Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

ДОСЛІДЖЕННЯ МЕТОДІВ ПЕРЕВІРКИ ВРАЗЛИВОСТІ ПРОГРАМНОГО КОДУ, ЯКІ ҐРУНТУЮТЬСЯ НА СТАТИЧНОМУ АНАЛІЗІ ПРОГРАМ
(тема)

Виконав:
студент 2 курсу, групи ІНФм-22-2
Теребецький М.А.
(прізвище, ініціали)

Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник проф. Кузьомін О. Я.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Кобилін О.А.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Теребецькому Микиті Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методів перевірки вразливості програмного коду, які грунтуються на статичному аналізі програм.

затверджена наказом по університету від 3 листопада 2023 року № 1280Ст

2. Термін подання студентом роботи до екзаменаційної комісії 19 грудня 2023 р.

3. Вихідні дані до роботи методи статичного аналізу даних, інустрменти що реалізують ці методи, перелік використовуваних програмних засобів: теоретичні відомості про статичний аналіх даних.

4. Перелік питань, що потрібно опрацювати в роботі
1. Огляд методів статичного аналізу даних.
2. Огляд інструментів, що реалізують ці методи.
3. Порівняння обраних інструментів.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Приклади та схеми роботи методів та іструментів статичного аналізу, тестові зображення.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	03.11.2023	
2	Аналіз завдання, підбір літератури	03.11.23-08.11.23	
3	Аналіз літератури з досліджуваної проблеми	08.11.23-20.11.23	
4	Аналіз технічних засобів	20.11.23-23.11.23	
5	Аналіз методів	25.11.23-01.12.23	
6	Програмна реалізація	01.12.23-05.12.23	
7	Оформлення пояснювальної записки	05.12.23-10.12.23	
8	Перевірка на плагіат	15.12.2023	
9	Рецензування	20.12.2023	
10	Підготовка презентації та доповіді	28.12.2023	
11	Занесення роботи в електронний архів	03.01.2024	
12	Попередній захист кваліфікаційної роботи	04.01.2024	

Дата видачі завдання 3 листопада 2023 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Кузьомін О.Я.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 55 с., 4 табл., 19 рис., 40 джерел.

СТАТИЧНИЙ АНАЛІЗ, ПОТІК ДАНИХ, ПОТІК КЕРУВАННЯ, СИСТЕМИ ТИПІВ, ЛІНТЕРИ.

Об’єктом дослідження є набір документів із програмним кодом.

Метою дослідження є аналіз та оцінка методів виявлення вразливостей, що базуються на статичному аналізі програм.

Розглянуто та систематизовано існуючі методи статичного аналізу програм для виявлення потенційних вразливостей в програмному коді. Оцінено ефективність в реальних програмах на прикладах з відкритих та комерційних проєктів. Порівняно їх точність, швидкодію та придатність для різних видів програмних проєктів та мов програмування. Розроблено рекомендації щодо використання найбільш ефективних методів статичного аналізу для забезпечення вищої якості програмного коду та зменшення ризику вразливостей в програмах.

У результаті дослідження здійснена програмна реалізація системи для комплексного аналізу коду.

STATIC ANALYSYS, DATA FLOW, CONTROL FLOW, TYPE SYSTEMS, LINTERS.

The object of the research is a set of documents with program code.

The aim of the research is to analysis and evaluate vulnerability detection methods based on static analysis of programs.

The existing methods of static analysis of programs for identifying potential vulnerabilities in the program code are considered and systematized. Their effectiveness in real programs is evaluated using examples from open source and commercial projects. Their accuracy, speed and suitability for different types of software projects and programming languages are compared. Recommendations on the use of the most effective static analysis methods to ensure higher quality of program code and reduce the risk of vulnerabilities in programs are developed.

As a result of implemented software implementation of the system for complex code analysis.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів 7
Вступ 8
1 Огляд основних методів статичного аналізу коду 9
1.1 Аналіз вихідного коду 9
1.1.1 Лексичний аналіз: 9
1.1.2 Синтаксичний аналіз: 9
1.1.3 Семантичний аналіз: 10
1.1.4 Аналіз потоку даних: 10
1.1.5 Аналіз контролю потоку: 10
1.1.6 Аналіз безпеки: 10
1.1.7 Аналіз продуктивності: 11
1.2 Автоматизовані засоби статичного аналізу 11
1.3 Аналіз потоку даних 13
1.4 Аналіз потоку керування 17
1.5 Абстрактна інтерпретація 19
1.6 Символьне виконання 20
1.7 Системи типів 22
1.8 Пошук за шаблонами 23
1.9 Аналіз метрик та складності коду 24
1.10 Постановка задачі дослідження 26
2 Інструменти статичного аналізу коду 27
2.1 Лінтери 27
2.2 Аналізатори вразливості безпеки 28
2.3 Аналізатори якості коду 30
2.4 Аналізатори типів та анотації типів (Type Checkers) 30
2.5 Аналізатори викликів функцій та залежностей 31

2.6 Аналізатори використання ресурсів та оптимізації 33
2.7 Підсумки 33
3 Порівняння інструментів статичного аналізу 35
3.1 Pylint 35
3.2 Mypy 37
3.3 Pyflakes 38
3.4 Pycodestyle 40
3.5 Flake8 41
3.6 Prospector 42
3.7 Bandit 44
3.8 Порівняння інструментів 46
Висновки 50
Перелік джерел посилання 51

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

IDE – Integrated Development Environment (інтегроване середовище розробки)

ПЗ – програмне забезпечення

PEP8 – сучасний стандарт написання коду на Python

ВСТУП

Розвиток та повсюдне використання інформаційних технологій спричиняють необхідність росту стандартів безпеки, оскільки разом з ними розвиваються способи та методи кібератак.

Потужний захист є головним гарантом життєздатності програмного забезпечення. Виявлення та виправлення вразливостей під час розробки є однією з ключових складових забезпечення конфіденційності, цілісності та доступності даних.

Сьогодні існує безліч методів та інструментів для захисту програмного забезпечення, одним із найпотужніших та ефективних є статичний аналіз [1-8]. Цей метод дозволяє аналізувати код без його виконання та визначає потенційні вразливості, викликані недотриманням норм безпеки або недоліками у коді, допущеними під час розробки, які можуть бути використані зловмисниками для кібератак.

Дослідження методів статичного аналізу програмного коду для виявлення вразливостей стає дедалі актуальнішим у зв’язку з постійним розвитком кіберзагроз.

Отже, слід дослідити різні способи та підходи до статичного аналізу з метою покращити якість безпеки програмного забезпечення та сприяти створенню надійніших та безпечних програмних продуктів.

1 ОГЛЯД ОСНОВНИХ МЕТОДІВ СТАТИЧНОГО АНАЛІЗУ КОДУ

1.1 Аналіз вихідного коду

Аналіз вихідного коду є методом статичного аналізу, що використовується для оцінки програмного коду без його фактичного виконання. Цей метод дозволяє виявити потенційні помилки, проблеми та аномалії в коді до того, як програма буде запущена або відладжена. Аналіз вихідного коду важливий для забезпечення якості програмного забезпечення та зменшення ризику виникнення помилок під час роботи програми [9-14].

1.1.1 Лексичний аналіз

На першому етапі аналізу вихідного коду програми її текст перетворюється на послідовність токенів (лексем). Токени представляють собою ключові слова, ідентифікатори, оператори та інші елементи мови програмування. Лексичний аналіз допомагає розбити вихідний код на окремі компоненти для подальшого аналізу.

1.1.2 Синтаксичний аналіз

На цьому етапі виконується перевірка синтаксису програми, тобто визначається, чи відповідає код синтаксичним правилам мови програмування. Якщо виявляються синтаксичні помилки, програма не може бути компільована або виконана.

1.1.3 Семантичний аналіз

Семантичний аналіз оцінює значення та коректність використання змінних, функцій, класів і інших структур даних в програмі. Цей аналіз допомагає виявляти семантичні помилки, такі як використання змінних без ініціалізації чи неправильні типи даних.

1.1.4 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 та інші проблеми, пов’язані з обробкою даних.

1.1.5 Аналіз контролю потоку

Аналіз контролю потоку досліджує шляхи виконання програми та визначає можливі проблеми з управлінням потоком, такі як недосяжний код, безкінечні цикли та інші аномалії.

1.1.6 Аналіз безпеки

Особливий вид аналізу вихідного коду, спрямований на виявлення потенційних вразливостей безпеки, таких як SQL-ін’єкція, переповнення буфера, витоки інформації та інші атаки.

1.1.7 Аналіз продуктивності

Під час аналізу вихідного коду можна виявити ділянки коду, які можуть призвести до погіршення продуктивності програми, і рекомендувати оптимізації. Аналіз вихідного коду може проводитися як вручну програмістами, так і за допомогою спеціалізованих інструментів аналізу коду, таких як статичні аналізатори або інтегровані середовища розробки (IDE) з підтримкою аналізу. В результаті аналізу вихідного коду можуть бути знайдені та виправлені помилки, що допомагає підвищити якість та безпеку програмного забезпечення.

1.2 Автоматизовані засоби статичного аналізу

Автоматизовані засоби статичного аналізу, також відомі як статичні аналізатори, є програмами і інструментами, призначеними для проведення аналізу вихідного коду програми без її виконання.

Інструменти статичного аналізу можуть використовувати різні методи аналізу, включаючи лексичний, синтаксичний, семантичний, аналіз потоку даних, аналіз контролю потоку та аналіз безпеки. Вони також можуть використовувати статичні моделі та евристику для виявлення проблем.

Інструменти статичного аналізу можуть виявляти різні типи помилок, такі як дублювання коду, недосяжний код, невизначені змінні, вразливості безпеки, проблеми з витоками пам’яті, недоцільне або неправильне використання API, недотримання стандартів коду, тощо.

Існує безліч технологій для аналізу безпеки, статичними інструменти, найуживаніші з них це:

– SAST;

– DAST;

– IAST;

– RASP.

SAST (Static Application Security Testing), розблений більше десяти років назад, дозволяє знаходити вразливості безпеки в вихідний код застосунку на початкових етапах розробки. SAST забезпечує відповідність коду до стандартів, без фактичного його виконання.

DAST (Dynamic Application Security Testing), виявляє потенційно вразливі та слабкі місця у коді, під час його виконання, частіше за все у вебзастосунках. DAST виявляє поширені вразливості безпеки, такі як sql-інєкції і міжсайтові сценарії, за допомогою впровадження помилок, наприклад передачею шкідливх даних в ПЗ.

IAST (Interactive Application Security Testing), включає в себе основі можливості двох попередників, проводить аналіз у реальному часі, в будь-якому місці процесса розробки в IDE, неперервному інтегрованому осердку, та в виробничому осердку. Може аналізувати код, потоки даних, конфігурації, http-запроси та відповіді, бібліотки, фреймворки, інші компоненти, у порівнянні із SAST і DAST ця технологія дає точніші результати, перевіряє більший спектр правил безпеки, та покриває більшу кількість коду.

RASP (Run-time Application Security Protection), на відміну від раніше описаних технологій, RASP не є засобом тестування, це інструмент, який надає захист системі у реальному часі, відражає атаки, превентивно завершуючі сессії злодіїв, так сповішує про них захисників.

Лінтери – це ще один вид засобів статичного аналізу, які перевіряють код на відповідність певних специфікаціям та стандартам.

Багато інструментів статичного аналізу надають детальні відгуки та рекомендації щодо виправлення виявлених проблем. Це допомагає розробникам швидше виправити помилки та покращити якість коду.

Однією з ключових переваг автоматизованих засобів статичного аналізу є їх здатність швидко та ефективно аналізувати великі обсяги коду без

необхідності ручного втручання. Вони можуть бути інтегровані в середовища розробки (IDE) або використовувати пакетну обробку для аналізу всього проєкту. Вони можуть бути інтегровані в процес розробки програмного забезпечення, дозволяючи автоматично аналізувати код під час компіляції або перед комітом змін в систему контролю версій.

Інструменти статичного аналізу підтримують різні мови програмування та платформи, включаючи C/C++, Java, C#, Python, JavaScript та інші.

Використання цих інструментів допомагає покращити якість програмного забезпечення, зменшити кількість помилок та витрат на виправлення їх після випуску застосунку. Приклад роботи одного із таких інстурментів, наведено на рисунку 1.1, цей інструмент буде розглянуто детальніше у наступних розділах.

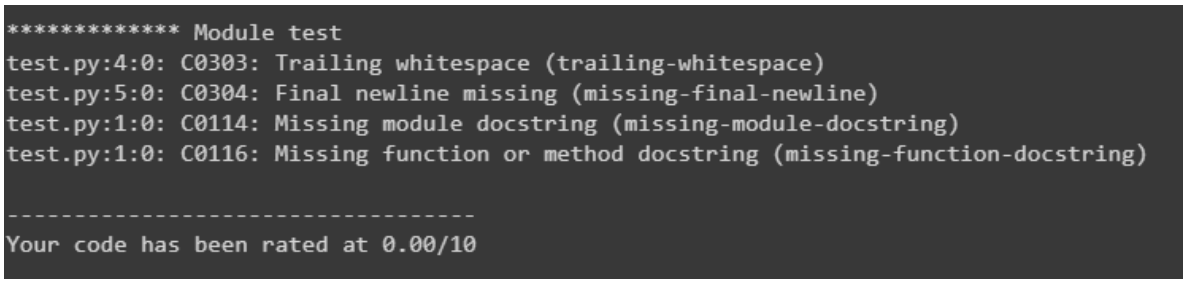

Рисунок 1.1 – Приклад роботи автоматизованого засобу статичного аналізу (Pylint)

1.3 Аналіз потоку даних

Аналіз потоку даних – це важливий метод статичного аналізу вихідного коду програми, спрямований на визначення та аналіз потоків даних в програмі. На рисунку 1.2 [1] зображено базові терміни аналізу потоку даних: *L1* – Точка визначення – визначає елемент даних, *L2* – Точка посилання – посилається на

відповідний елемент даних, *L3* – Точка оцінки – містить вираз що підлягає оцінюванню.

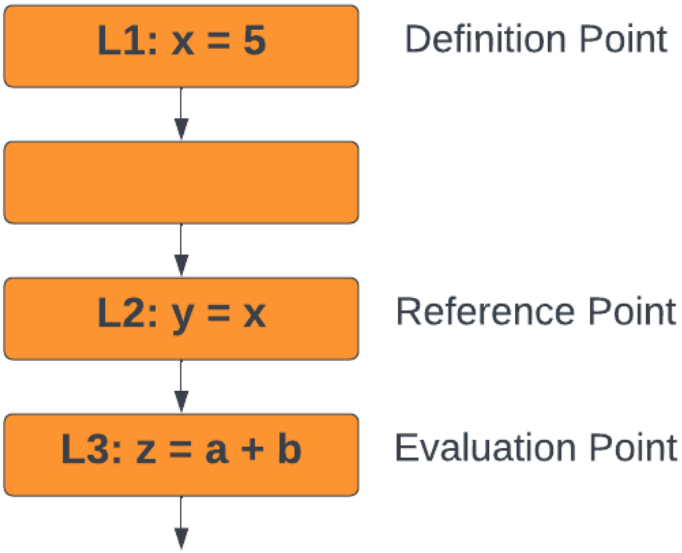

Рисунок 1.2 – Базові терміни аналізу потоку даних

Аналіз потоку даних досліджує, як дані передаються через програму (рис. 1.3), включаючи вхідні та вихідні дані, проміжні обчислені значення (рис. 1.4), передачу даних між функціями та модулями, а також використання даних в умовних конструкціях.

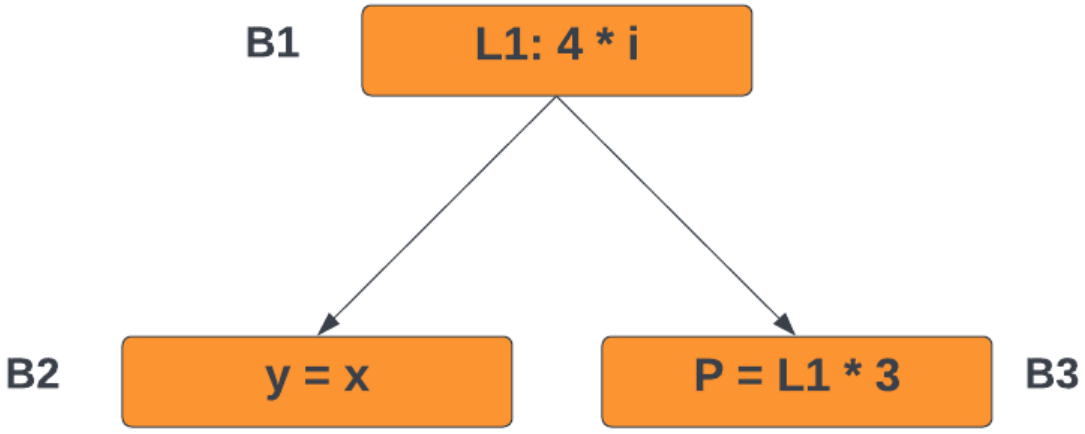

Рисунок 1.3 – Діаграмма доступності виразу

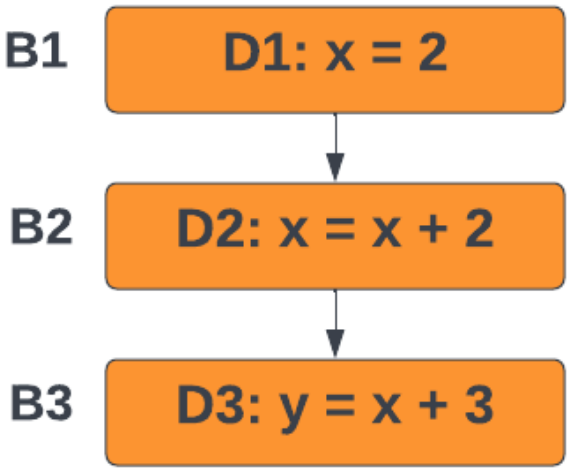

Рисунок 1.4 – Діаграмма досягнення визначенням точки

На рисунку 1.5 наведено схему життя змінної під час виконання програми, *B1* – блок у якому створено змінну *a* *B2* – блок у якому створено змінну с *B3* – блок у якому створено змінну *b*, *B4* – блок я якому використано змінну *a* для встановлення значення змінної *b*, *B5* – блок в якому встановлюється значення змінної *a*, оскільки далі вона ніде не використовується, вона знищується.

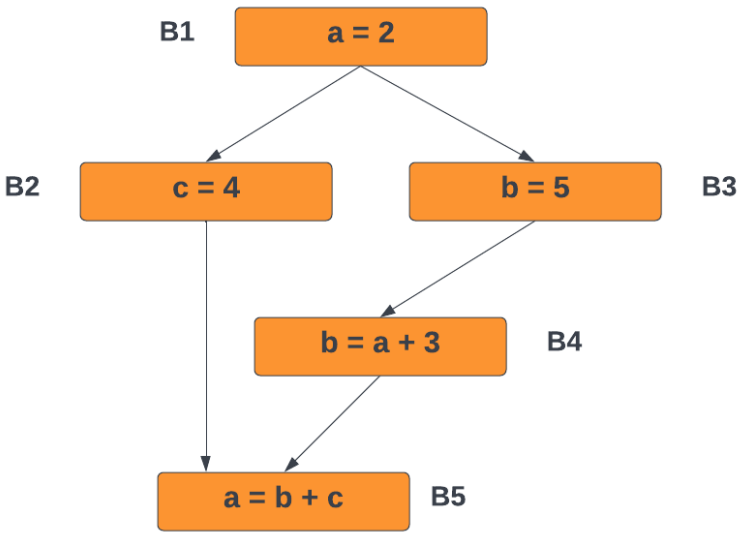

Рисунок 1.5 – Граф життя змінної під час виконання програми

Для сбору інформації необхідної щоб здійснити такий аналіз, використовується рівняння потоку даних (1.1). Цей аналіз може виявляти проблеми, такі як витоки пам’яті, недосяжність коду, неконтрольована змінна, невизначеність даних та інші аномалії.

$$Out[s] = gen[s] \cup In[s] - Kill[s], \quad (1.1)$$

де $Out[s]$ – інформація в кінці оператору s;

$gen[s]$ – інформація створена оператором s;

$In[s]$ – інформація на початку оператору s;

$Kill[s]$ – інформація знищена оператором s.

Аналіз потоку даних часто поєднується з аналізом контролю потоку, який досліджує шляхи виконання програми. Поєднання цих двох аспектів допомагає зрозуміти, як дані впливають на керування потоком в програмі. Аналіз потоку даних часто використовується в інструментах статичного аналізу, таких як компілятори, аналізатори коду та інші інструменти. Вони використовують інформацію про потік даних для виявлення помилок та оптимізації коду.

Зазвичай виділяють наступні види аналізу потоку даних:

– аналіз використання-визначення (Use-Def Analysis);

– аналіз визначення-використання (Def-Use Analysis);

– аналіз антонімів (Antonyms Analysis).

Аналіз використання-визначення – визначає, де визначається змінна, та як вона використовується в коді. Аналіз визначення-використання – визначає, які змінні визначені та як вони використовуються в коді. Аналіз антонімів – виявляє використання змінних, які мають протилежні значення (наприклад, змінна із значенням «*true*» і «*false*» в одному контексті).

Аналіз потоку даних допомагає виявляти помилки в коді, такі як невизначені змінні, витоки пам’яті, дублювання даних та інші проблеми.

Розробники можуть використовувати результати аналізу потоку даних для оптимізації коду та видалення зайвих обчислень. Цей аналіз може бути використаний для виявлення вразливостей безпеки, таких як витоки інформації або ін’єкції даних [15].

Аналіз потоку даних є важливим інструментом у розробці програмного забезпечення для забезпечення якості, безпеки та продуктивності коду.

1.4 Аналіз потоку керування

Аналіз потоку керування – це метод статичного аналізу вихідного коду програми, який досліджує та аналізує порядок виконання і керування потоком в програмі.

Цей метод аналізує структуру програми і визначає, які фрагменти коду виконуються у визначеному порядку. Він вивчає умовні оператори, цикли, вирази переходу та зміну потоку виконання в коді, та виявляє точки виходу з функцій та переходи між функціями. наліз потоку керування досліджує умовні вирази і визначає можливі шляхи виконання в залежності від значень умов. В аналізі потоку керування вивчаються умови виходу з циклів та перебіг виконання циклів.

Здебільшого цей метод застосовують для наступних речей:

– виявлення недосяжного коду: аналіз потоку керування допомагає виявити фрагменти коду, які ніколи не будуть виконані під час реального виконання програми;

– виявлення витрат коду: дослідження циклів та умовних виразів допомагає виявити надмірну складність та невиправдану витрату ресурсів;

– аналіз безпеки: цей метод може використовуватися для виявлення потенційних проблем безпеки, таких як можливі уразливості в коді, які

дозволяють некоректний доступ до даних. Схему аналізу потоку керування наведено на рисунку 1.6 [2].

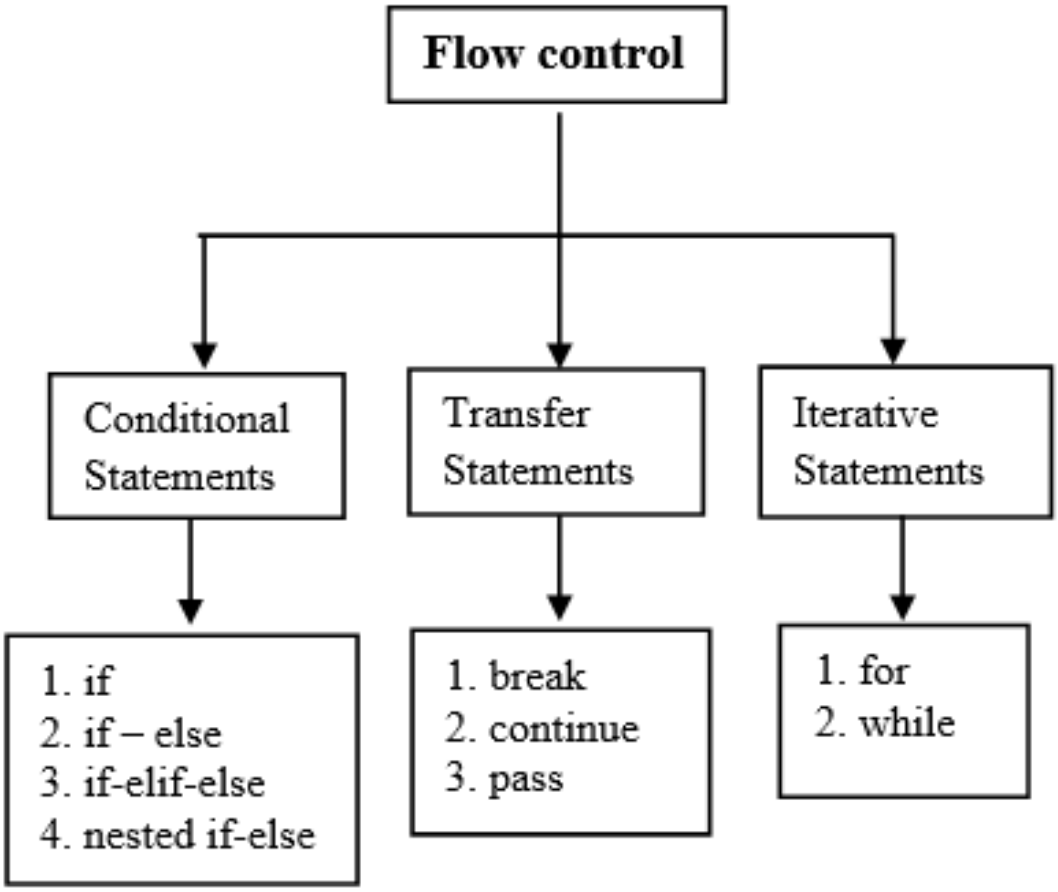

Рисунок 1.6 – Схема потоку керування

Існують спеціалізовані інструменти статичного аналізу, які використовують аналіз потоку керування для виявлення проблем у програмному коді. Такі інструменти можуть надавати відповідні відгуки та рекомендації розробникам щодо поліпшення якості та безпеки коду [16-19].

Аналіз потоку керування є важливим елементом процесу статичного аналізу, оскільки він допомагає зрозуміти, як програма працює та де можуть виникнути проблеми.

Використання цього методу допомагає розробникам покращити якість, безпеку та продуктивність програмного забезпечення.

1.5 Абстрактна інтерпретація

Абстрактна інтерпретація – це математичний метод статичного аналізу вихідного коду програми, спрямований на аналіз і прийняття висновків щодо властивостей програми шляхом створення абстрактних моделей і аналізу їх замість самого коду. Цей метод може бути використаний для виявлення помилок, оптимізації коду, а також для аналізу безпеки та верифікації програмного забезпечення [20].

Ядро абстрактної інтерпретації – це використання абстрактних доменів. Абстрактний домен – це математична структура, яка використовується для апроксимації значень та властивостей даних в програмі. Прикладами абстрактних доменів можуть бути інтервали чисел, решітки, булеві значення, та інші.

Схему абстрактних перетворень зображено на рисунку 1.7 [8]. Це допомагає зменшити складність аналізу та враховувати різноманітні варіанти виконання програми.

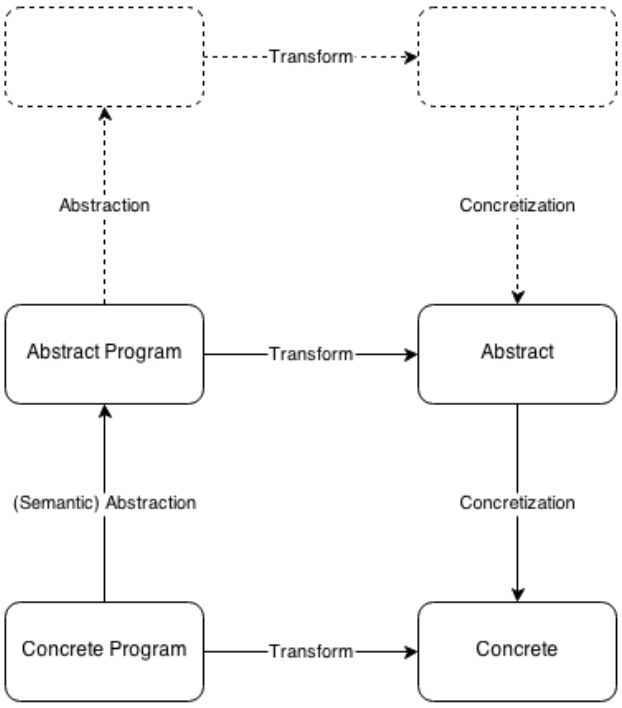

Рисунок 1.7 – Схема абстрактних перетворень

Абстракція полягає в тому, що аналіз робиться не над конкретними значеннями, а над абстрактними моделями, які представляють собою підсумок можливих значень.

Абстрактна інтерпретація зазвичай працює з апроксимованими значеннями та властивостями, тому результати аналізу є приблизними. Точність аналізу залежить від вибору абстрактного домена та апроксимацій, які використовуються.

Абстрактна інтерпретація може виявляти різні помилки в коді, такі як дублювання коду, витоки пам’яті, недосяжний код, невизначені змінні, тож її можна використовувати для виявлення шляхів оптимізації коду, таких як видалення надмірних обчислень чи зайвих перевірок.

1.6 Символьне виконання

Символьне виконання – це метод статичного аналізу програмного коду, який дозволяє моделювати виконання програми замість фактичного виконання. Під час символьного виконання замість конкретних вхідних значень використовуються символьні (символи або символьні вирази), що представляють собою обмеження на вхідні дані. Наприклад, замість вхідного числа «*x*» може використовуватися символьний вираз «*a*», який представляє всі можливі значення для «*x*» (рис. 1.8) [3]. Символьне виконання дозволяє вивчати всі можливі шляхи виконання програми, роблячи рішення на основі символьних виразів для умовних переходів. Це дозволяє виявити усі гілки виконання програми та визначити умови, за яких вони виконуються. Символьне виконання допомагає виявити різні види помилок, такі як дублювання коду, витоки пам’яті, недосяжний код, недопустимі операції та інші аномалії, шляхом аналізу усіх можливих розгалужень під час виконання. Символьне виконання може витрачати багато ресурсів для обчислення при

аналізі великих програм або складних умовних виразів. Також, символьне виконання може зазнавати труднощів при аналізі програм, які взаємодіють з зовнішніми ресурсами або іншими недетермінованими факторами [21-23].

Символьне виконання використовується для виявлення помилок та вразливостей в програмному коді, а також для аналізу безпеки та верифікації програмного забезпечення. Воно може бути корисним для критичних додатків, де важлива безпека та надійність коду.

Символьне виконання – це потужний метод аналізу програмного коду, який дозволяє ретельно досліджувати всі можливі шляхи виконання програми та виявляти помилки та недоліки.

Воно знаходить застосування в багатьох галузях розробки програмного забезпечення та допомагає підвищити якість та безпеку коду.

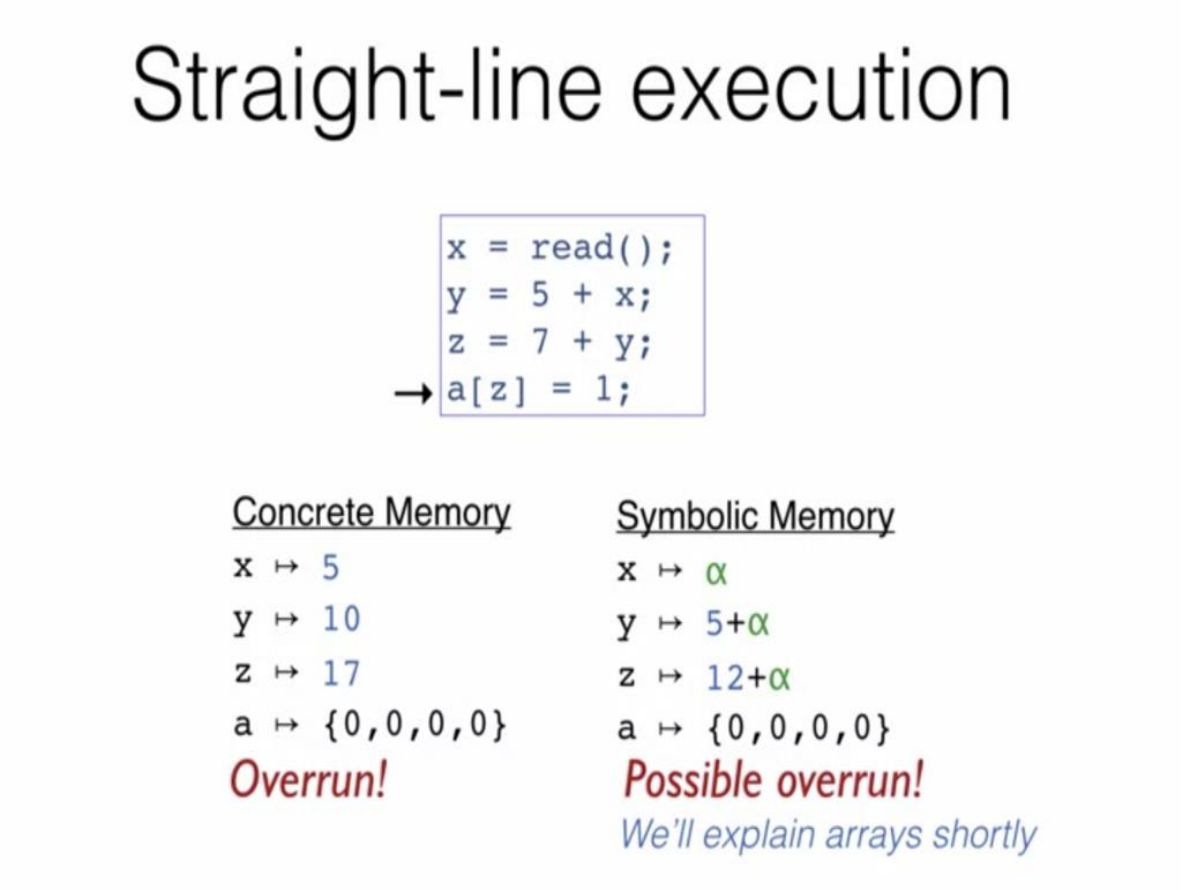

Рисунок 1.8 – Порівняння тестування із визначеними значеннями та зі змінними

1.7 Системи типів

Системи типів грають важливу роль у статичному аналізі коду, спрямованому на визначення та перевірку типів даних у програмах на етапі компіляції. Вони допомагають виявляти та усувати помилки, пов’язані з неправильним використанням типів, та забезпечують безпеку та надійність коду. Системи типів визначають, які види даних можуть бути представлені в програмі та як вони можуть бути використані. Типи даних можуть включати цілі числа, рядки, булеві значення, масиви, структури, об’єкти та інші конструкції. Системи типів використовуються для перевірки правильності використання типів у програмах. Вони допомагають виявляти невідповідності між типами та потенційні помилки, такі як неправильне приведення типів, використання неініціалізованих змінних та інші аномалії [24, 25].

Системи типів полегшують відладку коду, оскільки допомагають ідентифікувати місця, де виникають проблеми з типами даних. Вони надають інформацію про можливі помилки та вказують на точки, де можуть бути неправильно використані дані.

Системи типів сприяють безпеці програмного коду, оскільки дозволяють виявляти помилки на ранніх етапах розробки та уникати вразливостей. Вони гарантують, що дані обробляються коректно, та забезпечують надійність виконання програм.

Системи типів можуть бути статичними або динамічними. Статична типізація перевіряє типи на етапі компіляції, тоді як динамічна типізація робить це на етапі виконання. Багато мов програмування поєднують обидві підходи. Системи типів підтримують поліморфізм, що дозволяє одному функціональному виразу працювати з різними типами даних. Наслідування в системах типів використовується для створення ієрархії класів та об’єктів.

Різні мови програмування мають різні системи типів (рис. 1.9) [5]. Наприклад, Java, C++, C# використовують статичну типізацію, Python,

JavaScript – динамічну. Системи типів є важливою частиною розробки програмного забезпечення і допомагають забезпечити правильність та безпеку коду. Вони дозволяють розробникам ідентифікувати та виправляти помилки на ранніх етапах розробки, підвищуючи якість та надійність програм.

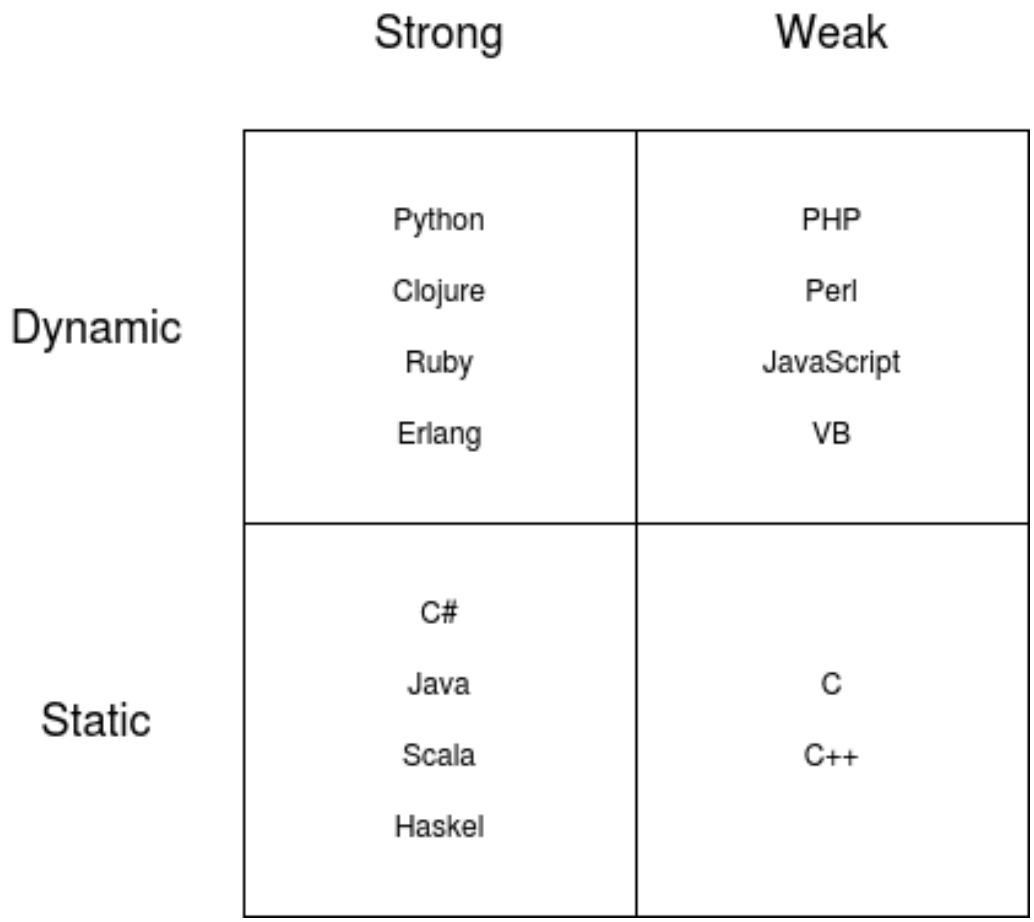

Рисунок 1.9 – Системи типів найпопулярніших мов програмування

1.8 Пошук за шаблонами

Пошук за шаблонами у статичному аналізі відноситься до техніки, яка використовується для пошуку конкретних шаблонів або структур у вихідному коді програми. Це може бути будь-яка структура даних, виклики функцій, використання змінних, або навіть послідовності дій, які специфікують певну логіку чи дію в програмі.

Основна мета пошуку за шаблонами є виявлення найкращих практик та антипатернів у коді. Наприклад, пошук за шаблонами може допомогти виявити витоки пам’яті, неоптимальне використання ресурсів, а також структури, які можна оптимізувати.

Пошук за шаблонами часто використовується для виявлення потенційних вразливостей у програмному коді, таких як sql-ін’єкція, витоки інформації, атаки на безпеку та інші проблеми безпеки.

Існують спеціалізовані інструменти та бібліотеки, призначені для пошуку за шаблонами у вихідному коді. Деякі з них використовують регулярні вирази, інші використовують статичний аналіз та інші методи для виявлення шаблонів.

Пошук за шаблонами в статичному аналізі передбачає аналіз вихідного коду програми без її виконання. Це дозволяє виявити проблеми та вразливості на етапі розробки та підвищити якість та безпеку програмного забезпечення [26].

Пошук за шаблонами використовується в різних галузях розробки програмного забезпечення, включаючи верифікацію коду, аналіз безпеки, аналіз якості, рефакторинг та багато інших. Він є важливим інструментом для підвищення якості та безпеки програм, та для покращення розробки програмного забезпечення.

1.9 Аналіз метрик та складності коду

Аналіз метрик та складності коду – це метод статичного аналізу програмного коду, спрямований на вимірювання, оцінку та аналіз якості та складності коду. Цей метод допомагає розробникам та командам розробки отримувати кількісну інформацію про програмний код для покращення його розуміння, підтримки та оптимізації.

Метрики коду – це вимірювання та числові характеристики, які використовуються для оцінки різних аспектів якості, розміру та складності програмного коду. Ці метрики можуть бути використані для оцінки продуктивності, підтримки, безпеки та інших аспектів програми.

До прикладів метрик коду включаються:

– кількість рядків коду: вимірює загальний обсяг коду;

– кількість коментарів: вказує на наявність документації та читабельності коду;

– кількість функцій та класів: вимірює структуру програми;

– складність коду: вимірює складність алгоритмів та функцій, часто використовуючи метрику cyclomatic complexity;

– кількість вимог на зміну (часто змінюваний код): вказує на потребу у підтримці та рефакторингу;

– метрики залежності між модулями: вимірюють структуру програми та залежності між її частинами;

– метрики продуктивності: вимірюють продуктивність програми, такі як час виконання.

Аналіз метрик та складності коду допомагає виявити потенційні проблеми та недоліки в програмному коді, полегшує рефакторинг, допомагає покращити розуміння коду та визначити обсяги робіт для підтримки.

Існують спеціалізовані інструменти так як: SonarQube, PMD, Checkstyle, Lint, та інші, для аналізу метрик та складності коду, які автоматизують процес вимірювання та аналізу.

Аналіз метрик та складності коду застосовується у різних галузях розробки програмного забезпечення, включаючи верифікацію коду, аналіз безпеки, аналіз якості, рефакторинг, планування проєктів та багато інших. Він допомагає покращити процес розробки та забезпечити високу якість та надійність програмного забезпечення [27].

1.10 Постановка задачі дослідження

Таким чином, проаналізувати існуючі методи, та виявити їхні переваги та недоліки є актуальним завданням для пошуку вразливостей у програмному коді. Тому ставиться завдання створення порівняльної характеристики методів та засобів статичного аналізу.

Об’єктом дослідження є набір документів із програмним кодом.

Метою дослідження є аналіз та оцінка методів виявлення вразливостей, що базуються на статичному аналізі програм.

Для досягнення мети необхідно вирішити такі завдання:

– провести аналіз існуючих методів статичного аналізу;

– провести аналіз існуючих інструментів, що реалізують ці методи;

– розробити систему оцінювання, скориставшись нею порівняти отримані результати аналізу.

2 ІНСТРУМЕНТИ СТАТИЧНОГО АНАЛІЗУ КОДУ

2.1 Лінтери

Як було згадано раніше, лінтери – інструменти статичного аналізу, які перевіряють відповідність коду специфікаціям, або відповідним стандартам. Далі розглянемо деякі з них.

Лінтери бувають різні, ті які реалізують базові потреби, наприклад flake8, який перевіряє лише стиль написання коду, складніші, які окрім стилистикі перевіряють семантику, зокрема pylint, та ті які впроваджують додаткові перевірки, такі як type checking, але для python це новий функціонал, та наразі він знаходиться в стороні [28-30].

Щодо Pylint, можна одразу відзначити деякі його недоліки, це те що він має дуже багато різних, не завжди корисних та налшатовуваних перевірок, не рідко стається те що він звертає увагу на місця в коді, які фактично не містять помилок. Також слід відзначити що в pylint власна реалізація abstract syntax tree, це може впливати на якість перевірок, часто у гіршу сторону.

Flake8 навідміну від попереднього лінтера, не має такої кількості перевірок, і не здатен перевіряти семантику, але він підтримує плагіни, і може ними розширювати свій функціонал, враховуючи специфіку окремого проєкта.

На інших мовах програмування лінтери представлені теж великою кількістю різних біблотек, зокрема на javascript – ESLint, JSHint, а на .Net Roslyn Anylyzers та FxCop. Від мови до мови, від біліотеки до бібілотеки, вони дещо відрізняються але у цілому містять схожий функціонал.

На рисунку 2.1 [4] наведені лінетри для найпопулярніших мов програмування.

Рисунок 2.1 – Популярні лінтери

2.2 Аналізатори вразливості безпеки

Окрім лінтерів, існує інший вид інструментів, що забезпечують статичний аналіз даних – аналізатори вразливостей безпеки. Ці інструменти призначені для виявлення потенційних вразливостей, які можуть бути використані зловмисниками для атаки на систему. Їх основне завдання виявлення вразливостей пов’язаних з недостатньою перевіркою введених даних, невірним керуванням доступом, вразливостями типу буферного переповнення та інші. Аналізатори виявляють можливості для виконання sql-інєкцій, введення шелл-команд, атак на основні XSS, CSRF та інші.

Такі можливості мають SonarQube, Checkmarx. Fortify, тощо.

Деякі аналізтори можуть перевіряти потік яким ширяться дані у застосунку, виявляючі шляхи атак та точки введення які можуть бути використані хакерами, такий функціонал має зокрема Fortify, також Veracode.

Аналізтори можуть застосовувати набір правил, для перевірки дотримання кодом стандартів безпеки. Це включає в себе правила для уникнення використання небезпечних функцій, неналежного використання криптографії, тощо. Для цього призначені ESLint(javascript), Bandit(python).

Деякі інструменти можуть інтегруватися з базами даних відомих загроз та використовувати їхні дані для виявлення вразливостей. Це дозволяє швидше виявляти вразливості, які вже були описані та класифіковані. Основні типи вразливостей безпеки наведені на рисунку 2.2 [7].

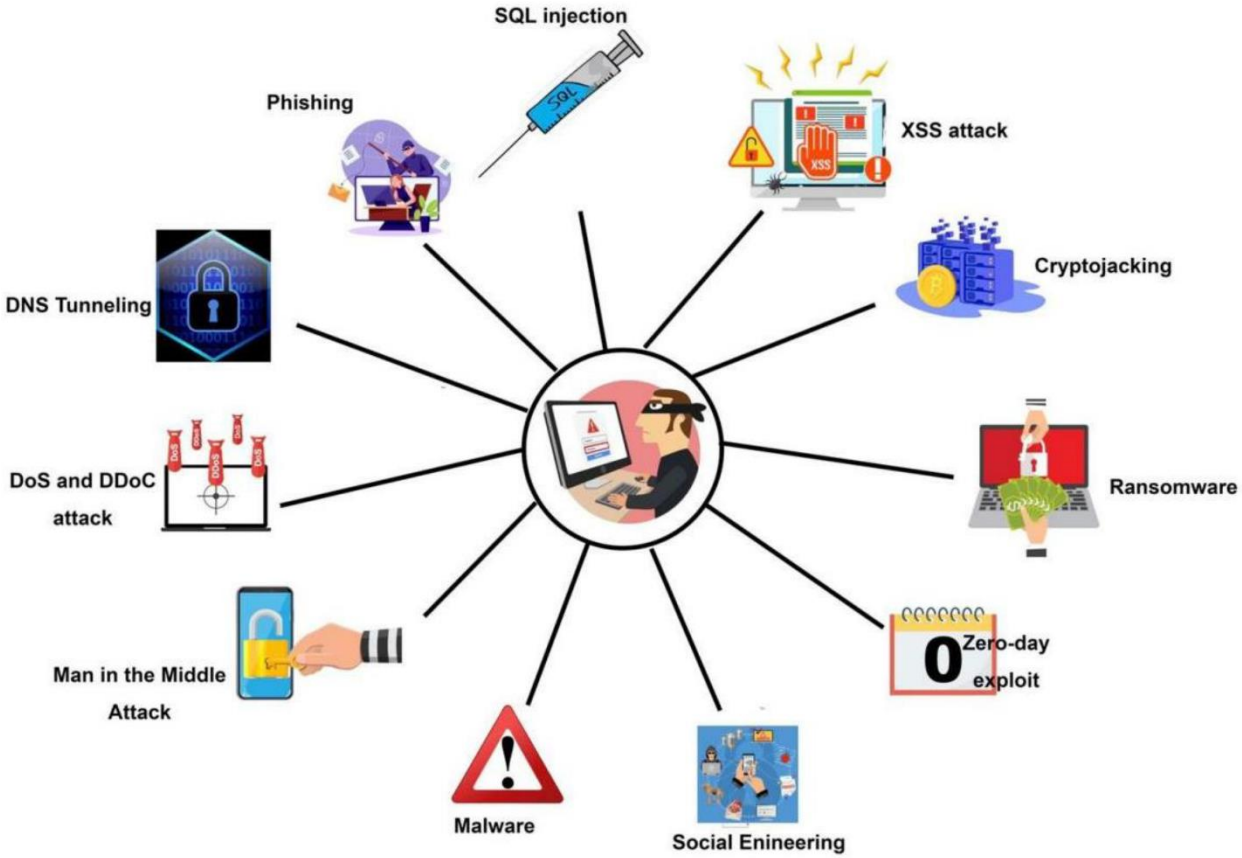

Рисунок 2.2 – Типи вразливостей безпеки

2.3 Аналізатори якості коду

Аналізатори якості коду у статичному аналізі є інструментами, спрямованими на визначення та оцінку якості програмного коду без його фактичного виконання. Вони стежать за відповідністю коду визначеним стандартам, виявляють потенційні проблеми та рекомендують поліпшення з точки зору читабельності, ефективності та обслуговуваності.

Аналізатори використовують визначені стандарти коду та метрики для оцінки якості програми. Це включає в себе перевірку належного форматування коду, дотримання стилів та визначених критеріїв якості.

Інструменти виявляють «забруднення» або погані практики в програмному коді, які можуть призводити до невиправних проблем у майбутньому. Це може включати в себе виявлення дублікатів коду, довгих методів, антипатернів та інших аномалій.

Деякі аналізатори обчислюють цикломатичну складність коду, яка визначає кількість незалежних шляхів у коді. Велика цикломатична складність може свідчити про потребу в рефакторингу для покращення читабельності та обслуговуваності [30].

Аналізатори можуть виявляти потенційні проблеми та недоліки в коді, такі як невикористані змінні, недоступний код, або інші конструкції, які можуть викликати проблеми.

2.4 Аналізатори типів та анотації типів (Type Checkers)

Аналізатори типів та анотації типів є інструментами, спрямованими на перевірку та забезпечення 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 виконання програми.

Аналізатори типів роблять це, спираючись на інформацію про типи, яку може надавати сам код чи спеціальні анотації типів.

Багато мов програмування дозволяють використовувати анотації типів, які допомагають аналізаторам правильно визначати типи даних. Анотації типів можуть бути використані для зазначення типів параметрів функцій, змінних, або інших елементів коду.

Деякі аналізатори дозволяють визначати межі для типів даних. Наприклад, вказуючи, що деяка змінна повинна бути підтипом певного типу.

Аналізатори типів можуть сприяти покращенню безпеки та надійності програмного коду, оскільки неправильне використання типів може призвести до важливих помилок під час виконання програми [31].

2.5 Аналізатори викликів функцій та залежностей

Аналізатори викликів функцій та залежностей у статичному аналізі є інструментами, спрямованими на виявлення та аналіз взаємозв’язків між різними функціями, класами, модулями та іншими компонентами програмного коду.

Ці інструменти допомагають розробникам розуміти структуру програми, виявляти залежності та можливі проблеми, такі як циклічні залежності.

Інструменти аналізують виклики функцій та методів у програмному коді. Це дозволяє виявляти, які частини коду взаємодіють між собою та визначати, які функції використовуються в конкретних контекстах.

Аналізатори можуть виявляти циклічні залежності між різними компонентами програми. Це допомагає уникнути ситуацій, коли модулі

взаємно залежать один від одного, утворюючи неприродню та складну структуру.

Приклад схеми виклуку функцій проілюстровано рисунком 2.3.

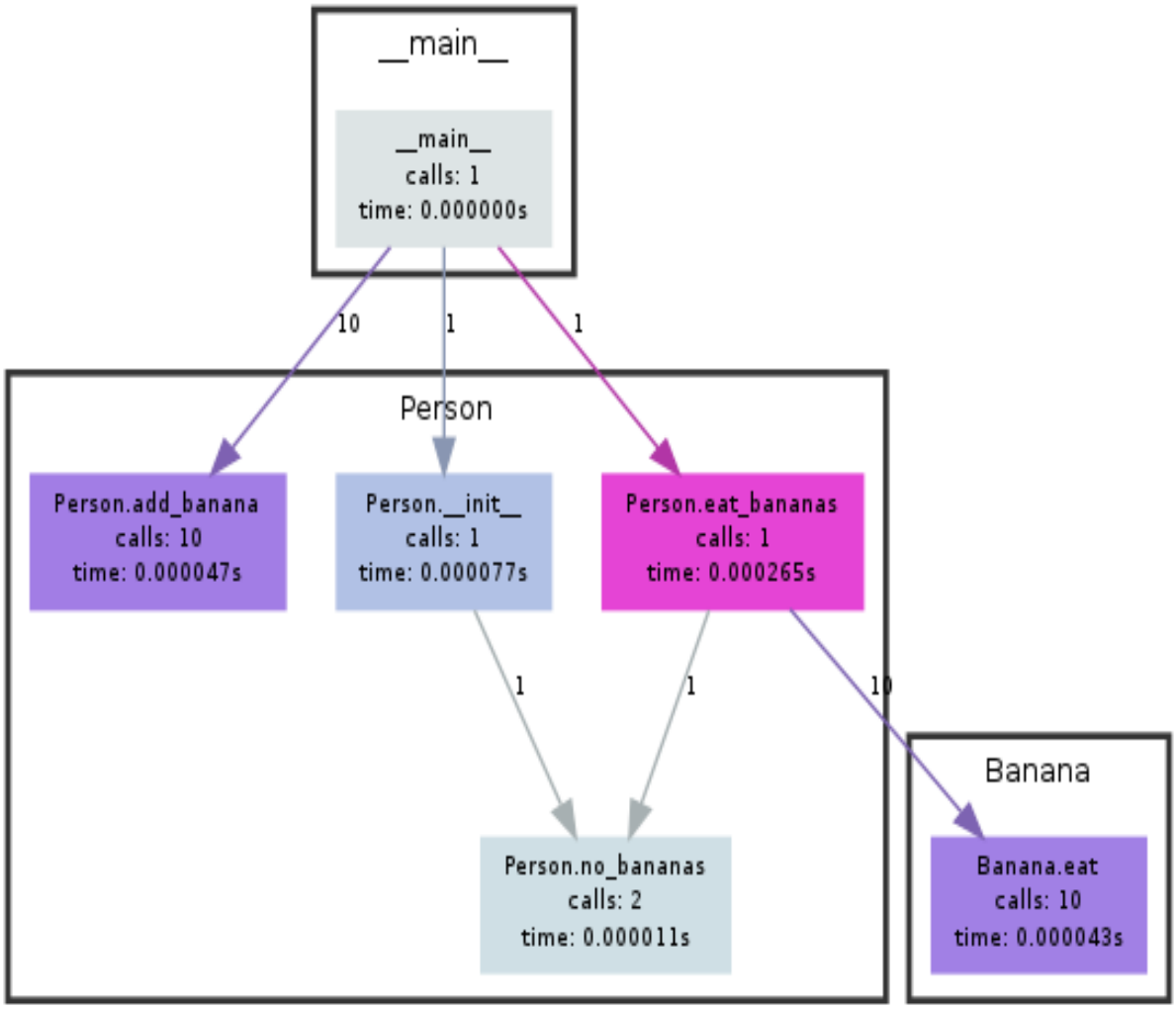

Рисунок 2.3 – Граф виклику функцій

Аналізатори можуть виявляти залежності від сторонніх бібліотек та компонентів, що дозволяє розробникам визначити, які бібліотеки використовуються у їхньому проєкті та як це впливає на структуру програми.

Аналізатори можуть надавати поради щодо рефакторингу та оптимізації коду на основі залежностей. Це може включати в себе рекомендації щодо розбиття функцій чи модулів на менші компоненти.

2.6 Аналізатори використання ресурсів та оптимізації

Аналізатори використання ресурсів та оптимізації у статичному аналізі це інструменти, які спрямовані на виявлення можливих проблем використання ресурсів (пам’яті, обчислювальної потужності, мережевих ресурсів тощо) та надають рекомендації для покращення ефективності програмного коду.

Аналізатори перевіряють код на наявність потенційних витоків пам’яті, де пам’ять виділяється, але не вивільняється після завершення виконання програми.

Деякі аналізатори виявляють частини коду, які можуть використовувати зайве обчислювальне навантаження або витрачати занадто багато ресурсів процесора Деякі інструменти спрямовані на виявлення можливих проблем у коді, що впливають на мережеві ресурси, такі як зайвий трафік чи неефективні мережеві запити.

Окремі аналізатори можуть виявляти неоптимальні алгоритми та структури даних, які можуть призводити до зайвого використання ресурсів.

Одні з інструментів, призначені для аналізу ресурсів у багатозадачних або розподілених системах, де ефективне використання ресурсів є ключовим завданням. Багато аналізаторів можуть надавати рекомендації щодо оптимізації конкретних частин коду для зменшення використання ресурсів.

2.7 Підсумки

Багато з наведених інструментів статичного аналізу, поєднують у собі декілька видів аналізу, і являють собою мульти-тули у світі аналізу вихідного коду.

Слід зауважити що подібні інструменти представлено у різних видах, як у вигляді бібліотек, інтегрованих модулів у IDE, так і у вигляді окремих застосунків.

Оскількі для кожної мови, існують свої відповідні іструменти, слід визначити мову, для якої буде розроблено порівняльну характеристику, і запропоновано набір інструментів, який надасть найкраший реузльтат.

3 ПОРІВНЯННЯ ІНСТРУМЕНТІВ СТАТИЧНОГО АНАЛІЗУ

3.1 Pylint

Розглянемо Pylint, він був згаданий у попередніх розділах, адже є одним із найпопулярніших засобів статичного аналізу. Pylint виявляє та класифікує можливі помилки в коді, включаючи невизначені змінні, неправильне використання функцій тощо. Він перевіряє відповідність коду стандартам оформлення, таким як PEP8. Аналізує складності коду за допомогою метрик, таких як цикломатична складність McCabe. Він потребує деякої настройки для оптимального використання.

Код приведений у лістингу 3.1, був проаналізований за допомогою цього інструменту, результати наведені на рисунку 3.1.

Лістинг 3.1 Тестовий код для аналізу за допомогою Pylint:

```
class ExampleClass:
def __init__(self, name):
self.name = name

def greet(self):
print(f"Hello, {self.name}!")

def add_numbers(a, b):
result = a + b
return result

def main():
user_name = input("Введіть ваше ім'я: ")
my_instance = ExampleClass(user_name)
```

```
my_instance.greet()

num1 = 10
num2 = "20"  # Це призведе до помилки типу

result = add_numbers(num1, num2)
print(f"Результат додавання: {result}")

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

```
************* Module test
test.py:26:0: C0304: Final newline missing (missing-final-newline)
test.py:1:0: C0114: Missing module docstring (missing-module-docstring)
test.py:3:0: C0115: Missing class docstring (missing-class-docstring)
test.py:7:4: C0116: Missing function or method docstring (missing-function-docstring)
test.py:3:0: R0903: Too few public methods (1/2) (too-few-public-methods)
test.py:10:0: C0116: Missing function or method docstring (missing-function-docstring)
test.py:14:0: C0116: Missing function or method docstring (missing-function-docstring)

-----------------------------------
Your code has been rated at 6.11/10
```

Рисунок 3.1 – Результат аналізу тестового коду за допомогою Pylint

Оскількі Pylint аналізує якість коду на основі різних метрик, було використано показовий код, який містить клас ExampleClass з методом greet та функцію add_numbers, яка додає два числа. Окрім того, у функції main є введення користувача та спроба додати числа різних типів, що призводить до помилки типу.

У приведених результатах видно декілька стилістичних помилок, таких як: відсутність пусто рядка в кінці, відсутність комментаря докумментації для модуля та для функцій, та деякі інші. І надано орієнтовну оцінку якості коду – 6.11 з 10.

3.2 Муру

Динамічно-типізований Python приваблює своєю зручністю та зрозумілістю, але статична типізація знижує ймовірність виникнення помилок пов’язаних із типами. Масштаб є головним чинником що створює необхідність використовувати статичну перевірку типів, адже із стростанням проєку зростає і необхідність її використовувати. Декларації типів виступають у ролі перевіреної машинної документації, а статична типізація робить ваш код більш зрозумілим, його легше модифікувати, не припускаючи помилок. В вирішенні цієї проблеми допоможе муру.

Муру – це статично перевірка типів для Python. Вона працює як лінтер, шо дозволяє писати статично-типізований код і перевіряти правильність типів у проєкті. При цьому, код потребує аннотування із використанням синтаксу функцій Python 3 (PEP484). В такому випадку МуРу зможе перевірити код на типи і знайти типові помилки. Його призначення – поєднати переваги динамічної та статичної (за допомгою модуля) типізації. Код приведений у лістингу 3.2, був проаналізований за допомогою цього інструменту, результати наведені на рисунку 3.2.

Лістинг 3.2 Тестовий код для аналізу за допомогою Муру:

```
def multiply(a: int, b: int) -> int:
return a * b

def greet(name: str) -> str:
return f"Hello, {name}!"

def main() -> None:
result = multiply(5, 3)
print(f"Результат множення: {result}")
```

```
person_name = "John"
greeting = greet(person_name) print(greeting)

Invalid_result = result + greeting

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

```
test.py:18: error: Unsupported operand types for + ("int" and "str")  [operator]
Found 1 error in 1 file (checked 1 source file)
```

Рисунок 3.2 – Результат аналізу тестового коду за допомогою Муру

Зважаючи на те, що Муру використовується для статичної типізації, тож код для розгляду включає помилку використання типів.

У цьому коді функція multiply приймає два цілочисельних аргументи та повертає їхній добуток, функція greet приймає рядок та повертає вітання у вигляді рядка. У функції main обчислюється результат множення, виводиться вітання, але також робиться спроба просумувати результат множення та вітання, це спричиняє помилку типу.

3.3 Pyflakes

Підхід Pyflakes схожий на Pylint, але він зосереджений на тому щоб не видавати хибних спрацьовувань. Pyflakes ніколи не буде звертати уваги на стиль. Це значить, що він не буде повідомлювати вас про відсутність рядка документації або імен аргументів які не відповідають стилю іменування. Він зосереджений на логічних проблемах кода та потенціальних помилках.

Pyflakes досліджує лише синтаксичне дерево, кожного файлу окремо. Це поєднанні з обмеженим набором помилок, робить його швидшим, ніж pylint. З іншої сторони, більш обмежений в можливостях перевірки.

Сам Pyflakes не робить жодної стилістичної перевірки, але існує інший інструмент, який поєднує Pyflakes з перевіркою стилю PEP8 – Flake8.

Код приведений у лістингу 3.3, був проаналізований за допомогою цього інструменту, результати наведені на рисунку 3.3.

Лістинг 3.3 Тестовий код для аналізу за допомогою Pyflakes:

```
def calculate_sum(a, b):
    result = a + b
    print(f"Результат додавання: {result}")

def main():
    num1 = 10
    num2 = 20

    calculate_sum(num1, num2)
    print(num3)

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

```
test.py:12:11: undefined name 'num3'
```

Рисунок 3.3 – Результат аналізу тестового коду за допомогою Pyflakes

Pyflakes спрямований на виявлення потенційних помилок та неправильно використання змінних, відповідно до цього було обрано запропонований код.

У цьому коді функція calculate_sum приймає два аргументи, додає їх та виводить результат. Функція main визначає дві змінні num1 та num2 та викликає функцію calculate_sum з цими аргументами. Далі робиться спроба вивести значення змінної num3, однак її не існує, тому виникає помилка.

3.4 Pycodestyle

Раніше Pycodestyle носив назву pep8, був переіменований у зв'язку із плутаниною з стандартом PEP8. Pycodestyle – офіційний лінтер для перевірки коду Python на відповідність стилям PEP8. Pycodestyle не забезпечує виконання усіх правил стандарту, він не носить вичерпний характер, лише допомагає увпевнитися у виконанні деяких узгоджень кодування. Деякі правила – це лише рекомендації, які за потреби можуть бути обійденеми, а інші – неможливо подати у вигляді просто алгоритму.

Код приведений у лістингу 3.4, був проаналізований за допомогою цього інструменту, результати наведені на рисунку 3.4.

Лістинг 3.4 Тестовий код для аналізу за допомогою Pycodestyles:

```
def calculate_sum(a, b):
    result = a + b
    print(f"Результат додавання: {result}")

def main():
    num1 = 10
    num2 = 20

    calculate_sum(num1, num2)

    invalid_variable=5

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

```
test.py:5:1: E302 expected 2 blank lines, found 1
test.py:12:21: E225 missing whitespace around operator
test.py:14:1: E305 expected 2 blank lines after class or function definition, found 1
test.py:15:11: W292 no newline at end of file
```

Рисунок 3.4 – Результат аналізу тестового коду за допомогою Pycodestyles

3.5 Flake8

Як і згадувалося раніше, Flake8 – інструмент побудований над PyFlakes їх використанням PyCodeStyle(pep8) і цикломатичної перевірки складності(використовується для пошуку складного кода).

Flake8 – може бути розширений різними модулями, існує велика кількість плагінів, під будь-які потреби. Flake8 має потужний механізм налаштування використовуючи один із файлів конфігурації «.flake8», «setup.cfg», «tox.ini». Плагіни, та попередження можуть бути вимкнені у кожному файлі, або рядку.

Код приведений у лістингу 3.5, був проаналізований за допомогою цього інструменту, результати наведені на рисунку 3.5.

Лістинг 3.5 Тестовий код для аналізу за допомогою Flake8:

```
def calculate_sum(a, b):
    result = a + b
    print(f"Результат додавання: {result}")

def main():
    num1 = 10
    num2 = 20

    calculate_sum(num1, num2)

    num3=5

    num4=5

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

```
test.py:5:1: E302 expected 2 blank lines, found 1
test.py:11:5: F841 local variable 'num3' is assigned to but never used
test.py:11:9: E225 missing whitespace around operator
test.py:13:5: F841 local variable 'num4' is assigned to but never used
test.py:13:11: E225 missing whitespace around operator
test.py:15:1: E305 expected 2 blank lines after class or function definition, found 1
test.py:16:11: W292 no newline at end of file
```

Рисунок 3.5 – Результат аналізу тестового коду за допомогою Flake8

Як було зазначено раніше Flake8 це інструмент, що включає в себе функції декількох інших, тож було створено код для показового тестування. На відміну від попереднього інструменту Flake8 має кольорове виділення інформації про помилки.

У цьому коді функція *calculate_sum* приймає два аргументи, додає їх та виводить результат. Функція main визначає дві змінні *num1* та *num2* та викликає функцію *calculate_sum* з цими аргументами. Далі створено, але так і не використано дві змінні *num3* та *num4*.

Так було допущено декілька помилок, у вигляді зайвих порожніх рядків, та пробілів.

3.6 Prospector

Потужний інструмент статичного аналізу для Python, який надає інформацію про помилки, потенційні проблеми, порушення узгоджень і складності. Включає в себе:

– Pylint – якість коду, відстеження помилок, відстеження дублікатів коду;

– pep8.py – якість кода у відповідності до PEP8;

– pep257.py – якість коментарів у відповідності до PEP27;

– PyFlakes – відстеження помилок;

– Mccabe – аналіз цикломатичної складності;

– Dodgy – відстеження витоків секретів;

– Pyroma – setup.py валідатор;

– Vulture – відстеження недосяжного, або невикористовуваного коду.

Більшість попереджень, надходячих, від таких інструментів як Pylint, pep8 або Pyflake, імовірно, будуть прискіпливими.

Існують попередження про довжину рядків, про пробіли у порожніх рядках, про те, скількі місця поміж методами класу и тощо. Але насправді список справжніх проблем у коді, не потрібен.

Тому Prospector має налаштування та поведінку за замовчування, що дають змогу пригнічувати настирні попередження і відображати лише те, що важливо.

Також можна налаштовувани серйозність, ігнорування деяких помилок, вмикати/вимикати інструменти, у файлі «. prospector.yml».

Код приведений у лістингу 3.6 був проаналізований за допомогою цього інструменту, результати наведені на рисунку 3.6.

Лістинг 3.6 Тестовий код для аналізу за допомогою Prospector:

```
import os

def calculate_sum(a, b):
    result = a + b
    print(f"Результат додавання: {result}")

def main():
    num1 = 10
    num2 = 20

    calculate_sum(num1, num2)

    # Помилка: зайвий пробіл перед знаком =
    invalid_variable=5

    # Помилка: відсутній пропуск перед знаком =
    invalid_variable2 =5

main()
```

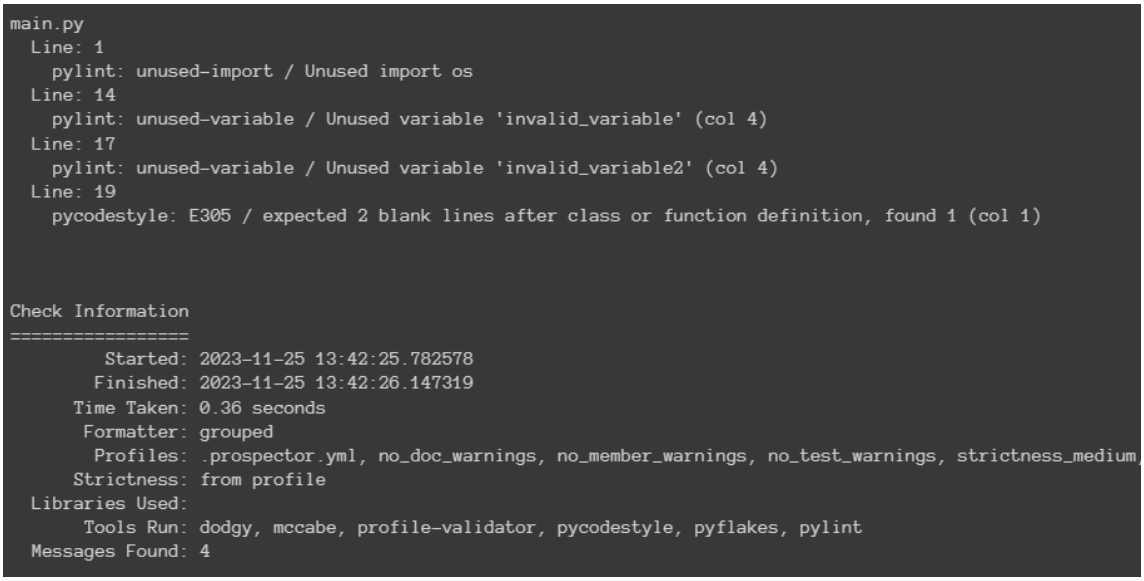

Рисунок 3.6 – Результат аналізу тестового коду за допомогою Prospector

Оскількі Prospector включає в себе одразу багато інструментів, відповідно і код використано у якому було допущено багато різних помилок.

3.7 Bandit

Bandit – інструмент, призначений для пошуку розповсюджених проблем безпеки в коді Python. Для цього він аналізує кожний файл, будує з нього AST і запускає відповдні плагіни для вузлів AST.

Він призначений для:

– виявлення дефектів безпеки;

– захардкожені паролі;

– невірна серіалізація/десеріалізація pickle;

– інєкції оболонки;

– sql-інєкції.

Код приведений у лістингу 3.7, був проаналізований за допомогою цього інструменту, результати наведені на рисунку 3.7

Лістинг 3.7 Тестовий код для аналізу за допомогою Bandit:

```
import os

def insecure_function(user_input):
    os.system("rm -rf /")  # Небезпечна операція

def safe_function(user_input):
    print(f"Введене значення: {user_input}")

if __name__ == "__main__":
    user_input = input("Введіть дані: ")

    insecure_function(user_input)
    safe_function(user_input)
```

```
Test results:
>> Issue: [B605:start_process_with_a_shell] Starting a process with a shell: Seems safe, but may be chang
   Severity: Low   Confidence: High
   CWE: CWE-78 (https://cwe.mitre.org/data/definitions/78.html)
   More Info: https://bandit.readthedocs.io/en/1.7.5/plugins/b605_start_process_with_a_shell.html
   Location: main.py:4:4
3       def insecure_function(user_input):
4           os.system("rm -rf /")  # Небезпечна операція
5

--------------------------------------------------
>> Issue: [B607:start_process_with_partial_path] Starting a process with a partial executable path
   Severity: Low   Confidence: High
   CWE: CWE-78 (https://cwe.mitre.org/data/definitions/78.html)
   More Info: https://bandit.readthedocs.io/en/1.7.5/plugins/b607_start_process_with_partial_path.html
   Location: main.py:4:4
3       def insecure_function(user_input):
4           os.system("rm -rf /")  # Небезпечна операція
5

--------------------------------------------------

Code scanned:
        Total lines of code: 9
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 2
                Medium: 0
                High: 0
        Total issues (by confidence):
                Undefined: 0
                Low: 0
                Medium: 0
                High: 2
Files skipped (0):
```

Рисунок 3.7 – Результат аналізу тестового коду за допомогою Bandit

3.8 Порівняння інструментів

Кожен з наведених інструментів має індивідуальні особливості, як з точки зору аналізу, так із точки зору представлення результатів. Хоча наразі більшість інструментів виходить за межі свого базового призначення, але все ще можуть бути віднесені до певного типу, зазначено в таблиці 3.1 [20-30].

Таблиця 3.1 – Типи інструментів

Інструмент	**Тип**
Pylint	Обширний аналіз
Mypy	Статична типізація
PyFlakes	Лінтер
Pycodestyle	Лінтер
Flake8	Комбінований (Лінтер + Інші)
Prospector	Комбінований (Лінтер + Інші)
bandit	Аналіз Безпеки

Усі наведені інструменти так чи інакше використовуються у реальних проєктах, хоча найчастіше їх ролі очевидні, у таблиці 3.2 наведено, мету з якою використовують ці засоби.

Таблиця 3.2 – Призгначення інструментів

Інструмент	**Напрямок**
Pylint	Якість коду
Mypy	Типізація, Помилки типів
PyFlakes	Лінтинг
Pycodestyle	Стиль коду
Flake8	Лінтинг + Стиль коду
Prospector	Загальний (Лінтинг, Стиль, Безпека)
bandit	Безпека

Кожна з представлених біліотек, досить популярна і використовуєтся розробниками, для покращення свого коду, але слід за значити що у різних випадках варто використовувати відповідні іструменти, у таблиці 3.3 наведно переваги та недоліки цих засобів.

Таблиця 3.3 – Переваги та недоліки інструментів

Інструмент	**Переваги**	**Недоліки**
Pylint	Високий рівень деталізації та широкий спектр перевірок, інтеграція з багатьма IDE, добре документований.	В деяких випадках надає багато хибнопозитивних результатів, вимагає додаткових налаштувань для окремих проєктів.
Муру	Спрощує введення статичної типізації в Python, допомагає виявляти помилки типів, інтеграція з іншими інструментами.	Вимагає анотацій типів, що може бути часоємним, може викликати виклики для великих проєктів.
PyFlakes	Швидкий та простий використовуваний, фокус на виявленні невизначених та не використовуваних змінних.	В окремих випадках надає хибні результати, обмежений в аспекті загального аналізу коду
Pycodestyle	Простий у використанні та налаштуванні, об’єднує PyFlakes, McCabe та Зycodesty.	Малодеталізований порівняно з іншими лінтерами, часом ігнорує деякі стилістичні проблеми.
Flake8	Комбінує PyFlakes, pycodestyle та ще декілька лінтерів, легко налаштовується.	Може бути менш деталізований порівняно з Pylint, зосереджений на стилі, але може бути менш суворим. Важко налаштовується для конкретних потреб.

Продовження таблиці 3.3

Інструмент	**Переваги**	**Недоліки**
Prospector	Комбінує багато різних інструментів, доступнх одразу після встановлення, має детальний вивід.	Може виводити багато повідомлень, що може бути, як перевагою, так і недоліком залежно від потреб.
bandit	Комбінує лінтери, стилістику та аналіз безпеки, не скалдне налаштування.	Може спричиняти велику кількість ложнопозитивних результатів. Може бути менш ефективним для більших проєктів.

Враховуючи виявлені ознаки, можна сформувати розуміння про ситуації в яких доречним буде використання того чи іншого іструменту, ці дані занесено у таблицю 3.4.

Таблиця 3.4 – Використання інструментів

Інструмент	**Використання**
Pylint	Для великих та серйозних проєктів, де важлива якість коду. Там, де необхідно строго визначати стиль та стандарти. Велика громада та підтримка.
Муру	Для проєктів, які вимагають статичної типізації та хочуть підтримувати безпеку. Там, де важливо забезпечити стабільність та уникнути помилок типів.

Продовження таблиці 3.4

Інструмент	**Використання**
PyFlakes	Для невеликих проєктів, де не важлива строга типізація. Для швидкого виявлення базових проблем.
Pycodestyle	Для проєктів, які вимагають строгого виконання стилістичних правил. Для швидкого виявлення стилістичних недоліків
Flake8	Для проєктів, де важлива стиль коду та хочеться скористатися кількома інструментами в одному. Для швидкого лінтингу та аналізу коду
Prospector	Для проєктів, де потрібен комбінований аналіз. Для тих, хто хоче швидко отримати висновки з кількох джерел.
bandit	Для проєктів, де важлива безпека інформації та виявлення потенційних загроз. Для підвищення безпеки великих та критичних проєктів.

ВИСНОВКИ

У рамках кваліфікаційної роботи було досліджено і визначено загальні методи та методології, які використовуються для статичного аналізу коду. Статичний аналіз коду є важливою частиною розробки програмного забезпечення, спрямованою на виявлення помилок, покращення якості коду, забезпечення дотримання стилістичних правил та збільшення безпеки програм.

Типи інструментів для статичного аналізу коду можна класифікувати на кілька категорій: лінтери, інструменти статичної типізації, аналізатори стилю коду та інструменти безпеки. Кожна категорія має свої унікальні завдання та переваги, спрямовані на покращення різних аспектів розробки програмного забезпечення.

У цьому дослідженні було розглянуто кілька конкретних інструментів для статичного аналізу коду в мові програмування Python, зокрема Pylint, MyPy, PyFlakes, Pycodestyle, Flake8, Prospector та Bandit. Кожен з цих інструментів має свої власні переваги та недоліки, а також відмінності у функціональності.

Для більш детального порівняння була використана таблиця, де враховані основні характеристики кожного інструменту. Зазначено, що вибір конкретного інструменту повинен базуватися на конкретних потребах проєкту, вимогах до якості коду, стилістичних вимогах та участі в аспектах безпеки.

На підставі аналізу можна визначити, що кожен інструмент має свої особливості, які можуть бути корисними в різних сценаріях розробки [30-40].

Результати дослідження апробовано у вигляді статті у студентському науковому журналі «UNIVERSUM» [40].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Data Flow Analysis. URL: https://www.codingninjas.com/studio/library/data-flow-analysis (дата звернення 14.10.2023)

2. Python Control Flow Statements and Loops. URL: https://pynative.com/python-control-flow-statements/ (дата звернення 14.10.2023)

3. Static and Symbolic Analysis. URL: https://www.talkcrypto.org/blog/2019/03/15/static-and-symbolic-analysis/ (дата звернення 14.10.2023).

4. Infographic – Open source linters, tools for code analysis – 2021. URL: https://www.promyze.com/open-source-linters-2021/ (дата звернення 14.10.2023).

5. Python Type Checking. URL: https://testdriven.io/blog/python-type-checking/ (дата звернення 14.10.2023).

6. Generating and using a Callgraph, in Python. URL: https://cerfacs.fr/coop/pycallgraph (дата звернення 14.10.2023).

7. Akhtar, M. S., & Feng, T. (2022). Malware analysis and detection using machine learning algorithms. Symmetry, 14(11), 2304. https://doi.org/10.3390/sym14112304 (дата звернення 14.10.2023).

8. Delmas, D. (2022). Static analysis of program portability by abstract interpretation (Doctoral dissertation). Sorbonne Université..

9. Monat, R., Ouadjaout, A., Miné, A. (2021). A Multilanguage Static Analysis of Python Programs with Native C Extensions. In: Drăgoi, C., Mukherjee, S., Namjoshi, K. Static Analysis. SAS 2021. Lecture Notes in Computer Science(), vol 12913. Springer, Cham. https://doi.org/10.1007/978-3-030-88806-0_16.

10. Vassallo, C., Panichella, S., Palomba, F., et al. (2020). How developers engage with static analysis tools in different contexts. Empirical Software Engineering, 25, 1419–1457. https://doi.org/10.1007/s10664-019-09750-5

11. Kim, S., Yeom, S., Oh, H., Shin, D., & Shin, D. (2021). Automatic malicious code classification system through static analysis using machine learning. Symmetry, 13, 35. https://doi.org/10.3390/sym13010035

12. Pan, Y., Ge, X., Fang, C., & Fan, Y. (2020). A systematic literature review of Android malware detection using static analysis. IEEE Access, 8, 116363–116379. https://doi.org/10.1109/ACCESS.2020.3002842

13. Nachtigall, M., Schlichtig, M., & Bodden, E. (2022). A large-scale study of usability criteria addressed by static analysis tools. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022) (pp. 532–543). https://doi.org/10.1145/3533767.3534374

14. Blazy, S., Bühler, D., & Yakobowski, B. (2017). Structuring abstract interpreters through state and value abstractions. In 18th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2017) (pp. 112–130). LNCS of Proceedings of the International Conference on Verification Model Checking and Abstract Interpretation, vol. 10145.

15. Boespflug, E., Ene, C., Mounier, L., & Potet, M. L. (2020). Countermeasures optimization in multiple fault-injection context. In Fault Diagnosis and Tolerance in Cryptography, FDTC 2020.

16. Cadar, C., & Sen, K. (2013). Symbolic execution for software testing three decades later. Communications of the ACM, 56(82–90).

17. Fourtounis, G., Triantafyllou, L., & Smaragdakis, Y. (2020). Identifying Java calls in native code via binary scanning. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020) (pp. 388–400).

18. Lee, S., Lee, H., & Ryu, S. (2020). Broadening horizons of multilingual static analysis: Semantic summary extraction from C code for JNI program analysis. In 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 127–137).

19. Schiewe, M., Curtis, J., Bushong, V., & Cerny, T. (2022). Advancing static code analysis with language-agnostic component identification. IEEE Access, 10, 30743–30761. https://doi.org/10.1109/ACCESS.2022.3160485

20. Ghaleb, A. A. E. (2023). Static analysis approaches for finding vulnerabilities in smart contracts (T). University of British Columbia. Retrieved from https://open.library.ubc.ca/collections/ubctheses/24/items/1.0435192

21. Li, S., Jiang, L., Zhang, Q., Wang, Z., Tian, Z., & Guizani, M. (2023). A malicious mining code detection method based on multi-features fusion. IEEE Transactions on Network Science and Engineering, 10(5), 2731-2739. https://doi.org/10.1109/TNSE.2022.3155187

22. Clem, T., & Thomson, P. (2021). Static analysis at GitHub: An experience report. Queue, 19(4), 20. https://doi.org/10.1145/3487019.3487022

23. Negrini, L., Shabadi, G., & Urban, C. (2023). Static analysis of data transformations in Jupyter Notebooks. In Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP 2023) (pp. 8–13). https://doi.org/10.1145/3589250.3596145

24. Ferrara, P., Negrini, L., Arceri, V., & Cortesi, A. (2021). Static analysis for dummies: Experiencing LiSA. In Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP 2021) (pp. 1–6). https://doi.org/10.1145/3460946.3464316

25. Tvoroshenko I., and Gorokhovatskyi V. (2022) The Application of Hybrid Intelligence Systems for Dynamic Data Analysis, *International Journal of Engineering and Information Systems*, 6(2), pp. 40–48.

26. Skanda V. C., S. Srinivasa Prasad, G. R. Dheemanth, & N. S. Kumar. (2019). Assessment of quality of program based on static analysis. In 2019 IEEE Tenth International Conference on Technology for Education (T4E) (pp. 276-277). Goa, India. https://doi.org/10.1109/T4E.2019.00072

27. Mordahl, A. (2023). Automatic testing and benchmarking for configurable static analysis tools. In Proceedings of the 32nd ACM SIGSOFT International

Symposium on Software Testing and Analysis (ISSTA 2023) (pp. 1532–1536). Association for Computing Machinery. https://doi.org/10.1145/3597926.3605232

28. de Souza, R. L., Zaffalon Ferreira, F., & da Silva Botelho, S. (2020). A proposal for source code assessment through static analysis. In 2020 IEEE Frontiers in Education Conference (FIE) (pp. 1-5). Uppsala, Sweden. https://doi.org/10.1109/FIE44824.2020.9274050

29. Li, Z., & Feng, G. (2020). Inter-language static analysis for Android application security. In 2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE) (pp. 647-650). Dalian, China. https://doi.org/10.1109/ICISCAE51034.2020.9236807

30. Gershuni, E., Amit, N., Gurfinkel, A., Narodytska, N., Navas, J. A., Rinetzky, N., Ryzhyk, L., & Sagiv, M. (2019). Simple and precise static analysis of untrusted Linux kernel extensions. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019) (pp. 1069–1084). Association for Computing Machinery. https://doi.org/10.1145/3314221.3314590

31. Daradkeh Y.I., Gorokhovatskyi V., Tvoroshenko I., and Zeghid M. (2022) Tools for fast metric data search in structural methods for image classification, *IEEE Access*, 10, pp. 124738-124746.

32. Gorokhovatskyi V., Tvoroshenko I., Kobylin O., and Vlasenko N. (2023) Search for visual objects by request in the form of a cluster representation for the structural image description, *Advances in Electrical and Electronic Engineering*, 21(1), pp. 19-27.

33. Гороховатський В.О., Творошенко І.С., Чмутов Ю.В. (2022) Застосування систем ортогональних функцій для формування простору ознак у методах класифікації зображень, *Сучасні інформаційні системи*, 6(3), С. 5-12.

34. Гороховатський В., Передрій О., Творошенко І., Марков Т. (2023) Матриця відстаней для множини компонентів структурного опису як

інструмент для створення класифікатора зображень, *Сучасні інформаційні системи*, 7(1), С. 5-13.

35. Pomazan V., Tvoroshenko I., and Gorokhovatskyi V. (2023) Development of an application for recognizing emotions using convolutional neural networks, *International Journal of Academic Information Systems Research*, 7(7), pp. 25-36.

36. Pomazan V., Tvoroshenko I., and Gorokhovatskyi V. (2023) Handwritten character recognition models based on convolutional neural networks, *International Journal of Academic Engineering Research*, 7(9), pp. 64-72.

37. Tvoroshenko I., Gorokhovatskyi V., Kobylin O., and Tvoroshenko A. (2023) Application of deep learning methods for recognizing and classifying culinary dishes in images, *International Journal of Academic and Applied Research*, 7(9), pp. 57-70.

38. Gorokhovatskyi V., Tvoroshenko I. (2023) Identification of visual objects by the search request. *International scientific symposium «INTELLIGENT SOLUTIONS-S». Computational intelligence (results, problems and perspectives). Decision making theory: proceedings of the international symposium*, September 28, 2023, Kyiv-Uzhorod, Ukraine, pp. 25-27.

39. Yakovleva O., Kovač M., Ardasov V. & Yeremenko I. (2023). Study on adding functionality to the Zoom online conference system for monitoring the participant activities, *Public Administration and Regional Development*, 19(1), pp. 158-184.

40. Теребецький, М., & Кузьомін, О. (2023). Дослідження методів перевірки вразливості програмного коду, які базуються на статичному аналізі програм. UINVERSUM, 2, 100–105.