens;
}

int main() {
    int x = 10;
    std::string y = "20";  // Помилка: використання рядка замість числа
    int result = addNumbers(x, std::stoi(y));  // Потенційна помилка
    std::cout << "The sum of " << x << " and " << y << " is " << result << std::endl;

    std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
    std::vector<int> evenNumbers = getEvenNumbers(numbers);
    std::cout << "Even numbers: ";
    for (int num : evenNumbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
```

Рисунок 5.6 – Приклад коду для перевірки C++ (PVS-Studio)

```
example.cpp:14: warning: V522: Return of 'stoi(y)' could result in undefined behavior if the string does not represent a valid integer.
example.cpp:18: warning: V519: The 'y' variable is assigned values twice successively. Perhaps this is a mistake. Check lines: 18, 18.
```

Рисунок 5.7 – Результат аналізу коду для перевірки C++ (PVS-Studio)

5.2 Порівняння практичних результатів

Результати аналізу показують, що різні інструменти статичного аналізу коду можуть виявляти різні типи помилок та проблем у коді. Зробимо таблицю порівняння (див. табл. 5.1) ESLint фокусується на стилістичних помилках у JavaScript-коді, Муру допомагає виявляти помилки типізації у Python-коді, PVS-Studio знаходить потенційні помилки у C++-коді, а SonarQube забезпечує комплексний аналіз для багатьох мов програмування. Використання цих

інструментів допомагає забезпечити високу якість коду та зменшити кількість помилок на ранніх етапах розробки.

Таблиця 5.1 – Порівняння інструментів (виповнена самостійно)

Інструмент	Виявленні помилки
ESLint	- Використання рядка замість числа
	- Використання подвійних лапок замість одинарних
	- Використання "==" замість "==="
	- Відсутня крапка з комою
Муру	- Неправильний тип аргументу для функції
	- Несумісні типи при присвоєнні значення
PVS-Studio	- Потенційна помилка при конвертації рядка в число
	- Повторне присвоєння змінної

ВИСНОВКИ

Протягом наукового дослідження були розглянуті різні аспекти та інструменти статичного аналізу коду, зокрема такі як ESLint, TSLint, SonarQube, PVS-Studio та Муру. Було детально розглянуто методи статичного аналізу, їхнє значення для забезпечення якості програмного забезпечення, а також практичні приклади використання цих інструментів на конкретних прикладах коду.

Статичний аналіз коду включає кілька основних методів, таких як лексичний аналіз, синтаксичний аналіз, семантичний аналіз, аналіз потоку даних, аналіз контролю потоку та аналіз безпеки. Кожен із цих методів виконує свою специфічну функцію і допомагає виявляти різні типи помилок та вразливостей у коді. Наприклад, лексичний аналіз розбиває код на токени, синтаксичний аналіз перевіряє відповідність коду правилам синтаксису, а аналіз потоку даних визначає передачу і використання даних у програмі. Аналіз безпеки виявляє вразливості, що можуть бути використані зловмисниками для атаки на програмне забезпечення. Всі ці методи в сукупності дозволяють забезпечити високий рівень якості та безпеки програмного коду.

Одним із найпопулярніших інструментів для статичного аналізу коду JavaScript є ESLint. ESLint допомагає виявляти синтаксичні помилки, порушення стилю та потенційні проблеми у коді. Наприклад, при аналізі коду з ESLint було виявлено помилки типу, неправильне використання оператора порівняння та інші стилістичні проблеми. Це свідчить про те, що ESLint є потужним інструментом для підтримки чистоти та якості коду, що допомагає розробникам уникати поширених помилок та дотримуватися найкращих практик програмування.

TSLint, інший інструмент для статичного аналізу, призначений для коду TypeScript, дозволяє виявляти синтаксичні та стилістичні помилки. Хоча TSLint офіційно припинив підтримку, він все ще використовується в багатьох проектах. Для тих, хто хоче перейти на сучасніший інструмент, рекомендується використовувати ESLint з плагіном для TypeScript, який забезпечує аналогічні функції з кращою підтримкою.

SonarQube є комплексною платформою для статичного аналізу, що підтримує багато мов програмування та надає розширені звіти про якість коду. SonarQube забезпечує глибокий аналіз, виявляє вразливості, дублювання коду, проблеми з безпекою та інші аспекти якості. Інструмент легко інтегрується з процесами CI/CD, що дозволяє автоматизувати аналіз коду на кожному етапі розробки. Приклад використання SonarQube показав, що він здатен надавати детальні звіти та рекомендації щодо виправлення виявлених проблем, що значно підвищує ефективність розробки та підтримки програмного забезпечення.

PVS-Studio є інструментом для статичного аналізу коду мов C, C++, C# та Java. Він допомагає виявляти помилки, вразливості та потенційні проблеми у вихідному коді. Приклад використання PVS-Studio показав, що він може виявляти різні типи помилок, включаючи помилки конвертації типів, дублювання змінних та інші логічні помилки. Це свідчить про те, що PVS-Studio є потужним інструментом для забезпечення високої якості та надійності коду.

Муру є інструментом для статичного аналізу типів у Python, який допомагає виявляти помилки типізації на ранніх етапах розробки. Використання Муру дозволяє підвищити якість коду, забезпечуючи правильність типів та зменшуючи кількість потенційних помилок. Наприклад, Муру допомагає виявити невідповідність типів аргументів функцій та змінних, що дозволяє розробникам уникати багатьох типових помилок, які можуть виникати під час виконання програми.

Варто також зазначити важливість використання технологій для аналізу безпеки, таких як SAST, DAST, IAST та RASP. SAST (Static Application Security Testing) дозволяє знаходити вразливості безпеки в вихідному коді на початкових етапах розробки, забезпечуючи відповідність коду стандартам безпеки без його виконання. DAST (Dynamic Application Security Testing) виявляє потенційно вразливі місця у коді під час його виконання, що дозволяє знаходити поширені вразливості безпеки, такі як SQL-ін'єкції та міжсайтові сценарії. IAST (Interactive Application Security Testing) поєднує можливості SAST і DAST, проводячи аналіз у реальному часі на будь-якому етапі процесу розробки, що забезпечує точніші

результати. RASP (Run-time Application Security Protection) забезпечує захист системи в реальному часі, відбиваючи атаки та повідомляючи про них захисників.

Загалом, використання автоматизованих засобів статичного аналізу коду значно підвищує якість та надійність програмного забезпечення. Вони дозволяють виявляти помилки та вразливості на ранніх етапах розробки, що допомагає зменшити витрати на виправлення помилок та забезпечити безпеку програмного забезпечення. Інтеграція інструментів статичного аналізу в процес розробки дозволяє автоматизувати перевірку коду, забезпечуючи постійний контроль якості на кожному етапі розробки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Chaves, K., Hafiz, M., & Khomh, F. (2018). A comprehensive study on the impact of static code attributes on the popularity of JavaScript GitHub repositories. Empirical Software Engineering, 23(4), 2043–2095. (дата звернення: 01.01.2024).

2. Păsăreanu, C. S., & Csallner, C. (2011). Combining Unit-level Symbolic Execution and Search-based Testing for Java Path Coverage. ACM SIGSOFT Software Engineering Notes, 36(2), 1–5. (дата звернення: 01.01.2024).

3. CodeSonar C/C++ [Електронний ресурс] — Режим доступу до ресурсу: https://www.grammatech.com/codesonar-cc, (дата звернення: 01.01.2024).

4. GDACS URL: https://www.gdacs.org (дата звернення: 01.01.2024).

5. Zhang Wei, Ma Zhen, Lu Qing, Wang Xiao, Liu Da. (2014). A Method of Software Maintainability Evaluation Based on Static Analysis. Applied Mechanics and Materials. [Електронний ресурс] — Режим доступу до ресурсу: www.scientific.net/AMM.651-653.1757, (дата звернення: 01.01.2024).

7. Sheridan Flash. (2022). Static Analysis Deployment Pitfalls (дата звернення: 01.01.2024).

8. VO Leshchynskyi, IO Leshchynska - Системи управління, навігації та зв'язку. Збірник …, 2017

9. Maksym Bekuzarov, Oleksandr Samantsov, Oksana Mazurova,Mariia Shirokopetleva. Neural Network Architecture Editor With Code Generation. Problem of Infocommunications. Science and Technology (PIC S&T'2020), Kharkiv, Ukraine.- 6- 9 October 2020.

ПЕРЕЛІК ДЖЕРЕЛЬ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

8. VO Leshchynskyi, IO Leshchynska - Системи управління, навігації та зв'язку. Збірник …, 2017

9. Maksym Bekuzarov, Oleksandr Samantsov, Oksana Mazurova,Mariia Shirokopetleva. Neural Network Architecture Editor With Code Generation. Problem of Infocommunications. Science and Technology (PIC S&T′2020), Kharkiv, Ukraine.- 6- 9 October 2020.