

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Інститут інформатики та радіоелектроніки,
Факультет комп'ютерних наук і технологій
(повне найменування інституту, назва факультету)

Кафедра програмних засобів
(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

магістр

(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ АНАЛІЗАТОРА КОДУ
RESEARCH AND SOFTWARE IMPLEMENTATION OF A CODE ANALYZER.

Виконав: студент(ка) 2 курсу, групи КНТ-120м
Спеціальності 121 Інженерія програмного
забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

Жуков Є.О.

(прізвище та ініціали)

Керівник Касьян М.М.

(прізвище та ініціали)

Рецензент Поляков М.О.

(прізвище та ініціали)

2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Інститут, факультет ПРЕ, ФКНТ
 Кафедра програмних засобів
 Ступінь вищої освіти магістр
 Спеціальність 121 Інженерія програмного забезпечення
 (код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
 (назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
С.О. Субботін
 “ ” 20 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

Жукова Євгенія Олексійовича

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) Дослідження та програмна реалізація аналізатора коду.
Research and SoftwareImplementation of a Code Analyzer.

керівник проекту (роботи) Касьян Микола Миколайович, к.т.н., доцент.
 (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від “05” листопада 2021 року № 435

2. Строк подання студентом проекту (роботи) 1 грудня 2021 року

3. Вихідні дані до проекту (роботи) рекомендована література, технічне
завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Дослідження методів аналізу кода. 3.
Проектування архітектури аналізатора. 4. Основні рішення щодо реалізації
компонентів аналізатора коду. 5. Тестування аналізатора коду. Додатки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-5 Основна частина	Касьян М.М., доцент		
Нормоконтролер	Белова А.В., асистент		

7. Дата видачі завдання «09» вересня 2021 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітки
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2 тиждень	Розділ 1
3	Розробка та удосконалення методів, моделей й алгоритмів вирішення задачі.	3-4 тижні	Розділ 2
4	Проектування архітектури аналізатора.	5 тиждень	Розділ 3
5	Розробка аналізатора.	6-7 тижні	Розділ 4
6	Тестування аналізатора коду.	8 тиждень	Розділ 5
7	Оформлення пояснювальної записки та документів до неї.	9-10 тижні	Додатки
8	Нормоконтроль та рецензування.	11-12 тижні	
9	Захист роботи.	13 тиждень	

Студент (ка) _____ Жуков Є.О.
(підпис) (прізвище та ініціали)

Керівник проєкту (роботи) _____ Касьян М.М.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи магістра: 85 с., 13 табл., 42 рис., 2 дод., 36 джерел.

АНАЛІЗАТОР КОДА, БАЗА ДАНИХ, ВРАЗЛИВОСТІ ПРОГРАМ, ПРОГРАМНИЙ КОД, СТАТИЧНИЙ АНАЛІЗ, ІНТЕРФЕЙС, CWE, PYTHON, UML.

Об'єкт дослідження – пошук вразливостей коду.

Предмет дослідження – статичний аналізатор коду мови C.

Мета роботи – провести дослідження методів пошуку вразливостей програмного забезпечення та розробити аналізатор коду мови C.

Матеріали, методи та технічні засоби: абстрактні синтаксичні дерева, сигнатурний аналіз, мова моделювання UML, мова програмування Python.

Результати. На основі аналізу предметної області та дослідження методів пошуку вразливостей програмного коду розроблено статичний аналізатор коду мови C, призначений для підвищення безпеки застосунків.

Висновки. Розроблений аналізатор дозволяє шукати вразливості коду.

Галузь використання – розробка програмного забезпечення.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 85 pages, 13 tables, 42 figures, 2 appendices, 36 sources.

CODE ANALYZER, DATABASE, PROGRAM VULNERABILITIES, SOFTWARE CODE, STATIC ANALYSIS, INTERFACE, CWE, PYTHON, UML.

The object of research is to search for code vulnerabilities.

The subject of research is a static code analyzer C language.

The purpose of the work is to conduct research on methods of searching for software vulnerabilities and to develop a language code analyzer C.

Materials, methods and technical means: abstract syntax trees, signature analysis, UML modeling language, Python programming language.

Results. Based on the analysis of the subject area and the study of methods for searching for software vulnerabilities, a static C code analyzer was developed to increase the security of applications.

Conclusions. The developed analyzer allows to look for code vulnerabilities.

Scope - software development.

ЗМІСТ

	С.
Перелік скорочень та умовних познач	8
Вступ	9
1 Аналіз предметної області	11
1.1 Помилки та вразливості програмного коду	11
1.2 Огляд сучасних аналізаторів коду	17
1.3 Порівняння аналізаторів коду з компіляторами	23
1.4 Висновки до розділу	25
2 Дослідження методів аналізу коду	26
2.1 Статичний та динамічний аналіз коду	26
2.2 Дослідження методів статичного аналізу коду на С	31
2.3 Висновки до розділу	41
3 Проектування аналізатора коду	43
3.1 Архітектура аналізатора коду	43
3.2 Каталоги інспекції коду	47
3.3 Моделювання аналізатора коду	48
3.4 Висновки до розділу	51
4 Основні рішення щодо реалізації компонентів аналізатора коду	52
4.1 Розробка додатку	52
4.2 Побудова АСД	54
4.3 Правила коду	56
4.4 Висновки до розділу	60
5 Тестування аналізатора коду	61
5.1 Інтерфейс аналізатора коду	61
5.2 Дослідження роботи аналізатора	63
5.3 Висновки до розділу	64

	7
Висновки.....	66
Перелік джерел посилання	68
Додаток А Програми тестування аналізатора	72
Додаток Б Слайди презентації.....	78

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

AST	—	abstract syntax tree;
ASG	—	abstract semantic graph;
CFG	—	control flow graph;
DFG	—	data flow graph;
SSA	—	social security administration;
АК	—	аналізатор коду;
АСД	—	абстрактне-синтаксичне дерево;
БД	—	база даних;
ІТ	—	інформаційні технології;
СА	—	статичний аналізатор;
ОС	—	операційна система;
ПЗ	—	програмне забезпечення;
ПВК	—	потенційні вразливості коду;
ПрО	—	предметна область.

ВСТУП

Останніми роками компанії, розробники програмного забезпечення, стурбовані загрозами безпеці, пов'язаними з вихідним кодом їхніх продуктів. Це пов'язано з тим, що клієнти вимагають високої безпеки коду і що програмне забезпечення використовуються в мережах, де можуть бути використані вразливості коду (наприклад, для вторгнення) [1]. З цими новими загрозами компанії-розробники повинні підвищити безпеку своїх продуктів, запобігаючи вразливості коду. Як наслідок, розробники тепер повинні надавати продукти із захищеним кодом та перевіряти безпеку застарілого коду. Це завдання, яке може уповільнити поточну розробку і дуже коштовне, якщо продукт має велику базу застарілого коду.

Для вирішення цього завдання необхідно виявити дефекти, що викликають уразливість вихідного коду. Можливі дві основні напіваавтоматичні альтернативи, а саме статичне та динамічне виявлення вразливостей [2]. Динамічне виявлення помилок виконується на пізніх стадіях процесу, оскільки для цього потрібно, щоб система була працюючою. У разі відмови системи можливо визначити причину несправності. Статичний аналіз не вимагає, щоб система була виконуваною. Натомість його можна запускати в неповних частинах системи та виділяти причину можливого збою. Це дає дві основні переваги: несправності можуть бути виявлені на ранній стадії розробки і, отже, набагато дешевше виправити порівняно з пізнім виявленням [3], та не потрібно визначати місце збою. З точки зору безпеки, система не може визначити, які з виявлених проблем є вразливими. Натомість це судження повинні робити розробники, які використовують результати роботи інструментів статичного аналізу коду. Тим не менш, кілька досліджень і статей показали, що статичний аналіз коду дійсно є корисними інструментами [4], [5].

Мета даної магістерської роботи – розробити аналізатор коду для пошуку вразливостей програмного забезпечення.

Реалізація поставленої мети передбачає вирішення таких завдань:

- провести огляд помилок та вразливостей програмного коду;
- дослідити сучасні аналізатори коду;
- провести порівняння аналізаторів коду з компіляторами;
- дослідити методи статичного аналізу коду;
- розробити структуру аналізатору коду;
- побудувати інспекції коду;
- розробити програмне забезпечення аналізатора;
- проаналізувати результати аналізу програмного коду.

Предметом дослідження дипломної кваліфікаційної роботи магістра є статичний аналізатор коду.

Практична цінність роботи полягає в тому, що на підставі досліджень розроблено програмне забезпечення для розробки безпеченого коду на мові С.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Помилки та вразливості програмного коду

У сучасному суспільстві ПЗ використовується в будь-яких приладах: у телефонах та комп'ютерах, у медичному обладнанні, у банках, у літаках та автомобілях, у дитячих іграшках та електростанціях.

Залежність від ПЗ з кожним днем підвищується, а самі програми стають складнішими і більшими. Було досліджено, що густина помилок також зростає з розміром програмного забезпечення [6]. Багато помилок можуть призводити до катастрофічних наслідків. Тому при розробці ПЗ пошуку та виправленню помилок приділяють особливу увагу.

У роботах [7] та [8] наведено короткий огляд причин появи помилок у програмах, серед яких можна виділити наступні.

Вплив кваліфікації команди розробників програми:

- поганий або недостатній рівень опрацювання архітектури програми: погано спроектована архітектура програми може призвести до внесення помилок у процесі розвитку програмної системи;

- зневажливе ставлення програмістів до результату своєї роботи: замість ретельного планування змін у програмі застосовується метод швидкого кодування з наступним виправлення помилок, знайдених шляхом тестування програми.

Приділяється мало уваги різним рівням тестування програми у процесі розробки:

- недостатня кількість та якість автоматичних тестів на компоненти (модульних тестів), що призводить до пізнього виявлення порушення контракту використання модуля та функції;

- недостатність або відсутність інтеграційних тестів, що призводить до

інтеграційних помилок у процесі зміни коду компонентів, у тому числі поділюваних компонентів операційної системи, які не враховують контекст їх використання;

– недостатній рівень тестування програми на стійкість під час обробки некоректних зовнішніх даних.

Вплив змін, що вносяться у середу виконання програми. Багато сучасних програми використовують бібліотеки, що входять до комплекту постачання ОС або використовувані разом із різними програмами. Зміна у коді бібліотек при нестачі інтеграційного тестування може призвести до неправильної роботи програми [4].

Все безліч причин появи помилок у програмах у результаті зводяться до однієї – недостатньої кваліфікації розробників програмного забезпечення. У зв'язку з цим стає очевидним необхідність створення автоматичних засобів виявлення помилок у програмне забезпечення, які дозволять виявляти помилки на різних етапах життєвого циклу ПЗ.

У літературі [9] пропонується трактування програмної помилки (дефекту), як деякої сутності, що призводить до однієї чи кількох помилок у такому артефакті як програмний код. У той же час проводиться аналіз досліджень на тему порівняння інспекцій коду та тестування, в яких виділяються дефекти на рівні вимог, на рівні специфікацій та коду ПЗ.

Загальноприйнятого визначення програмного дефекту виявити не вдалося. У різних роботах, присвячених виявленню дефектів у програмах, дається своє визначення з метою використання їх у рамках роботи [2], [3]. У 2010 році підрозділом Комп'ютерного співтовариства (IEEE Computer Society) Інституту інженерів електротехніки та електроніки (IEEE – Institute of Electrical and Electronics Engineers) випущено стандарт IEEE 1044-1999 «Стандартна класифікація для програмних аномалій» (IEEE Standard Classification for Software Anomalies) [10], термінів, які використовуються в рамках даної

класифікації:

– дефект (defect) – недосконалість чи недолік у працюючому продукті, коли цей працюючий продукт відповідає вимогам чи специфікаціям і потрібно або його виправлення, чи заміна;

– помилка (error) – дія людини, що призводить до некоректного результату;

– ушкодження (failure) – припинення можливості продукту виконувати необхідну функцію чи нездатність виконувати функцію у раніше зазначених обмеженнях;

– помилка (fault) – повідомлення про помилку у програмі;

– проблема (problem) – (А) складність чи неясність, випробувана одним чи декількома користувачами, що сталася у процесі використання системи. (В) Негативна ситуація, яку потрібно подолати.

Комп'ютерне програмування - це програмування ОС, прикладне програмування, вбудоване кодування системи, веб-розробка, програми для мобільних платформ, розвиток програм, розгорнутих в інтернеті, наукові обчислення. У таблиці 1.1 подано основні види помилок.

Таблиця 1.1 – Види помилок

Тип помилки	Опис
Логічна помилка	Коли написана програма будь-якою мовою і копілюється правильно, але видає неправильний висновок, недолік полягає в логіці основного програмування. Це помилка, яка була успадкована від нестачі базового алгоритму. Сама логіка, на якій базується вся програма, є неповною.
Синтаксична помилка	Кожна комп'ютерна мова, така як C, Java, Perl і Python має специфічний синтаксис, в якому буде написаний код. Коли програміст не дотримуватиметься "граматики" специфікаціями комп'ютерної мови, виникне помилка синтаксису. Такі помилки легко усуваються на етапі компіляції.

Продовження таблиці 1.1

Тип помилки	Опис
Помилка компіляції	Компіляція це процес, у якому програма, написана мовою високого рівня, перетворюється на машиночитаемую форму. Багато видів помилок можуть відбуватися цьому етапі, зокрема і синтаксичні помилки.
Помилка середовища виконання	Програмний код успішно скомпільований, і виконуваний файл було створено. Помилки під час виконання програми можуть виникнути внаслідок аварії чи нестачі ресурсів носія.
Арифметична помилка	Багато програм використовують числові змінні, і алгоритм може містити кілька математичних обчислень. Арифметичні помилки виникають, коли комп'ютер не може впоратися з проблемами, такими як "Поділ на нуль", або які ведуть до нескінченного результату. Це знову логічна помилка, яку можна виправити лише шляхом зміни алгоритму.
Помилка ресурсу	Помилка ресурсу виникає, коли значення змінної переполює максимально допустиме значення. Переполювання буфера, використання неініціалізованої змінної, порушення прав доступу та переполювання стека – приклади деяких поширених помилок.
Помилка взаємодії	Вони можуть виникнути через невідповідність ПЗ з апаратним інтерфейсом або інтерфейсом прикладного програмування.

Класифікація вразливостей захисту залежно від програмних помилок [11]:

– переполювання буфера. Ця вразливість виникає через відсутність контролю над виходом межі масиву у пам'яті під час виконання програми. Коли надто великий пакет даних переполює буфер обмеженого розміру, вміст сторонніх осередків пам'яті перезаписується, і відбувається збій та аварійний вихід із програми. За місцем розташування буфера в пам'яті процесу розрізняють переполювання буфера в стеку) та області статичних даних;

– вразливості можуть виникати у випадках, коли дані, що вводяться користувачем, без достатнього контролю передаються інтерпретатору деякої зовнішньої мови. У цьому випадку користувач може таким чином задати вхідні дані, що запущений інтерпретатор виконає зовсім не ту команду, яку передбачали автори вразливої програми [5];

– помилки форматних рядків. Даний тип уразливостей захисту є підкласом уразливості. Він виникає через недостатній контроль параметрів при використанні функцій форматного введення-виводу `printf`, `fprintf`, `scanf` і т. д. стандартної бібліотеки мови C.;

– вразливість як наслідок помилок синхронізації. Проблеми, пов'язані з многозадачністю, призводять до ситуацій, званих : програма, не розрахована виконання у багатозадачному середовищі, вважатимуться, що, наприклад, використовувані нею під час роботи файли неспроможна змінити інша програма. Як наслідок, злоумисник, який вчасно підміняє вміст цих робочих файлів, може нав'язати програмі виконання певних дій.

Звичайно, крім перерахованих, існують і інші класи вразливостей захисту. На рисунку 1.1 видно, що кількість виявлених вразливостей ПЗ з відкритим вихідним кодом у 2019 році перевищило 6000 помилок (рис. 1.1) [12].

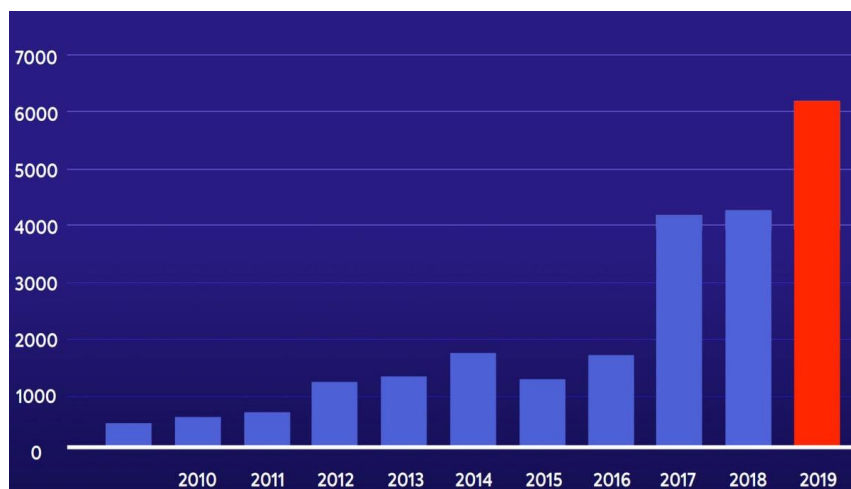


Рисунок 1.1 – Кількість виявлених вразливостей ПЗ

Аналіз знайдених помилок надходить із відкритих інструментів від Google, таких як OSS-Fuzz, які до 2018 року допомогли знайти 9000 недоліків за два роки. Станом на січень 2020 року він допоміг знайти 16 000 помилок у 250 проектах з відкритим кодом [13].

WhiteSource виявив, що 85% уразливостей з відкритим кодом розкриті і вже доступні виправлення. Однак у ньому зазначається, що деякі користувачі не знають про ці виправлення, оскільки лише 84% відомих помилок із відкритим кодом потрапляють до Національної бази даних уразливостей (NVD). У 2019 році WhiteSource переніс свою базу даних уразливостей на GitHub для підтримки служби сповіщень безпеки. GitHub сканує залежності проекту на наявність вразливостей у проектах, написаних на PHP, Java, Python, .NET, JavaScript та Ruby. Це допомогло розробникам знайти та виправити мільйони відомих недоліків коду.

WhiteSource також розглянув частку вразливостей, виявлених у популярних мовах програмування (рис. 1.2). Найбільша частка вразливого коду була написана на C з часткою 30%, що зменшилася з 47% десять років тому. Компанія зазначає, що високий відсоток C, ймовірно, пов'язаний з тим, що в ньому написано так багато коду. З іншого боку, код, написаний на PHP, був відповідальним за 27% помилок безпеки в порівнянні з 15% 10 років тому, незважаючи на те, що PHP став менш популярним серед розробників. Навпаки, код Python був відповідальним за лише 5% помилок, порівняно з 6% 10 років тому. Найпоширенішими типами недоліків безпеки у 2019 році були міжсайтові сценарії, неправильна перевірка введених даних, помилки буфера, читання за межами та розкриття інформації. Помилки міжсайтових скриптів були найпоширенішим типом уразливості для Java, JavaScript, PHP, Python і Ruby. Для мови C, небезпечної для пам'яті, це було «невірне обмеження операцій у межах буфера пам'яті» [7].

На рисунку 1.2 показані співвідношення кількості вразливостей з відкритим кодом для кожної мови за 2019 рік проти 2009-2018 років. Мова С все ще має більшість через великий обсяг написаного на ньому коду.

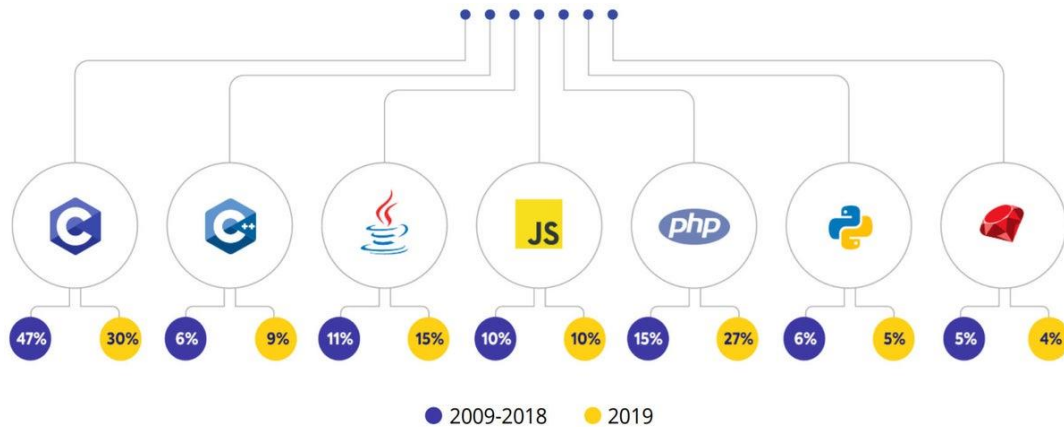


Рисунок 1.2 - Уразливості з відкритим кодом для кожної мови

1.2 Огляд сучасних аналізаторів коду

Для виявлення вразливостей захисту у програмах застосовують такі інструментальні засоби:

- динамічні інструменти, які дозволяють проводити налагодження програми у процесі її виконання;
- статичні аналізатори - інструменти, які використовують інформацію, накопичену під час статичного аналізу програми.

Статичні аналізатори вказують на ті місця в програмі, в яких може бути помилка. Ці підозрілі фрагменти коду можуть як містити помилку, так і виявитися абсолютно безпечними [7].

Наведемо огляд кількох статичних аналізаторів для різних мов. Розглянемо докладніше кожен із них.

CQual – інструмент аналізу для виявлення помилок у C-програмах. Програма розширює мову C додатковими специфікаторами типу, що

визначаються користувачем. Програміст коментує свою програму з відповідними специфікаторами і `squal` перевіряє помилки. Неправильні інструкції вказують на потенційні помилки. `Squal` може використовуватися, щоб виявити потенційну вразливість форматowanego рядка [14].

MOPS (MObel checking Programs for Security) – інструмент для пошуку вразливостей у захисті програм. Його призначення: динамічне коригування, що забезпечує відповідність програми на статичній моделі. MOPS використовує модель аудиту програмного забезпечення, яка має допомогти з'ясувати, чи відповідає програма набору правил, визначеному для створення безпечних програм [15].

ITS4, RATS, PScan, Flawfinder - для пошуку помилок переповнення буфера та помилок форматних рядків використовують такі статичні аналізатори [16]:

- ITS4. Простий інструмент, який статично переглядає вихідний C/C++-код виявлення потенційних вразливостей захисту. Він наголошує на викликах потенційно небезпечних функцій і виконує поверхневий семантичний аналіз;

- RATS. Утиліта RATS (Rough Auditing Tool for Security) обробляє код, написаний на C/C++, також може обробити ще й скрипти на Perl, PHP і Python. RATS переглядає вихідний текст, знаходячи потенційно небезпечні звернення до функцій. Мета цього інструменту - забезпечити обґрунтовані висновки, спираючись на які фахівець зможе вручну виконувати перевірку коду. RATS використовує поєднання перевірок надійності захисту від семантичних перевірок ITS4 до глибокого семантичного аналізу в пошуках дефектів, здатних призвести до переповнення буфера, отриманих з MOPS;

- PScan. Сканує вихідні тексти у пошуках потенційно некоректного використання функцій, аналогічних `printf`, та виявляє вразливі місця у рядках формату;

- Flawfinder. Як і RATS, це статичний сканер вихідних текстів програм,

написаних C/C++. Виконує пошук функцій, які найчастіше використовуються некоректно, надає їм коефіцієнти ризику (спираючись на таку інформацію, як параметри, що передаються) і складає список потенційно вразливих місць, впорядковуючи їх за ступенем ризику.

Всі ці інструменти схожі і використовують лише лексичний та найпростіший синтаксичний аналіз. Тому результати звітів можуть містити до 100% неправдивих повідомлень.

FlexeLint (PC-Lint) – цей аналізатор призначений для аналізу вихідного коду з метою виявлення помилок різного типу [17]. Програма проводить семантичний аналіз вихідного коду, аналіз потоків даних та управління.

Наприкінці аналізу видаються повідомлення кількох основних типів:

- можливий нульовий покажчик;
- проблеми із виділенням пам'яті (наприклад, немає `free()` після `malloc()`);
- проблемний потік управління (наприклад, недосяжний код);
- можливе переповнення буфера, арифметичне переповнення;
- попередження про поганий і потенційно небезпечний стиль коду.

Алгоритм аналізаторів у тому, що на вхід подають байт-код. А на виході одержують звіт із попередженнями. Щоб виявити потенційні проблеми, вихідний код перетворюється на внутрішнє уявлення аналізатора з подальшим застосуванням різних методологій для отримання всієї необхідної інформації.

Статичний аналізатор Java-коду, вбудований в IntelliJ IDEA, не поступається спеціалізованим інструментам статичного аналізу. Пошуком підозрілого, неоптимального чи помилкового коду займаються інспекції. Вони використовують різні сучасні методології статичного аналізу: аналіз потоку даних, зіставлення із шаблоном. IntelliJ IDEA налічує велику кількість інспекцій. Правду кажучи, багато з них повідомляють не про помилку, а скоріше, про неакуратність у коді або про можливість заміни на альтернативу [18].

При використанні IntelliJ IDEA безліч помилок можна виправляти за

секунди після того, як їх припустилися. Оскільки статичний аналіз проводиться завжди як редагування коду. Код із виявленими дефектами підсвічується у редакторі (рис. 1.3).

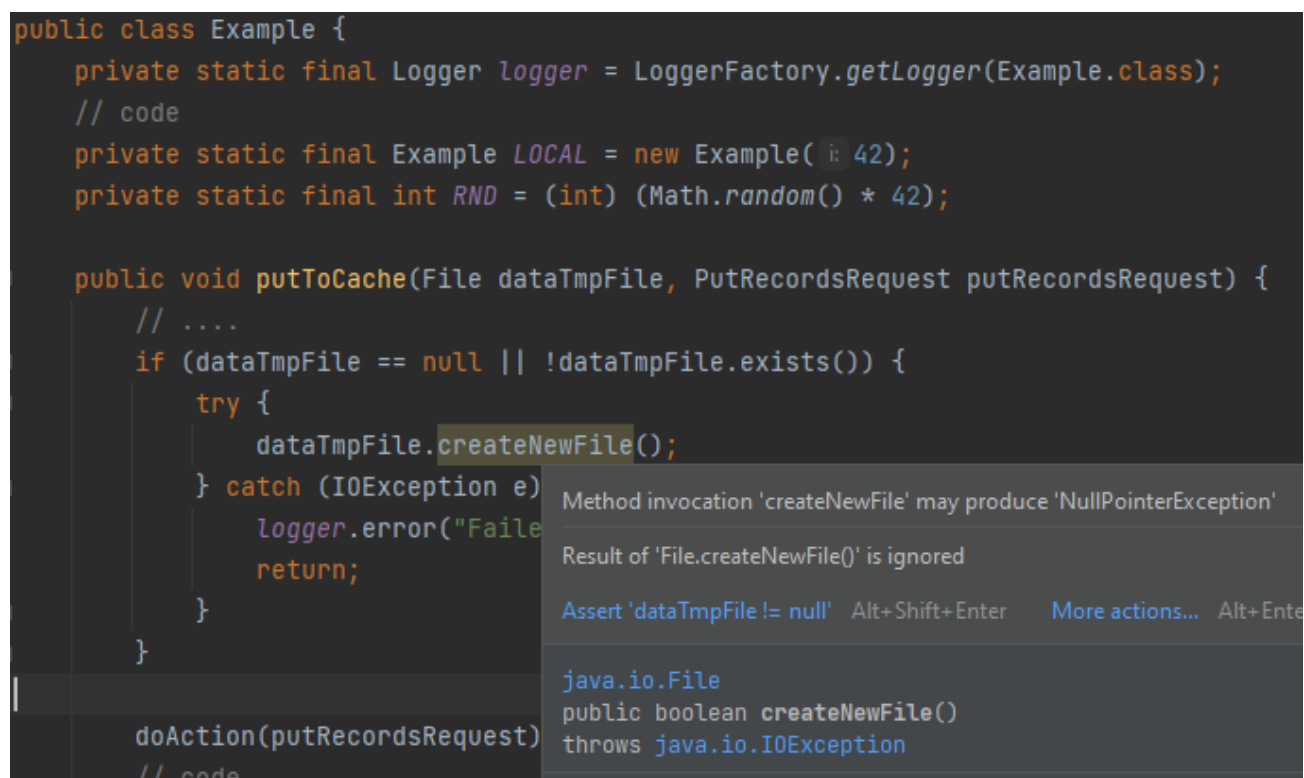


Рисунок 1.3 - Статичний аналізатор Java-коду, вбудований в IntelliJ IDEA

Також аналіз можна запустити вручну «Analyze > Inspect Code», або запустити окрему інспекцію за допомогою «Analyze > Run Inspection by Name», вказавши перед цим область дії аналізу. При цьому запуску стають доступними деякі інспекції, які через трудомісткість не працюють у режимі редагування. Після аналізу результати будуть згруповані за категоріями директорій в окремому вікні. З нього можна перейти до конкретного спрацювання інспекції.

SonarQube – статичний аналізатор коду для Java від компанії SonarSource, яка позиціонує його найкращим інструментом серед конкурентів. Для побудови AST моделі, заповненої семантичною інформацією, SonarQube на вхід приймає вихідний код файлу. Якщо код використовує зовнішні залежності (а вони

завичай завжди є), то для повної побудови моделі також необхідно передавати байт-код залежностей. Але за це не варто хвилюватися, якщо ви інтегрували аналізатор у збірну систему (наприклад, Gradle, Maven) [14].

Далі вже побудованої моделі застосовуються сучасні методології статичного аналізу (аналіз потоку даних, символічне виконання, зіставлення з шаблоном). Вони виявляють помилки коду та вразливості.

Особливості аналізатора:

- 150+ правил виявлення помилок;
- 350+ правил виявлення пошкоджень коду (Code Smells);
- 40+ правил виявлення потенційних уразливостей;
- інтеграція з Maven, Gradle, Ant, Eclipse, IntelliJ IDEA, VS Code;
- можливість розширення користувальницькими діагностичними правилами.

SAST-спеціалізований інструмент: більшість діагностичних правил зіставлено CWE, CERT, OWASP. Аналіз можна запустити як у межах різних IDE (через плагін SonarLint), і у рамках SonarQube.

У IntelliJ IDEA SonarLint, аналогічно до вбудованого аналізатора коду, виявляє помилковий код в режимі редагування коду з можливістю ручного запуску на всіх файлах (рис. 1.4).

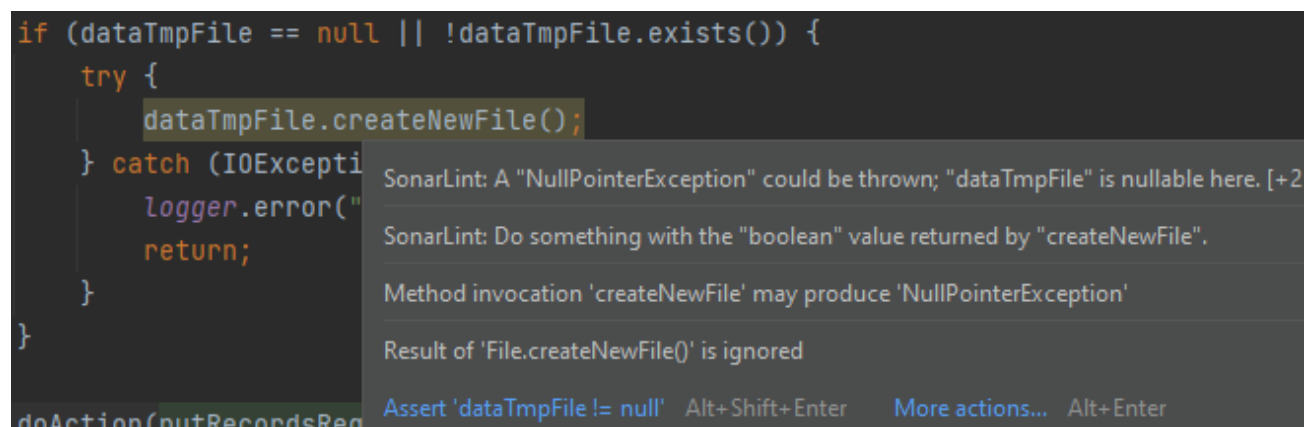


Рисунок 1.4 – Аналізатор IntelliJ IDEA SonarLint

SonarLint працює пліч-о-пліч із вбудованим аналізатором коду IntelliJ IDEA. Тому, якщо навести курсор на підсвічений фрагмент коду, то часто можна побачити попередження обох аналізаторів [18].

В основі PVS-Studio лежить open-source бібліотека Spoon. Вона приймає на вхід вихідний код та генерує добре спроектовану АСД-модель із семантичною інформацією [20]. На основі отриманої моделі застосовуються передові методології статичного аналізу:

- аналіз потоку даних (Data-Flow Analysis);
- символічне виконання (Symbolic Execution);
- анотування методів (Method Annotations);
- зіставлення із шаблоном (Pattern-based analysis).

На даний момент аналізатор зберігає більш ніж 105 діагностичними правилами, які дозволяють виявляти різні помилки в коді: друкарські помилки, розіменування нульового посилання, недосяжний код, вихід за межі масиву, порушення використання методу і т.п.

Особливості аналізатора:

- орієнтований на пошук реальних помилок;
- крім CLI версії аналізатора також є інтеграція до IntelliJ IDEA, Maven, Gradle, Jenkins, SonarQube;
- можливість запуску аналізу в інкрементальному режимі;
- дозволяє виявляти потенційні проблеми сумісності Java SE API при міграції проекту з Java 8 на новіші версії;
- PVS-Studio дозволяє конвертувати звіт у різні зручні для використання формати: json, xml, html, txt;
- SAST-спеціалізований інструмент: більшість діагностичних правил зіставлено CWE, CERT, OWASP.

Всі попередження аналізатора зводяться до спеціальної таблиці, в якій можна зручно з ними працювати (сортувати, фільтрувати, позначати як False Alarm) (рис.1.5).

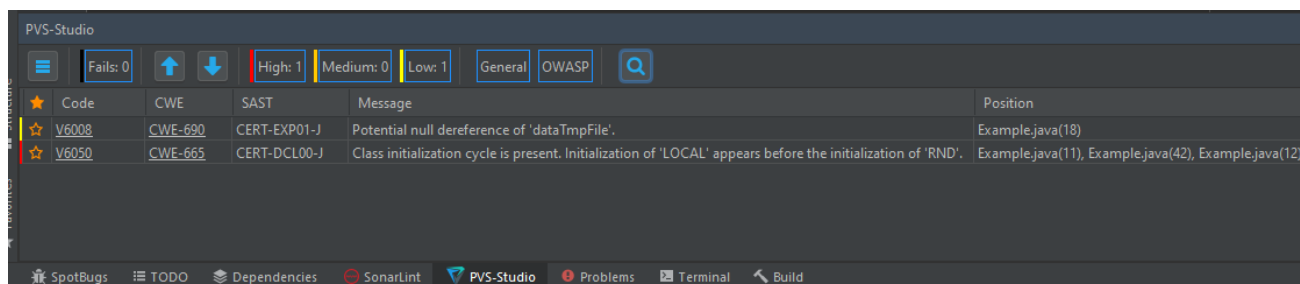


Рисунок 1.5 – Аналізатор PVS-Studio

Звичайно ж, крім розглянутих аналізаторів є й інші: як платні (Coverity, Klockwork, JArchitect і так далі), так і безкоштовні (Error Prone, Infer, Checkstyle і так далі). Всі вони орієнтовані на одне: запобігти попаданню помилкового або потенційно помилкового коду в готовий продукт.

Важливу роль у виборі статичного аналізатора грають:

- інтеграція у різні IDE;
- інтеграція у складальні системи;
- зручність запуску аналізатора на сервері;
- можливість виявлення помилок у режимі редагування коду;
- можливість зручної роботи із попередженнями;
- SAST-орієнтованість;
- відсоток хибних спрацьовувань;
- складність конфігурування.

1.3 Порівняння аналізаторів коду з компіляторами

У компілятора та сторонніх інструментів аналізу коду є спільне завдання

– виявлення небезпечних фрагментів коду. Проте існує суттєва різниця у тому, аналіз якого типу вони здійснюють.

Розглянемо аналіз проєкту з погляду компілятора та окремого спеціалізованого інструменту. Компілятор завжди тяжить до попереджень, які можна видати, обробивши тільки маленький локальний фрагмент коду. Така перевага є наслідком того, що на компілятор накладені дуже жорсткі вимоги продуктивності. Час компіляції середніх та великих проєктів є істотним фактором, що впливає на вибір методики розробки. Тому якщо існуватиме можливість додати 5% приросту продуктивності у компілятора, це буде здійснено [21].

Подібна оптимізація робить компілятор більш монолітним і насправді там навряд чи чітко виділяються такі етапи як препроцесування, побудова АСД або генерація коду.

Перейдемо до спеціалізованих інструментів статичного аналізу коду. Вони вже не мають таких жорстких обмежень швидкодії, оскільки частота їх запуску значно нижче, ніж у компілятора. Швидкість їхньої роботи цілком може бути в десятки разів повільніша, ніж компіляція коду.

Опишемо ще один наслідок «локального» аналізу компіляторів та більш глобального у спеціалізованих інструментах. При «локальному аналізі» складно уточнити, чи дійсно дана ситуація є небезпечною чи ні. В результаті виникає на порядок більше помилкових спрацьовувань [22].

На закінчення відзначимо основні моменти про засоби аналізу вихідного коду:

- аналіз коду не є основним завданням компілятора;
- використання різних компіляторів для аналізу важко, хоч і бажано;
- компілятори включають невеликий перелік поширених помилок;
- статичні аналізатори спеціалізуються лише на аналізі коду;
- статичні аналізатори мають велику основу діагностичних правил;

- ідеологія деяких діагностик не виключає хибні спрацьовування;
- можна використовувати різні аналізатори, незалежно від компілятора.

1.4 Висновки до розділу

Розмір і складність ПЗ постійно збільшуються. Розмір вихідного тексту сучасних програм та програмних систем може досягати сотень мільйонів рядків. Ускладнення призводить до того, що створення якісного ПЗ стає фактично неможливим без застосування автоматичних засобів перевірки програм на відповідність функціональним вимогам та відсутність програмних помилок.

В даному розділі перелічені найпоширеніші помилки та вразливості програмного кода. Наведена статистика їх появи. Поваден огляд сучасних аналізаторів коду. Виділені основні переваги розробки аналізаторів коду перед компіляторами.

2 ДОСЛІДЖЕННЯ МЕТОДІВ АНАЛІЗУ КОДУ

2.1 Статичний та динамічний аналіз коду

При виборі аналізатора коду важливо враховувати не швидкість, а якість сканування: його широту, повноту і кількість помилкових спрацьовувань. Кількість помилкових спрацьовувань має бути мінімальною, оскільки вони збільшують трудовитрати фахівців на верифікацію [17].

Для того, щоб знизити кількість помилкових спрацьовувань і знаходити вразливість нульового дня, якісний аналізатор коду повинен вирішувати такі завдання:

- обчислювати умови експлуатації уразливостей. Наприклад, у коді виявилася SQL-ін'єкція, а далі йде умова, що цей фрагмент коду має виконуватися лише 31 лютого. Очевидно, що таку вразливість не можна проексплуатувати, і експерту, який перевіряє результати аналізу, не варто витрачати час;

- формувати тестові експлойти, які дозволяють в режимі реального часу (найкраще - прямо в інтерфейсі аналізатора при розборі результатів) проексплуатувати вразливість та побачити, як реагує програма;

- підтверджувати знайдену вразливість за допомогою інших механізмів (функція автоперевірки). Підтвердження вразливості двома незалежними механізмами свідчить, що ніякого хибного спрацьовування немає, вразливість реальна;

- розраховувати всі можливі вектори атак зломисника і наслідки, до яких вони можуть призвести. Пошук за шаблонами (сигнатурний чи синтаксичний аналіз) з цим завданням не справляється. Останнім часом поширення набув механізму побудови діаграми потоків даних (data flow diagram, DFD), яка відображає послідовність перетворень даних,

контрольованих користувачем, від точки їх виникнення в програмі до точки виходу потенційно небезпечної операції.

Більшість методів аналізу коду, що використовується в аудиті безпеки ПЗ можна розділити на дві групи (рис. 2.1) - динамічні та статичні методи [23].

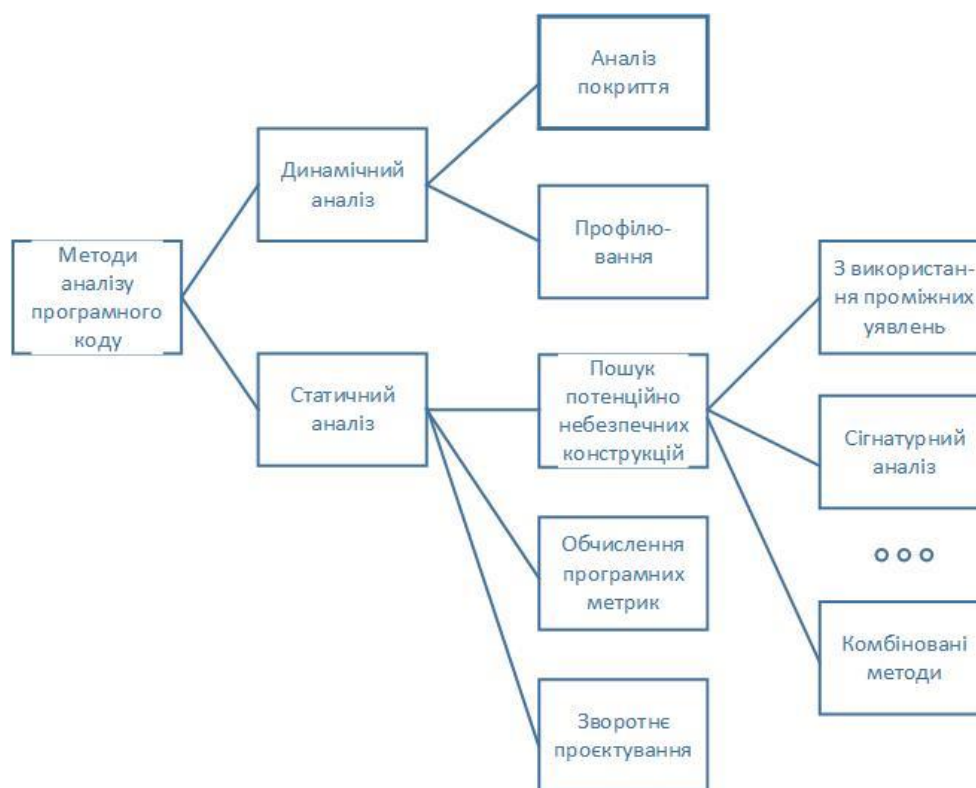


Рисунок 2.1 – Класифікація методів аналізу коду

Динамічний аналіз – це сукупність усіх методів аналізу ПЗ, реалізованих за допомогою програм на реальному чи віртуальному процесорі. Такі методи найбільш затребувані при дослідженні програм методом чорного ящика, коли є доступ лише до зовнішніх інтерфейсів програмного забезпечення без урахування їх структури, внутрішніх інтерфейсів та стану [5].

Такий підхід не завжди ефективний при пошуку помилок, пов'язаних із комбінаціями рідко використовуваних вхідних даних, а також при виявленні прихованих програмних закладок (програмних закладок), які там навмисно

введені.

Для своєї роботи статичний аналіз не вимагає реалізації програмного коду і дозволяє повністю або частково автоматизувати процес, але передбачає доступ не тільки до модулів навантаження та об'єктів, а й до вихідного коду програмної системи [9].

Інструменти динамічного аналізу виявляють програмні помилки в коді, запущеному на виконання. При цьому розробник має можливість спостерігати або діагностувати поведінку програми під час її виконання.

У багатьох випадках в інструментах динамічного аналізу проводиться модифікація вихідного або бінарного коду, щоб встановити пастки або процедури-перехоплювачі для проведення інструментальних вимірювань. За допомогою цих пасток можна виявити програмні помилки на етапі виконання, проаналізувати використання пам'яті, покриття коду та перевірити інші умови.

Переваги та недоліки динамічного аналізу наведені в таблиці 2.1.

Таблиця 2.1 – Переваги та недоліки динамічного аналізу

Переваги	Недоліки
Рідко виникають "хибні спрацьовування" - висока продуктивність знаходження помилок.	Відбувається втручання у поведінку системи у часі; ступінь втручання залежить від кількості використовуваних інструментальних вставок.
Захоплюються помилки як у реальних умовах, і у режимі імітаційного моделювання.	Повнота аналізу помилок залежить від рівня покриття коду.

Тому коли інструменти динамічного аналізу знаходять помилку, то це справжня помилка, яку програміст може швидко ідентифікувати та виправити.

Слід зазначити, що для створення помилкової ситуації на етапі виконання повинні існувати необхідні умови, при яких проявляється програмна помилка [5]. Відповідно, розробники повинні створити певний контрольний приклад для реалізації конкретного аналізу.

Інструменти статичного аналізу знаходять програмні помилки ще на ранній фазі створення проєкту, зазвичай перед тим, як створюється файл, що виконується. Таке раннє виявлення особливо корисне для проєктів великих вбудованих систем, де розробники не можуть використовуватися засоби динамічного аналізу доти, доки програмне забезпечення не буде завершено настільки, щоб його можна було запустити [20].

На етапі статичного аналізу виявляються і описуються області вихідного коду зі слабкими місцями, включаючи приховані вразливості, логічні помилки, дефекти реалізації, некоректності при виконанні паралельних операцій, граничні умови, що рідко виникають. Ці інструментальні засоби виконують також складний міжпроцедурний аналіз керуючих потоків і потоків даних та використовують удосконалену техніку відсікання хибних шляхів, оцінюють значення, які прийматимуть змінні та моделюють потенційну поведінку програми під час виконання.

Розробники можуть використовувати інструменти статичного аналізу у час на етапі розробки, навіть коли написані лише фрагменти проєкту. Однак, чим більш завершеним є код, тим краще. При статичному аналізі можуть проглядатися всі потенційні шляхи виконання коду – при звичайному тестуванні таке відбувається рідко, якщо в проєкті не потрібно забезпечення 100% покриття коду. Наприклад, при статичному аналізі можна виявити програмні помилки, пов'язані з крайовими умовами чи помилками шляхів, які не тестуються під час розробки [22].

Оскільки під час статичного аналізу робиться спроба передбачити поведінку програми, засновану на моделі вихідного коду, іноді виявляється

«помилка», якої фактично не існує - це так зване «хибне спрацьовування». Багато сучасних засобів статичного аналізу реалізують покращену техніку, щоб уникнути подібної проблеми та виконати виключно точний аналіз. Переваги та недоліки використання статичного аналізу коду наведені в таблиці 2.2.

Таблиця 2.2 – Переваги та недоліки статичного аналізу

Переваги	Недоліки
Використовується вже на ранніх етапах життєвого циклу програмного забезпечення. Можуть аналізуватись існуючі бази кодів, які вже пройшли тестування.	Можуть виправлятися програмні помилки та вразливості, які обов'язково призводять до відмови програми.
Низькі вартісні витрати.	Ненульова ймовірність "хибного спрацьовування".

Відповідно до вищевикладеного пропонується використовувати методи статичного аналізу для одночасного вирішення завдань виявлення дефектів випадкового коду та програмних помилок.

Для виконання різних форм аналізу в інструментах статичного аналізу мають справу з двома основними типами перевірок коду:

– перевірка абстрактного синтаксичного дерева – для перевірки базового синтаксису та структури коду;

– аналіз кодових шляхів — виконання більш повного аналізу, який залежить від розуміння стану програмних об'єктів даних у конкретній точці на шляху виконання коду.

Абстрактне синтаксичне дерево (АСД) є представленням вихідного коду у вигляді деревоподібної структури, як це може генеруватися на попередніх стадіях роботи компілятора. Дерево містить докладне однозначне розкладання

структури коду, що дозволяє інструментам виконати простий пошук аномальних місць синтаксису [24].

При аналізі кодових шляхів простежуються об'єкти всередині шляхів виконання, в результаті програми перевірки можуть визначити, наскільки чітко і коректно використовуються дані. Використання аналізу кодових шляхів розширює коло питань, куди можна отримати відповіді у процесі проведення статичного аналізу. Замість того щоб просто аналізувати правильність написання коду програми, при аналізі кодових шляхів робиться спроба визначити "намір" коду і перевірити, чи написаний код відповідно до цих намірів.

На даний час використання статичних аналізаторів стає об'єктивною потребою для забезпечення захисту сучасних програмних систем з високим рівнем довіри. Введення нових стандартів розробки безпечного коду підтверджує необхідність ефективних інструментів, які підтримують якість програмного коду.

Використання програмного аналізатора дозволяє істотно спростити та скоротити витрати на перегляд коду при розробці програми, а також збільшити якість коду та скоротити кількість помилок та вразливостей ще на етапі проектування.

2.2 Дослідження методів статичного аналізу коду на С

Все безліч причин появи помилок у програмах у результаті зводяться до однієї – недостатньої кваліфікації розробників ПЗ. У зв'язку з цим стає очевидним необхідність створення автоматичних засобів виявлення помилок, які дозволять виявляти помилки на різних етапах життєвого циклу ПЗ. Заявляється широкий набір підтримуваних правил аналізатора [25].

Аналізатор має підтримку як перевірок, заснованих на АСД, так і

перевірок на основі шляху виконання, з підтримкою міжпроцедурного аналізу. До недоліків аналізатора варто віднести пропрієтарність продукту, закритість його вихідного коду (недоступність вивчення алгоритмів роботи аналізатора), а також високу вартість.

На даний момент поширення отримав ряд комерційних розробок для статичного аналізу програм, розроблених з використанням мов C і C++.

Статичний аналізатор Coverity Save (раніше – Coverity Prevent) компанії Synopsys [26] є найбільш відомим комерційним статичним аналізатором і часто служить еталоном для оцінки інших аналізаторів (рис. 2.2).

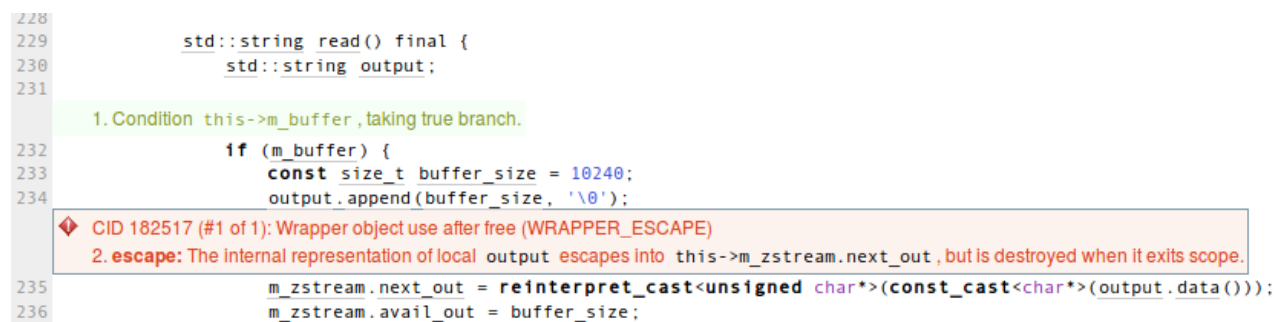


Рисунок 2.2 – Статичний аналізатор Coverity Save

Проте Coverity є найдорожчим інструментом статичного аналізу. Coverity Save позиціонується як система аналізу, яка здатна забезпечити виявлення великого набору можливих дефектів. До складу аналізатора входять засоби взаємодії з системами побудови проекту, безпосередньо статичний аналізатор, засоби перегляду та управління звітами аналізатора. Один з небагатьох аналізаторів, які мають можливість міжпроцедурного аналізу.

Виробником заявляється дуже висока точність аналізу - відсоток помилкових спрацьовувань від 5% до 15%. Однак його вихідний код повністю закритий, доступний лише програмний інтерфейс для розробки власних перевірок. Це виключає вивчення принципів його роботи та доопрацювання ядра аналізатора. Дані по хибним спрацьовуванням публікуються на основі

відкритих тестових наборів статичних аналізаторів, а не реального коду [26].

Зовсім недавно Klocwork оголосила про розробку нового інструменту статичного аналізу, який має на меті забезпечити якість та безпеку процесу розробки коду – Klockwork Insight [27]. Він застосовує складну методику аналізу на C, C++, Java і C# для автоматичного пошуку критичних програмних помилок і вразливостей безпеки у вихідному коді (рис. 2.3).

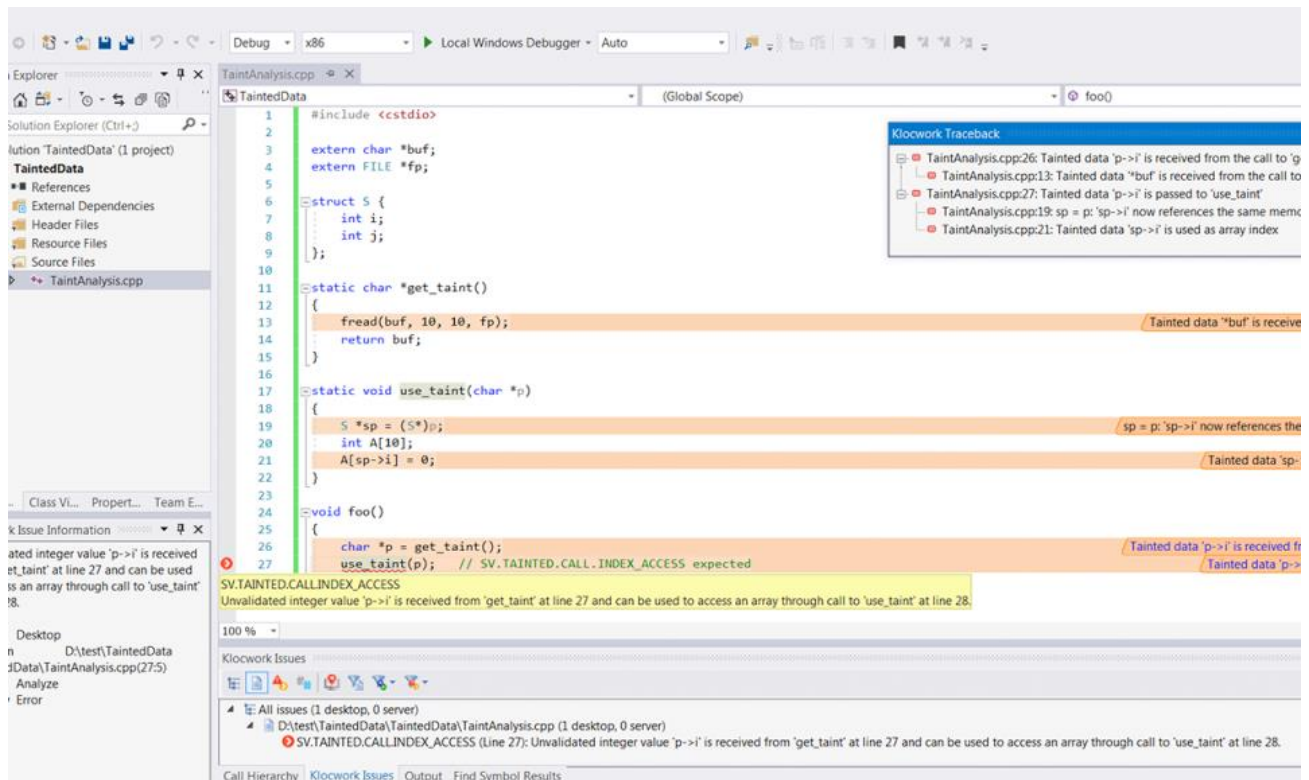


Рисунок 2.3 – Аналізатор коду Klockwork Insight

Застосовуючи міжпроцедурний потік керування, потік даних, поширення діапазону значень та оцінку символічної логіки, цей інструмент може знайти сотні помилок під час добре перевіреного коду.

Крім того, Insight розроблено, щоб вписатися в існуючий процес розробки та масштабований для великих організацій завдяки контролю доступу на основі ролей і розширеним можливостям аналізу, таким як паралелізація,

розподілений та інкрементальний аналіз. Klocwork Insight — це новаторський підхід до аналізу вихідного коду, який був доведений у деяких із найвимогливіших середовищ розробки програмного забезпечення у світі [26].

Компанія Hewlett-Packard випускає ще один досить відомий статичний аналізатор – HP Fortify Static Code Analyzer. Його основним призначенням є пошук дефектів, що стосуються безпеки [28].

Аналізатор комбінує дані від різних видів аналізу: аналіз потоків даних, семантичний аналіз, аналіз потоку управління, аналіз конфігурації та структурний аналіз. Недоліками аналізатора є його закритість і недоступність вихідного коду, а також висока вартість (рис. 2.4).

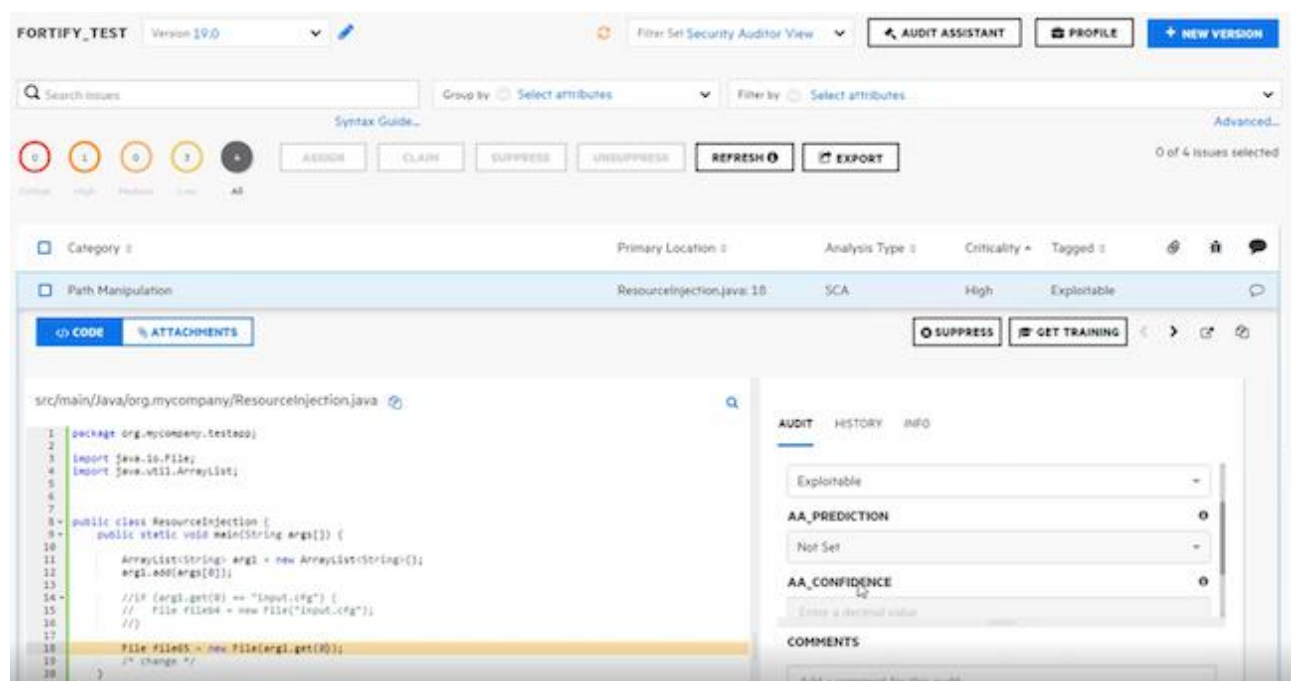


Рисунок 2.4 – Статичний аналізатор HP Fortify Static Code Analyzer

Загальними недоліками комерційних статичних аналізаторів є вкрай висока вартість, пропріетарність і закритість їх вихідних кодів, і, отже, недоступність вивчення алгоритмів роботи аналізатора. Тому, найчастіше, для глибокої модифікації та розробки перевірок має сенс вибирати вільні програми-

аналізатори, вихідний код яких є відкритим.

Ряд статичних аналізаторів розповсюджується вільно і доступні для використання. Насамперед, у числі необхідно відзначити статичні аналізатори CppCheck та різні види lint-аналізаторів.

Вільний (ліцензія GNU GPL) аналізатор CppCheck [29] досить широко використовується для перевірок різних проектів і доступний для використання в різних ОС і дистрибутивах, а також як доповнення до середовищ розробки (рис. 2.5).

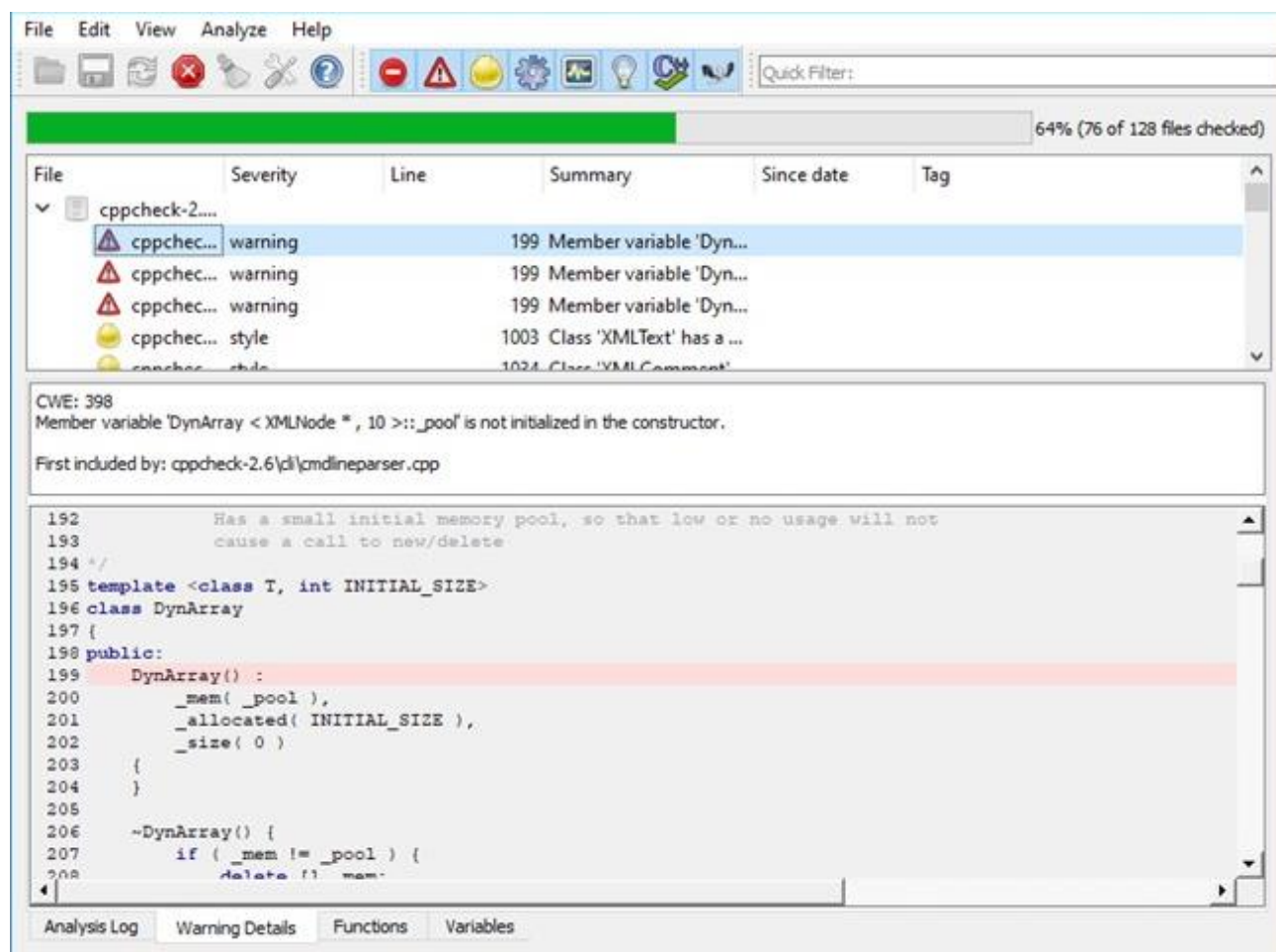


Рисунок 2.5 – Аналізатор CppCheck

Основна увага приділяється мінімізації кількості помилкових спрацьовувань. Аналізатор виконує перевірки на основі аналізу потоку

управління, проте складного аналізу потоку управління не проводиться, через що, як зауважують і самі розробники, багато дефектів не можуть бути виявлені.

Під вільними ліцензіями доступно багато lint-аналізаторів, зокрема, Splint [30]. Він може бути використаний для пошуку різних видів дефектів, включаючи переповнення буфера, вразливості форматного рядка, арифметичні обмеження та ін. Проте, значним недоліком даного інструменту є необхідність анотувати вихідний код вручну. Подібне обмеження різко знижує практичну цінність тих, що використовують дану методику інструментів, оскільки на анотування коду витрачається додатковий час розробника.

Серед вільних статичних аналізаторів з підтримкою мов C та C++ виділяється Clang Static Analyzer [31]. Цей статичний аналізатор підтримує різні види перевірок: як прості, засновані на аналізі синтаксичного дерева, так і складні - на основі аналізу шляхів виконання (рис. 2.6).

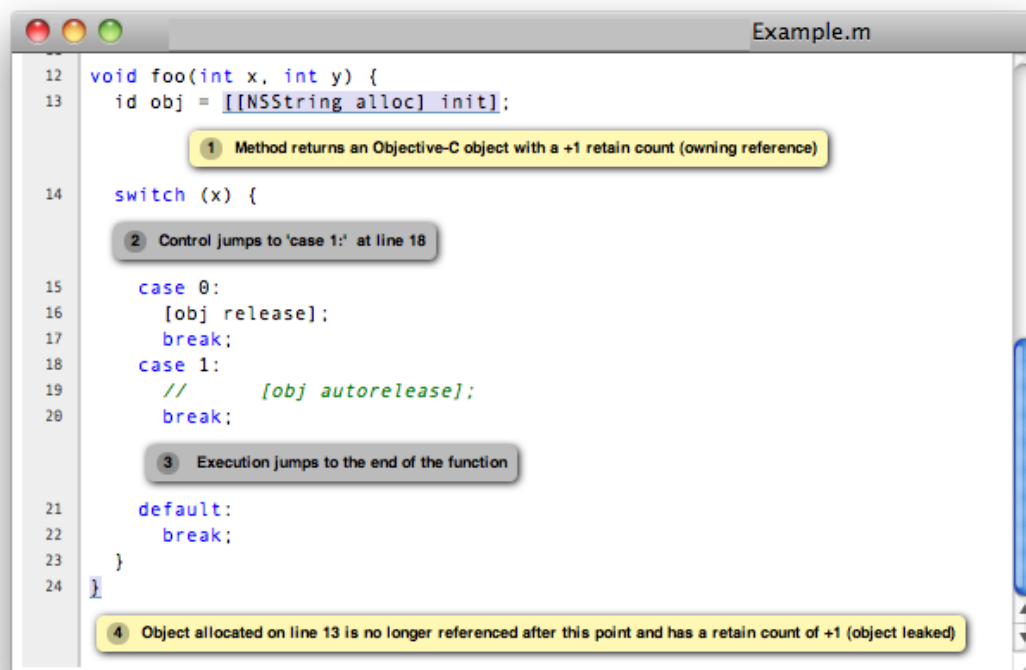


Рисунок 2.6 – Аналізатор Clang

Для аналізу шляхів виконання використовується символічне виконання, за допомогою якого проводиться обхід можливих шляхів виконання програми та моделювання операторів. Підтримується міжпроцедурний аналіз з використанням методу вбудовування. Аналізатор інтегрований для використання у середовищах програмування XCode та QtCreator. Будучи одним з найбільш функціональних з вільних статичних аналізаторів для C/C++, так і одним з найбільш широко використовуваних, Clang Static Analyzer використаний для реалізації розроблених у рамках даної роботи.

Статичний аналізатор здійснює пошук помилок у програмі без фактичного запуску. Після аналізу видається список попереджень про дефекти у вихідному коді програми. Попередження називають істинним, якщо він відповідає дефекту вихідного коду.

В результаті дослідження інструменти статичного аналізу коду можна класифікувати за такими ознаками:

- спосіб аналізу: автоматичний чи напівавтоматичний;
- мова програмування: одна, кілька, аналіз проміжного уявлення, аналіз виконуваного коду;
- типи програмних помилок, що виявляються: фіксований список, можливість створення додаткових детекторів помилок;
- модель коду програми: текстовий пошук, синтаксичний аналіз, аналіз дерева абстрактного синтаксису, аналіз шляхів виконання, міжпроцедурний аналіз.

У схемі роботи статичного аналізатора можна виділити три основні кроки.

- побудова проміжного уявлення (проміжне уявлення також називають внутрішнім уявленням чи моделлю коду);
- застосування алгоритмів статичного аналізу, внаслідок яких модель

коду доповнюється новою інформацією;

- застосовує правила пошуку уразливостей до доповненої моделі коду.

Сучасні методи статичного аналізу з метою унфікування алгоритмів використовують різні моделі подання коду:

- лексичний код;
- синтаксичне дерево;
- абстрактне синтаксичне дерево (AST);
- граф потоку керування (CFG);
- граф залежності за даними (DFG);
- модель одноразового статичного присвоєння (SSA);
- абстрактний семантичний граф (ASG).

AST, являє собою структуру даних, у якій кожен внутрішній вузол дерева відповідає оператору, яке дочірні вузли представляють операнди цього оператора. Фактично ця структура описує структуру вихідного коду програми. З використанням даної структури можлива побудова аналізаторів, які виконують обхід вузлів абстрактного синтаксичного дерева або таблиці символів [10] та направлені на перевірку найпростіших синтаксичних правил. При цьому аналіз абстрактного синтаксичного дерева є відносно швидким методом аналізу програм і дозволяє створювати аналізатори на спеціалізованих декларативних мовах [13].

За допомогою аналізу абстрактного синтаксичного дерева можлива побудова аналізаторів, перевіряючих не лише такі досить прості правила кодування, як перевірка правил іменування або відступів у вихідному коді програм, та помилок, що впливають на логіку виконання програми.

Граф потоку програми є структурою, в якій вузлами є базові блоки, а ребра вказують порядок прямування базові блоки. Базовий блок – це послідовність команд, що має одну точку входу та одну точку виходу. Аналіз

потоків програми дозволяє обчислювати властивості потоків даних програми з урахуванням потоків управління, що дозволяє виявляти найцікавіші помилки у програмі, пов'язані з послідовністю операцій над даними програми.

Однією з можливих реалізацій аналізу потоків програми є представлення потоків операцій програми у вигляді проміжного уявлення на основі графа потоків управління програми, в якому базові блоки складаються з операцій представлених у триадресному коді: операція представляється у вигляді четвірок $\langle op, arg1, arg2, res \rangle$, де op – операція, $arg1$ – перший аргумент операції, $arg2$ – другий аргумент операції, res – результат операції. При цьому у кожного базового блоку є адреса, яка може бути використана для здійснення передачі управління в операціях умовного та безумовного переходу [9].

У процесі обходу графа потоків управління програми здійснюється аналіз потоків даних програми, який полягає в тому, що для кожної операції обчислюються властивості аргументів та результату операцій залежно від семантики операції. Залежно від виду аналізу розрізняють потоково-чутливий аналіз, аналіз, чутливий до шляхів, контекстно-чутливий (міжпроцедурний) аналіз.

Аналізатор обсягу або аналізатор контексту створює структуру даних, яка представляє інформацію про область програми, розширюючи інформацію, доступну з AST або ASG. Елемент, що представляє область у цій структурі даних, може містити, серед іншого, метадані про саму область, вузол AST, пов'язаний із цією областю, доступні активи (змінні, функції тощо) у цій області.

Абстрактний семантичний граф (ASG) відрізняється від абстрактного синтаксичного дерева двома основними поняттями: ASG не обов'язково є деревами і ASG виражають більше, ніж синтаксичну інформацію; ASG несуть семантичну інформацію, виражену додатковими ребрами.

На рисунку 2.7 наведений приклад побудови AST та ASG.

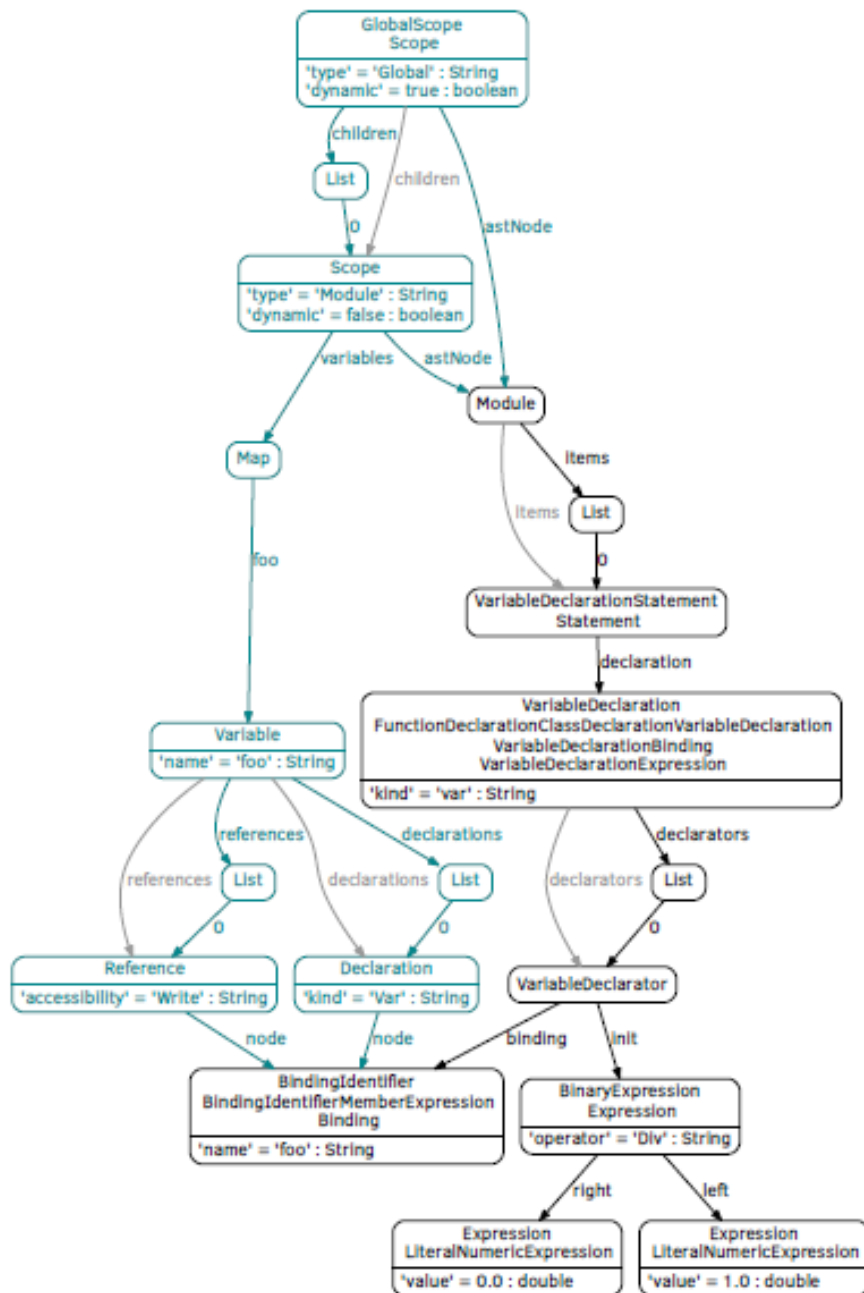


Рисунок 2.7 – Приклад побудови AST та ASG

Порівняльний аналіз різних методик та алгоритмів статичного аналізу показав, що сигнатурний аналіз може бути реалізований на різному рівні представлення коду, вибір якого може залежати від ефективності та особливостей використовуваного парсера, що відповідає конкретної мови

програмування.

До переваг статичного сигнатурного аналізу можна віднести відносну простоту реалізації аналізаторів та баз сигнатур, високу швидкість роботи (для рівня вихідних текстів та АСД), легкість модифікації сигнатур.

Переваги статичного аналізу:

- раннє виявлення помилок (під час розробки);
- код може не збиратися і не запускатися;
- повне покриття коду;
- не потрібно задавати тестові вхідні дані;
- найкраще перевіряє ділянки коду, складні для тестування (обробка граничних ситуацій).

Проте статичний аналіз перестав бути заміною тестування, оскільки перевіряє логіку програм. Для статичного аналізу є низка обмежень:

- відсутність знань про вхідні значення (невизначені значення змінних);
- експонентне зростання кількості шляхів у програмі;
- велика довжина окремих шляхів програми (цикли, рекурсивні виклики);
- складність аналізу циклів та рекурсивних викликів;
- великий обсяг даних, що описують стан програми (можливі значення).

Проведені дослідження методів статичного аналізу коду дозволяють сформулювати перелік завдань і вимоги, що пред'являються до системи, що проектується.

2.3 Висновки до розділу

При виборі аналізатора коду важливо враховувати не швидкість, а якість сканування: його широту, повноту і кількість помилкових спрацьовувань. На

даний час використання статичних аналізаторів стає об'єктивною потребою для забезпечення захисту сучасних програмних систем з високим рівнем довіри. Введення нових стандартів розробки безпечного коду підтверджує необхідність ефективних інструментів, які підтримують якість програмного коду.

У цьому розділі розглядається метод статичного аналізу. З його допомогою можна запобігти проблемам до їх проникнення до складу основного програмного коду та гарантувати, що новий код відповідає стандарту. Порівняльний аналіз різних методик та алгоритмів статичного аналізу показав, що сигнатурний аналіз може бути реалізований на різному рівні представлення коду, вибір якого може залежати від ефективності та особливостей використовуваного парсера, що відповідає конкретної мови програмування.

До переваг статичного сигнатурного аналізу можна віднести відносну простоту реалізації аналізаторів та баз сигнатур, високу швидкість роботи (для рівня вихідних текстів та АСД), легкість модифікації сигнатур.

Використовуючи різні техніки аналізу перевірку абстрактного синтаксичного дерева та аналіз кодових шляхів. Інструменти статичного аналізу можуть виявити приховані вразливості, логічні помилки, дефекти реалізації та інші проблеми. Це може відбуватися як на етапі розробки, так і під час компонування системи.

3 ПРОЄКТУВАННЯ АНАЛІЗАТОРА КОДУ

3.1 Архітектура аналізатора коду

Вихідний код програми — це послідовність інструкцій, сформульованих програмною мовою. Граматика формальних мов — це набір правил, що описують синтаксис компілятора. Об'єкт обробки вихідного коду (компілятор) надає значення (семантику) і перетворює інструкцію в байт-код [32].

На рисунку 3.1 показано загальний процес обробки вихідного коду та перетворення потоку символів у структуру даних зі значенням (семантична інформація).

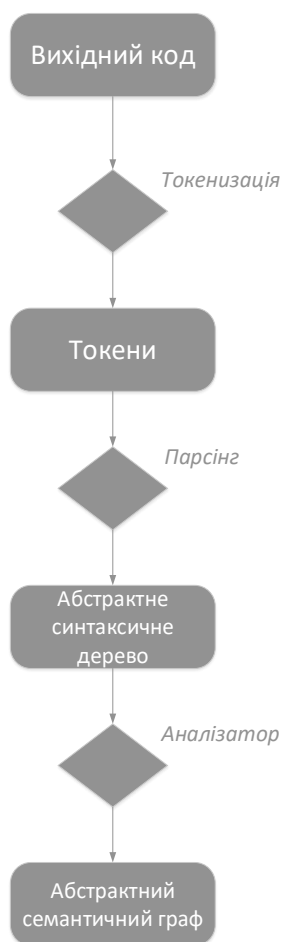


Рисунок 3.1 – Процес обробки вихідного коду

Основним недоліком відомих аналізаторів є не стільки обмежене використання, скільки неможливість врахування результатів власного розбору вихідної мови при пошуку для сигналів. Цей недолік не дозволяє створити відкриту масштабовану архітектуру аналізу з точки зору додавання нових мов і середовищ програмування, а також нових класів недоліків безпеки.

Подібно до компіляторів, лексичний і синтаксичний аналіз використовуються для побудови внутрішньої структури АСД. Лексичний аналіз розбиває текст програми на мінімальні значення елементів, на виході отримує потік лексем. Синтаксичний розбір перевіряє, що потік маркерів відповідає граматиці мови програмування. За допомогою розбору відбувається побудова структури, яка моделює вихідний текст програми. Далі застосовується семантичний аналіз.

Завдяки застосуванню вищевказаних алгоритмів проміжний вигляд доповнюється інформацією, необхідною для пошуку вразливостей (рис. 3.2). Правила пошуку вразливостей сформульовані в термінах моделі коду. Правила описують, які функції в остаточному проміжному поданні можуть свідчити про наявність уразливості. Таким чином, важливою частиною статичного аналізатора, окрім якості алгоритмів, є конфігурація та база правил: опис яких конструкцій у коді генерує прапори чи іншу інформацію, які конструкції перевіряють такі дані, для яких конструкцій критично важливі. використання таких даних.



Рисунок 3.2 – Проміжний вигляд структури аналізатора коду

АСД є базовим елементом, навколо якого будується аналізатор. При виконанні перевірки аналізатор переміщується по дереву і досліджує його окремі частини.

Крім АСД та семантичної моделі, аналізатору необхідний механізм аналізу потоку даних. Аналіз потоку даних дозволяє обчислювати допустимі значення змінних і виразів у кожній точці програми і завдяки цьому знаходити помилки. Допустимі значення створюються для змінних, полів класів,

параметрів методів та іншого при першій згадці.

Алгоритм роботи статичного аналізу коду на основі розглянутих методів і процедур представлений на рисунку 3.3.

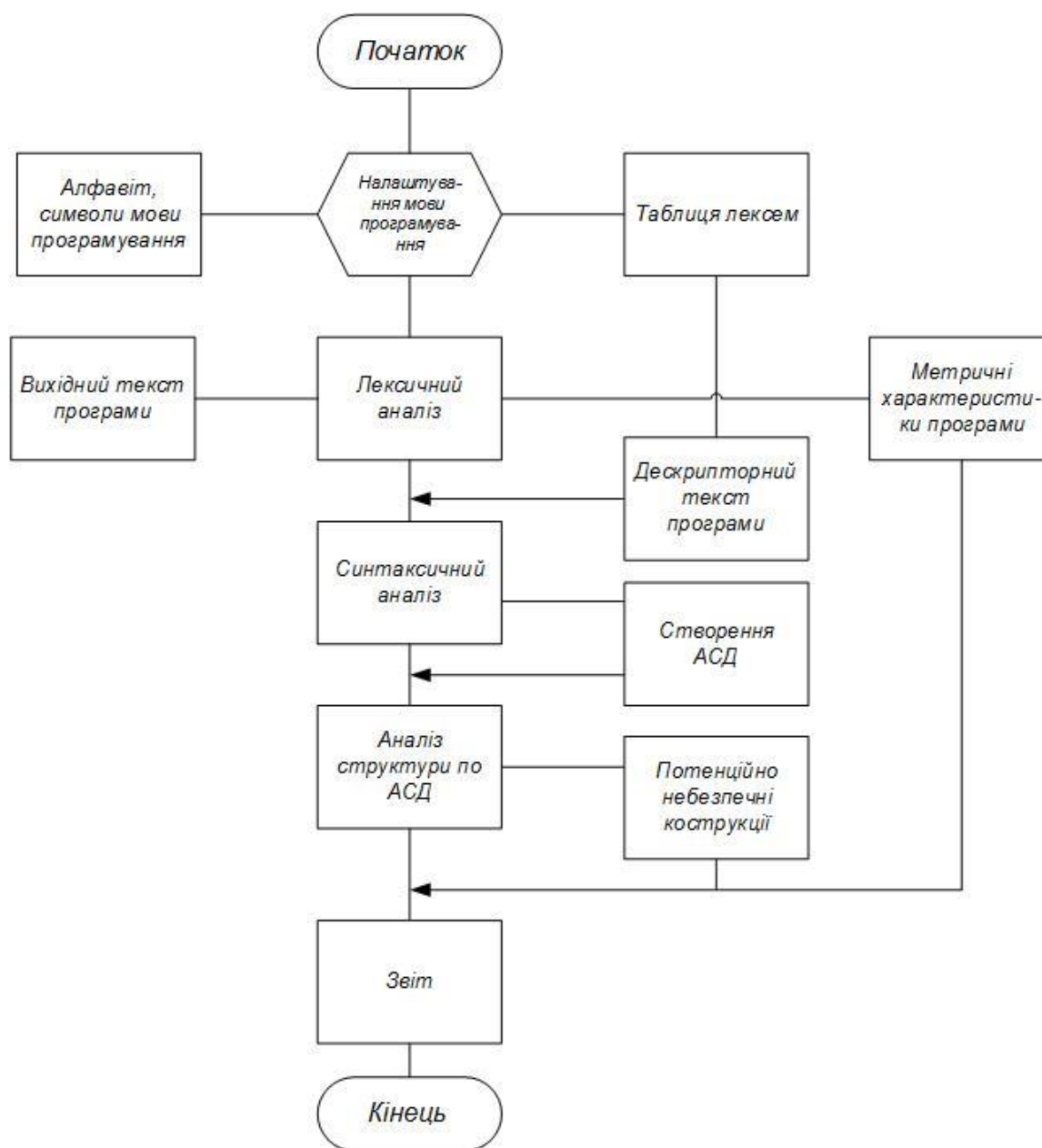


Рисунок 3.3 - Схема алгоритму аналізатору коду

Завдання лексичного аналізатора полягає в розпізнаванні лексем у коді програми, файлах конфігурації та бінарних файлах.

Завдання синтаксичного аналізатора заключається в тому, щоб на основі ланцюжка розпізнаних лексем сформувати АСД. Сигнатурний аналізатор здійснює наскрізний пошук у вихідних текстах програми патернов вразливостей, використовуючи базу сигнатур.

Модуль логічного висновку порівнює структурну інформацію коду та дані сигнатурного аналізу, на основі чого формує висновок про можливості присутності вразливостей у тому чи іншому ділянці коду.

3.2 Каталоги інспекції коду

Метою інструментального аналізу коду програми є визначення наявності у ньому помилок. У широкому значенні під помилкою (або дефектом) розуміється неправильність, похибка або спотворення об'єкта або процесу [28]. Аналізатори зазвичай оперують вужчим поняттям дефекту, розуміючи його як закодовану у програмі помилку чи недогляд людини [29]. Дане визначення має на увазі опис якості програми, яке необхідно аналізатору для пошуку дефекту. Для багатьох видів дефектів, що зустрічаються при розробці програм, існують описи та класифікація. Найбільш відомими серед них є каталоги Common Weaknesses Enumeration (CWE) [33], Computer Emergency Readiness Team (CERT) [34], Motor Industry Software Reliability Association (MISRA) [35].

CWE Top 25 – це список найнебезпечніших і найпоширеніших недоліків. Ці недоліки є небезпечними тому, що часто їх легко знайти і використовувати. Вони можуть дозволити зловмисникам завадити роботі програми, вкрати дані або навіть повністю захопити систему. CWE Top 25 – це значний ресурс спільноти, який може допомогти розробникам, тестувальникам, користувачам, а також керівникам проектів, дослідникам безпеки та викладачам отримати уявлення про найпоширеніші та найнебезпечніші наразі недоліки безпеки.

Нижче наведено короткий перелік недоліків у рейтингу CWE Top 25

2020 року, включаючи загальну оцінку кожного (табл. 3.1.).

Таблиця 3.1 – CWE Top 10 2020 року

Місце	ID	Вразливість	Оцінка
1	CWE-119	Неправильне обмеження операцій у межах буфера пам'яті	75.56
2	CWE-79	Неправильна нейтралізація введення під час генерації веб-сторінок («Міжсайтовий скрипт»)	45.56
3	CWE-20	Неправильна перевірка введених даних	43.61
4	CWE-200	Інформаційна експозиція	32.12
5	CWE-125	Поза межами видимості	26.53
6	CWE-89	Неправильна нейтралізація спеціальних елементів, що використовуються в команді SQL ("SQL Injection")	24.54
7	CWE-416	Використовуйте після звільнення пам'яті	17.94
8	CWE-190	Цілочисельне переповнення	17.35
9	CWE-352	Підробка міжсайтових запитів (CSRF)	15.54
10	CWE-22	Неправильне обмеження назви шляху до каталогу з обмеженим доступом	14.10

3.3 Моделювання аналізатора коду

Моделювання дозволяє вирішити чотири різні завдання:

– візуалізувати систему в її поточному або бажаному для замовника стані;

- описати структуру або поведінку системи;
- отримати шаблон, що дозволяє сконструювати систему;
- документувати прийняті рішення, використовуючи отримані моделі.

Перед додатком ставляться наступні завдання (рис. 3.4).

Аналіз вихідного програмного коду мовою C/C++ на наявність у ньому вразливостей, саме:

- виявлення небезпечних стандартних функцій;
- виявлення місць, де використовувалися ресурси після звільнення;

Реалізація власної частини програми, де користувач має такі можливості:

- додавання вихідного коду програми для аналізу самостійно;
- додавання файлу з вихідним кодом програми для аналізу через файлову систему;
- збереження результатів статичного аналізу у файл.

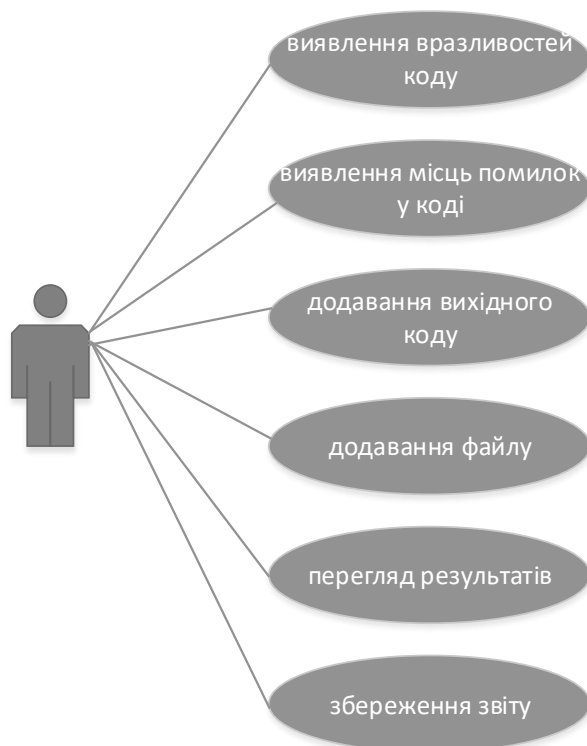


Рисунок 3.4 – Функції додатку

Лексер, токенізатор або сканер утворюють першу фазу процесу розбору. Він сканує вхідні дані потік символів і сегментує їх на послідовність груп, лексем, рядків із «значенням».

Він також класифікує ці групи на різні класи лексем, типи лексем. На рисунку 3.5 показано, як можна токенізувати рядок виразу з інформацією про тип маркера.

Token	Token type
foo	IDENTIFIER (Ident)
=	ASSIGN (Punctuator)
1	NUMBER (NumericLiteral)
/	DIV (Punctuator)
0	NUMBER (NumericLiteral)

Рисунок 3.5 - Потік символів і токенізації результат

Синтаксичний аналізатор формує другу фазу процесу розбору, використовуючи вихідний потік маркерів лексера. Він приймає вхідні дані та створює ієрархічну структуру даних (АСД), що представляє вхідні дані. Якщо вхідні дані не відповідають правилам синтаксису, і дерево не може бути побудовано, вихідний код синтаксично неправильний.

Далі представлення синтаксичного аналізатора порівнюється на основі синтаксису та семантики вихідної мови. Але база даних вразливостей не може імпортувати дані в такому вигляді, тому її потрібно трансформувати. На рисунку 3.6 представлено перетворення структури об'єкта мови.

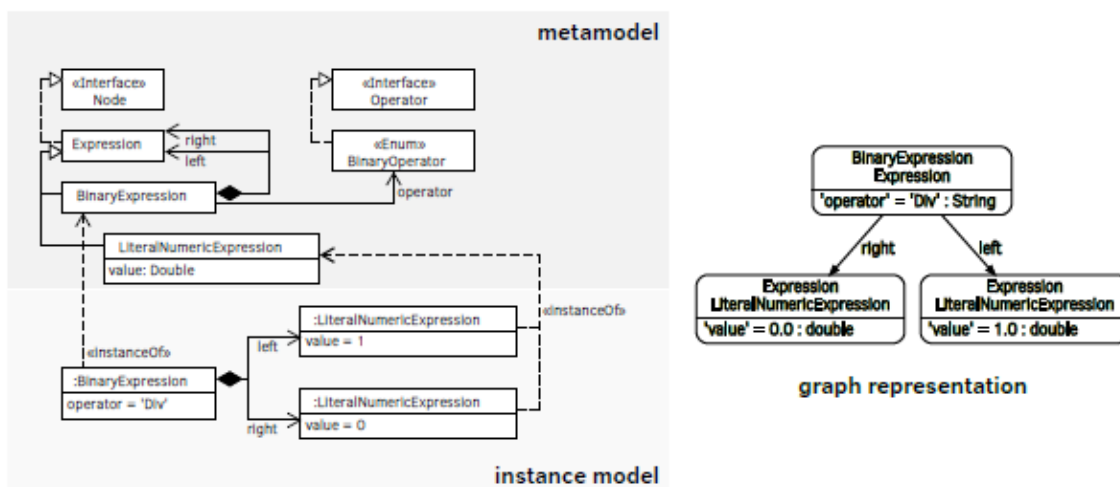


Рисунок 3.6 – Перетворення програмного коду

3.4 Висновки до розділу

Подібно до компіляторів, лексичний і синтаксичний аналіз використовуються для побудови внутрішньої структури абстрактного синтаксичного дерева. Лексичний аналіз розбиває текст програми на мінімальні значення елементів, на виході виходить потік лексем. Синтаксичний розбір перевіряє, що потік маркерів відповідає граматиці мови програмування, тобто отриманий потік маркерів правильний з точки зору мови. Далі застосовується семантичний аналіз. Завдяки застосуванню вищевказаних алгоритмів проміжний вигляд доповнюється інформацією, необхідною для пошуку вразливостей. Правила пошуку вразливостей сформульовані в термінах моделі коду.

Статичний аналізатор коду складається з наступних частин: база даних сигнатур вразливостей, база даних структурної інформації про код, програмний модуль лексичного аналізу, програмний модуль синтаксичного аналізу, програмний модуль сигнатурного аналізу, програмний модуль логічного виводу, програмний модуль побудови звітів.

4 ОСНОВНІ РІШЕННЯ ЩОДО РЕАЛІЗАЦІЇ КОМПОНЕНТІВ АНАЛІЗАТОРА КОДУ

4.1 Розробка додатку

Як мову написання програми був обраний Python, оскільки він дозволяє використовувати існуючі рішення найбільш ефективно і має великий набір додаткових бібліотек. Інтерфейс користувача реалізований за допомогою бібліотеки Tkinter, логіка програми побудована на аналізі вихідного коду за допомогою регулярних виразів.

Для зберігання даних про функції був використаний асоціативний контейнер dictionary (рис. 4.1). Ключом є небезпечна функція, а значення за цим ключем відповідає альтернативна їй безпечна функція.

```
methods_replacement_rule = {  
    'strcpy': 'strcpy_s',  
    'strcat': 'strcat_s',  
    'sprintf': 'sprintf_s',  
    'strlen': 'strlen_s',  
    'printf': 'printf_s',  
    'ctime': 'ctime_s',  
    'rewind': 'fseek'  
}
```

Рисунок 4.1 – Приклад структури даних

Пошук функцій з цього контейнера здійснюється за допомогою інструменту «регулярні вирази», який шукає по всьому вихідному коду програми небезпечні функції, а також запам'ятовує їх місцезнаходження, щоб наочніше вивести результати аналізу (рис. 4.2).

```

def unsafe_functions_warning(unsafe_method, file, positions):
    if len(positions) == 1:
        pos_str = "position "
    else:
        pos_str = "positions "
    warning = "Method " + unsafe_method + " in " + pos_str + ", ".join(map(str, positions)) + \
        " is unsafe. You should better use " + methods_replacement_rule[unsafe_method] + "\n"
    file.write(warning)

def find_unsafe_functions(data, result_file):
    for method in methods_replacement_rule:
        matches = [i.start() for i in re.finditer(method, data)]
        if len(matches):
            unsafe_functions_warning(method, result_file, matches)

```

Рисунок 4.2 - Функції, що забезпечують пошук небезпечних функцій мови C

Для пошуку місць, де відбувається звернення до ресурсів після їх звільнення, було створено регулярне вираження наступного виду (рис. 4.3), яке спочатку здійснює пошук оголошення покажчиків певних типів даних, а саме тих, що зберігаються в змінній `var_types`.

```

var_types = ['char', 'int', 'uint8_t', 'uint16_t', 'uint32_t']

def find_using_variables_after_deleting(data, result_file):
    r = re.search("((" + '|'.join(var_types) + ")*\s[a-zA-Z_][a-zA-Z0-9_]*", data)
    while r:
        variable = r.group(0).split(" ")[1:][0]
        cut_data = data.split(r.group(0), 1)[1]
        data = cut_data
        r = re.search("delete\s*" + variable, data)
        if r:
            cut_after_delete_data = cut_data.split(r.group(0), 1)[1]
            delete_variable = r.group(0).split(" ")[1:][0]
            if variable == delete_variable:
                r = re.search("((" + variable + "\s*=\s*)" + variable + "\s*[" + "[a-zA-Z_0-9_][a-zA-Z0-9_]*" + "\s*" +
                    "=\s*[" + "[a-zA-Z_0-9_][a-zA-Z0-9_]*" + "\s*",
                    cut_after_delete_data)
                if r:
                    deleted_pointer_warning(variable, r.group(0), result_file)
            r = re.search("((" + '|'.join(var_types) + ")*\s[a-zA-Z_][a-zA-Z0-9_]*", data)

```

Рисунок 4.3 – Функція пошуку звернення до ресурсів після їхнього звільнення

Для пошуку місць, де відбувається переповнення буфера, було створено регулярне вираження наступного виду (рис. 4.4), яке спочатку здійснює пошук оголошення буферів різної довжини, а також оголошення змінних та їх

поточного значення, а потім інформація про всі буфери та змінні заноситься в контейнер `array_name_and_range`, де ключу відповідає назва змінної або буфера, а значення - поточний розмір буфера або поточне значення змінної.

```
array_name_and_range = {}

def find_out_of_range_arrays(data, result_file):
    r = re.search("((" + '|'.join(var_types) + ")\\s[a-zA-Z_][a-zA-Z0-9_]*\\[ [0-9]*\\];|)" +
                  "(((" + '|'.join(var_types) + ")\\s[a-zA-Z_][a-zA-Z0-9_]*\\s*=\\s*[0-9]*)", data)

    while r:
        r_string = r.group(0)
        if r_string.find("(") != -1:
            var = r.group(0).split(" ")[1:][0]
            array_name = var.split("[")[0]
            array_range = (var.split("[")[1]).split("]")[0]
        else:
            var = r.group(0).split(" ")[1:][0]
            array_range = r.group(0).split("=")[1]
            array_name = var.split(" ")[0]

        array_name_and_range[array_name] = array_range
        cut_data = data.split(r.group(0), 1)[1]
        data = cut_data
        find_direct_overflow(array_name, array_range, data, result_file)
        r = re.search("((" + '|'.join(var_types) + ")\\s[a-zA-Z_][a-zA-Z0-9_]*\\[ [0-9]*\\];|)", data)
```

Рисунок 4.4 – Функція пошуку переповнення буфера

4.2 Побудова АСД

Статичний аналіз вихідних текстів ПЗ схожий з принципами роботи компіляторів. [36].

Сигнатурний аналіз виконує пошук дефектів в програмному коді шляхом зіставлення фрагментів коду зі зразками з бази даних шаблонів (сигнатур) дефектів безпеки. В [18] наведені приклади правил формування сигнатур помилок, що відповідають стандарту CWE.

Парсинг — процес перетворення вихідного коду в структурований вид. Типовий парсер являє собою комбінацію лексера і парсера (рис. 4.5). Лексер групує символи вихідного коду в значення послідовності, які називаються лексемами. Після цього визначається тип лексеми (ідентифікатор, число, строка і т.п.). Токеном називається сукупність значення лексеми і її типу.

Парсер з потоку токенів будує зв'язну древовидну структуру, яка

називається АСД, з якого видалені незначущі токени, такі як дужки, коми.

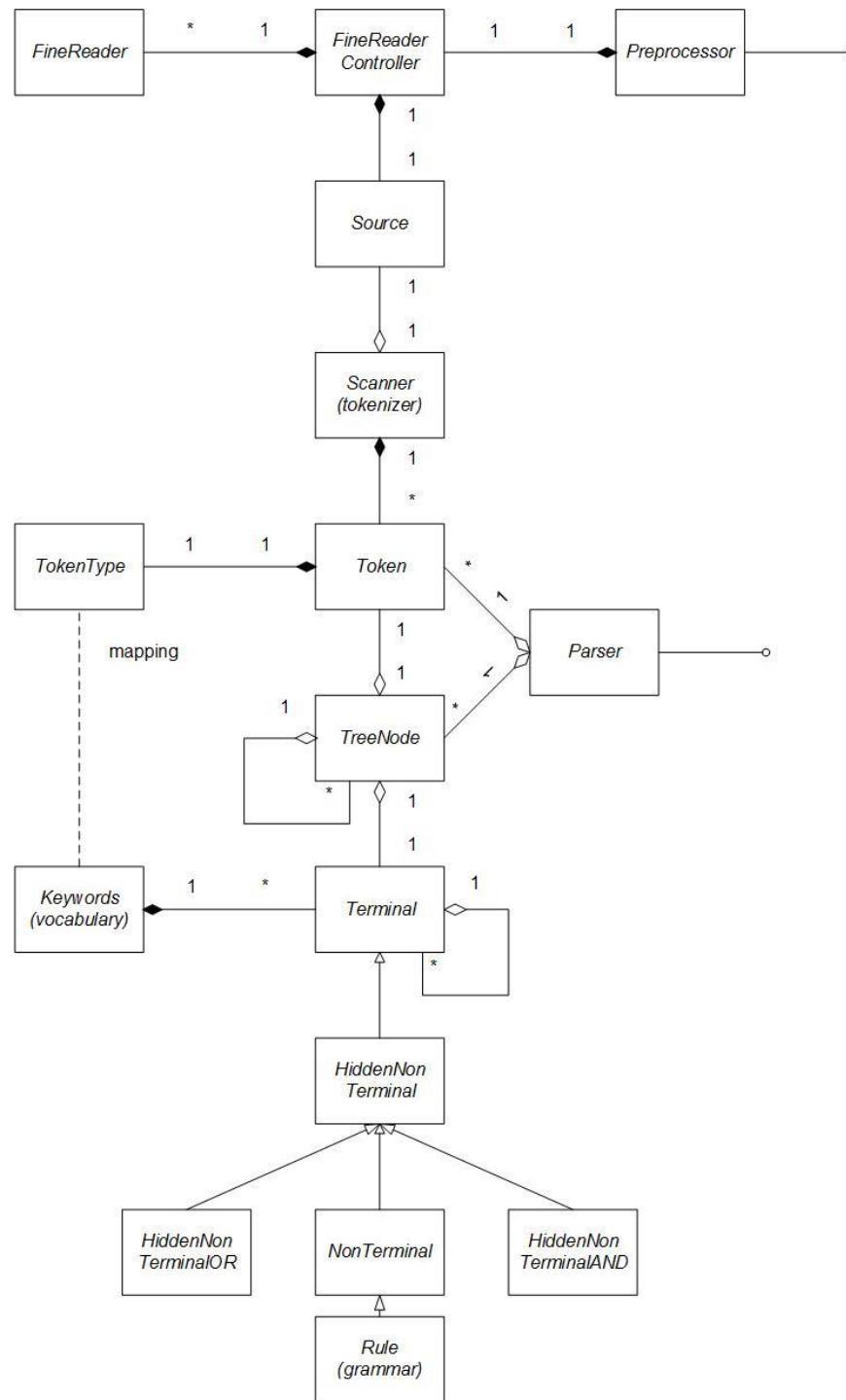


Рисунок 4.5 - Структура створення абстрактного синтаксичного дерева

4.3 Правила коду

Ми обрали 36 правил написання коректного коду C, які поділили на вісім категорій (табл. 4.1 – 4.8).

Таблиця 4.1 – Загальні правила

№	ID	Опис
1	E1	Використовування <code>const</code> замість <code>#define</code> для констант у глобальній та файловій області
2	E2	Використовування приведення в новому стилі замість приведення в стилі C
3	E3	Не розглядайте вказівник на <code>Derived[]</code> як вказівник на <code>Base[]</code>

Таблиця 4.2 – Використання `new` / `delete`

№	ID	Опис
1	E5	Використовування однакової форми для викликів <code>new</code> та <code>delete</code>
2	E6	Коли результат нового виразу в конструкторах зберігається в члені класу німого покажчика, переконайтеся, що для цього члена в деструкторі викликано <code>delete</code>
3	E9	Уникання приховування підпису за замовчуванням для оператора <code>new</code> та <code>operator delete</code>

Таблиця 4.3 – Конструктори/деструктори/визначення

№	ID	Опис
1	E11	Оголосити конструктор для кожного класу покажчика
2	E11	Оголосити оператор присвоєння для кожного класу покажчика
3	E12	Ініціалізація кожного члена даних класу за допомогою списку ініціалізації членів
4	E13	Список членів класу у списку ініціалізації членів в порядку, що відповідає порядку, в якому вони фактично ініціалізовані
5	E14	Зробіть деструктори віртуальними в базових класах
6	E15	Мати визначення <code>operator=</code> повернути посилання на <code>*this</code> .
7	E16	Призначити кожному локальному члену даних всередині <code>operator=</code>
8	E17	Викликати <code>operator=</code> . базового класу із похідного класу <code>operator=</code> .
9	E18	Використовувати список ініціалізації членів, щоб переконатися, що конструктор базового класу викликається з конструктора похідного класу
10	E19	Не викликайте віртуальні функції в конструкторах або деструкторах

Таблиця 4.4 – Правила розробки коду

№	ID	Опис
1	E20	Використовування функції, які не є членами, для бінарних операцій, таких як $+/-/*$
2	E21	Уникайте публічних даних
3	E22	Використовуйте метод <code>pass-by-ref-to-const</code> замість <code>pass-by-value</code>
4	E23	Нехай такі оператори, як $+/-/*$ повертають об'єкт, а не посилання
5	E24	Не перевантажуйте покажчик і <code>int</code>
6	E25	Зробіть <code>non-leaf</code> класи абстрактними
7	E26	Уникайте порожнього використання віртуального успадкування, тобто переконайтеся, що є принаймні два шляхи успадкування до кожного віртуального базового класу

Таблиця 4.5 – Правила реалізації коду

№	ID	Опис
1	E27	Не повертайте покажчики/посилання на внутрішні дані
2	E28	Ніколи не визначайте статичну змінну всередині вбудованої функції
3	E29	Не розглядайте вказівник на <code>Derived[]</code> як вказівник на <code>Base[]</code> .

Таблиця 4.6 – Правила спадкування

№	ID	Опис
1	E30	Не перевизначайте успадковану невіртуальну функцію
2	E31	Не перевизначайте успадковане значення параметра за замовчуванням

Таблиця 4.7 – Правила операторів

№	ID	Опис
1	E32	Уникайте використання визначених користувачем операторів перетворення
2	E33	Не перевантажуйте &&, , or ,.
3	E34	Переконайтеся, що оператори ++ і -- мають правильний тип повернення
4	E35	Використовуйте префікс ++ і --, якщо результат виразу збільшення або декременту не використовується

Таблиця 4.8 – Правила винятків

№	ID	Опис
1	E36	Переловлювати винятки за посиланням.

Кожне правило починається з номера правила, за яким слід посилання на опис правила C++.

Деякі правила можуть здатися суперечливими, особливо в світлі C++, що міститься в багатьох популярних бібліотеках класів. Наприклад, правило 15 широко порушується в базових класах Microsoft, в той час як майже жодна бібліотека не дотримується правила 19. Тому ми вважаємо, що важливо, щоб

програмісти мали можливість застосовувати ці обмеження, навіть якщо більшість програмістів вирішить цього не робити. Крім того, наше рішення включити правила, які зазвичай порушуються, допомагає нам оцінити ефективність можливостей фільтрації інструментів.

4.4 Висновки до розділу

Мовою Python був розроблений інструмент статичного аналізу, що дозволяє виявити місця використання небезпечних функцій, переповнення буфера, а також звернення до ресурсів після їх звільнення у вихідному коді програми мовами C. Загалом інструмент зберігає 36 правил написання коректного коду.

Функціональність інструменту можна розширити і додати пошук вразливостей, специфічніших для програмування вбудованих систем, оскільки таких статичних аналізаторів недостатньо.

5 ТЕСТУВАННЯ АНАЛІЗАТОРА КОДУ

5.1 Інтерфейс аналізатора коду

Вихідний код, написаний мовою C/C++, який необхідно проаналізувати можна додати у додаток двома способами: вручну, тобто ампостійно вставити його у ліве вікно програми або через файлову систему (рис. 5.1).

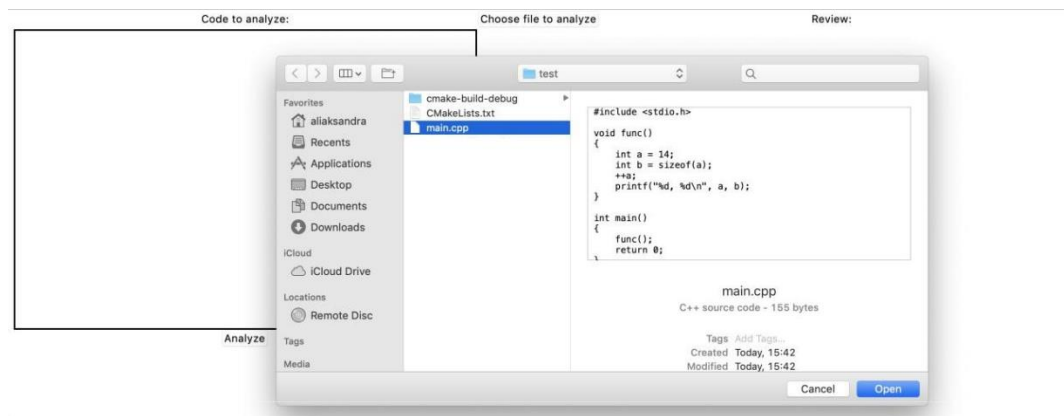


Рисунок 5.1 - Додавання вихідного коду

На рисунку 5.2 зображено результат роботи аналізатора з прикладу. У цьому випадку було виявлено небезпечні функції printf, strcpy, strcat.

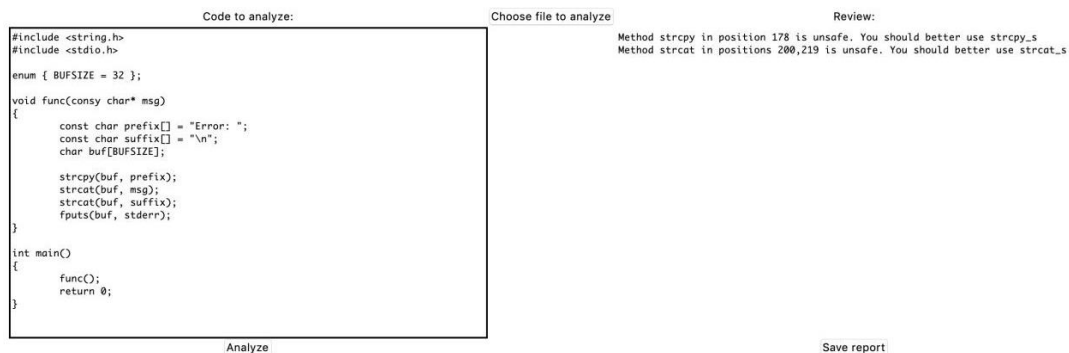


Рисунок 5.2 – Результат роботи аналізатора

На рисунку 5.3 зображено результат роботи аналізатора де відбувається звернення до ресурсів після їх звільнення. В даному випадку було виявлено некоректне звернення змінної s.

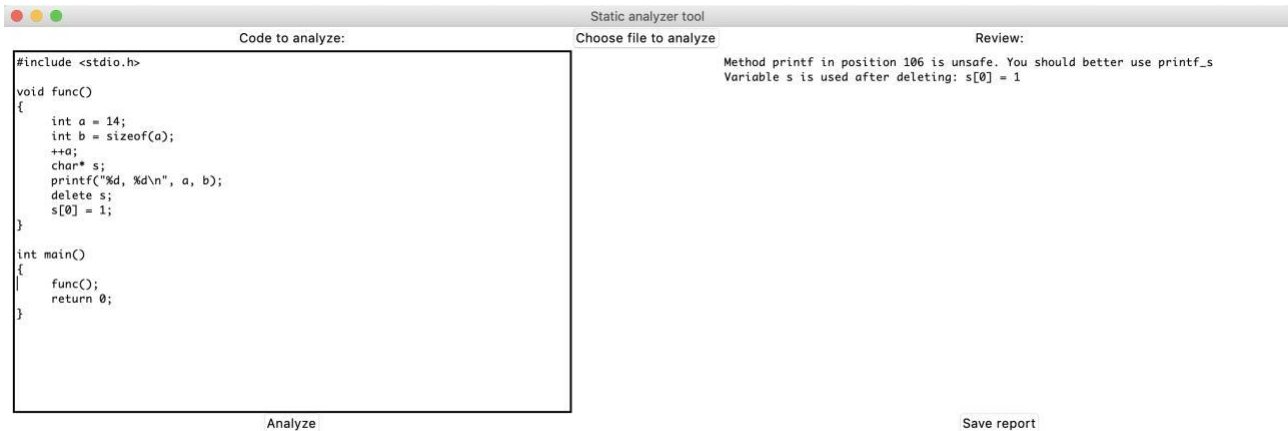


Рисунок 5.3 – Результат роботи аналізатора

На рисунку 5.4 зображено результат роботи аналізатора з прикладом переповнення буфера buf.

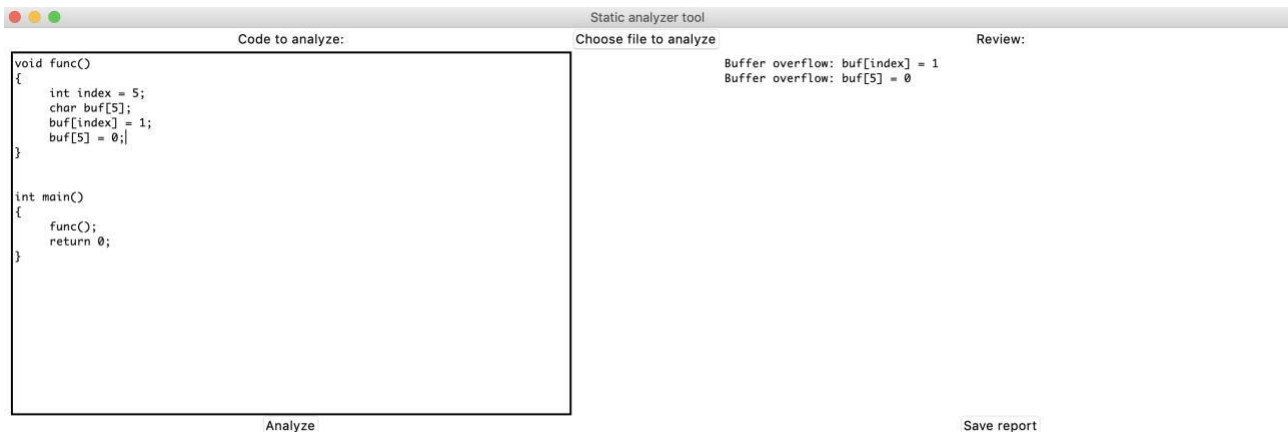


Рисунок 5.4 – Результат роботи аналізатора

Отримані результати, які аналізатор виводить на екран праворуч, також може зберегти в обраний файл (рис. 5.5).

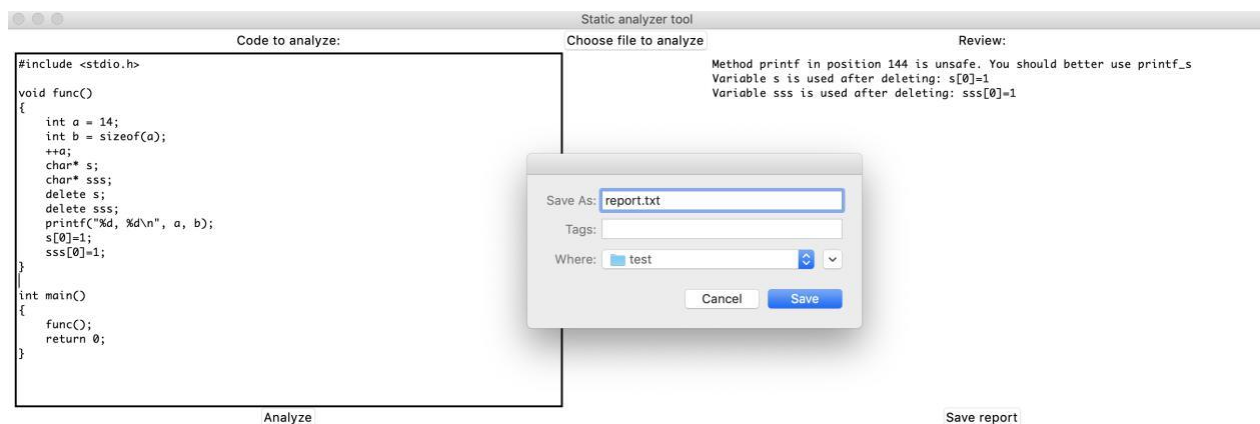


Рисунок 5.5 – Вибір файлу для збереження результату роботи аналізатора

5.2 Дослідження роботи аналізатора

Оцінюючи інструмент статичного аналізу вихідного коду програм нерідко використовуються три метрики:

- повнота аналізу, що показує відношення кількості знайдених помилок до повної кількості помилок у програмі;
- точність аналізу показує відношення кількості істинних попереджень про помилки до кількості хибних попереджень про помилки знайдених інструментом;
- продуктивність, це кількість обчислювальних ресурсів, потрібних для отримання результату.

Для перевірки аналізатора ми взяли тестові програми з відомими помилками у коді. Ці вихідні файли були невеликими, тому що мали довжину менше 10 рядків. Наша мета з цими файлами полягала не в тому, щоб забезпечити не навантажене тестування, а лише в тому, щоб побачити, чи може аналізатор виявити порушення правил у найпростіших випадках. Приклади тестових програм наведені у додатку А.1.

Серед попереджень діяла випадкова вибірка, яка розглядалася на предмет істинності чи хибності спрацювання. Істинні з точки зору аналізатора попередження, які можуть бути некритичними, також зараховані.

Результати запуску аналізатору наведені в таблиці 5.1.

Таблиця 5.1 – Результати аналізу

Помилка	Всього	Переглянутих	Істинних	Точність, %	Повнота, %
E1	17	17	17	100	100
E5	14	14	14	100	100
E6	4	4	4	100	100
E10	55	48	44	91	80
E11	21	21	21	100	100
E14	19	17	12	71	63
E20	4	4	2	50	50
E24	54	27	20	74	37
E25	144	61	47	77	32
E32	12	6	6	100	50

Можна бачити, що середній показник істинних спрацювань для критичних помилок, тобто точність роботи аналізатора дорівнює 87%. Це дає змогу зробити висновок, що аналізатор працює правильно.

5.3 Висновки до розділу

Вихідний код, написаний мовою C/C++, який необхідно проаналізувати можна додати двома способами: самостійно вставити його у ліве вікно програми або через файлову систему. Аналіз доданого коду проводиться автоматично після натискання кнопки «Аналіз». Результати перевірки коду

відображаються в правій частині вікна програми. Результати перевірки можна зберегти окремим файлом. Тестування аналізатору проходило на основі розроблених текстових програм для кожного правила. Можна бачити, що точність роботи аналізатора дорівнює 87%. Це дає змогу зробити висновок, що аналізатор працює правильно.

ВИСНОВКИ

Розмір і складність ПЗ постійно збільшуються. Ускладнення коду призводить до того, що створення якісного програмного забезпечення стає фактично неможливим без застосування автоматичних засобів перевірки програм на вразливості безпеки та відсутність програмних помилок.

В даній роботі проведений огляд найпоширеніших помилок та вразливостей програмного кода. Наведена статистика їх появи. Поведен аналіз сучасних аналізаторів коду. Виділені основні переваги розробки аналізаторів коду перед компіляторами. Досліджені методи аналізу програмного коду.

При виборі аналізатора коду важливо враховувати не швидкість, а якість сканування: його широту, повноту і кількість помилкових спрацьовувань. Метод статичного аналізу може знайти вразливості коду до їх проникнення до складу основного програмного коду та гарантувати, що новий код відповідає стандарту. Використовуючи різні техніки аналізу перевірку абстрактного синтаксичного дерева та аналіз кодових шляхів, інструменти статичного аналізу можуть виявити приховані вразливості, логічні помилки, дефекти реалізації та інші проблеми.

Статичний аналізатор коду складається з наступних частин: база даних сигнатур вразливостей, база даних структурної інформації про код, програмний модуль лексичного аналізу, програмний модуль синтаксичного аналізу, програмний модуль сигнатурного аналізу, програмний модуль логічного виводу, програмний модуль побудови звітів.

Мовою Python був розроблений інструмент статичного аналізу, що дозволяє виявити місця використання небезпечних функцій, переповнення буфера, а також звернення до ресурсів після їх звільнення у вихідному коді програми мовами C та C++. Загалом інструмент зберігає 32 правила написання

коректного коду. Також реалізована можливість зручного додавання файлу з вихідним кодом та можливість збереження результатів роботи аналізатора у файл. Функціональність інструменту можна розширити і додати пошук вразливостей, специфічніших для програмування вбудованих систем.

Тестування аналізатору проходило на основі розроблених текстових програм для кожного правила. Можна бачити, що відсоток істинних спрацьовувань для критичних помилок знаходиться між 50% і 80%. Точність аналізатора складає 87 %. Це дає змогу зробити висновок, що аналізатор працює правильно.

В даній валіфікаційній роботі можна зробити висновок, що поставлені задачі були вирішені.

Практична цінність роботи полягає в тому, що на підставі аналізу предметної області та дослідження методів пошуку вразливостей коду розроблено статичний аналізатор мови C. Аналізатор коду дозволяє покращити розробляємий код та ідентифікувати помилки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Hunter, M.T. Security issues with the IP multimedia subsystem (IMS) / M.T. Hunter, R.J. Clark, F.S. Park // Proceedings of the 2017 Workshop on Middleware for next-generation converged networks and applications, Port Beach, California: ACM. – 2017. Vol. 2. – P. 1-6.
2. Рашитов, Н. р. С. л. Анализ вредоносного кода с помощью интерактивных инструментов дизассемблера и составление сравнительного анализа / Н. р. С. л. Рашитов // Студенческий вестник. – 2021. – № 13-4(158). – С. 53-55.
3. Boehm, B. Top 10 list software development / B. Boehm, V. Basili // Computer. – 2001. – Vol. 34. – P. 135-137.
4. An Automated Framework for Testing Source Code Static Analysis Tools / D. M. Gimatdinov, A. Y. Gerasimov, P. A. Privalov [et al.] // Proceedings of the Institute for System Programming of the RAS. – 2021. – Vol. 33. – No 3. – P. 41-50. – DOI 10.15514/ISPRAS-2021-33(3)-3.
5. Сидорин, А.В. Реализация межмодульного анализа для языков С и С++ в статическом анализаторе, использующем для анализа исходный код программы / А.В. Сидорин // Наука и образование: электронное научно-техническое издание. Информатика, вычислительная техника и управление. – 2015. – № 9. – С. 145–169.
6. Булах, Б.В. Применение семантических технологий для анализа и рефакторинга программного кода / Б.В. Булах // Международный научный журнал Интернаука. – 2018. – № 12(52). – С. 37-40.
7. Rus, S. Hybrid Analysis: Static & Dynamic Memory Reference Analysis / S. Rus, L. Rauchwerger, Ja. Hoeflinger // International Journal of Parallel Programming. – 2003. – Vol. 31. – No 4. – P. 251-283.

8. Kremenek T. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations / T. Kremenek, D. Engler // Static Analysis. – Springer Berlin Heidelberg, 2003. – P. 295-315.

9. McGraw, G. Static Analysis for Security / G. McGraw, B. Chess // IEEE Security & Privacy. – 2004. – Vol. 2, Issue: 6. – P. 76-79.

10. Компиляторы. Принципы, технологии и инструментарий / В. Ахо Альфред, С. Лам Моника, Р. Сети, Д. Ульман Джеффри. – Вильямс, 2003. – 1184 с.

11. Пути совершенствования метода динамического анализа кода C/C++-программ / М. А. Кондауров, А. Е. Кузнецов, А. А. Стенин, М. С. Стуков // Известия Института инженерной физики. – 2010. – № 1(15). – С. 3-7.

12. Количество уязвимостей в ПО [Электронный ресурс]. - Режим доступа: <https://servernews.ru/1028105>

13. Международная статистика уязвимостей [Электронный ресурс]. - Режим доступа: http://www.infosecurity.ru/_gazeta/content/081024/article01.shtml

14. Жидков, И. В. О признаках потенциально опасных событий в информационных системах / И.В. Жидков, И.В. Кадушкин // Вопросы кибербезопасности. – 2014. – № 1 (2). – С. 40-48.

15. Поиск уязвимостей в программах с помощью анализаторов кода [Электронный ресурс]. - Режим доступа: <http://www.codenet.ru/progr/other/code-analysers.php>

16. Безопасность систем с открытым кодом [Электронный ресурс]. - Режим доступа: <https://www.osp.ru/os/2003/07-08/183264>

17. Клімов, І.З. Оцінка надійності систем захисту інформації від несанкціонованого доступу / І.З. Клімов, А.А. Пономарьов // Вісник Іжевського державного технічного університету. – 2008. – №-3. – С. 102-103.

18. Анализ кода [Электронный ресурс]. - Режим доступа: <https://www.jetbrains.com/ru-ru/objc/features/code-analysis.html>

19. Контролируем качество кода с помощью платформы SonarQube [Электронный ресурс]. - Режим доступа: <https://habr.com/ru/company/pvs-studio/blog/315422/>

20. PVS-Studio – статический анализатор на страже качества, защищённости (SAST) и безопасности кода [Электронный ресурс]. - Режим доступа: <https://pvs-studio.com/ru/pvs-studio/>

21. Компиляторы: принципы, технологии и инструментарий / Ахо А. В. [и др.]. – М.: ООО «И. Д. Вильямс», 2016. – 1184 с.

22. Аветисян, А. И. Технологии статического и динамического анализа уязвимостей программного обеспечения / А.И. Аветисян, А.А. Белеванцев, И.И. Чукляев // Вопросы кибербезопасности. – 2014. – № 3 (4). – С. 20-28.

23. Лебедев, В.И. Методы и перспективы статического анализа программного кода / В.И. Лебедев // Молодежный научно-технический вестник. – 2015. – № 12. – С. 28.

24. Кривдюк, Н. М. Современные средства анализа программного кода / Н.М. Кривдюк, А.Е. Янковская // Естественные и технические науки: опыт, проблемы, перспективы. – 2016. – № 3. – С. 46-49.

25. Липаев, В.В. Проектирование и производство сложных заказных программных продуктов / В.В. Липаев . – Москва: СИНТЕГ, 2011. – 408 с.

26. Coverity Scan [Electronic resource]. – Access mode: <https://scan.coverity.com/>

27. Static Code Analysis for C, C++, C#, Java, JavaScript, and Python [Electronic resource]. – Access mode: <https://www.perforce.com/products/klocwork>

28. Fortify Static Code Analyzer [Электронный ресурс]. - Режим доступа: <https://www.netwell.ru/products/fortify-static-code-analyzer/>

29. Cppcheck [Электронный ресурс]. – Режим доступа: <https://sourceforge.net/projects/cppcheck/>

30. Okamura, H. Quantitative Security Evaluation for Software System from

Vulnerability Database / H. Okamura, M. Tokuzane, T. Dohi // Journal of Software Engineering and Applications. – 2013. – Vol. 6 No. 4A. – P. 15-23.

31. Clang Static Analyzer [Электронный ресурс] – Режим доступа: <https://clanganalyzer.llvm.org/>

32. Марков, А. С. Систематика уязвимостей и дефектов безопасности программных ресурсов / А.С. Марков, А.А. Фадин. // Защита информации. Инсайд. – 2013. – № 3 (51). – С. 56-61.

33. CWE [Electronic resource]. – Access mode: <https://cwe.mitre.org/>

34. CERT [Electronic resource]. – Access mode: <https://cert.gov.ua/>

35. MISRA C: борьба за качество и безопасность кода [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/company/pvs-studio/blog/579420/>

36. Рибер, Г. Многоуровневый подход к оценке безопасности программных средств / Г. Рибер, К. Малмквист, А. Щербаков // Вопросы кибербезопасности. – 2014. – № 1 (2). – С. 36-39.

ДОДАТОК А
Програми тестування аналізатора

Лістинг А.1 – Приклади програм тестування аналізатора коду

// Приклад 1

```

if (!is_void_return && tm) {
if (tm) {
Replaceall(tm, "$result", "lresult");
Printf(f->code, "%s\n", tm);
Printf(f->code, " return lresult;\n");
Delete(tm);
} else {
Swig_warning(WARN_TYPEMAP_OUT_UNDEF, input_file, line_number,
"Unable to use return type %s in function %s.\n",
SwigType_str(t, 0), name);
}}

```

// Приклад 2

```

if (rawptr >= end) {
pubk->u.fortezza.DSSKey.len= pubk->u.fortezza.KEYKey.len;
pubk->u.fortezza.DSSKey.data=pubk->u.fortezza.KEYKey.data;
pubk->u.fortezza.DSSprivilege.data=pubk->u.fortezza.DSSprivilege.data;
goto done; }

```

// Приклад 3

```

if(!p) {
*p = 0;
}
if(q) {
exit(0);
}
*q = 0;

```

//Приклад 4

```

old = new;
new = NextIndicatorName(info);
if (!new)
{
WSGO1("Couldn't allocate name for %d\n", new->ndx);

```

```

ACTION("Ignored\n");
return False;
}

```

```

//Приклад 5
struct namehashent *namehashent = insert_name (ah, alias, strlen (alias),
replace);
if (namehashent == NULL && ! replace)
return;

```

```

// Приклад 6
while(a == true);
{
a=b;
}

```

```

// Приклад 7
while(a = true)
{
a=1;
}

```

```

//Приклад 8
if(a.x == a.x)
{
a.x=0;
}

```

```

// Приклад 9
void emptystring(char * p)
{
if(p)
return;
*p = '\0';
}

```

```

//Приклад 10
char buffer[10]
void init(char **p, int a)
{
if(a > 0)

```

```

*p = &buffer[0];
if(a >= 0)
**p = '\0';
}

```

```

//Приклад 11
int main()
{
    cin >> y;
    cout << y;
}

```

```

//Приклад 12
int ct;
while(ct < 100)
{
    cout << ct;
}

```

```

//Приклад 13
class Base { int x; };
class Derived: virtual public Base {};
Derived d;

```

```

//Приклад 14
void test(void *ptr)
{
    free(ptr);
    free(ptr);
}

```

```

//Приклад 15
static jmp_buf env;
static void inner(void)
{
    longjmp(env, 1);
}
static void middle(void)

```

```

{
    void *ptr = malloc(1024);
    inner();
    free(ptr);
}
void outer(void)
{
    int i;
    i = setjmp(env);
    if (i == 0)
        middle();
}

```

//Приклад 16

```

void test(const char *filename)
{
    FILE *f = fopen(filename, "r");
    void *p = malloc(1024);
    /* do stuff */
}

```

// Приклад 17

```

struct link { struct link *next; };
int free_a_list_badly(struct link *n)
{
    while (n) {
        free(n);
        n = n->next;
    }
}

```

//Приклад 18

```

void test(int n)
{
    int buf[10];
    int *ptr;
    if (n < 10)
        ptr = buf;
    else
        ptr = (int *)malloc(sizeof (int) * n);
    free(ptr);
}

```

```
}
```

//Приклад 19

```
extern void body_of_program(void);  
void custom_logger(const char *msg)  
{  
    fprintf(stderr, "LOG: %s", msg);  
}  
static void handler(int signum)  
{  
    custom_logger("got signal");  
}  
int main(int argc, const char *argv)  
{  
    custom_logger("started");  
    signal(SIGINT, handler);  
    body_of_program();  
    custom_logger("stopped");  
    return 0;  
}
```

ДОДАТОК Б
Слайди презентації

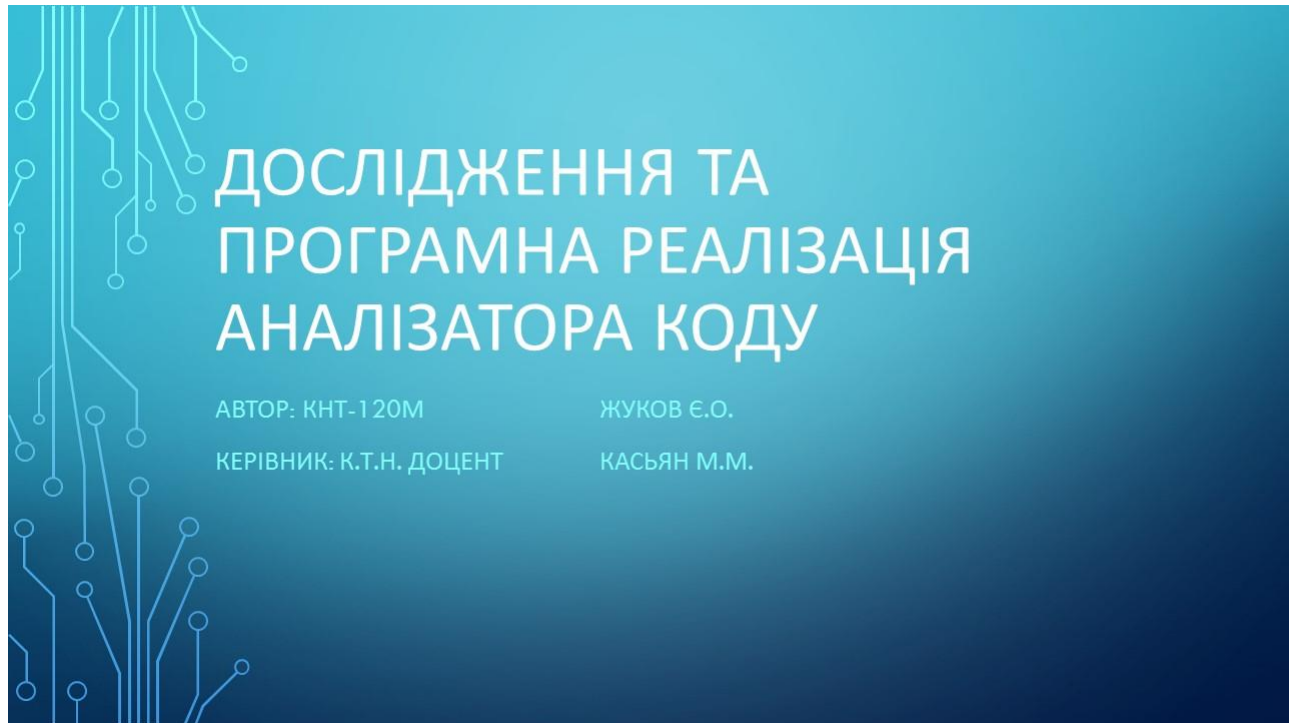


Рисунок Б.1 – Слайд № 1

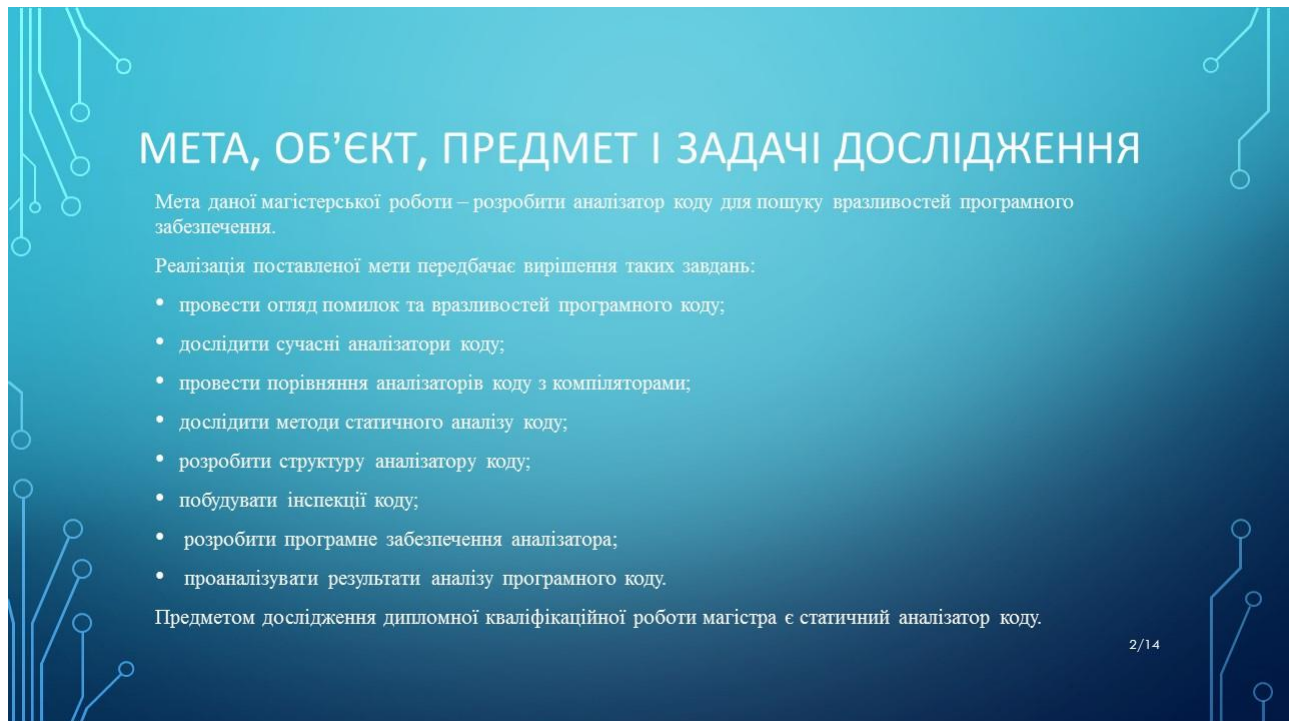


Рисунок Б.2 – Слайд № 2

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Таблиця 1 – Переваги та недоліки динамічного аналізу

Переваги	Недоліки
Рідко виникають "хибні спрацьовування" - висока продуктивність знаходження помилок.	Відбувається втручання у поведінку системи у часі; ступінь втручання залежить від кількості використовуваних інструментальних вставок.
Захоплюються помилки як у реальних умовах, і у режимі імітаційного моделювання.	Повнота аналізу помилок залежить від рівня покриття коду.

3/14

Рисунок Б.3 – Слайд № 3

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Таблиця 2 – Переваги та недоліки статичного аналізу

Переваги	Недоліки
Використовується вже на ранніх етапах життєвого циклу програмного забезпечення. Можуть аналізуватись існуючі бази кодів, які вже пройшли тестування.	Можуть виправлятися програмні помилки та вразливості, які обов'язково призводять до відмови програми.
Низькі вартісні витрати.	Ненульова ймовірність "хибного спрацьовування".

4/14

Рисунок Б.4 – Слайд № 4

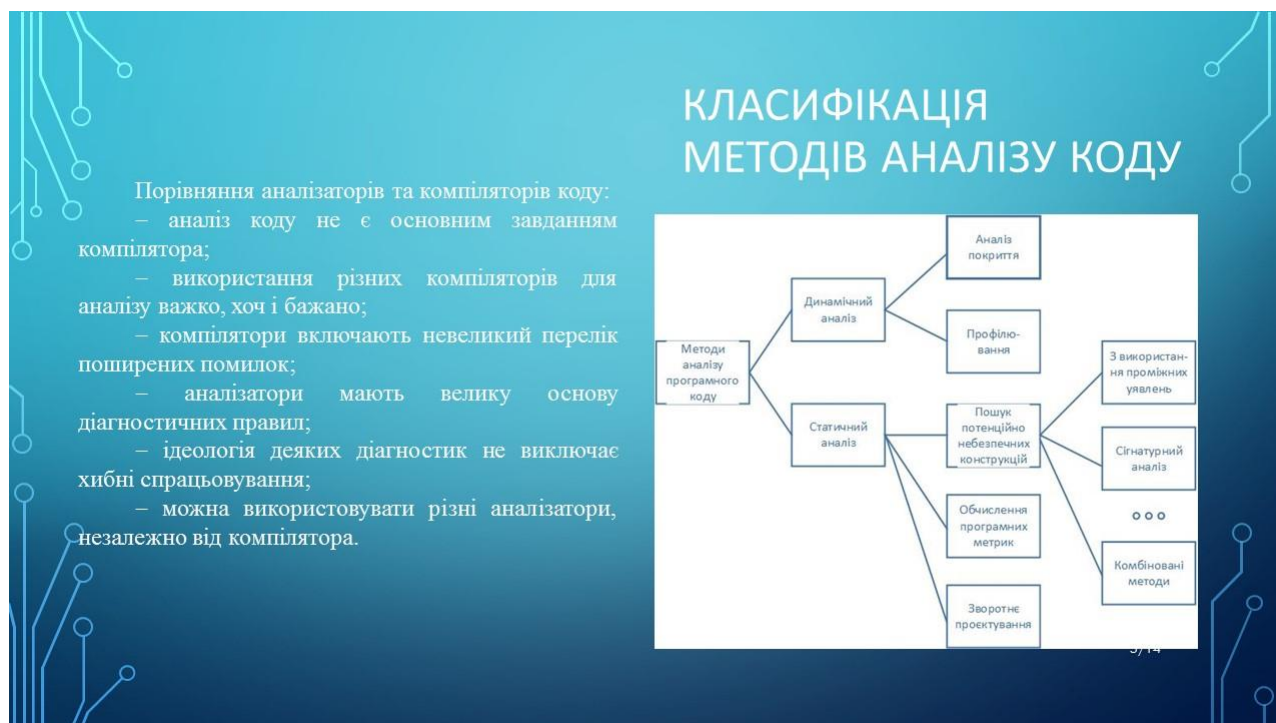


Рисунок Б.5 – Слайд № 5

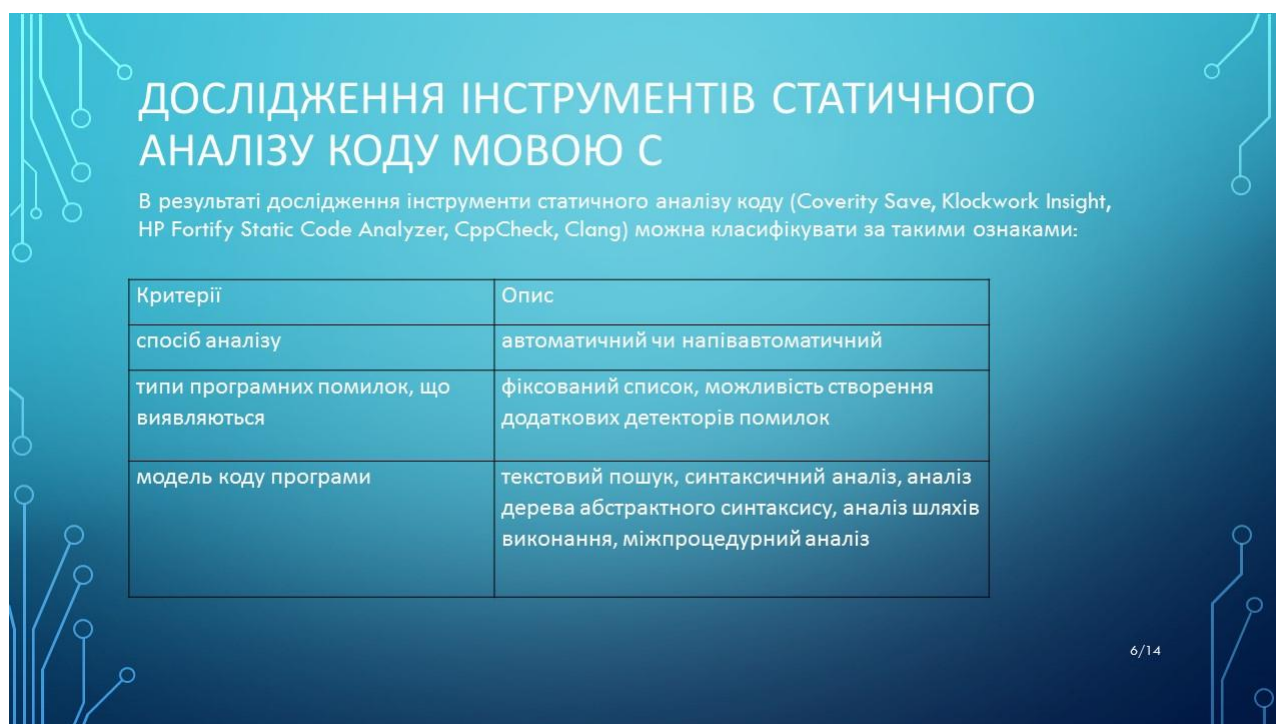


Рисунок Б.6 – Слайд № 6

СТРУКТУРА АНАЛІЗАТОРА



Рисунок 4 - Проміжний вигляд структури аналізатора коду

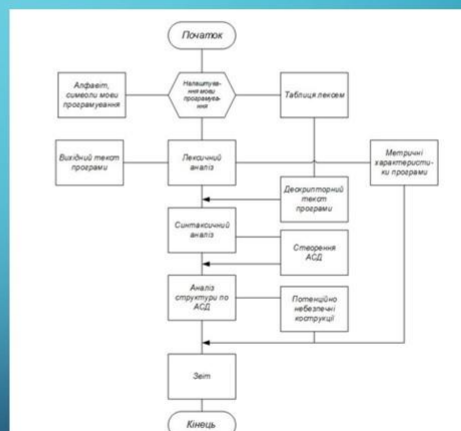


Рисунок 5 - Схема алгоритму аналізатора коду

7/14

Рисунок Б.7 – Слайд № 7

РОЗРОБКА СИСТЕМИ

Критерій	Опис
Мова програмування	Python, оскільки він дозволяє використовувати існуючі рішення найбільш ефективно і має великий набір додаткових бібліотек
Інтерфейс користувача	Інтерфейс користувача реалізований за допомогою бібліотеки Tkinter.
Інспекції коду	Python, описує 36 правил написання коректного коду C, які поділені на вісім категорій: загальні правила, використання new/delete, конструктори-деструктори-визначення, розробка, реалізація, спадкування, оператори, винятки

8/14

Рисунок Б.8 – Слайд № 8

ІНСПЕКЦІЇ КОДУ

Таблиця 4.1 – Загальні правила

№	ID	Опис
1	E1	Використовуйте <code>const</code> замість <code>#define</code> для констант у глобальній та файлової області
2	E2	Використовуйте приведення в новому стилі замість приведення в стилі C.
3	E3	Не розглядайте вказівник на <code>Derived[]</code> як вказівник на <code>Base[]</code> .

Таблиця 4.6 – Правила спадкування

№	ID	Опис
1	E30	Не перевизначайте успадковану невіртуальну функцію
2	E31	Не перевизначайте успадковане значення параметра за замовчуванням

9/14

Рисунок Б.9 – Слайд № 9

ІНТЕРФЕЙС ПРОГРАМИ

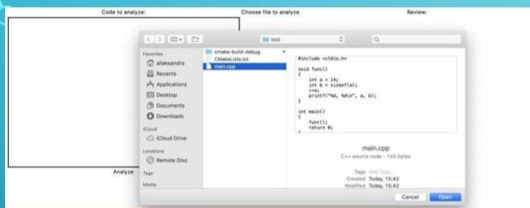


Рисунок 6 - Додавання вихідного коду



Рисунок 7 – Результат роботи аналізатора

10/14

Рисунок Б.10 – Слайд № 10

ІНТЕРФЕЙС ПРОГРАМИ



Рисунок 8 – Результат роботи аналізатора



Рисунок 9 – Результат роботи аналізатора

11/14

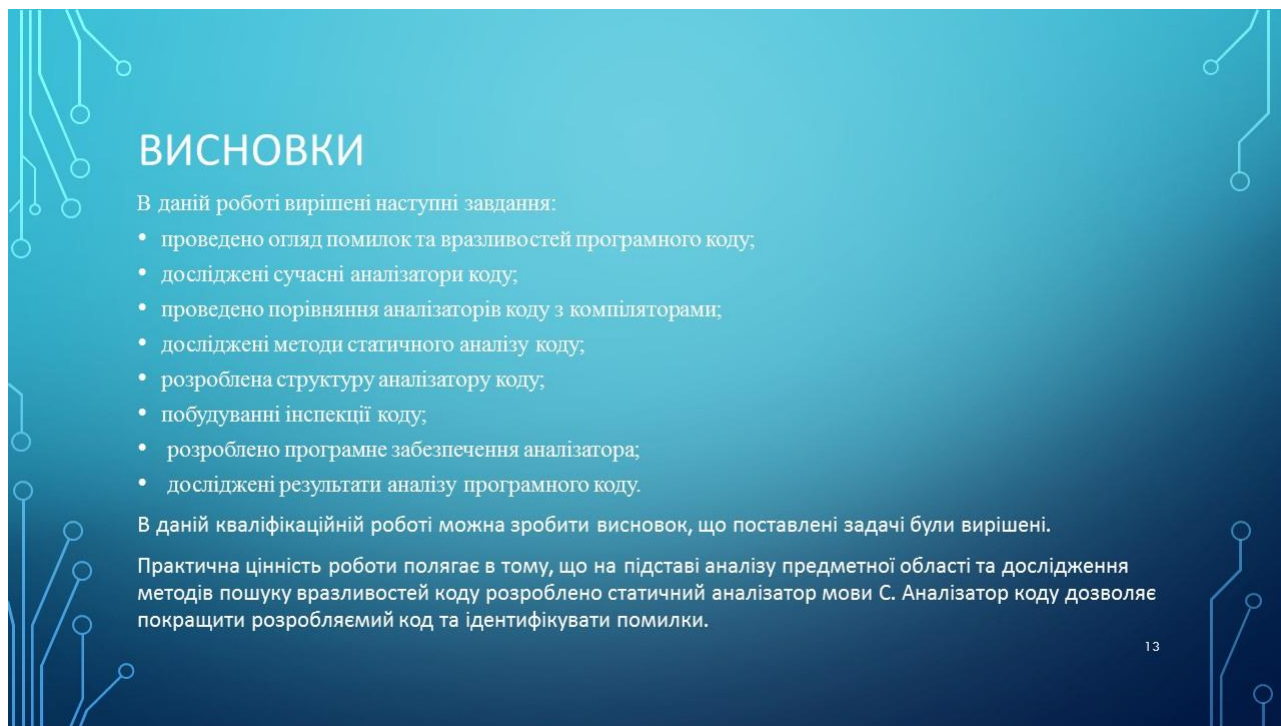
Рисунок Б.11 – Слайд № 11

РЕЗУЛЬТАТИ АНАЛІЗУ

Помилка	Всього	Переглянутих	Істинних	Точність, %	Повнота, %
E1	17	17	17	100	100
E5	14	14	14	100	100
E6	4	4	4	100	100
E10	55	48	44	91	80
E11	21	21	21	100	100
E14	19	17	12	71	63
E20	4	4	2	50	50
E24	54	27	20	74	37
E25	144	61	47	77	32
E32	12	6	6	100	50

12/14

Рисунок Б.12 – Слайд № 12



ВИСНОВКИ

В даній роботі вирішені наступні завдання:

- проведено огляд помилок та вразливостей програмного коду;
- досліджені сучасні аналізатори коду;
- проведено порівняння аналізаторів коду з компіляторами;
- досліджені методи статичного аналізу коду;
- розроблена структура аналізатору коду;
- побудовані інспекції коду;
- розроблено програмне забезпечення аналізатора;
- досліджені результати аналізу програмного коду.

В даній кваліфікаційній роботі можна зробити висновок, що поставлені задачі були вирішені.

Практична цінність роботи полягає в тому, що на підставі аналізу предметної області та дослідження методів пошуку вразливостей коду розроблено статичний аналізатор мови С. Аналізатор коду дозволяє покращити розробляємий код та ідентифікувати помилки.

13

Рисунок Б.13 – Слайд № 13