Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерної інженерії та управління_____
(повна назва)

Кафедра _____Безпеки інформаційних технологій_____
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти _____другий (магістерський)_____

Застосування технологій комбінованого аналізу для пошуку вразливостей
_____при розробці програмного забезпечення_____
(тема)

Виконав:
студент __2_ курсу, групи __БІКСм-20-1_____
_____Ярещенко В.В._____
(прізвище, ініціали)

Спеціальність _____125 Кібербезпека _____
(код і повна назва спеціальності)

Освітня програма ___«Безпека інформаційних
_____і комунікаційних систем»_____
(повна назва освітньої програми)

Керівник __доцент Федюшин О.І._____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ _____Халімов Г.З._____
(підпис) (прізвище, ініціали)

2021 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління

Кафедра Безпеки інформаційних технологій

Рівень вищої освіти другий (магістерський)

Спеціальність 125 Кібербезпека
(код і повна назва)

Освітня програма «Безпека інформаційних і комунікаційних систем»
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Ярещенко Владиславу Валерійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Застосування технологій комбінованого аналізу для пошуку вразливостей при розробці програмного забезпечення

затверджена наказом по університету від 08 11 2021 р. № 1685ст

2. Термін подання студентом роботи до екзаменаційної комісії _____2021 р.

3. Вихідні дані до роботи Перелік бінарних вразливостей в програмному забезпеченні та їх аналіз. Сучасні види класифікації бінарних вразливостей. Дослідження технологій статичних та динамічних аналізаторів програм. Дослідження проблематики тестування програмного забезпечення в сучасних методологіях розробки.

4. Перелік питань, що потрібно опрацювати в роботі: сучасні методології розробки безпечного програмного забезпечення; сучасні підходи до пошуку вразливостей в програмному забезпеченні, їх переваги та недоліки; можливість використання сучасних підходів до аналізу програм в комплексі для підвищення ефективності виявлення вразливостей; розробка інструменту комбінованого аналізу вразливостей; тестування інструменту комбінованого аналізу вразливостей; можливість використання комбінованого аналізу вразливостей в сучасних методологіях розробки програмного забезпечення.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) презентаційний матеріал у вигляді слайдів

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Видача завдання	02.09.2021	Виконано
2	Аналіз літературних джерел за отриманим завданням	02.09.2021 – 17.09.2021	Виконано
3	Вивчення фундаментальних понять про вразливості в ПЗ	17.09.2021 – 30.09.2021	Виконано
4	Вивчення сучасних підходів до аналізу вразливостей в ПЗ	01.10.2021 - 15.10.2021	Виконано
5	Дослідження переваг та недоліків сучасних підходів до аналізу вразливостей в ПЗ	15.10.2021 – 31.10.2021	Виконано
6	Створення вимог, завдань та схеми комбінованого аналізу вразливостей в ПЗ	01.11.2021 – 07.11.2021	Виконано
7	Розробка інструменту комбінованого аналізу ПЗ	07.11.2021 – 21.11.2021	Виконано
8	Тестування розробленого інструменту та оцінка ефективності запропонованого підходу	21.11.2021 – 30.11.2021	Виконано
9	Оформлення пояснювальної записки	01.12.2021 – 10.12.2021	Виконано

Дата видачі завдання 02 вересня 2021 р.

Студент _____
(підпис)

Керівник роботи _____ _____ доцент Федюшин О.І.
(підпис) (посада, прізвище, ініціали

РЕФЕРАТ

Пояснювальна записка містить 116 сторінок, 33 рисунка, 9 таблиць, 3 додатки, 27 джерел за переліком посилань.

ВРАЗЛИВІСТЬ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, РОЗРОБКА, ВИХІДНИЙ КОД, БІНАРНИЙ КОД, СТАТИЧНИЙ АНАЛІЗ, ДИНАМІЧНИЙ АНАЛІЗ, КОМБІНОВАНИЙ АНАЛІЗ, ІНСТРУМЕНТ, ЗАСІБ.

Об’єкт дослідження – процес пошуку вразливостей при розробці програмного забезпечення.

Предмет дослідження – технології та інструменти пошуку вразливостей на етапі розробки програмного забезпечення.

Метою даної роботи є дослідження можливості комбінування інструментів аналізу вихідного та бінарного коду програм для підвищення ефективності пошуку вразливостей в процесі створення ПЗ.

За результатами роботи розроблені: інструмент комбінованого аналізу пошуку вразливостей програм; схема використання комбінованого аналізу розробником та послідовність дій щодо інтеграції комбінованого аналізу в сучасну методологію розробки програмного забезпечення.

ABSTRACT

Explanatory note includes: 116 pages, 33 pictures, 9 tables, 3 applications, 27 sources for references.

VULNERABILITY, SOFTWARE, DEVELOPMENT, STATIC ANALYSIS, DYNAMIC ANALYSIS, COMBINED ANALYSIS, APPLICATION, TOOL

The object of research is the process of finding vulnerabilities in software development.

The subject of research - technologies and tools for finding vulnerabilities at the stage of software development.

The aim of this work is to study the possibility of combining tools for analyzing the source and binary code of programs to improve the efficiency of finding vulnerabilities in the software development process.

Based on the results of the work, the following were developed: a tool for combined analysis of the search for program vulnerabilities; the scheme of using the combined analysis by the developer and the sequence of actions for the integration of the combined analysis into the modern methodology of software development.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення.

Embedded System – англ. вбудовані системи.

Proof-Of-Concept – англ. Доказ здійсненності концепції.

S-SDLC – англ. Secure Software Development Life Cycle – цикл розробки безпечного програмного забезпечення.

AST - англ. Abstract Syntax Tree - абстрактне синтаксичне дерево.

SSL – англ. Secure Sockets Layer – рівень захищених сокетів.

CVE – англ. Common Vulnerabilities and Exposures – загальні вразливості та їх вплив.

CWE – англ. Common Weakness Enumeration – загальний перелік вразливостей.

CVSS – англ. Common Vulnerability Scoring System – загальна система оцінювання вразливостей.

WYSINWYX – абревіатура з англ. What You See Is Not What You eXecute.

ОС – операційна система.

ЦП – центральний процесор.

ОП – оперативна пам’ять.

GUI – англ. Graphical User Interface – графічний інтерфейс користувача.

ВСТУП

На сьогоднішній день компанії-розробники програмного забезпечення зустрічаються з серйозними ризиками, пов’язаними з безпекою власних продуктів. Ці ризики виникають внаслідок того, що процес розробки програмного забезпечення неможливо уявити без помилок, що можуть бути допущені розробниками та архітекторами в ході своєї роботи. В результаті таких помилок можуть з’являтися серйозні вразливості, пов’язані з безпекою розроблюваного програмного продукту та оброблюваної ним інформації. Це, в свою чергу, буде мати негативний вплив на безпеку інформаційних систем, в яких будуть функціонувати таке програмне забезпечення. У міжнародній базі даних про вразливості програмного забезпечення CVE (Common Vulnerabilities and Exposures) щодня реєструються десятки вразливостей різного рівня, які можуть призводити до компрометації інформації на мільйонах обчислювальних пристроїв у всьому світі [1].

Сьогодні все більше поширюється пропрієтарне ПЗ, вихідні тексти якого представляють інтелектуальну власність компанії, яка, як правило, не зацікавлена в передачі своїх розробок третім сторонам. Не дивлячись на те, що в таких компаніях дуже часто добре розвинена інженерна практика Code Review, ціллю якої є виявлення недоліків, помилок, невідповідність бізнес-логіки, такий підхід не дозволяє виявити цілий ряд вразливостей, які можуть буде приховані та одразу непомітні людині (переповнення декількох байт буфера, зчитування/запис по довільній або не ініціалізованій адресі і т.п.), або взагалі програмна закладка може бути введена навмисно. Крім цього аналіз вихідного тексту програми людиною не дозволяє виявити деякі типи помилок, які можуть бути пов'язані з особливостями компілятора або взагалі можуть бути виявлені тільки в рамках динамічного аналізу.

Інструменти аналізу вихідних текстів програми (також відомі як Інструменти статичного тестування безпеки додатків, з англ. Static Application Security Testing) – інструменти розробки програмного забезпечення, що призначені для аналізу вихідних текстів програми з ціллю виявлення потенційних погроз безпеці програмного забезпечення.

Ці інструменти можуть бути як інтегровані в середовище розробки, так і бути самостійними програмними продуктами. Для типів проблем, які можна виявити під час фази написання вихідних текстів програми, інтеграція в середовище розробника (наприклад в його редактор вихідних текстів) може мати значний вплив на якість написаних цим розробником кодів, так як таке впровадження забезпечує негайний зворотній зв'язок щодо можливих помилок [2].

Крім статичного аналізу вихідних текстів, виділяють також і засоби статичного аналізу бінарного коду, тобто аналізу скомпільованої програми без її виконання. При такому підході зазвичай використовуються ті ж самі методи і технології, що й для вихідного коду. Але їх використання потребує попереднього отримання уявлення про потік виконання для кожної функції. Для отримання такого уявлення спеціально створюється проміжне представлення (по даним і керуванню програмою).

При статичному аналізі можна виявити багато різноманітних дефектів і слабких місць вихідного коду навіть до того, як код буде готовий для запуску. Можливість такого використання особливо корисна для проектів вбудованих систем (Embedded Systems), так як на таких проектах розробники часто не можуть використовувати засоби динамічного аналізу та повномасштабного тестування до тих пір, поки програмне забезпечення не буде завершено настільки, щоб його можна було запустити на цільовій системі.

З іншого боку, динамічний аналіз, або аналіз під час виконання, відбувається на працюючому програмному забезпеченні і виявляє проблеми в міру їх виникнення, як правило, використовуючи складні інструментальні засоби. Хтось може заперечити, що одна форма аналізу передує іншу, але

розробники можуть комбінувати обидва способи для прискорення процесів розробки і тестування, а також для підвищення якості розроблюваного продукту.

Використовуючи різні техніки аналізу, наприклад, перевірку абстрактного синтаксичного дерева і аналіз кодових шляхів, інструменти статичного аналізу можуть виявити приховані уразливості, логічні помилки, дефекти реалізації та інші проблеми. При динамічному аналізі не тільки виявляються помилки, пов'язані з посиланнями в пам’яті, потенційні аномалії в логіці, синтаксисі та семантиці, але і є можливість також оптимізувати використання циклів ЦП, ОП, мережевого інтерфейсу і інших ресурсів.

Слід також чітко відрізняти динамічний аналіз від тестування логіки та очікуваного результату роботи програмного забезпечення, що може проводитись за допомогою інфраструктури автоматизованого тестування ПЗ. Принципове і математично поки не розв’язане обмеження полягає в тому, що за допомогою аналізаторів ми можемо виявити лише заздалегідь детерміновані та описані типи вразливостей, що можуть призводити до нетривіальних проблем.

Таким чином, необхідність проведення аналізу вразливостей програм обумовлена низкою вищеописаних факторів, які можуть породжувати серйозні наслідки, пов'язані з безпекою інформації, що захищається, або обробку якої здійснює уразливий програмний продукт. В протидію таким ситуаціям та для їх запобігання запропоновано розглянути засоби аналізу програм та їх коду.

Метою даної роботи є дослідження можливості використання інструментів аналізу програм (статичний аналіз вихідного коду, динамічний аналіз бінарного коду, статичний аналіз бінарного коду) для виявлення бінарних вразливостей та можливості їх комбінування для підвищення якості програм при їх розробці. Для досягнення даної мети необхідно вирішити наступні завдання:

1. Дати визначення досліджуваним інструментам. Розглянути технології що реалізовані в інструментах. Визначити основні переваги та недоліки засобів аналізу.

2. Розглянути проблематику виникнення бінарних вразливостей в процесі розробки ПЗ. Визначити основні фактори що перешкоджають в виявленні дефектів ПЗ інструментами аналізу.

3. На базі розглянутих знань сформувати вимоги щодо системи яка ефективно поєднувати засоби аналізу.

4. Створити схеми систем комбінованого аналізу які б задовольняли висунутим вимогам.

5. Розробити інструмент комбінованого аналізу, який реалізує ключові ідеї запропонованої схеми (так званий Proof-Of-Concept).

6. Провести тестування розробленого інструменту та зробити висновки щодо відповідності між теоретичними гіпотезами та результатами практичного використання.

7. На основі отриманих висновків запропонувати підхід до використання комбінованого аналізу під час розробки ПЗ в сучасних методологіях.

1 ВРАЗЛИВОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Загальноприйнятого в усьому світі поняття «вразливість» на сьогодні не існує. В різних роботах посвячених виявленню дефектів в програмах дається своє визначення що найбільш чітко відображає суть досліджуваного явища в рамках роботи. В 2010 році підрозділом Комп'ютерного спільництва (IEEE Computer Society) Інституту інженерів електротехніки і електроніки (IEEE – Institute of Electrical and Electronics Engineers) випущено стандарт IEEE 1044 2009 «Стандартна класифікація для програмних аномалій» (IEEE Standard Classification for Software Anomalies) [3], в якому дається декілька визначень для термінів, використовуваних в рамках даної класифікації:

- дефект (defect) – недолік в працюючому програмному продукті, коли цей працюючий продукт не відповідає вимогам або специфікаціям і потребується його виправлення або заміна;

- помилка (error) – дії людини, які призводять до некоректного результату;

- провал (failure) – припинення можливості продукту виконувати необхідну функцію чи нездатність виконувати функції у вказаних раніше обмеженнях;

- несправність (fault) – повідомлення про помилку в програмі;

- проблема (problem) – труднощі або невизначеність, з якими стикається один або більше користувачів, які виникли внаслідок роботи із системою, що використовується.

Наведені вище визначення не дають розуміння що таке «вразливість», адже ці поняття в контексті даної роботи скоріше є результатом використання зловмисником одного з наявних в програмі дефектів. В термінах безпеки інформаційних технологій, вразливість – це дефект, що може бути експлуатований третьою стороною, такою як зловмисник, для виконання неавторизованих дій з інформаційною системою. Це означає, що ресурси (фізичні або логічні) можуть мати деякі дефекти, що можуть використовуватися

зловмисником і в результаті його дій може створюватись негативний вплив на конфіденційність, цілісність або доступність ресурсів (не обов'язково вразливих) що відносяться до організації та інших залучених сторін (замовники, постачальники). Схематичне зображення того, як зловмисник може завдавати збитки організаціям, в склад яких входять вразливі ресурси, зображено на рисунку 1.1.

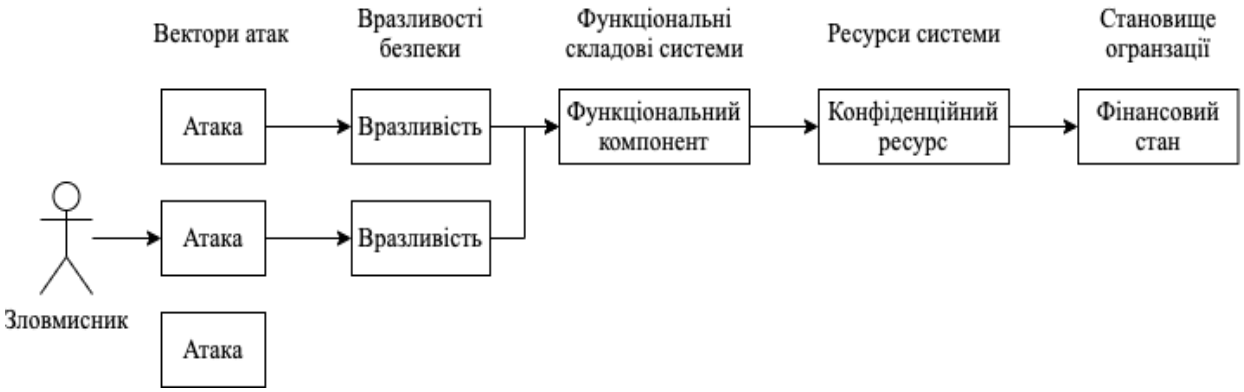

Рисунок 1.1 – Вплив зловмисника на становище організації.

На сьогодні в світі загальноприйнятими класифікаціями вразливостей є:

- Common Weakness Enumeration (CWE) – класифікатор типів вразливостей. Тут описано що означає той чи інший тип вразливості, наведені приклади в вигляді коду та описані можливі наслідки експлуатації [4].

- Common Vulnerabilities and Exposures (CVE) – класифікатор конкретних вразливостей. Кожний дефект в серйозному програмному продукті представлена в класифікаторі цього типу [5]. Вразливості представлені тут в вигляді префіксу "CVE", року виявлення та довільного числа (Наприклад: CVE-2014-1776).

- Common Vulnerability Scoring System (CVSS) – стандарт оцінювання серйозності вразливостей. По суті, являє собою раціональне число в проміжку (0, 10], яке говорить про те, наскільки серйозною є ця вразливість, де 10 - означає найбільш серйозну уразливість [6]. CVSS часто використовується організаціями для того, щоб прийняти рішення про те, наскільки швидко потрібно виправити вразливість.

Найбільш поширені види дефектів в програмному забезпеченні зображено на рисунку [7].

Рисунок 1.2 – Найпоширеніші типи дефектів в програмному забезпеченні.

Для того щоб зрозуміти як в програмному забезпеченні з’являються вразливості, необхідно ознайомитись з його життєвим циклом. Слід зауважити, що ми не будемо розглядати так зване «самомодифікуюче» програмне забезпечення, тобто таке ПЗ, яке в процесі своєї роботи може змінювати свою структуру та граф виконання. Найпопулярнішим алгоритмом створення програмного забезпечення серед компаній-розробників на сьогодні є Цикл розробки безпечного програмного забезпечення (Secure Software Development

Lifecycle). Нижче описані та зображені на рисунку основні складові цієї методології [8]:

- Визначення завдання. Першим етапом в рамках створення програми є формулювання завдання. Необхідно дати відповіді на запитання «Що можна зробити?», «Що потрібно клієнту?», тобто необхідно отримати інформацію від всіх зацікавлених сторін (клієнти, розробники, співробітники).

- Планування та аналіз. Наступним кроком є визначення вимог до розроблюваного продукту. Зазвичай це робиться за допомогою документу SRS (Специфікація вимог до програмного забезпечення). З точки зору безпеки розробки, на цьому етапі проводиться оцінка загроз та створюється їх модель.

- Дизайн. Цей етап зазвичай дає відповідь на запитання «Як ми отримаємо те, що нам потрібно?». Визначаються елементи системи, компоненти, рівень та вимоги безпеки, модулі, архітектуру, різні інтерфейси і типи даних, якими оперує система.

- Розробка. В циклі розробки зазвичай тут пишеться вихідний код та створюються прототипи, збірки (Build) програмного продукту.

- Тестування. На цьому етапі перевіряється наявність дефектів та недоліків, виконується перевірка вимог до безпеки (Security Quality Assurance). Пізніше, після виявлення, намагаються вирішити всі проблеми, поки продукт не буде задовольняти специфікації.

- Експлуатація та інтеграція. Наступним кроком є отримання зворотного зв’язку від кінцевих користувачів і внесення змін в залежності від нього.

- Підтримка. Ця фаза включає в себе технічну підтримку системи, оцінка продуктивності, оновлення компонентів відповідно до стандартів та нових технологій і т. д.

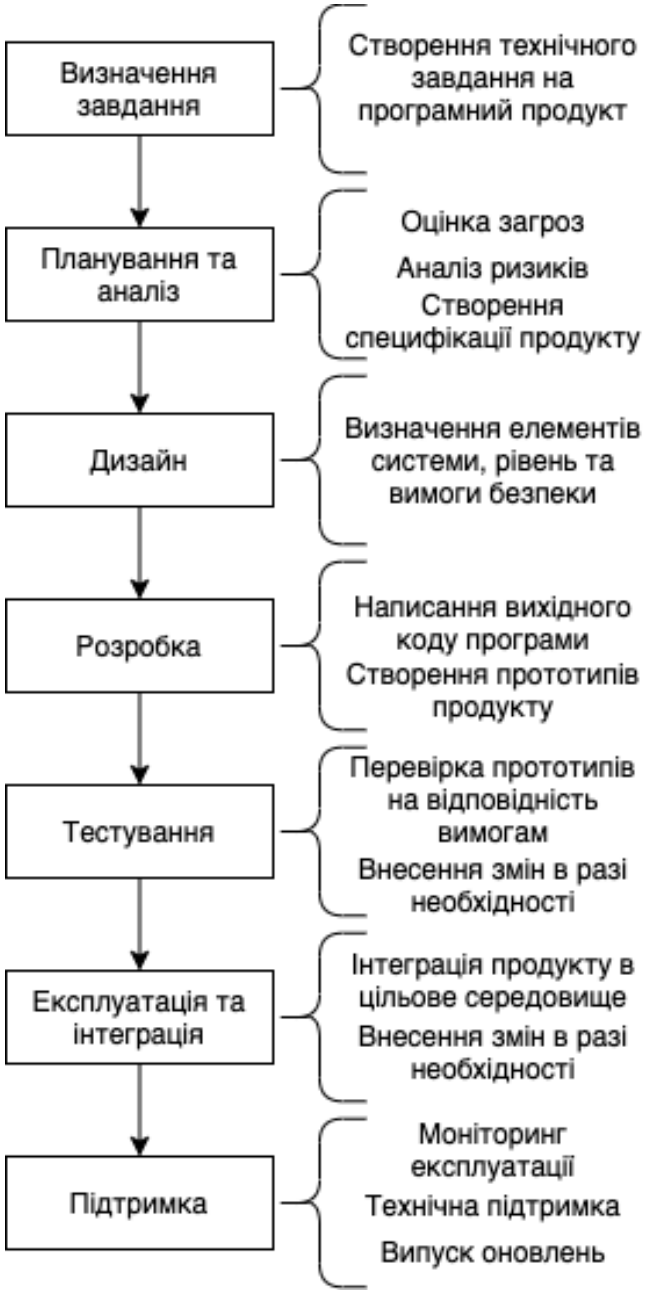

Рисунок 1.3 – Етапи процесу розробки ПЗ при використанні методології S-SDLC.

Таким чином, в даному розділі було розглянуто поняття вразливості. Дано визначення цьому явищу та розглянуті їх основні види та класифікації. Також було розглянуто сучасну методологію розробки ПЗ та виявлено що основним етапом на якому з’являються вразливості є етап розробки, в свою чергу найоптимальнішим етапом для виявлення та усунення дефектів є етапи розробки та тестування.

2 СУЧАСНІ ПІДХОДИ ДО ПОШУКУ ВРАЗЛИВОСТЕЙ В ПЗ

Пошук і усунення дефектів в ПЗ вимагає значних витрат, при цьому багато з них можуть залишитися непоміченими. За даними дослідження, проведеного на замовлення Національного інституту стандартів і технологій США, збитки, що виникають через недостатньо розвинуту інфраструктуру усунення дефектів в ПЗ (вразливостей і некритичних помилок), складають від 22 до 60 мільярдів доларів в рік [9]. Вартість усунення дефекту, пропущеного на етапах розробки і тестування, може зрости після поставки програми від 2 до 100 разів [10].

Як наслідок, найбільш поширеними на сьогодні є інструменти, що базуються на обробці вихідного тексту програми – тобто методи статичного аналізу вихідного коду. Такі технології легко інтегруються в цикл розробки ПЗ, застосовуються на ранніх етапах розробки, причому однаково ефективно як під час нічних збірок на спеціалізованих віддалених стендах, так і безпосередньо на апаратурі розробника.

Достатньо популярними сьогодні також стають методи аналізу що використовуються при умові відсутності вихідного коду програми. Актуальність такого підходу багато в чому обумовлена високим рівнем поширення пропрієтарного програмного забезпечення, вихідний код якого є інтелектуальною власністю тієї або іншої компанії, які, як правило, не зацікавлені в публікації або передачі своїх розробок третім особам.

Також навіть при наявності вихідного коду існує проблема того, що перетворення виконувані компілятором і різними оптимізаторами можуть значно змінити поведінку програми, зробивши результати аудиту вихідного коду недостовірними. У іноземній літературі ця проблема носить назву «What You See Is Not What You eXecute». Для наочності розглянемо наступний приклад:

Перед тим, як звільнити динамічно виділений буфер пам'яті (в якому зберігається конфіденційна інформація), його вміст заповнюється нулями. Компілятор (в залежності від налаштувань оптимізації) може вирішити, що

значення, записані в буфер в результаті виконання функції memset, ніде далі не використовуються і видалить відповідний виклик. В результаті в купі протягом якогось часу буде зберігатися конфіденційна інформація, що є потенційною вразливістю.

```
memset(password, '\0', len);
free(password);
```

Рисунок 2.1 – Фрагмент програмного коду який потенційно може бути оптимізований компілятором.

Існують і інші класи систем виявлення дефектів у вихідному коді програм, але точність і необхідні для використання ресурси обмежують їх область застосування. Дані системи не отримали такого поширення, як згадані системи автоматичного пошуку дефектів на основі аналізу програмних кодів. Серед класів таких методів слід згадати наступні [7]:

- автоматизації експертного аудиту;
- верифікації вихідного коду;
- перевірка коректності дій користувача.

2.1 Аналіз вихідного коду програми

2.1.1 Визначення інструментів статичного аналізу вихідного коду

Так як аналізу вихідних текстів виконується без запуску програми на виконання, будемо називати такий процес як статичний аналіз коду – це технологія пошуку потенційних помилок в програмах шляхом розбору вихідних текстів програми та пошуку в ньому шаблонів відомих помилок. Інструменти, що реалізують цю технологію, називаються статичними аналізаторами коду. Найбільш відомими виробниками таких інструментів є компанії Coverity, Klocwork, Gimpel Software.

Слово «статичний» означає що код аналізується без запуску програми на виконання. Такі інструменти використовуються для пошуку програмних помилок ще на ранній фазі розробки проекту, зазвичай перед тим, як створюється виконуваний файл. Можливість такого використання особливо корисна для проектів вбудовуваних систем (Embedded Systems), так як на них розробники часто не можуть використовувати засоби динамічного аналізу та повномасштабного тестування до тих пір, поки програмне забезпечення не буде завершено настільки, щоб його можна було запустити на цільовій системі. Актуальними такі засоби також є і для інтерпретованих мов програмування (JavaScript, Python, PHP, Ruby). Так як процес розробки з використанням цих мов виключає фазу компіляції, інструменти статичного аналізу можуть використовуватись як для пошуку синтаксичних розбіжностей, так і більш складних помилок.

2.1.2 Перевірка абстрактного синтаксичного дерева

Статичний аналіз існує майже так само довго, як існує сама індустрія розробки програмного забезпечення. В своєму первозданному вигляді, такі інструменти представляли собою засоби, що перевіряли відповідність до стилю програмування, знаходили підозрілі конструкції, тощо. Першим таким відомим засобом є Lint (Linter) – розроблений в 1978 році Стівеном Куртісом Джонсоном, науковим співробітником Bell Labs, цей засіб був створений під час написання науковцем компілятора «уасс». Розробник використовував цей інструмент для виявлення потенційних помилок під час переносу своєї розробки на 32-бітну систему Unix [11]. Lint дозволяв знаходити наступні типи потенційних помилок, часто виникаючих через неуважність розробника:

- Ініціалізовані невикористовувані змінні (зараз такі типи перевірок часто вбудовані в компілятор) [12].

- Перевірку порядку виконання – інструмент знаходить «гілки» виконання програми, в яких немає умови виходу (нескінченні цикли, рекурсивні функції, завжди невірні умови типу 0 > 1, тощо). Такі перевірки дозволяють дослідити

логіку програми без використання знань про поведінку ПЗ в процесі виконання [12].

- Перевірка аргументів функції – деякі типи даних можуть неявно приводитись до типів з меншою місткістю [12]. Наприклад: виклик функції з параметрами char за допомогою аргументів типу int.

- Невизначений порядок обчислень – вирази типу f(i, j++) можуть мати неочікуваний для розробника результат [12].

Lint використовує один з найпростіших методів – перевірку абстрактного синтаксичного дерева. Абстрактне синтаксичне дерево (з англ. Abstract Syntax Tree, AST) – представлення вихідного коду в вигляді деревовидної структури, в якому кожна гілка є представленням вираз мови програмування. Таке представлення вихідного коду дає можливість інструментам аналізувати аномальні місця програми, наприклад: гілки з безумовним переходом, що конфліктують з іншими гілками; не використані в жодній із гілок коду об’єкти; використання не ініціалізованих змінних, тощо. В результаті еволюції комп’ютерних наук, конструкції абстрактних синтаксичних дерев пізніше стали основою для більш складних структур, таких як графи управління виконання програми, направлених ациклічних графів, таблиці умовних виразів та інших [13].

Для того, щоб до кінця зрозуміти як працює аналіз синтаксичних дерев, розглянемо для прикладу відому вразливість CVE-2014-1266, що була знайдена в компанії Apple в 2014 році і через особливість свого існування отримала назву «goto fail» [14]. Ця помилка в реалізації протоколу SSL/TLS дозволяла реалізовувати атаку типу Man in The Middle з підміною трафіку. Переглянемо частину відкритого вихідного коду, що став причиною дефекту в безпеці операційних систем iOS та збитків для компанії Apple та її клієнтів [15]. На рисунку 2.2 наведено фрагмент функції що перевіряє підпис, слід звернути увагу на два рядки «goto fail», які слідують один за одним:

```
{
        OSStatus err;
        ...
        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        ...
fail:
        …
        return err;
}
```

Рисунок 2.2 – Фрагмент коду програми, що спричиняє вразливість CVE-2014-1266

Перші інструменти аналізу використовували для своєї роботи технологію співставлення відомим шаблонам AST. На рисунку 2.3 представлено як інструмент статичного аналізу, в спрощеному для більшої ясності вигляді, розглядатиме цей уривок.

Тут використання зайвого виразу безумовного переходу може призвести до потенційно катастрофічних наслідків. Особливої уваги потребує об’єкт “err” ініціалізований значеннями по замовчуванню (зазвичай це 0 для числових типів данних та символ ‘\0’ для символьних типів), далі цей об’єкт стає результатом роботи функції і повертається за допомогою виразу “return” і використовуються кодом, що викликав цю функцію.

```
Statement block
    Expression-statement: initialize “err” with default constructor
    …
    If-statement
        Check-Expression
            …
        True branch
        Jump to “fail”-statement
    Jump to “fail”-statement
    …
    “fail”-statement
        …
     return “err”, which could be initialized with default values;
```

Рисунок 2.3 – Представлення вразливого фрагменту коду за допомогою технології AST.

Далі слідує помилка в порядку виконання функції – оператор goto немає виразу перевірки і стає безумовним, перевірки що слідують після виразу “goto” не виконуються. Отже, цей фрагмент вихідного коду не пройшов би перевірку порядку виконання аналізатора вихідного коду Lint, розробленого в 1978 році, що є одним із найпростіших і найстаріших інструментів аналізу.

Розглянутий вище приклад добре відображає те, що проста з точки зору синтаксису помилка, може призводити до катастрофічних наслідків для логіки та безпеки програми. Слід також сказати про те, що так як під час статичного аналізу інструмент намагається передбачити поведінку програми по моделі вихідного коду, то іноді виявляються помилки, якої фактично не існує. Помилки такого роду називають хибним висновком (false positive). Багато сучасних засобів статичного аналізу реалізують покращену техніку щоб уникнути таких помилок та надати максимально точний аналіз. Інструменти динамічного

аналізу, які будуть розглядатися в подальших розділах, в меншій мірі схильні до похибок, так як їх результат – висновки про виконання програми.

2.1.3 Аналіз кодових шляхів

Попередній досліджуваний метод є одним з найпростіших, тому він не позбавлений недоліків. Розглянемо більш складний приклад для того, щоб отримати уявлення про те, які синтаксичні конструкції аналіз синтаксичного дерева не дозволяє виявити як помилкові. Для дослідження візьмемо помилку в вихідному коді типу «Висяче посилання (Dangling pointer reference)». Нижче наведено простий приклад такого недоліку на мові програмування C++:

```
{
    char* pointer = nullptr;
    …
    {
        char data;
        pointer = &data;
    }
    ….
}
```

Рисунок 2.4 – Приклад помилки типу «Висяче посилання» в програмному коді.

Розглянемо що саме тут відбувається. Спочатку створюється змінна “pointer” та ініціалізується пустим значенням, далі починається нова область видимості, в якій створюється змінна типу char та кладеться на стек. Момент стекової змінної дуже важливий, так як такі дані знищуються по виходу з області видимості. Далі ми ініціалізуємо змінну “pointer” за допомогою посилання на стекову змінну та виходимо з області видимості. Тепер у нас закривається область видимості: всі дані, що знаходились на стекові знищуються і “pointer” тепер вказує на дані, що перестають існувати. Це означає, що нижче, при спробі використання змінної, виконання програми буде мати невизначений характер.

Використання висячого посилання на знищені дані далі в програмі стає причиною появи бінарної вразливості типу “Use-after-free”, розглянутого в попередньому розділі. Для прикладу, вразливість CVE-2014-1776 була знайдена в Internet Explorer 6, розроблюваною компанією Microsoft, в 2014 році, та була прикладом появи помилки висячого посилання [16].

У подібних випадках пошук по дереву шляхів або просте перерахування вузлів не зможуть виявити спробу використання покажчика. Причиною цього є те, що всі шляхи виконання програми є: безумовних переходів немає, всі змінні правильно ініціалізовані, порядок обчислені чітко визначений. Відповідно, інструмент аналізу не може просто проводити пошук по синтаксичної моделі. Необхідно також аналізувати життєвий цикл об'єктів даних у міру їх появи і використання всередині керуючої логіки в процесі виконання.

При аналізі кодових шляхів простежуються об'єкти всередині шляхів виконання, в результаті чого програми перевірки можуть визначити, наскільки чітко і коректно використовуються дані. Використання аналізу кодових шляхів розширює коло питань, на які можуть бути отримані відповіді в процесі проведення статичного аналізу. Замість того щоб просто аналізувати правильність написання коду програми, при аналізі кодових шляхів робиться спроба визначити "наміри" коду і перевірити, чи написаний код відповідно до цих намірами [17]. При цьому можуть бути отримані відповіді на наступні питання:

- Чи не був новостворений об'єкт звільнений перш, ніж з області видимості були видалені всі звертаються до нього посилання?

- Чи проводилася для деякого об'єкта даних перевірка дозволеного діапазону значень, перш ніж об'єкт передається функції ОС?

- Чи перевірявся рядок символів на наявність в ній спеціальних символів перед передачею цього рядка в якості SQL-запиту?

- Чи не призведе операція копіювання до переповнення буфера?

- Чи безпечно в даний час викликати цю функцію?

За допомогою такого аналізу шляхів виконання коду, як в прямому напрямку від запуску події до цільового сценарію, так і в зворотному напрямку від запуску події до необхідної ініціалізації даних, інструмент зможе дати відповіді на поставлені питання і видати звіт про помилки в разі, якщо цільовий сценарій або ініціалізація виконуються або не виконуються так, як очікується.

2.1.4 Переваги та недоліки статичного аналізу та його інструментів

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 і багато інших проблем. Враховуючи особливості інструментів аналізу, можна визначити наступні їх переваги:

- Результат аналізу не залежить від ступеня покриття коду вхідними даними – це означає, що імовірність виявлення вразливості не залежить від того, чи перейде потік управління програми по підозрілого елемента коду.

- Добре масштабуються – ці інструменти можуть бути використані на багатьох видах програмного забезпечення та може бути налаштований автоматичний повторний запуск (наприклад, при розподіленій компіляції програмного забезпечення вночі, коли більшість обчислювальних потужностей використовується як кластери; автоматичний запуск після модифікації коду в системі контролю версій).

- Дані інструменти з високою ймовірність автоматично ще на етапі написання програмного коду визначають такі вразливості, синтаксис яких добре досліджений (наприклад відсутність звільнення ресурсів, потенційно помилкове використання безумовних переходів, SQL-ін’єкції, тощо).

Перераховані вище переваги дозволяють використовувати інструменти статичного аналізу ще на етапі написання вихідного коду, що дозволить зменшити витрати на розробку та тестування під час впровадження системи. Слід також сказати і про основні недоліки розглянутого інструменту:

- Можуть виявлятися програмні помилки і уразливості, які не обов'язково призводять до відмови програми або впливають на поведінку програми під час її реального виконання.

- Реалізація та впровадження інструментів статичного аналізу є тяжким процесом та потребує високої кваліфікації.

- Інструмент має ненульову ймовірність хибних висновків.

2.2 Аналіз бінарного (виконуваного) коду програми

2.2.1 Визначення інструментів аналізу бінарного коду

Для аналізу бінарного коду використовуються методи статичного і динамічного аналізу. Як можна здогадатися, так само як і при статичному аналізі вихідного коду, бінарний код досліджується без виконання самої програми, а при динамічному аналізі, програма досліджується на підставі інформації, отриманої під час її виконання. Слід зазначити, що при аналізі бінарного коду можуть застосовуватися ті ж методи статичного аналізу, що і для вихідного коду. Однак їх застосування вимагає попереднього отримання графа викликів функцій програми і набору графів потоку управління для кожної функції. Деякі види аналізу потоку даних вимагають крім того побудови графа залежностей (за даними і залежності управління) [18].

2.2.2 Статичний аналіз бінарного коду програми

У разі аналізу вихідного коду програма представлена на мові високого рівня, що дозволяє досить просто отримати уявлення про структуру у вигляді набору графів: виклики функцій здійснюються по іменах, передача управління по міткам або умовним переходам, всюди використовуються змінні і їх типи описані явно. Єдиним винятком є виклик функцій за вказівником, при якому не завжди можна сказати, яка функція (або їх набір) викликається в даній точці програми. Але і в цьому випадку доступний прототип (сигнатура) функції що викликається, це в свою чергу часто дозволяє істотно звузити набір потенційних функцій що можуть бути викликаними. У бінарному коді відсутні змінні і типи

даних в явному вигляді - замість цього інструкції оперують регістрами і елементами пам'яті, відсутні і об’явлення прототипів функцій і їх меж в коді [18].

Слід зауважити, що в даній роботі розглядаються програми, бінарний код яких займають фіксовані області пам’яті, і ті, код яких не може самомодифікуватися. На сьогодні до такого класу програм відносяться довгий список програм, які отримуються за допомогою мов C/C++, а також похідні від них.

Виконуваний файл містить області коду і статичних даних програми, таблиці функцій, імпортованих зі сторонніх динамічно завантажуваних бібліотек, а також список функцій, що експортуються програмою, в разі, якщо програма сама є бібліотекою. Область коду містить послідовність функцій програми, яка не є неперервною, так як між функціями можуть виникати невикористовувані ділянки пам'яті внаслідок вирівнювання. Крім того, багато компіляторів вставляють дані функції поряд з її кодом (статичні дані). Бібліотечні функції, які при збірці програми були статично пов'язані з нею, зберігаються в її коді і не відрізняються від функцій програми [18].

Одним з поширеніших і найбільш розвинутих засобів статичного аналізу бінарного коду є система інтерактивного IDA Pro [19], що підтримує широкий ряд бінарних форматів і процесорних архітектур. При аналізі виконуваних файлів IDA Pro дозволяє проводити їх автоматичний аналіз. При цьому виконується:

- виділення таблиць імпортованих і експортованих функцій;

- виділення меж статичних змінних в області даних;

- виділення параметрів функцій в стеку і на регістрах, а також локальних змінних функції і значення, що повертається;

- побудова графа викликів і графа потоку керування за рахунок аналізу адрес переходів;

Однак в деяких випадках статичний аналіз виконати складно або неможливо. Як приклад розглянемо програму, забезпечену захистом від аналізу: код програми в виконуваному файлі знаходиться в зашифрованому вигляді і

розшифровується в процесі виконання. В цьому випадку можна спробувати починати статичний аналіз після запуску програми, що в свою чергу наштовхує на думку про використання статичного та динамічного аналізу в поєднанні.

Статичний аналіз може бути ускладнений і через відсутність коду динамічно завантажуваних бібліотек в виконуваному файлі. Зокрема, можуть бути відсутні компоненти ОС, з якими взаємодіє програма. Це може перешкодити виявленню шкідливого коду, одним із способів впровадження якого є перезапис початку однієї або декількох системних функцій з метою вставки інструкції переходу на шкідливий код [18].

2.2.3 Динамічний аналіз бінарного коду програми

Динамічний аналіз програми – технологія пошуку дефектів в програмному забезпеченні, що здійснюється шляхом аналізу стану ресурсів процесу в режимі реального часу. По аналогії з засобами статичного аналізу, інструменти, що реалізують цю технологію, отримали назву динамічних аналізаторів програм [20].

Ключовою відмінністю засобів динамічного аналізу від статичних аналізаторів є те, що такий процес обов’язково потребує запуск аналізованого коду на виконання. Одним з переваг цього підходу є відсутність будь-яких припущень про те, що відбувається під час виконання програми, і те, що всі необхідні вимоги перевіряються під час або відразу після закінчення роботи. Це в свою чергу дозволяє зменшити об’єм ресурсів, що необхідно витрачати на тестування. З іншого боку, якщо деяка частина програми не виконується, інструмент динамічного аналізу не зможе дати висновки про правильність цього фрагменту. При виявленні дефекту в процесі динамічного аналізу, як правило, можна згенерувати вхідні дані для програми, на яких помилка відтворюється.

Для виявлення програмних помилок в інструментах динамічного аналізу часто проводиться вставка незначних фрагментів коду в вихідний текст програми або в виконуваний (об’єктний код). При цьому, однією із головних вимог, що накладаються на такі засоби є те, що проведення аналізу повинно мати

як можна менший вплив на хід виконання [21]. Такі методи також можуть мати значний вплив на продуктивність програми, тому їх не рекомендується використовувати під час фінального тестування. В вставлених кодових сегментах виконується перевірка стану програми і формується звіт про помилки, якщо виявляється якась некоректна поведінка. Основними технологіями динамічного аналізу на сьогодні є:

- Розміщення вставок в вихідний код на етапі препроцесорної обробки – в текст програми до запуску компіляції вставляється спеціальний фрагмент для виявлення вразливостей [21]. При такому підході не потребуються детальні знання про середовище виконання, в результаті чого такий метод часто використовується в програмах, які до свого випуску не можуть бути запущені на цільових системах.

- Розміщення вставок в об’єктний код – такий метод потребує достатніх знань про середовище виконання, щоб мати можливість вставляти код безпосередньо в виконуваний файли та бібліотеки [21]. Перевагою такого підходу є те, що не потрібно мати доступ до вихідного коду чи проводити перекомпоновку програми.

- Вставка коду під час компіляції – розробник використовує спеціальні ключі (опції) компілятора для впровадження в вихідний код. Використовується особливість компілятора виявляти помилки.

- Трансляція та спостереження за програмою без вставок в код – такі інструменти не використовують вставок в програмний код, а виконується перевірка роботи в тому ж середовищі що й при звичайному виконанні [21].

- Спеціалізовані бібліотеки етапу виконання – при використанні такого методу для аналізу використовуються модифіковані версії системних бібліотек [21]. Метод не потребує внесення змін до коду програми щоразу при компіляції. В попередньо підготованих бібліотеках додається спеціальний код, ціллю якого є перевірка достовірності вхідних параметрів в системних викликах. При цьому не можливо виявити помилки, що не використовують системні функції (наприклад, використання неініціалізованого посилання).

Наведений вище опис дає зрозуміти, що основною відмінністю технологій аналізу є місце перевірки правильної роботи програми. Спостереження може бути вбудованим в код програми, додатковий модуль що слідкує за ресурсами системи або в функції системних викликів.

Спроба генерації всіх можливих вставок в код програми може призводити до експоненційного росту кількості таких фрагментів при збільшенні об'єму вихідного коду. Це в свою чергу накладає певні вимоги, такі як той факт, що сам аналіз і необхідні для цього дії повинні надавати мінімальний вплив на хід виконання. Також однією з умов аналізу програми є детермінованість продукту, що дозволяє уникнути однієї з основних проблем в аналізі - помилкових спрацьовувань. Ця умова означає, що на кожному кроці виконання стан програми повністю і однозначно залежить від її стану на попередньому кроці. Тому будь-який наступний крок буде залежати від попереднього. Будь-яка інформація, яка буде використовуватися в контексті аналізованої програми, будь то змінні або вхідні дані, які використовуються в програмі, є дискретної і остаточною. Оскільки всі ітерації аналізу коду залежать виключно від вхідних даних, як зазначено вище, необхідно формалізувати, як дані будуть відбиратися для введення і т. п.

Однак динамічний аналіз має деякі недоліки. Одним з них є те, що для отримання дійсно якісного покриття вихідного коду аналізованої програми, фактично, потрібно не один запуск програми, а кілька повторів запуску з різними вхідними даними, що тягне за собою чимало часу витрати. Але якщо під час динамічного аналізу виявлена помилка, завжди можна відновити вхідні дані для програми, на яких ця помилка буде відтворена в майбутньому. Таким чином, ви можете уникнути помилкових спрацьовувань аналізатора.

2.2.4 Переваги та недоліки засобів динамічного аналізу

Технології динамічного аналізу використовуються для пошуку дефектів (в тому числі й бінарних вразливостей) в розроблюваному програмному забезпеченні. Головною особливістю таких інструментів є необхідність запуску

аналізованої програми для виконання. Враховуючи наведені вище особливості інструментів аналізу, можна визначити наступні їх переваги:

- Низька ймовірність виникнення хибних спрацьовувань, так як виявлення помилки відбувається в момент її виникнення в програмі; таким чином, виявлена помилка є не припущення, зробленим на основі аналізу моделі програми, а констатацією факту її виникнення;

- Для відстеження причини помилки може бути проведена повне дослідження стека і середовища виконання.

- Для тестування часто не потрібно мати доступ до вихідного коду програми – це дозволяти використовувати інструменти динамічного аналізу на етапах розробки, що йдуть після фази написання вихідного коду.

- Захоплюються помилки в контексті діючої системи в реальних умовах - це дозволяє знаходити дефекти, що зазвичай складно без запуску програми на виконання на цільовій системі.

Описані вище переваги є серйозними аргументами для використання інструментів в повсякденній роботі. Деякі особливості засобів динамічного аналізу дозволять зменшити негативний вплив засобів статичного аналізу на процес розробки ПЗ (наприклад, можна зменшити кількість хибних висновків при використанні засобів в комплексі).

Слід також розглянути й основні недоліки інструментів динамічного аналізу:

- Відбувається втручання в поведінку системи в реальному часі; ступінь втручання залежить від кількості використовуваних інструментальних вставок.

- Повнота аналізу помилок залежить від ступеня покриття коду. Таким чином, кодовий шлях, що містить помилку, повинен бути обов'язково пройдено, а в контрольному прикладі повинні створюватися необхідні умови для створення помилкової ситуації.

3 АНАЛІЗ ВИМОГ ДО СИСТЕМИ КОМБІНОВАНОГО АНАЛІЗУ

Проаналізувавши дані наведені в попередніх розділах, можна зробити наступний висновок щодо наявних на сьогодні підходів до аналізу програм. Технології пошуку дефектів (вразливостей і помилок) при створенні програмного забезпечення розробляються за трьома основними напрямками:

- По-перше, використовуються технології пошуку дефектів за допомогою статичного аналізу вихідного коду програм. Такі інструменти не потребують запуску програми, що дозволяє використовувати їх на самих ранніх етапах розробки [7].

- По-друге, застосовуються системи динамічного аналізу бінарного коду програм. Це передбачає запуск програми на деякому наборі вхідних даних і відстеження ситуації виникнення дефектів. Ефективність такого аналізу напряму залежить від набору вхідних даних, але при знаходженні помилки відразу дозволяють отримати дані, на яких ця помилка проявляється (тобто не мають помилкових спрацьовувань) [21].

- По-третє, займають своє місце засоби статичного аналізу бінарного коду. Подібно до аналізу вихідного коду, для обробки бінарного коду непотрібно запускати програму на виконання. Недоліком таких інструментів є те, що вони передбачають складні перетворення низького рівня, що означає роботу з асемблерним кодом. Але такий підхід дозволяє будувати та оцінювати графи виконання програми на більш детальному рівні, ніж при аналізі вихідного коду. Ефективно та швидко можна знаходити та виправляти ситуації WYSINWYX («What You See Is Not What You eXecute») [18].

Для того щоб сформулювати основні проблеми, що потрібно вирішити, та висунути вимоги до використання інструментів аналізу, необхідно оцінити основні переваги та недоліки засобів. Для наочності, порівняння підходів до аналізу будемо проводити за допомогою схеми, зображеної на рисунку 3.1.

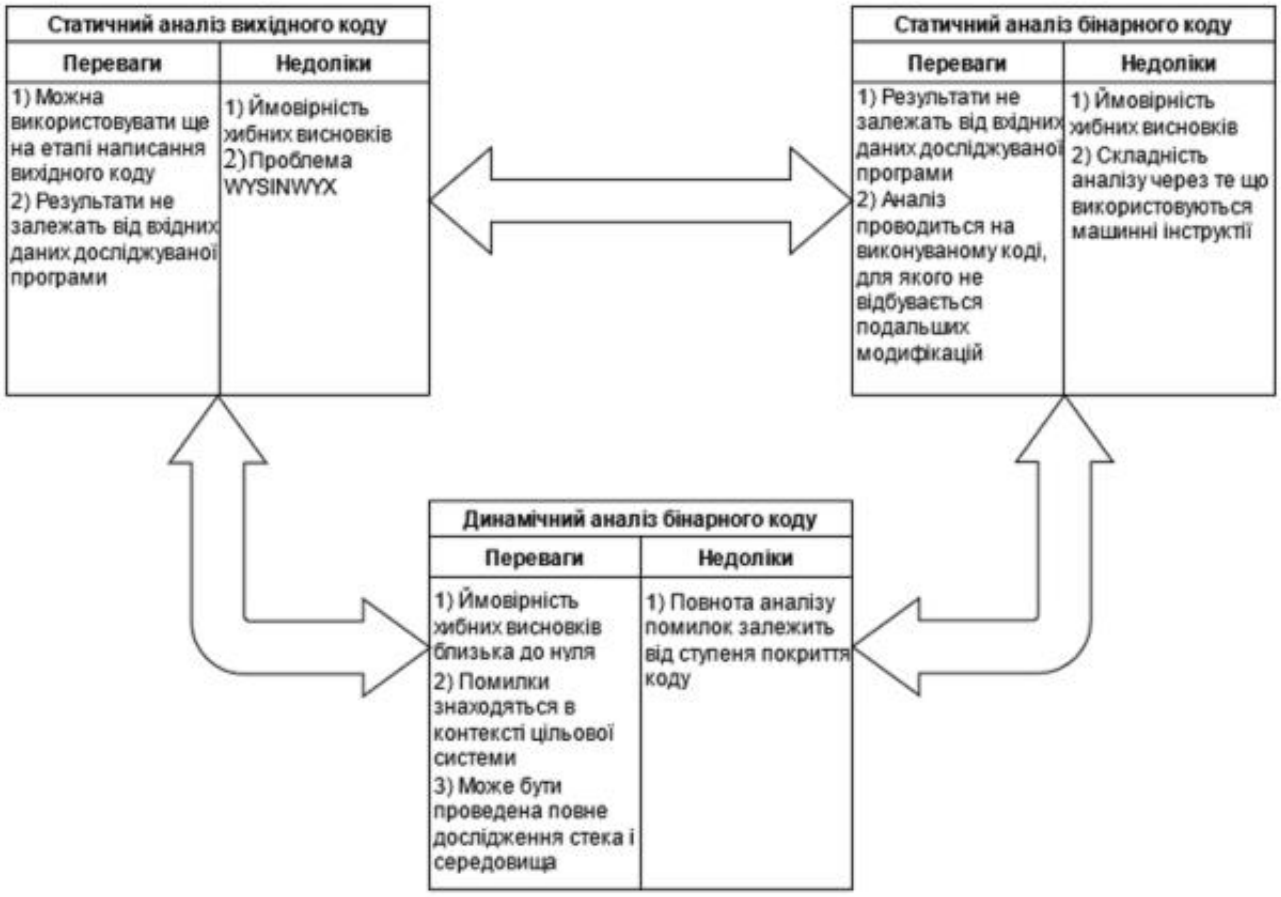

Рисунок 3.1 – Порівняння переваг та недоліків підходів до аналізу вразливостей програм.

Основною проблемою інструментів статичного аналізу є ймовірність хибних висновків, що призведе до росту звіту про аналіз та збільшення часу необхідного на його обробку. В свою чергу такий засіб можна використовувати ще на етапі написання вихідного коду. Така особливість дозволить ефективно виконувати компіляцію програми та статичний аналіз програми паралельно. Якщо подивитись на інструменти динамічного аналізу, то їх найважливішим недоліком є необхідність формування вхідних даних для програми. Так як зазвичай експлуатація вразливості передбачає введення в програму граничних значень, часто дефект може бути пропущений поза увагою через недостатнє покриття вхідними даними для аналізу [22].

- В свою чергу, аналіз вихідного коду не дозволить виявити проблеми, що можуть виникнути внаслідок внесення компілятором змін в потік виконання

програми для оптимізації роботи [7]. Для уникнення вище описаної ситуації пропонується використовувати порівняння результатів отриманих під час аналізу вихідного та об’єктного коду [22].

Таким чином, основними проблемами які необхідно вирішити для підвищення ефективності аналізу програм є:

- проблема відсіювання хибних висновків статичного аналізу – може бути вирішена шляхом додаткової обробки звітів про проблеми інструментами динамічного аналізу;

- проблема покриття програми тестовими сценаріями – пов’язана з попередньою проблемою. Пропонується використовувати статичний аналіз бінарного коду, що виконується на внутрішньому представленні програми та не потребує запуску програми на виконання та отримання тестових даних;

- проблема невідповідності вихідного коду реальному бінарному коду програми – пропонується використовувати порівняння результатів отриманих під час аналізу вихідного та об’єктного коду [22].

Окрім описаних вище проблем виникає ще одна – проблема обробки великої кількості даних [8]. Очевидно, що перегляд да оцінка результатів аналізу може займати значний час у розробника або аналітика. Для вирішення таких проблем на сьогодні найпопулярнішим є підхід створення автоматичного середовища, яке дозволяло б відсіювати дані що не несуть реальної загрози.

Розглянувши ключові недоліки інструментів аналізу окремо та сформулювавши основні проблеми що потрібно вирішити, можна сформулювати наступні вимоги до алгоритму роботи з інструментами аналізу:

1) Система повинна дозволяти відсіювати висновки, що не несуть небезпеки під час реальної роботи програми.

2) Необхідно забезпечити якомога краще покриття програми тестовими сценаріями.

3) Створений алгоритм повинен надавати можливість порівнювати результати аналізу на основі вихідного та бінарного коду.

4) Передбачається, що алгоритм буде використовуватися на ранніх етапах розробки ПЗ, тому виправлення буде найдешевшим.

5) Так як розробник взаємодіє з вихідним кодом програми, необхідно щоб представлення результатів надавалось в вигляді перетворень високого рівня, уникаючи машинних інструкцій [22].

Таким чином, після того як було сформовано основні проблеми та висунуто основні вимоги до засобу що їх вирішує, перейдемо до формування загального алгоритму роботи з інструментами аналізу програм.

4 ТЕХНІКА КОМБІНОВАНОГО АНАЛІЗУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Проаналізувавши дані наведені в попередніх розділах, стає зрозуміло, що бінарні вразливості в програмному забезпеченні є серйозною проблемою, з якою зустрічаються розробники в процесі своєї роботи. В той самий час, розглянувши особливості дефектів, стає зрозуміло, що для виявлення можна використовувати вихідний або бінарний код програми. В попередньому розділі було сформовано основні проблеми та висунуто вимоги до засобу комбінованого аналізу програм. Передбачається, що основним завданням такого засобу є підвищення ефективності пошуку дефектів в програмному забезпеченні [20]. В якості такого засобу пропонується розглянути аналітичну платформу, яка дозволить поєднати, підкреслити переваги та усунути недоліки підходів до аналізу вихідного та бінарного коду програм [22].

У даному розділі описується аналітична платформа, що дозволяє комбінувати аналіз бінарного та вихідного коду, забезпечуючи подолання обмежень кожного підходу. Як було сказано раніше, для того щоб зменшити час необхідний розробнику на ознайомлення з результатами аналізу, передбачається, що система має бути автоматичною, тобто розробнику буде надано лише кінцевий перевірений в процесі роботи список проблем.

Загальну схему роботи аналітичної платформи, що створюється в даній роботі, можна коротко описати наступним чином. Спочатку розробник вносить зміни до вихідного коду програми, це в свою чергу може внести потенційні дефекти до розроблюваної програми. Після цього, розробник запускає компіляцію програми. Передбачається, що паралельно з цим запускається процес статичного аналізу вихідного коду. Після отримання виконуваного файлу розробник може перейти до аналізу бінарного коду. Для цього необхідно використати інструмент що поєднує статичний та динамічний підходи. Після отримання результатів проводиться їх автоматичне порівняння на предмет виявлення хибних висновків. В результаті роботи з платформою розробник

отримує записи щодо підозрілих областей вихідного коду та може швидко усунути потенційний дефект. Більш детально схема роботи комбінованого аналізу зображена на рисунку нижче.

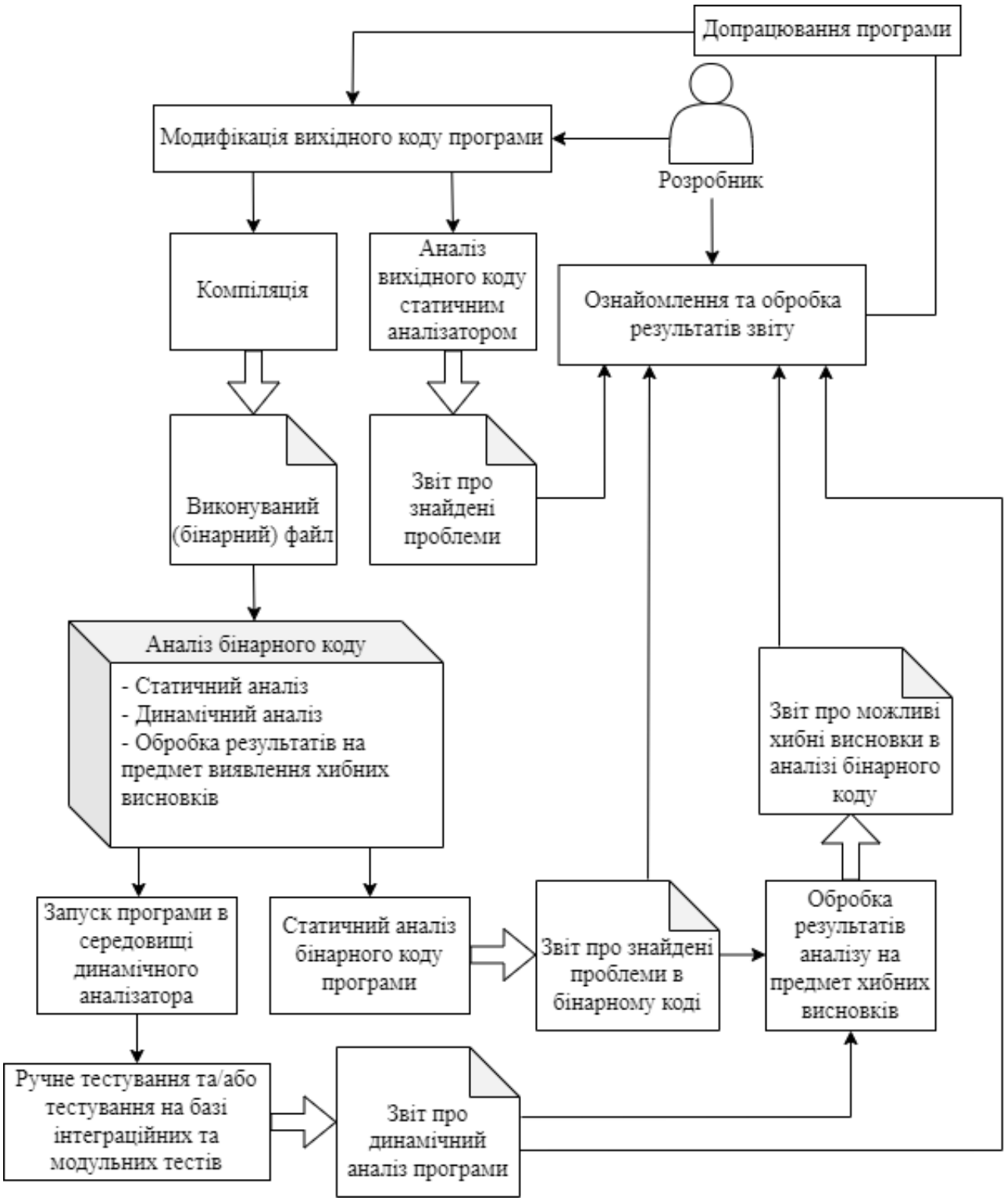

Рисунок 4.1 – Схема аналітичної платформи для комбінованого аналізу.

Таким чином, переглянувши схему що реалізує комбінований аналіз програми, можна перерахувати наступні основні етапи взаємодії з аналітичною платформою:

1) Модифікація вихідного коду.

2) Статичний аналіз вихідного коду (виконується паралельно з компіляцією) – результатом є звіт про знайдені проблеми.

3) Отриманий виконуваний файл запускається на виконання в середовищі аналізу бінарного коду. Паралельно запускається статичний аналізатор бінарного коду що створює граф на основі машинних інструкцій [20], проводить його перевірку та надає дані про знайдені вразливості. Ці дані порівнюються з результатами динамічного аналізу, в результаті такого порівняння виноситься рішення про потенційні хибні висновки в програмі [22].

4) Якщо статичний аналіз не знаходить підозрілих областей, виконується ручне та/або тестування на базі інтеграційних та модульних тестів.

5) Результатом роботи є звіт про аналіз бінарних файлів програми, звіт про потенційні хибні висновки та звіт про статичний аналіз вихідного коду [22].

Наведена вище послідовність дій задовольняє основним вимогам, сформованим в попередньому розділі: відсіювання відбувається на етапі динамічного аналізу, статичний аналіз забезпечує більше покриття, а використання такого підходу комбінованого аналізу дозволить виявляти дефекти ще на ранніх етапах розробки [22]. Передбачається що створена платформа буде використовуватись розробниками на персональних комп’ютерах. У звіті по аналізу наводяться виявлені дефекти поряд з інформацією, яка допоможе в їх усуненні, в тому числі посилання на вихідний код. Після цього розробником проводиться виправлення фактично присутніх помилок.

Розробники також можуть використовувати таку систему на спеціальних кластерах, що використовуються під час нічної компіляції. Розробник передає зміни в систему управління версіями, після цього змінений код бере участь в процесі подальшої нічної збірки. На початку робочого дня розробник переглядає звіт про результати нічного компонування. В даний звіт включаються як власне

помилки компіляції, так і результати аналізу, проведеного під час компіляції та аналіз проведений під час автоматичного тестування програми.

Підсумуємо, що в даному розділі було запропоновано схему взаємодії розробника з комбінованим аналізом ПЗ та виділено ключові етапи роботи. Далі пропонується за наведеними теоретичними даними розробити інструмент, що реалізує ключові ідеї комбінованого аналізу та дозволяє зробити висновки щодо перспектив використання комбінованого аналізу ПЗ.

5 РЕАЛІЗАЦІЯ ІНСТРУМЕНТУ КОМБІНОВАНОГО АНАЛІЗУ

Як було розглянуто в попередніх розділах, вразливості в програмному забезпеченні є серйозною проблемою, з якою зустрічаються розробники в процесі своєї роботи. В той самий час, розглянувши особливості дефектів, стає зрозуміло, що для виявлення можна використовувати вихідний або бінарний код програми. Для вирішення цих завдань запропоновано розглянути підхід комбінованого аналізу програми на вразливості. В даному розділі спробуємо реалізувати інструмент для проведення комплексного аналізу програм, використовуючи основні вимоги, завдання та ідеї, сформовані в попередніх розділах.

Як було описано в попередніх розділах, ефективне використання інструментів аналізу програм передбачає:

1. Відсіювання висновків, що не несуть небезпеки під час реальної роботи програми;

2. Вхідні дані для програми повинні покривати якомога більше потенційно загрозливих сценаріїв.

3. Передбачається використання на як можна ранніх етапах розробки ПЗ, так як виправлення дефектів на цих етапах буде найдешевшим.

4. Так як розробник взаємодіє з вихідним кодом програми, необхідно щоб представлення результатів надавалось в вигляді перетворень високого рівня;

5. Для покращення ефективності аналізу та зменшення часу необхідного на ознайомлення з інструментом передбачається використання графічного інтерфейсу.

Таким чином, переглянувши наведені ключові ідеї аналізу, пропонується реалізувати інструмент, що буде являти собою аналітичну платформу для комбінованого аналізу вразливостей в програмному забезпеченні. Для реалізації передбачається використовувати наступні технології:

1. Мова програмування C++.

2. Система збірки CMake.

3. Інструмент розробки графічного інтерфейсу користувача Qt.

4. Інструмент статичного аналізу вихідного коду CppCheck.

5. Інструмент статичного аналізу бінарного коду CWE Checker.

6. Інструмент динамічного аналізу бінарного коду Valgrind.

Перейдемо до детального розгляду розроблюваного інструменту.

5.1 Загальна характеристика розроблюваного інструменту комбінованого аналізу

В рамках даної роботи пропонується реалізувати прототип (так званий Proof of Concept) проекту, що включає в себе всі перераховані вище інструменти та слідує ключовим ідеям аналітичної платформи, що була запропонована в попередньому розділі.

Перед програмним додатком ставляться наступні завдання:

1. Аналіз вихідного програмного коду, бінарного коду програми та виконуваного файлу на наявність в ньому основних вразливостей, що були описані в рамках даної роботи.

2. Відображення результатів аналізу в зрозумілому для користувача вигляді.

3. Можливість знаходити потенційні хибні висновки в аналізі.

4. Реалізація частини користувача засобу, де користувач має наступні можливості:

4.1. Вибір директорії з вихідним кодом

4.2. Вибір виконуваного файлу

4.3. Можливість налаштування конфігурації для аналізу: ввімкнення асинхронного виклику процесів, ввімкнення/вимкнення окремих інструментів аналізу.

4.4. Відображення прогресу аналізу та часу що був витрачений на аналіз (що дозволить приймати рішення про найоптимальніший час для

проведення аналізу, наприклад запускати найбільш часозатратний процес на окремих кластерах в нічний час).

В якості мови програмування для розроблення проекту був вибраний C/C++, так як дана мова програмування дозволяє легко поєднувати виклики дочірніх процесів на низькому рівні та створення інтерфейсу користувача з низьким часом відклику. Графічний інтерфейс реалізований за допомогою бібліотеки QtWidgets, а логіка що пов’язана з асинхронним виконанням програми використовує QtCore [23]. Також для розгортання та налаштування компонентів третіх сторін, що були описані в попередніх параграфах, вирішено використовувати скрипти збірки створені за допомогою CMake [24].

Програмний засіб складається з наступних основних компонентів:

1. GUI – модуль взаємодії користувача з графічним інтерфейсом. Даний компонент також інкапсулює в себе функціонал виклику аналізаторів в паралельних потоках, що дозволить виконувати аналіз в повній мірі використовуючи обчислювальні потужності комп’ютера.

2. ExternalDependencies – скрипти розгортання інструментів аналізу вихідного та бінарного коду. Дочірніми є компоненти третіх сторін:

2.1. CppCheck – компонент статичного аналізу вихідного коду.

2.2. CWE Checker – компонент статичного аналізу бінарного коду.

2.3. Valgrind – компонент динамічного аналізу бінарного коду.

3. AnalysisCaller – програмний інтерфейс виклику дочірніх процесів для аналізу, що були розгорнуті в рамках компоненту ExternalDependencies.

На рисунку 6.1 представлена UML-діаграма компонентів розроблюваного проекту. З діаграми видно, що інструмент має багатошарову структуру, що дозволяє вносити правки до компонентів незалежно один від одного, а також можна інкапсулювати роботу з різними етапами спрощуючи структуру проекту. Роботу можна умовно поділити на три етапи.

На першому етапі користувач (програміст) використовує графічний інтерфейс, що дозволяє налаштувати інструмент під свої потреби. Здійснюється вибір вихідних текстів аналізованої програми та вказується шлях до

виконуваного файлу на файловій системі. Також користувач повинен вказати які види аналізу повинен провести інструмент, і вказується, чи потрібно створювати окремий асинхронний виклик для кожного виду аналізу.

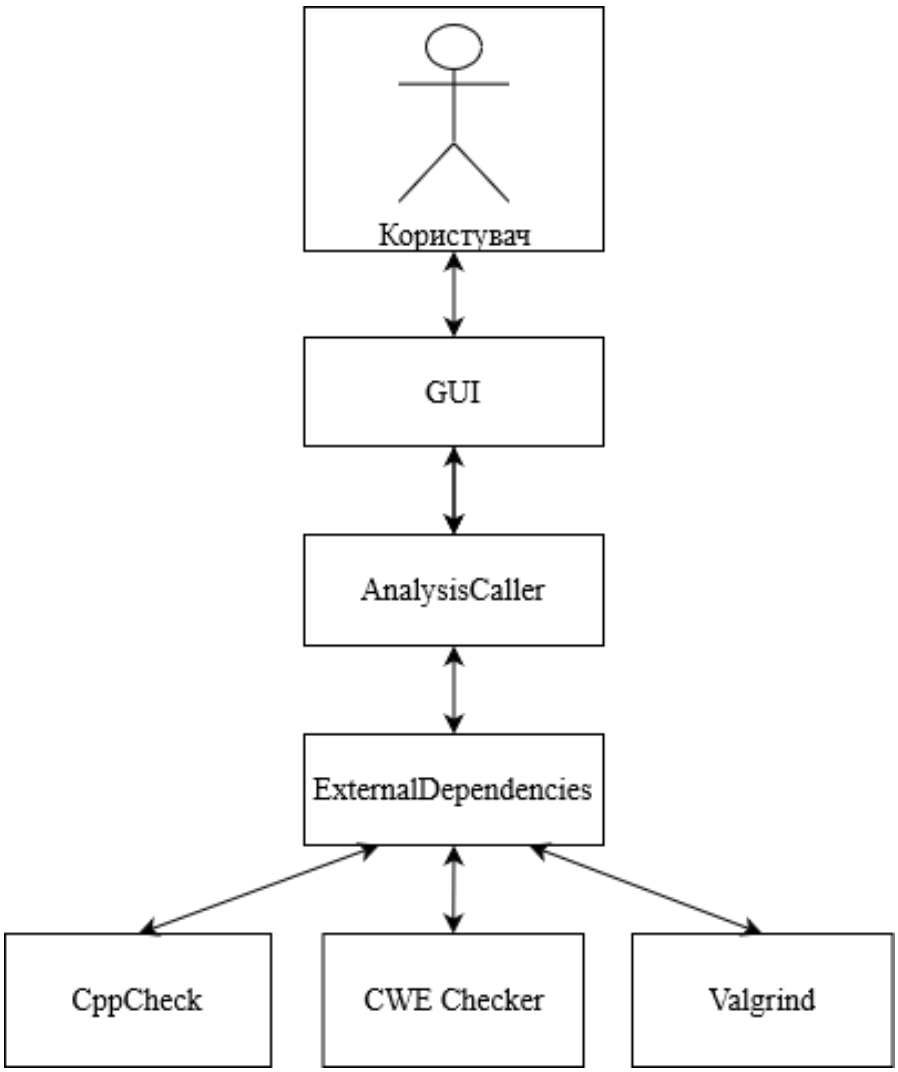

Рисунок 5.1 – UML-діаграма компонентів розроблюваного інструменту комбінованого аналізу.

Після запуску інструменту починається другий етап. На цьому етапі додаток, використовуючи налаштування користувача, запускає сам аналіз. Для цього використовується програмний інтерфейс AnalysisCaller, що реалізує виклики до компонентів ExternalDependencies. Для кожного інструменту також створюється підписка на сигнал завершення аналізу, щоб дані були представлені користувачу одразу після завершення процесу та без затримок.

Одразу по завершенні всіх створених процесів розпочинається третій етап – обробка результатів аналізу. Після того як графічний інтерфейс отримує дані про знайдені вразливості, виконується їх відображення для користувача. По-перше, для зручності інтерфейс має окремі поля для виводу даних про аналіз вихідного та бінарного коду програми. По-друге, користувачу виводиться час в мілісекундах, що був витрачений на проведення аналізу. По-третє, проводиться обробка результатів на предмет помилкових спрацьовувань, якщо результати статичного та динамічного аналізу виконуваного файлу будуть відрізнятися, користувач отримає відповідне попередження.

Закінчивши розгляд загальної структури інструменту, розглянемо кожен з компонентів більш детально.

5.2 GUI – компонент графічного інтерфейсу користувача розроблюваного проекту комбінованого аналізу

Як було вказано вище, для реалізації графічного інтерфейсу використовується бібліотека QtWidgets [23]. Для того щоб зробити інтерфейс якомога більше інтуїтивно зрозумілим, було здійснено розбиття на наступні логічні блоки:

1. Налаштування аналізу вихідного коду програми – складається з графічного предиката для запуску аналізу вихідного коду програми та поля для вибору шляху до вихідних текстів. .

2. Налаштування аналізу бінарного коду програми – складається з двох графічних предикатів для запуску статичного та/або динамічного аналізу бінарного коду програми; та поля вибору шляху до виконуваного файлу.

3. Налаштування асинхронного запуску процесів – складається з графічного предикату що визначає необхідність запуску інструментів в окремих потоках.

4. Вивід результатів роботи інструменту – складається з окремих полів для виводу результатів аналізу вихідного коду та аналізу бінарного коду. Також

містить поле, що відображає час витрачений на аналіз і поле з попередженням про потенційні хибні висновки в аналізі.

5. Статус роботи інструменту – складається з графічного відображення прогресу аналізу та кнопки запуску інструмента.

Загальне розташування блоків описаних вище блоків зображено на рисунку 5.2.

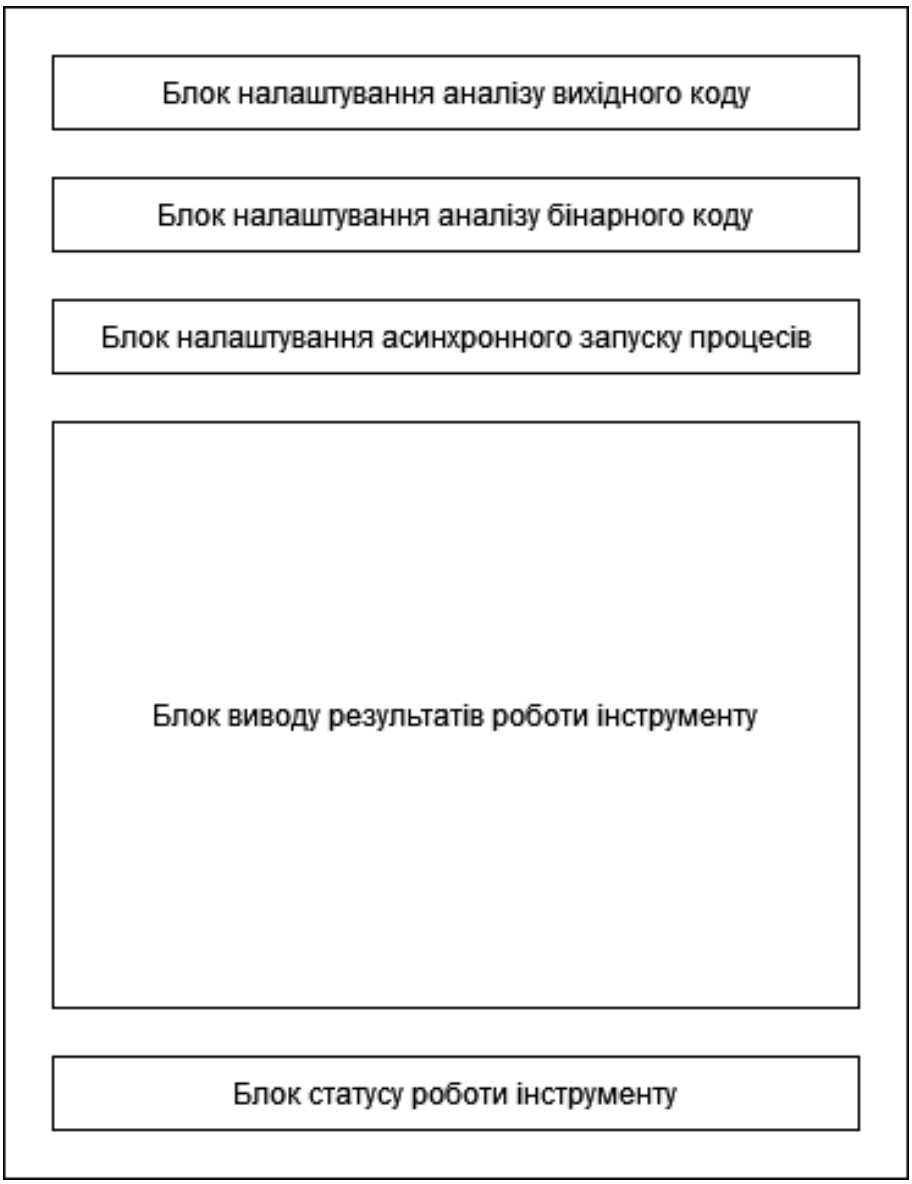

Рисунок 5.2 – Загальне розташування блоків на графічному інтерфейсі користувача.

Програмний код графічного інтерфейсу реалізований в рамках класу MainWindow, в якому відповідно до кожного з блоків графічного інтерфейсу

було реалізовано метод їх створення (рисунок 5.3). Повний програмний код цих методів наведено в додатку Б.

```
void setupUI();
QWidget* createSourcesAnalysisWidgetsGroup();
QWidget* createBinariesAnalysisWidgetsGroup();
QWidget* createRunAsyncAnalysisPredicateWidget();
QWidget* createOutputWidgetsBlock();
QWidget* createAnalysisRunningStateWidget();
```

Рисунок 5.3 – Програмні методи створення блоків графічного інтерфейсу.

Оскільки клас MainWindow містить налаштування по асинхронному запуску процесів, вирішено також в цьому класі помістити і саму роботу по створенню та контролю цих процесів. Для цього було реалізовано два допоміжних класи Operation та OperationsPool. Як можна здогадатися з назви, перший клас являє собою програмний компонент для зручного виконання окремих операцій. Повний програмний код класу наведено в додатку Б. Він приймає на вхід саму операцію та параметр що визначає, чи потрібно проводити виконання в окремому потоці. Основним методом цього класу є run, програмний код якого для наочності наведено на рисунку 5.4. Коротко цей метод можна описати так: відбувається виконання операції та нотифікація за допомогою сигналу finished [23].

Клас OperationsPool також є допоміжним програмним засобом для роботи з операціями, але цей клас працює с їх колекцією, та використовується для нотифікації про завершення всіх операцій за допомогою сигналу allOperationsFinished [23]. Повний програмний код цього класу також наведено в додатку Б.

Клас MainWindow також має метод performAnalysis, що викликається при натисканні кнопки запуску інструменту. Як можна здогадатися з назви, цей метод реалізує сам виклик процесів аналізу через компонент AnalysisCaller. Для цього використовуються допоміжні класи Operation та OperationsPool, при чому

їх конфігурація залежить від налаштувань користувача. Крім цього, при натисканні на кнопку запуску здійснюється підписка на завершення всіх операцій, щоб вивести результати роботи в відповідному блоці графічного інтерфейсу.

```
void run()
{
    if (_runAsync)
    {
        QThread *thread = new QThread();
        moveToThread(thread);

        connect(thread, &QThread::started, this, [this, thread] {
            // execute
            _result = _operation();
            thread->quit();
            emit finished(_result);
            thread->deleteLater();
            deleteLater();
        });

        thread->start();
    }
    else
    {
        _result = _operation();
        emit finished(_result);
        deleteLater();
    }
}
```

Рисунок 5.4 – Програмний код методу run класу Operation.

Підсумовуючи розгляд, можна виділити наступні завдання що виконує компонент:

1. Взаємодію користувача (розробника) з налаштуваннями інструменту.

2. Інкапсуляція роботи з компонентом AnalysisCaller.

3. Створення та управління асинхронних викликів, необхідних для проведення аналізу.

4. Інтерпретація та відображення результатів в відповідному блоці інтерфейсу.

Розглянувши компонент GUI, перейдемо до розгляду компоненту AnalysisCaller, від якого прослідковується безпосередня залежність.

5.3 Компонент AnalysisCaller

Програмний код реалізовано за допомогою мови програмування C++. Крім цього, використовується бібліотека QtCore, що дозволяє викликати та отримувати вивід дочірніх процесів. Компонент AnalysisCaller складається з одного класу, що має таку ж назву, та допоміжних функцій, що власне реалізують створення та контроль процесів статичного аналізу вихідного коду, статичного аналізу бінарного коду та динамічного аналізу бінарного коду. Програмний інтерфейс класу наведено на рисунку 5.5. Повна реалізація класу наведена в додатку Б. Як можна побачити з наведеного рисунка, інтерфейс є інтуїтивно зрозумілим та складається з трьох методів для кожного з видів аналізу. Кожен метод приймає рядок, що вказує на шлях на файловій системі до цільового об’єкту аналізу. Метод повертає рядок, що містить результати проведення аналізу для кожного з видів.

```
class AnalysisCaller {
public:
    std::string callStaticSourcesAnalysis(std::string forPath);
    std::string callStaticBinaryAnalysis(std::string forPath);
    std::string callDynamicBinaryAnalysis(std::string forPath);
};
```

Рисунок 5.5 – програмний інтерфейс класу AnalysisCaller.

Допоміжні функції, що реалізовані в рамках даного компоненту, використовують клас QProcess із бібліотеки QtCore[23] та змінні, що містять шлях на файловій системі до виконуваних файлів інструментів що виконують сам аналіз для заданих параметрів. Ці змінні встановлюються в рамках

компоненту ExternalDependencies під час розгортання проекту на робочій станції користувача (розробника). Для кожного з видів аналізу використовується схожий шаблон виклику (псевдокод такого шаблону зображено на рисунку 5.6).

```
string callProcess(string forPath)
{
    QProcess process - створення процесу
    String pathToExecutable - отримання шляху до процесу, що викликається
    String executableArguments - отримання аргументів процесу
    process.setData(pathToExecutable, executableArguments) - встановлення даних процесу

    process.start - виклик процесу
    waitForProcess - очікування завершення процесу
    String output = process.getOutput - отримання результатів виконання процесу

    return output - повернення результатів виконання з функції
}
```

Рисунок 5.6 – псевдокод шаблону виклику дочірнього процесу аналізу.

Як бачимо з рисунка вище, клас QProcess дозволяє виконувати запуск дочірніх процесів досить легко, при цьому також можна передавати додаткові параметри, що необхідні для проведення аналізу.

Підсумовуючи розгляд даного компоненту можна виділити його основну функцію: створення та отримання результатів дочірнього процесу, в рамках якого проводиться один з видів аналізу програми на вразливості.

Як було вказано вище, компонент AnalysisCaller має пряму залежність від іншого компоненту проекту – ExternalDependencies. Розглянемо його детальніше в наступному підрозділі.

5.4 Компонент ExternalDependencies

5.4.1 Загальна характеристика компонента ExternalDependencies

В рамках розроблюваного проекту компонент являє собою скриптовий програмний елемент, що забезпечує розгортання та налаштування інструментів

третіх сторін, які повинні виконувати сам аналіз програми та її вихідного коду на вразливості. Компонент повністю реалізовано за допомогою кросплатформенної системи збірки CMake, яка виконує генерацію файлів управління процесом збірки з вихідних текстів «CMakeLists.txt» [24]. Загальна структура компоненту представлена на рисунку 5.7, а повний програмний код в додатку В.

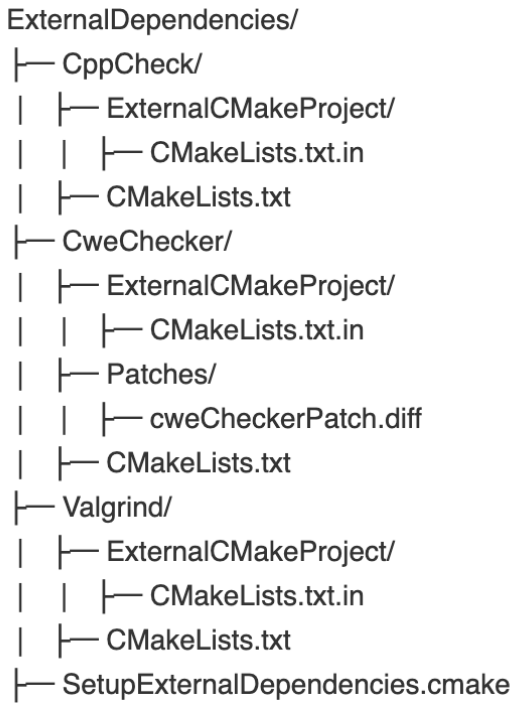

Рисунок 5.7 – Структура компоненту ExternalDependencies.

Як можна побачити, кореневим скриптом є «SetupExternalDependencies.cmake», який почергово виконує розгортання всіх інструментів аналізу на файловій системі користувача. Етапи налаштування кожного з дочірніх елементів зберігаються в окремих директоріях, що відповідають назвам інструментів аналізу. Кожна така директорія обов’язково містить основний файл «CMakeLists.txt» та окремий шаблонний файл «CMakeLists.txt.in», який конфігурується одразу після запуску процесу розгортання [24]. Додатково для інструменту CweChecker створена директорія з правками, що необхідно застосувати перед етапом вбудовування інструменту в

наш проект. Ці правки необхідні для зручності використання засобу в рамках розроблюваного проекту. Слід також зазначити, що ці правки вносяться автоматично, та не потребують додаткових дій від користувача.

Розглянемо детальніше процес розгортання окремого інструменту на прикладі CppCheck. Умовно його можна поділити на наступні етапи:

1. Встановлення індикаторів та змінних, що вказують шлях на файловій системі, де потрібно розмістити інструмент. Приклад таких налаштувань наведено на рисунку нижче:

```
set(CPP_CHECK_SRC_DIR "cppcheck_lib")
set(CPP_CHECK_HOME_PATH "${EXTERNAL_DEPENDENCIES_PATH}/${CPP_CHECK_COMPONENT_NAME}")
set(CPP_CHECK_SRC_PATH "${CPP_CHECK_HOME_PATH}/${CPP_CHECK_SRC_DIR}")
set(CPP_CHECK_BUILD_PATH "${CMAKE_BINARY_DIR}/${CPP_CHECK_COMPONENT_NAME}")
```

Рисунок 5.8 – Початкові налаштування процесу розгортання інструменту.

2. Конфігурація шаблонного файлу для розгортання змінними, що були встановлені на попередньому етапі. Для цього використовується команда, що наведена на рисунку 5.9.

```
configure_file("ExternalCMakeProject/CMakeLists.txt.in" "ExternalCMakeProject/CMakeLists.txt")
```

Рисунок 5.9 – Команда конфігурації шаблонного файлу для розгортання інструменту.

Сам шаблонний файл для наочності наведено на рисунку 5.10.

3. Сконфігурований шаблонний файл запускається через утиліту CMake. На цьому етапі виконується завантаження та розгортання на файловій системі. Для цього використовується команда «execute_process», якій на вхід передається параметр «CMAKE_COMMAND» без аргументів, що вказує на необхідність проведення попереднього налаштування інструменту [24].

4. Останнім етапом є збірка та встановлення. Для виконання цього завдання також використовується команда «execute_process» з параметром «CMAKE_COMMAND», але також додатково передається аргумент «–build» [24].

```
ExternalProject_Add("${CPP_CHECK_COMPONENT_NAME}_Setup"
    PREFIX                          "${CPP_CHECK_HOME_PATH}"
    TMP_DIR                         "${CPP_CHECK_HOME_PATH}/tmp"
    STAMP_DIR                       "${CPP_CHECK_HOME_PATH}/stamp"
    DOWNLOAD_DIR                    "${CPP_CHECK_SRC_PATH}"
    SOURCE_DIR                      "${CPP_CHECK_SRC_PATH}"
    GIT_REPOSITORY                  "https://github.com/danmar/cppcheck"
    GIT_TAG                         "main"
    CONFIGURE_COMMAND               cmake ${CPP_CHECK_SRC_PATH} -DUSE_MATCHCOMPILER=ON
    BINARY_DIR                      "${CPP_CHECK_BUILD_PATH}"
    BUILD_COMMAND                   cmake --build ./
    BUILD_IN_SOURCE                 0
    INSTALL_COMMAND                 ""
)
```

Рисунок 5.10 – Шаблонний файл для розгортання інструменту аналізу.

Після завершення останнього етапу інструмент готовий до використання нашим додатком через спеціальну змінну, що вказує на шлях на файловій системі до виконуваного файлу. Як було вказано раніше, ці змінні використовуються компонентом AnalysisCaller.

Якщо підсумувати розгляд даного компоненту, то стає зрозуміло, що основним його завданням є підготовка інструментів третіх сторін до використання нашим додатком. Після розгляду процесу розгортання засобів аналізу, доцільно вивчити особливості їх використання, завдання що вони виконують та переваги і недоліки.

5.4.2 Компонент статичного аналізу вихідного коду Cppcheck

Cppcheck – це інструмент статичного аналізу коду написаного на мові програмування C/C++. Він забезпечує унікальний аналіз коду для виявлення помилок і фокусується на виявленні невизначеної поведінки та небезпечних

кодових конструкцій. Метою є виявлення лише реальних помилок у коді та уникненні помилкових спрацьовувань (помилкових попереджень). Cppcheck призначений для аналізу C/C++ коду, навіть якщо він має нестандартний синтаксис, як це часто буває, наприклад, , у проектах для вбудовуваних систем [25]. Даний інструмент аналізу підтримує широкий набір платформ та систем:

- Cppcheck перевіряє нестандартний код, що містить різноманітні розширення компілятора, вбудований асемблерний код і т.п.

- Cppcheck повинен компілюватися будь-яким компілятором, який підтримує C++11 або пізнішої версії.

- Cppcheck є кросплатформенним і може використовуватися на багатьох системах (Linux, Windows, MacOS).

Гнучкість в використанні є важливою перевагою інструменту Cppcheck. Іншим важливим плюсом є те, що даний засіб аналізу виконує широкий набір перевірок в програмному коді та дозволяє знаходити помилки наступних типів:

- вказівники на область пам’яті що вже не використовується або використовується іншим модулем програми;

- ділення на нуль;

- переповнення цілих чисел;

- невірні операнди бітового зсуву;

- неправильні перетворення типів;

- помилки при управлінні пам'яттю;

- використання нульових покажчиків;

- перевірка на вихід за кордон використовуваного області пам’яті;

- використання неініціалізованих змінних;

- запис даних в константну область пам’яті [25].

Як було вказано вище, основною метою поставленою перед інструментом аналізу є уникнення помилкових спрацьовувань в результатах аналізу. Це є важливою особливістю в рамках створення нашого засобу комбінованого аналізу, адже одним із завдань розроблюваного проекту є можливість фільтрації помилкових спрацьовувань при аналізі вразливостей в процесі розробки. Для

досягнення цієї цілі інструмент Cppcheck реалізує широкий набір категорій аналізу, кожна з яких може включатися незалежно одна від одної [25]. Налаштування відбувається за допомогою ключа запуску enable, а можливі значення якого наведені в таблиці 5.1.

Таблиця 5.1 – Категорії фільтрації результатів аналізу.

Категорія	Перевірки що здійснюються для заданої категорії
error	Очевидні помилки, які засіб аналізу вважає критичними та які зазвичай призводять до дефектів в програмі. Ця категорія ввімкнена за замовчуванням.
warning	Попередження, що надають інформацію про небезпечний програмний код.
style	Стилістичні помилки – рекомендації про оформлення програмного коду.
performance	Попередження про потенційні проблеми продуктивності програми.
portability	Помилки сумісності систем, що пов’язані з різною поведінкою компіляторів та розрядності платформ.
information	Інформаційні повідомлення, виникаючі в ході перевірки та не пов’язані з помилками в коді.
unusedFunction	Попередження про функції в програмному коді що не використовуються
missingInclude	Перевірка на відсутність директив include для використовуваних функцій.

Інструмент Cppcheck є відкритим, та може бути встановленим користувачем на будь-якому робочому комп’ютері [25]. В рамках розробки нашого засобу комбінованого аналізу, ми будемо розгортати його за допомогою скриптів збірки, як це було вказано вище. Для початку роботи з Cppcheck не

потребується додаткових навичок. Щоб запустити процес аналізу, необхідно вказати шлях до вихідних текстів програми, умовно назвемо його «SourcesPath». Тоді запуск буде виглядати наступним чином:

```
cppcheck --enable=error,warning,performance,portability SourcesPath
```

Рисунок 5.11 – Запуск інструменту статичного аналізу вихідного коду через командний рядок.

Отже, робота з засобом є простою та може бути легко інтегрованою в наш додаток. Крім розглянутих опцій конфігурації засіб також реалізує гнучке налаштування за допомогою опцій командного рядка [25]. Найкорисніші з них наведено в таблиці 5.2.

Таблиця 5.2 – Опції налаштування інструменту Cppcheck.

Опція	Загальна характеристика опції конфігурації
-j	Опція дозволяє запускати перевірку в мультипоточному режимі. Для цього потрібно передати в якості параметр кількість ядер процесора.
-q	Опція вмикає тихий режим аналізу. За замовчуванням інструмент видає інформаційні повідомлення про хід перевірки, яких може бути досить багато. Дана опція повністю вимикає інформаційні повідомлення, залишаються тільки повідомлення про помилки.
-v	Режим відладки. В рамках цієї опції Cppcheck видає внутрішню інформацію про хід перевірки.
--xml	Опція дозволяє виводити результати перевірки в форматі XML, що дозволяє зберігати дані в файл.
--suppress	Опція використовується для фільтрації окремих типів помилок, для фільтрації необхідно вказати ідентифікатор помилки.
--errorlist	Опція виводить список ідентифікаторів помилок, що можуть використовуватись як ключі опції «suppress».

Після завершення процесу аналізу, користувачу виводиться повний список знайдених проблем в програмі. Приклад виводу результатів роботи наведено на рисунку 5.12.

```
cwe_415.c:24:9: note: Memory pointed to by 'buf1R1' is freed twice.
free(buf1R1);
^

cwe_457.c:7:17: error: Uninitialized variable: a [uninitvar]
int c = a + b;
^

cwe_476.c:14:15: error: Memory is allocated but not initialized: data [uninitdata]
printf("%i", data[0]);
^

memory_access.c:7:17: error: Uninitialized variable: a [uninitvar]
int c = a + b;
^
```

Рисунок 5.12 – Результат роботи аналізатора CppCheck.

Як можна побачити на наведеному вище рисунку, кожен запис про потенційну вразливість містить інформацію про файл, в якому було знайдено загрозу, номер рядка в файлі, коротку інформацію про тип та сам рядок програмного коду. Стає очевидно, що для представлення результатів інструмент Cppcheck не використовує машинних інструкцій, а представляє вивід в вигляді перетворень високого рівня, що полегшує роботу розробнику та є однією з вимог до засобу, що розробляється в рамках даної роботи.

Підсумовуючи розгляд інструменту статичного аналізу вихідного коду Cppcheck, сформулюємо основні властивості, що є важливими для розробки засобу для комбінованого аналізу програм:

1. Основним завданням засобу є зменшення ймовірності хибних висновків в результатах роботи. Для цього реалізовано різні категорії фільтрації, що дозволяють підвищувати або понижувати деталізацію результатів.

2. Інструмент може використовуватися на найбільш ранніх етапах розробки ПЗ, адже для аналізу потребується лише вихідний код.

3. В результатах аналізу містяться помилки про ті частини програми, які можуть ніколи не виконуватись, тим часом збільшуючи покриття програми тестовими сценаріями. Така особливість реалізована за допомогою того, що перевірка всіх кодових шляхів виконується безумовно.

4. Дані про потенційні вразливості представляються користувачу в вигляді перетворень високого рівня, уникаючи машинних інструкцій. Це в свою чергу покращує та полегшує роботу розробника.

Якщо проаналізувати вимоги до розроблюваного засобу комбінованого аналізу та особливості інструменту Cppcheck стає зрозуміло, що його використання є оптимальним в рамках створюваного проекту..

5.4.3 Компонент статичного аналізу бінарного коду CWE Checker

Cwe Checker – система статичного аналізу бінарного коду програм, що реалізує набір перевірок для виявлення найбільш поширених класів вразливостей, що можуть міститися в програмному забезпеченні [26]. Переліком таких класів є ресурс Common Weaknesses Enumeration, що являє собою систему класифікації недоліків та вразливостей ПЗ [4]. Основним завданням засобу є автоматичний пошук та розподіл відповідно системи класифікації. Основним напрямом є бінарні файлам, що використовуються на операційних системах Linux та Unix. Для своєї роботи CWE Checker виконує перетворення бінарного файлу в проміжний стан та реалізує власний аналіз цього стану. Подібно іншим засобам аналізу, наведений інструмент також підтримує широкий набір архітектур: x86/x64, ARM, MIPS та інші.

Інструмент позиціонується як гнучкий засіб для проведення аналізу програм, коли до вихідного коду немає доступу, наприклад при використанні пропрієтарних бібліотек з закритим вихідним кодом [26]. Крім цього, так як аналіз проводиться відносно вже готового (скомпільованого) бінарного файлу, важливою перевагою є те, що в процесі аналізу враховуються особливості використовуваного компілятора.

Як було вказано вище, CWE Checker використовує однойменну систему класифікації вразливостей, та реалізовує ті класи перевірок, що найбільш часто зустрічаються при розробці ПЗ. Перелік цих класів наведено в таблиці.

Таблиця 5.3 – Класи вразливостей що можуть бути знайдені інструментом.

Ідентифікатор вразливості	Назва вразливості
CWE-78	Ін’єкція команд операційної системи.
CWE-119 та підвиди CWE-125 і CWE-787	Переповнення буфера.
CWE-134	Використання форматних рядків, що контролюються поза програмою.
CWE-190	Переповнення цілочисленних типів.
CWE-215	Виток інформації через дані відладки.
CWE-243	Використання файлів поза областю команди «chroot».
CWE-332	Недостатня ентропія в псевдовипадкових послідовностях.
CWE-367	Стан гонки за даними програмі – ситуація в якій програма може виконуватись по різному в залежності від потоку виконання.
CWE-415	Звільнення ресурсів що були звільнені іншим модулем програми.
CWE-416	Використання ресурсів після їх звільнення.
CWE-426	Використання програмою ресурсів, що не підлягають безпосередньому контролю програмою.
CWE-467	Використання функції «sizeof» на типах вказівників.

Продовження таблиці 5.3.

CWE-476	Використання ресурсів після їх звільнення.
CWE-676	Використання потенційно небезпечних системних функцій.

CWE Checker в своїй роботі використовує проміжне представлення бінарного файлу та дозволяє знаходити проблеми не запускаючи програму на виконання [25]. Це дозволяє знаходити вразливості, що потенційно можуть не виникати в реальній роботі програми в залежності від умов кодових шляхів, збільшуючи покриття програми тестовими сценаріями. Це є важливою перевагою використовуваного інструменту статичного аналізу бінарних файлів над інструментами динамічного аналізу та дозволяє зробити позитивний висновок про доцільність використання комбінованого аналізу при розробці ПЗ.

Інструмент є відкритим, а для його використання необхідно провести його встановлення. В рамках розроблюваного проекту комбінованого аналізу налаштування та розгортання проводиться автоматично, як це було вказано вище, а детальний програмний код можна знайти в додатку В.

Для початку роботи, подібно з іншими інструментами аналізу, не потребується додаткових навичок. Необхідно лише вказати шлях на файловій системі до цільового виконуваного файлу, наприклад «Program», тоді запуск можна провести за допомогою командного рядка «cwe_checker Program» [25].

Подібно до інших використовуваних інструментів аналізу, засіб CWE Checker дозволяє налаштовувати процес аналізу за допомогою опцій, що передаються при запуску аналізу. Основні такі опції розглянуто в таблиці 5.4.

Таблиця 5.4 – Опції налаштування інструменту CWE Checker.

Опція	Загальна характеристика опції конфігурації
--out	Дозволяє вказати файл, в який необхідно записати звіт про знайдені проблеми.

Продовження таблиці 5.4.

--json	Опція використовується для того, що результати аналізу надавалися в JSON форматі, що дозволяє здійснювати структурний розбір знайдених вразливостей.
--module-versions	Параметр дозволяє вивести класи вразливостей, що можуть бути знайдені під час аналізу.
--statistics	Вказує інструменту що в результати необхідно включити статистичні дані про проведений аналіз.
--partial	Дозволяє проводити часткову перевірку програми, вказуючи лише ті класи вразливостей, на які необхідно звертати. Класи необхідно вказати через кому, наприклад «CWE-119,CWE-367,CWE-416».

Одразу після завершення процесу аналізу, користувачу надається звіт про знайдені вразливості. Приклад такого виводу зображено на рисунку 5.13.

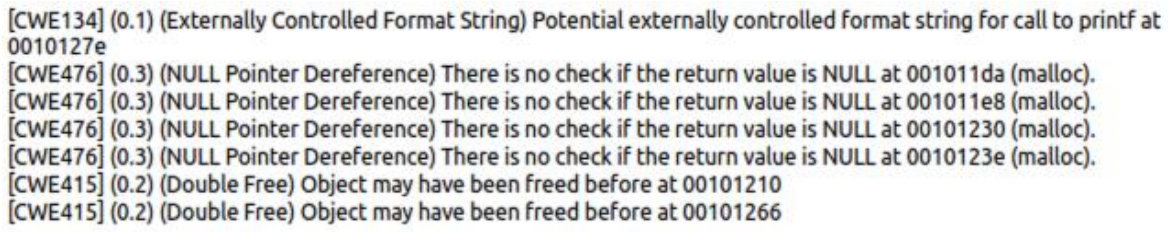

Рисунок 5.13 – Вивід інструменту CWE Checker про знайдені вразливості.

Як бачимо з рисунка, для кожної зі знайдених вразливостей створюється окремий запис, який містить коротку інформацію: ідентифікатор та назву класу вразливості, короткий опис проблеми, адресу внутрішнього представлення де була знайдена проблема. Детальної інформації про вразливість та її місце знаходження не надається. Але як було вказано вище, інструмент розроблено як розширення ресурсу Common Weaknesses Enumeration, тому за необхідності користувач, використовуючи ідентифікатор класу, може звернутися на веб-ресурс та знайти там необхідну йому інформацію. Для кожного з класу

наводиться детальна інформація: розширений опис, можливі наслідки (що виникають внаслідок використання вразливості), програмні приклади, та інше [4].

Отже, підсумовуючи розгляд інструменту статичного аналізу бінарного коду CWE Checker, слід виділити наступні особливості, що є важливими в рамках розроблюваного в даній роботі інструменту комбінованого аналізу ПЗ:

1. Для використання інструменту непотрібно мати доступ до вихідного коду програми, що дозволяє використовувати його не тільки розробникам, а також тест-інженерам, плануючи запуск аналізу в неробочий час в рамках розроблюваного засобу комбінованого аналізу.

2. В результатах аналізу містяться помилки про ті частини програми, які можуть не виконуватись, тим часом збільшуючи покриття програми тестовими сценаріями. Така особливість реалізована за допомогою того, що перевірка здійснюється на внутрішньому проміжному представленні.

3. Так як перевірка здійснюється на цільовому бінарному файлі, що вже є скомпільованим, надається можливість знаходити помилки, що пов’язані з особливостями використовуваного компілятора та платформи розробки.

Розглянувши та проаналізувавши вимоги до розроблюваного проекту та наведені вище особливості, можна зробити висновок про те, що використання інструменту CWE Checker в комбінації з іншими засобами аналізу дозволить підвищити ефективність тестування ПЗ в процесі його розробки.

5.4.4 Компонент динамічного аналізу Valgrind

Valgrind — система з відкритим вихідним кодом, відомий як потужний інструмент для пошуку помилок при роботі з пам’яттю, що робить програми більш стабільними та надійними. Але крім цього, у його складі є кілька додаткових утиліт, призначених для профілювання програм, аналізу споживання пам'яті та пошуку помилок пов'язаних із синхронізацією в мультипотокових програмах [26]. На жаль, він не може вирішити всі проблеми за допомогою

фрагмента коду і вимагає взаємодії з програмістом, який повинен ретельно проаналізувати вихід програми.

Valgrind має модульну архітектуру і складається з ядра, що виконує функцію емуляцію процесора, а конкретні модулі виконують збір та аналіз інформації, отриманої під час виконання коду на емуляторі. Valgrind працює під управлінням операційної системи Linux на процесорах x86, amd64, ppc32 та ppc64 (також ведуться роботи по перенесенню інструменту і на інші ОС) [26].

Основні модулі що формують структуру інструменту Valgrind наведені в таблиці 5.5 [26].

Таблиця 5.5 – Модулі інструменту динамічного аналізу Valgrind.

Модуль	Функції що виконуються окремим модулем
memcheck	Даний модуль є основним, він забезпечує виявлення витоків пам’яті та інших помилок, пов’язаних з неправильної роботою з областями пам’яті – читанням та записом за межами виділених областей та інше.
cachegrind	Аналізує виконання коду, збираючи дані про (неправильні) звернення до кешу та точки розгалуження (коли процесор неправильно прогнозує розгалуження). Ця статистика збирається для всієї програми, окремих функцій і рядків коду.
callgrind	Аналізує виклики функцій, використовуючи приблизно ту ж техніку, що й модуль cachegrind. Дозволяє будувати дерево викликів функцій, і, відповідно, аналізувати вузькі місця в програмі.
Massif	Даний модуль дозволяє аналізувати виділення та використання пам’яті усіх частин програми.
helgrind	Аналізує виконуваний код програми на предмет помилок синхронізації, при використанні багатопотокового коду.

Бачимо що засіб Valgrind дозволяє виконувати багатопрофільний аналіз програми на вразливості різних видів та походження. Для реалізації нашого інструменту будемо використовувати модуль memcheck, так як цей модуль є найбільш стабільним та простим в використанні, що дозволить зменшити складність експериментів при тестуванні.

Згідно документації [26], модуль memcheck дозволяє знаходити наступні вразливості:

- використання неініціалізованої пам'яті;
- пам'ять для читання/запису після її звільнення;
- читання/списування кінця блоків malloc;
- читання/запис невідповідних областей на стеку;
- витоки пам'яті - де вказівники на блоки malloc втрачені назавжди;
- невідповідні виклики malloc/new/new[] та free/delete/delete[];
- перекриття покажчиків src та dst у memcpy() та пов’язаних функціях;

В наш час інструмент Valgrind входить в склад більшості дистрибутивів Linux, однак, як було вказано раніше, ми будемо встановлювати його під час розгортання нашого засобу за допомогою скриптів збірки.

Робота з інструментом Valgrind достатньо проста – ним можна управляти за допомогою опцій командного рядка. Також важливим є те, що аналізована програма не потребує додаткової підготовки. Хоча розробку програми рекомендується проводити з використанням debug-інформації та вимкненою оптимізацією, використовуючи прапори «–g» та «–O0» [26]. Розглянемо звичайний запуск програми, яка називається «Program» та отримує аргументи «Arguments» під управлінням середовища Valgrind. Для цього необхідно додати слово «valgrind» на початку рядка виклику та вказати параметри аналізу. Приклад виклику наведено нижче:

```
valgrind Program Arguments
```

Рисунок 5.14 – Запуск інструменту за допомогою командного рядка.

За замовчуванням використовується модуль «memcheck», однак можна вказати який модуль використовувати за допомогою опції «--tool», наприклад:

```
valgrind –tool=callgrind Program Arguments
```

Рисунок 5.15 – Передача цільового модуля для запуску через опцію командного рядка.

Крім опції «tool» існують і інші, що дозволяють контролювати роботу інструменту [26]. Опції що найбільш часто використовуються наведені в таблиці 5.6.

Таблиця 5.6 – Опції конфігурації інструменту Valgrind.

Опція	Загальна характеристика опції конфігурації
--quiet	Заглушати непотрібну інформацію, виводячи лише інформацію про помилку.
--verbose	Відображати детальну інформацію про роботу інструменту
--log-file	Дозволяє вказати ім'я файлу, в який буде виведений звіт про роботу. У наведеній назві можуть використовуватися спеціальні шаблони, куди будуть підставлені різні значення, наприклад, ID процесу (шаблон% p).
--log-socket	Дозволяє встановити адресу та порт, на які буде надсилатися звіт про роботу.
--log-fd	Дозволяє вказати дескриптор файлу, в який буде виводитися звіт про роботу (за замовчуванням це число дорівнює 2 – стандартний вихід повідомлень про помилки).
--track-fds	Можливі значення yes чи no, за замовчуванням no. Змушує valgrind відображати дескриптори відкритих файлів під час виходу.

Продовження таблиці 5.6.

--trace-children	Можливі значення yes чи no, за замовчуванням no. Дозволяє відстежувати процеси, запущені аналізованою програмою, за допомогою системного виклику exec.
--time-stamp	Можливі значення yes чи no, за замовчуванням no. Призводить до видачі відміток часу у звіті про роботу (час відраховується від початку роботи програми).
--leak-check	Вмикає (значення yes, summary або full) або вимикає (значення no) виявлення витоку пам’яті. Варто зазначити, що при використанні значення summary, memcheck дає лише коротку інформацію про витік пам’яті, тоді як при інших значеннях, окрім підсумкової інформації, буде також відображатися інформація про місце, де відбувається цей витік пам’яті.
--leak-resolution	Можливі значення low, med або high. Вказує, як порівнюється стек викликів функції. Для low і med порівняння використовуються останні два або чотири виклики відповідно, а для рівня high порівнюється повний стек викликів. Цей параметр впливає лише на спосіб представлення результатів пошуку помилок.
--undef-value-erros	Можливі значення yes чи no. Визначає, чи показувати помилки щодо використання неініціалізованих значень.

Існують і інші опції, але їх використання зустрічається значно рідше, в випадку необхідності їх можна знайти в документації до засобу [26].

Інтерпретація результатів Модуль memcheck виявляє кілька типів помилок. Помилки читання та запису за межами виділеної пам’яті (і деякі інші типи помилок) виникають негайно під час роботи програми. А помилки, що призводять до витоку пам'яті, виробляються valgrind після завершення

аналізованої програми. Формат цих помилок дещо інший, тому вони будуть описані окремо. Кожен рядок у виводі valgrind має префікс форми «==12345==», де число вказує на ідентифікатор запущеного процесу.

Як було вказано в попередніх розділах, модуль memcheck виявляє кілька видів помилок пам'яті:

- читання або запис за неправильними адресами пам'яті – поза межами виділених блоків пам'яті, тощо.

- використання неініціалізованих значень, включаючи змінні, виділені в стеку

- помилки звільнення пам’яті, наприклад, коли блок пам’яті вже звільнено десь в іншому місці

- використання «неправильної» функції для звільнення пам'яті, наприклад використання «delete» для пам'яті, виділеної за допомогою «new[]»;

- передача неправильних параметрів системним викликам, наприклад, використання неправильних покажчиків для операцій читання з буфера, визначеного користувачем;

- перетин меж блока пам'яті при копіюванні/переміщенні даних між двома блоками пам'яті.

Для цих помилок дані надаються в міру їх виявлення, і зазвичай вони виглядають так як зображено на рисунку 5.16.

```
Mismatched free() / delete / delete []
   at 0x40043249: free (vg_clientfuncs.c:171)
   by 0x4102BB4E: QGArray::~QGArray(void) (tools/qgarray.cpp:149)
   by 0x4C261C41: PptDoc::~PptDoc(void) (include/qmemarray.h:60)
   by 0x4C261F0E: PptXml::~PptXml(void) (pptxml.cc:44)
 Address 0x4BB292A8 is 0 bytes inside a block of size 64 alloc'd
   at 0x4004318C: operator new[](unsigned int) (vg_clientfuncs.c:152)
   by 0x4C21BC15: KLaola::readSBStream(int) const (klaola.cc:314)
   by 0x4C21C155: KLaola::stream(KLaola::OLENode const *) (klaola.cc:416)
   by 0x4C21788F: OLEFilter::convert(QCString const &) (olefilter.cc:272)
```

Рисунок 5.16 – Дані про знайдені вразливості в програмі.

Перший рядок містить опис відповідної помилки, а потім стек викликів функцій, які викликали цю помилку. Там де необхідно (як у нашому прикладі), також надається адресу області пам’яті та місце, де був виділений блок пам’яті. Як бачимо, інструмент Valgrind представляє результати аналізу в вигляді перетворень високого рівня, що є однією з вимог до розроблюваного нами засобу.

Розглянувши інструмент динамічного аналізу Valgrind, стає зрозуміло, що його основними перевагами є:

1. Ймовірність хибних висновків при здійсненні аналізу близька до нуля, а сам інструмент реалізує рівні градації помилок, що дозволяє поступово розглядати виникаючі проблеми.

2. Помилки знаходяться в контексті цільової системи, що дозволяє враховувати особливості операційної системи на якій проводиться тестування.

3. В результаті аналізу користувачу надається стек виклику, що містить перетворення високого рівня, уникаючи машинних інструкцій.

Переглянувши наведені вище переваги можна зробити висновок, що інструмент Valgrind задовольняє основні вимоги до динамічних аналізаторів програм, що були сформовані в попередніх розділах.

Отже, в даному розділі було описано програмну структуру інструменту, що реалізує ключові принципи комбінованого аналізу ПЗ, такі як можливість запуску різних видів аналізу послідовно чи паралельно, відсіювання хибних висновків про вразливості, представлення результатів в зрозумілому для розробника вигляді. Таким чином, необхідно перейти до тестування розробленого інструменту, що дозволить нам зробити висновок про доцільність та перспективи теоретичних пропозицій, висунутих в попередніх розділах.

6 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ІНСТРУМЕНТУ КОМБІНОВАНОГО АНАЛІЗУ

В рамках попередніх розділів було розглянуто основні види вразливостей в програмному забезпеченні та сучасні підходи до їх пошуку. Виявлено що кожна з методик аналізу має свої переваги та недоліки, після їх розгляду було запропоновано використовувати систему комбінованого аналізу ПЗ. Сформулювавши вимоги та схему такої платформи, в попередньому розділі було описано програмну реалізацію, що була розроблена в рамках даної роботи. Після вивчення програмної структури необхідно провести тестування такого засобу для того щоб зробити висновок про доцільність та перспективи використання такого засобу в процесі розробки ПЗ. Для вирішення цього завдання пропонується провести наступні тестові дослідження:

1. Розглянути графічний інтерфейс розробленого інструменту та описати сценарій його використання.

2. Дослідити якісні характеристики комбінованого аналізу. Для цього доцільно розглянути використовувані підходи окремо та оцінити результати на предмет того, які типи вразливостей можуть чи не можуть бути виявленими.

3. Оцінити характеристики швидкості проведення комбінованого аналізу. Для цього доцільно провести заміри часу аналізу для кожного з підходів окремо.

Запропоновані дослідження будемо проводити на спеціально підготованих програмних прикладах, в яких спеціально створимо вразливості різних видів.

6.1 Розгляд графічного інтерфейсу інструменту комбінованого аналізу ПЗ

Графічний інтерфейс користувача було розбито на логічні блоки, що дозволяє зробити процес аналізу більш інтуїтивно зрозумілим та за рахунок цього підвищити ефективність аналізу. Опис блоків що формують інтерфейс

було проведено в рамках попереднього розділу, а сам вигляд головного вікна інструменту наведено на рисунку 6.1.

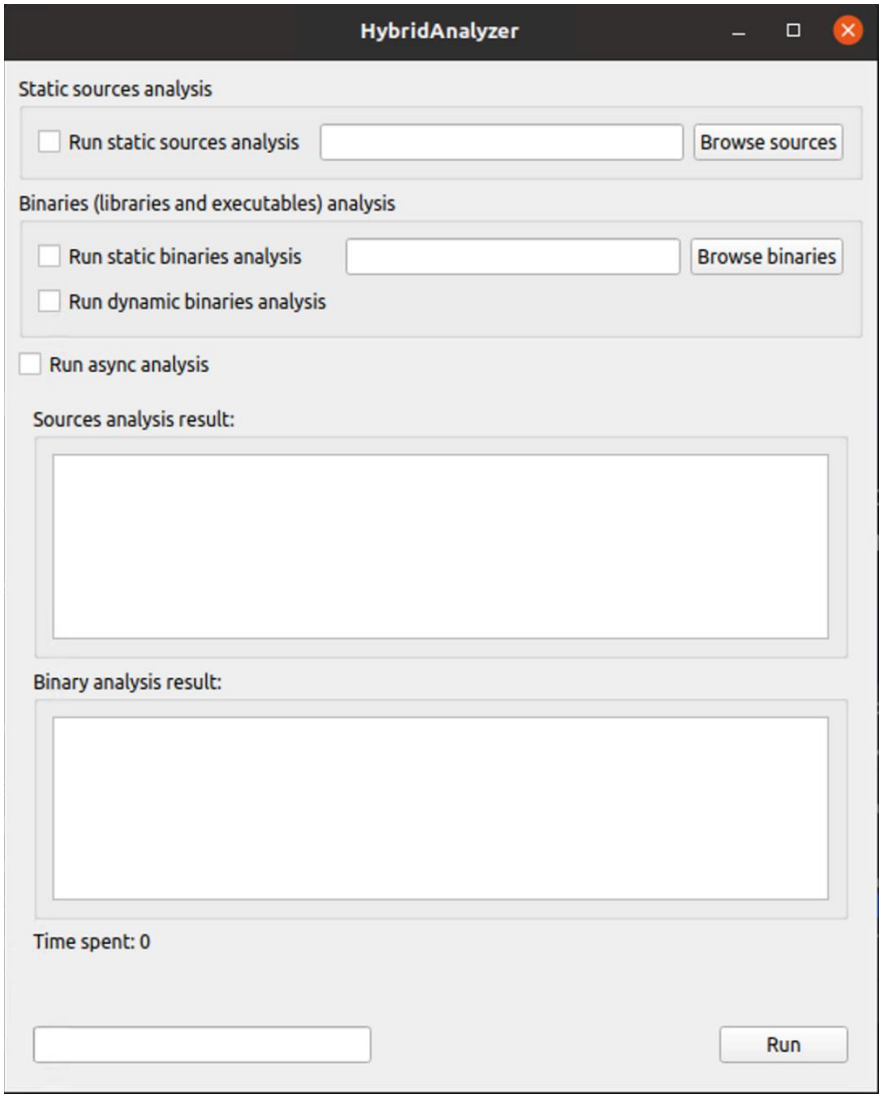

Рисунок 6.1 – Графічний інтерфейс користувача розробленого інструменту комбінованого аналізу.

З наведеного рисунка видно, що інтерфейс дозволяє конфігурувати процес аналізу, включаючи чи виключаючи різні типи аналізу. Також на головному вікні знаходиться налаштування щодо асинхронного процесу виконання, в разі ввімкнення такої опції, кожен з видів аналізу буде проводитись в окремому програмному потоці.

Щоб почати роботу необхідно вказати шляхи на файловій системі до вихідних текстів програми та виконуваного файлу програми. Для цього треба натиснути на кнопки «Browse sources» та «Browse binaries» відповідно. Після цього користувачу відкриється вікно файлового менеджера, що зображено на рисунку 6.2, в якому необхідно вибрати виконуваний файл або директорію з файлами вихідних текстів.

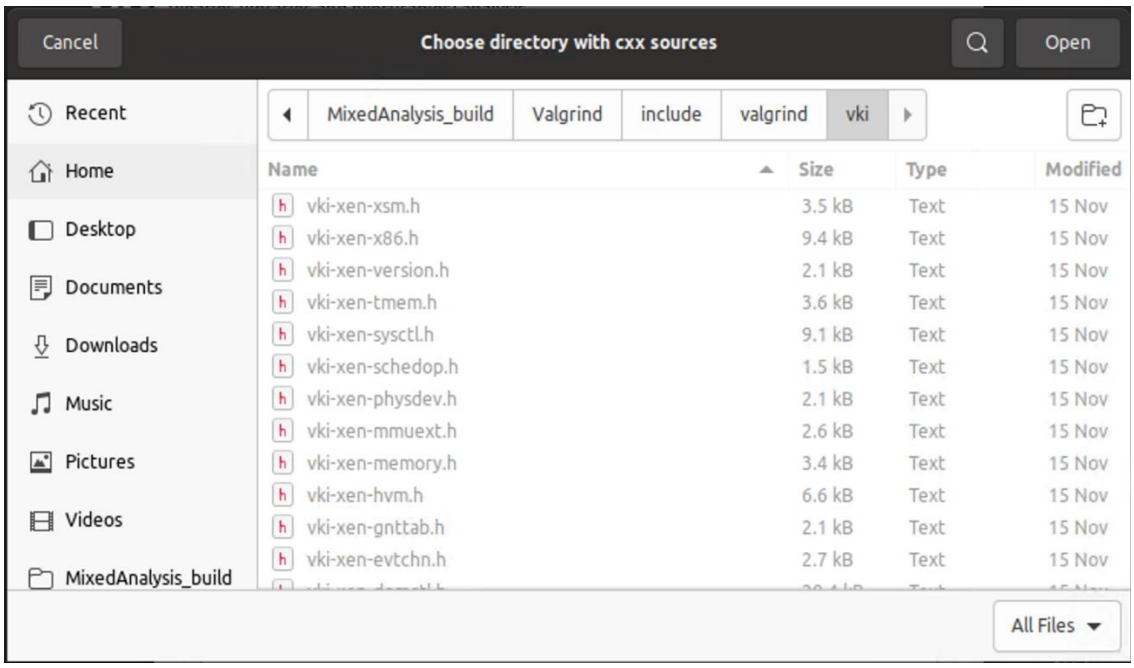

Рисунок 6.2 – Вікно файлового менеджера для вибору файлів для аналізу.

В якості зворотного зв’язку програми з користувачем, в блоках аналізу вихідного коду та бінарних файлів в текстових полях буде наведено шлях, що вибрав користувач так, як це зображено на рисунку 6.3.

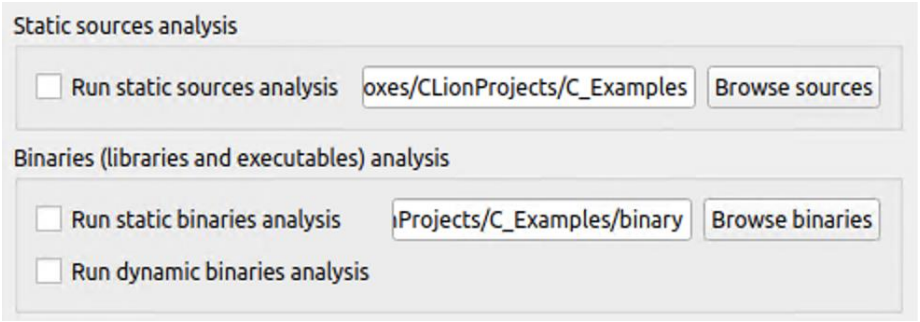

Рисунок 6.3 – Блоки налаштувань аналізу на графічному інтерфейсі.

Для запуску процесу аналізу необхідно виконати вибір типів аналізу за допомогою відповідних графічних індикаторів «Run static sources analysis», «Run static binaries analysis», «Run dynamic binaries analysis» та натиснути кнопку «Run».

Після старту аналізу його прогрес буде відображатися внизу головного вікна програми, тим самим надаючи зворотній зв’язок для користувача. Результати аналізу виводяться в текстових полях, що займають більшість простору головного вікна для зручності роботи з ними. Приклад виводу інструменту зображено на рисунку 6.4.

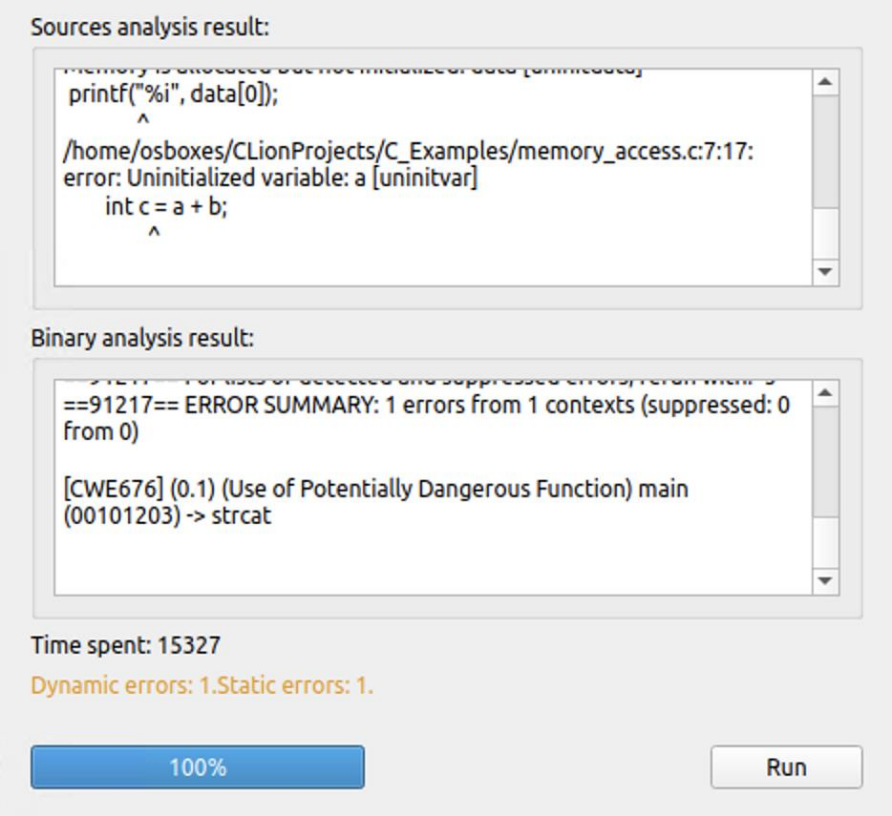

Рисунок 6.4 – Блок результатів виконання аналізу на графічному інтерфейсі.

Як можна побачити з рисунка вище, крім звіту про знайдені вразливості, користувачу також надаються статистичні дані проведеного аналізу. Це дозволяє проводити дослідження характеристик, що допомагають робити висновки про доцільність використання інструменту на різних етапах розробки ПЗ. Для фільтрації хибних висновків користувачу надається підказка про кількість

знайдених помилок в статичному та динамічному аналізах бінарного коду. На наведеному рисунку статистика проведених перевірок співпадає, тому повідомлення користувачу не надається. В протилежному випадку, користувач отримає повідомлення подібне тому, що зображено на рисунку 6.5. Далі розробнику необхідно переглянути звіт проведеного аналізу, та зробити висновок про те, чи несуть реальну небезпеку знайдені вразливості.

Рисунок 6.5 – Повідомлення для користувача що вказує на потенційні хибні висновки в результатах аналізу.

Підсумовуючи розгляд графічного інтерфейсу виділимо наступну послідовність використання інструменту комбінованого аналізу:

1. Вибір шляху на файловій системі до директорії вихідних текстів та виконуваного файлу програми за допомогою кнопок на відповідних графічних блоках.

2. Вибір типів аналізу що необхідно провести за допомогою графічних індикаторів.

3. Користувач вказує про необхідність проведення кожного з процесів в окремих програмних потоках.

4. Запуск процесу перевірки за допомогою відповідної кнопки інтерфейсу.

5. Отримання та перегляд результатів наведених в текстових полях. В рамках перегляду необхідно звернути увагу на попередження про можливі хибні висновки та провести дослідження результатів на предмет реальних загроз програмі.

Запропонований сценарій використання інструменту комбінованого аналізу є досить простим, та дозволяє легко виконувати перевірку програм, тим самим зменшуючи час що необхідно витрачати на тестування розроблюваного програмного забезпечення.

6.2 Дослідження якісних характеристик комбінованого аналізу за допомогою розробленого інструменту

Для проведення дослідження слід сформувати ряд програмних прикладів, що дозволять сформулювати уявлення про якість проведення аналізу окремо взятими видами. В якості таких прикладів запропоновано використати згадану раніше систему класифікації Common Weaknesses Enumeration, яка для кожного з видів вразливостей надає приклад програмного коду, що є потенційно небезпечними [4]. Також для повноти експерименту слід додати приклад, який не несе реальної небезпеки, але може бути розцінений інструментом як вразливість, тобто сформує хибний висновок. Для цього було проведено тестування на 10 прикладах вразливостей та 2 програмних прикладах які не несуть реальної небезпеки. Результати тестування наведені в таблиці 6.1.

Таблиця 6.1 – Результати перевірки тестових програмних прикладів інструментами аналізу.

Тип вразливості	Вразливість була знайдена		
	Статичний аналіз вихідного коду	Статичний аналіз бінарного коду	Динамічний аналіз бінарного коду
CWE-457	Так	Ні	Так
CWE-782	Ні	Так	Ні
CWE-560	Ні	Так	Ні
CWE-476	Так	Так	Так
CWE-457	Так	Ні	Так
CWE-416	Ні	Так	Так
CWE-415	Так	Так	Так
CWE-367	Ні	Так	Ні
CWE-190	Ні	Так	Так

Продовження таблиці 6.1.

CWE-119	Ні	Так	Так
Потенційний хибний висновок про CWE-787	Ні	Так	Ні
Потенційний хибний висновок про CWE-467	Ні	Так	Ні

Наведена вище таблиця являє собою мапу ефективності аналізу, тобто відображає результати виконання для кожного з окремих видів. В першому стовпці наводяться дані статичного аналізу вихідного коду, а в другому та третьому дані для статичного та динамічного аналізу бінарного коду відповідно. Додатково для наочного представлення нижче результати тестування наведені в вигляді гістограми на рисунку 6.6.

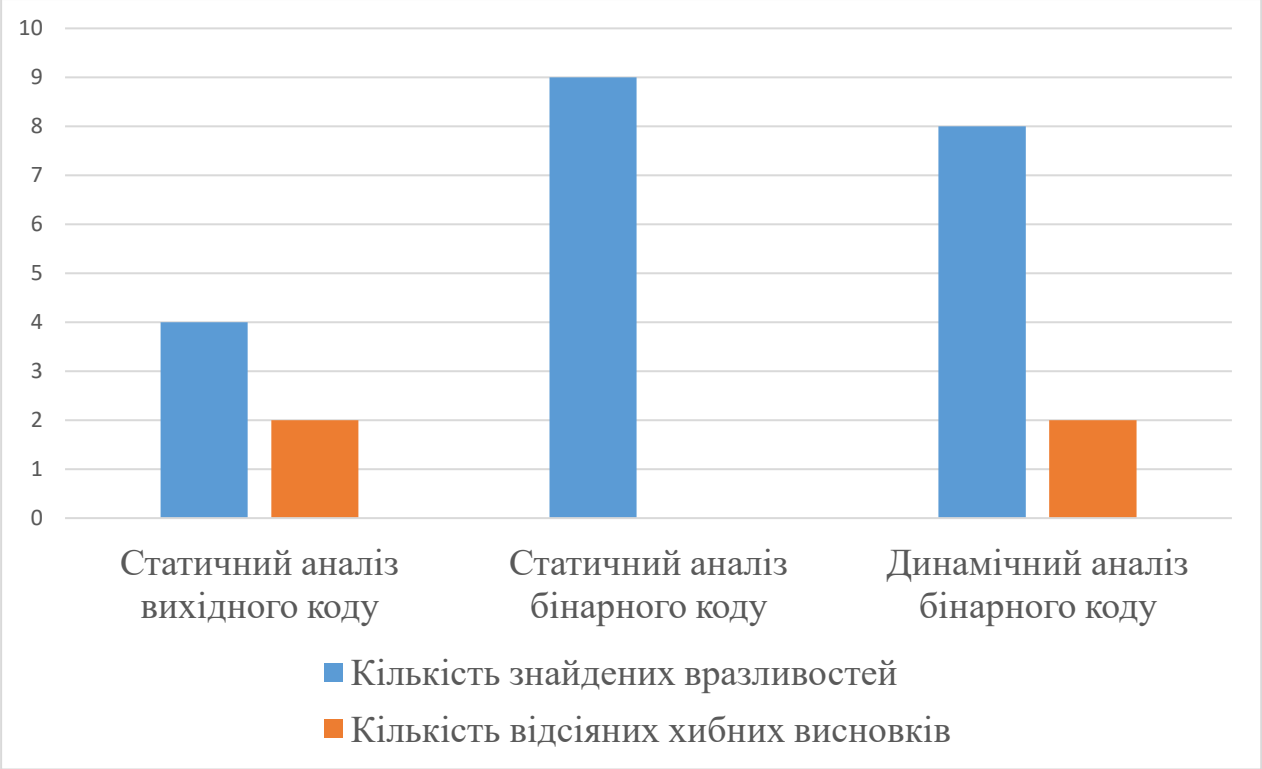

Рисунок 6.6 – Гістограма ефективності окремих інструментів аналізу.

Проаналізуємо отримані дані. Як бачимо, статичний аналіз вихідного коду має найнижчу ефективність виявлення вразливостей – вдалося знайти 4 вразливості. При цьому хибного спрацьовування не відбулося, тобто такі невірні висновки були відсіяні. Така поведінка пояснюється тим, що статичний аналіз має найменше даних щодо використовуваного компілятора, платформи на якій програма буде виконуватись та інші програмні модулі, вихідний код яких є недоступним для аналізу. Слід також зазначити, що в результаті аналізу було вдало відсіяно хибні висновки, що є одним з основних пріоритетів використовуваного інструменту статичного аналізу, який було описано в попередньому розділі.

Далі розглянемо результати статичного аналізу бінарного коду. Як можна побачити з отриманих даних, цей вид аналізу має найбільшу ефективність порівняно з іншими видами – вдалося виявити 9 вразливостей, що складає 90% від всіх програмних прикладів наведених до тестування. Така поведінка має декілька причин. По-перше, аналіз проводиться на готовому бінарному файлі, тобто в контексті цільової системи, що дозволяє враховувати всі непрямі впливи на процес виконання програми. По-друге, як було розглянуто в попередньому розділі, для проведення перевірки використовується внутрішнє представлення програми, що дозволяє перевіряти максимальну кількість кодових шляхів, тим самим підвищуючи покриття програми тестовими сценаріями. По-третє, такий тип аналізу дозволяє перевіряти також програмні модулі, вихідний код яких є недоступним, але використовуючи внутрішнє представлення їх перевірка стає можливою. Основною проблемою в використанні такого інструменту виявилась неефективність в відсіюванні хибних висновків, що є побічним ефектом того, що внутрішнє представлення програми не дозволяє передбачити реальне виконання програми, тобто зробити висновки про вплив на програму в процесі її реального виконання не вдасться.

Третім та останнім протестованим видом є динамічний аналіз бінарного коду програми. Отримані результати показують, що цей вид також має достатньо

високу ефективність в виявленні вразливостей – було знайдено 8 вразливостей, що становить 80% програмних прикладів. Подібно до інструменту статичного аналізу бінарного коду, розглянутий інструмент також виконує перевірку готового виконуваного файлу, що дозволяє оцінити непрямий вплив на процес виконання програми. Але на відміну від згаданого типу, динамічний аналіз не використовує жодних проміжних представлень, а процес перевірки проводиться під час самого виконання програми, що має свої переваги та недоліки. З одного боку, такий підхід звужує покриття програми тестовими сценаріями, адже ефективність аналізу напряму залежить від наданих програмі тестових даних. З іншого боку, під час виконання програми можна слідкувати за її поведінкою та робити висновки про її внутрішній стан, тим самим відсіюючи хибні висновки про вразливості.

Підсумовуючи розгляд якісних характеристик кожного з видів аналізу окремо можна зробити наступні висновки:

1. Статичний аналіз вихідного коду не потребує запуску програми на виконання. Має низьку ефективність в виявленні вразливостей порівняно з іншими інструментами, при цьому відсіювання хибних висновків проводиться на достатньому рівні.

2. Статичний аналіз бінарного коду має найвищу ефективність, проте найнижчу ефективність в відсіюванні хибних висновків через відсутність можливості слідкувати за станом програми під час реального виконання.

3. Динамічний аналіз бінарного коду має високу ефективність, при цьому дозволяє відсіювати висновки що не несуть реальної загрози. Недоліком є залежність ефективності від тестових сценаріїв.

Важливим є те, що сформовані характеристики співпадають з теоретичними даними, які були наведені під час розгляду підходів до аналізу програм в попередніх розділах. В свою чергу, використовуючи ці теоретичні дані, було сформовано підхід до комбінованого аналізу ПЗ. Така відповідність між теорією та практикою дозволяє зробити висновок про перспективу використання комбінованого аналізу ПЗ в процесі його розробки.

6.3 Дослідження характеристик швидкості проведення аналізу

Для проведення дослідження спочатку проведемо опис оточення, на якому проводиться тестування. В якості платформи будемо використовувати спеціально створену віртуальну машину, яка має наступні характеристики:

- Операційна система Ubuntu Linux x64
- Кількість ядер процесора – 4.
- Оперативна пам’ять – 8192 Мб.

Також слід сформувати ряд прикладів, які дозволять отримати уявлення про залежність швидкості виконання аналізу від розміру файлів що перевіряються. Слід зауважити, що підібрати послідовність з лінійним зростанням розміру файлу досить важко, тому що код потрібно компілювати. Було вибрано програмні приклади вихідного коду наступних розмірів: 440 байт, 1660 байт, 3440 байт, 4400 байт, 6200 байт, 7000 байт. Вибрані приклади було скомпільовано, в результаті отримано відповідні бінарні файли наступних розмірів: 16800 байт, 17400 байт, 17800 байт, 18400 байт, 18800 байт, 24000 байт. Результати замірів часу що був витрачений на аналіз вихідного коду наведено в таблиці 6.2, а дані про аналіз бінарного коду в таблиці 6.3.

Таблиця 6.2 – Результати замірів часу виконання аналізу файлів вихідного коду.

	Розміри файлів що перевіряються, байт					
	440	1660	3440	4400	6200	7000
Час, витрачений на аналіз, мс	25	50	85	95	115	130

Проглянувши наведені дані тестування видно, що час необхідний на проведення аналізу зростає зі збільшенням цільового файлу, що й можна було передбачити. Статичний аналіз вихідного коду займає найменше часу. Така поведінка пояснюється тим, що для аналізу використовується лише сам файл

вихідного тексту, при цьому не проводиться перевірка сторонніх бібліотек та модулів та не проводиться запуск програми на виконання.

Таблиця 6.3 – Результати замірів часу виконання аналізу бінарних файлів.

		Розміри файлів що перевіряються, байт					
		16800	17400	17800	18400	18800	24000
Час, витрачений на аналіз, мс	Статичний аналіз	6200	6400	6480	6580	6650	6800
	Динамічний аналіз	500	550	585	600	630	700

Протилежна ситуація спостерігається для статичного аналізу бінарного коду – для цього аналізу потрібно в десятки разів більше часу. Ця поведінка слідує з того, що для проведення такої перевірки потрібно виконати значно більше роботи. По-перше, необхідно створити внутрішнє проміжне представлення програми. По-друге, потрібно врахувати вплив сторонніх модулів на програму. Лише після проведення цих дій можна переходити безпосередньо до перевірки програми на вразливості. Що стосується динамічного аналізу бінарного коду, то цей тип показав середні результати. Процес виконується значно швидше за статичний аналіз бінарного коду, при цьому повільніше за статичний аналіз вихідного коду, а час виконання напряму залежить від продуктивності самої програми, адже для проведення перевірки необхідно запускати виконуваний файл.

Підсумовуючи розгляд характеристик швидкості проведення аналізу, можна сформувати наступні особливості:

1. Статичний аналіз вихідного коду займає мізерну частину порівняно з іншими інструментами. Це дозволяє швидко проводити перевірку вихідного коду та підтверджує висунуту в попередніх теорію про використання такого типу аналізу розробником, при цьому не вносячи затримок в його роботу.

2. Статичний аналіз бінарного коду займає значно більший час. Така поведінка вказує на те, що цей інструмент краще використовувати незалежно від самої розробки. Це означає, що програмісту слід запустити процес аналізу бінарного файлу та продовжити редагування вихідного коду програми. Іншим підходом може бути використання такого інструменту на спеціальних кластерах, де буде перевірка буде проводитись окремо від робочої станції розробника.

3. Динамічний аналіз бінарного коду демонструє середні показники швидкості роботи. Таким чином, можна зробити висновок що розробник може використовувати такий тип аналізу на свій погляд. Наприклад, він може запустити процес аналізу та слідкувати за процесом роботи самої програми, адже для проведення перевірки також запускається виконуваний файл.

Отже, отримавши уявлення про графічний інтерфейс розробленого інструменту комбінованого аналізу вразливостей ПЗ, його якісні характеристики та отримані результати швидкості проведення перевірки, можна сформулювати рекомендації щодо використання розробленого проекту.

7 ВИКОРИСТАННЯ КОМБІНОВАНОГО АНАЛІЗУ В СУЧАСНІЙ МЕТОДОЛОГІЇ РОЗРОБКИ S-SDLC

Кожен з інструментів аналізу має свої сильні сторони. В рамках даної роботи були виявлені та оцінені переваги та недоліки окремих типів аналізу програм на вразливості. Наприклад, інструменти статичного аналізу здатні виявляти помилки, які пропускаються інструментами динамічного аналізу, тому що інструменти динамічного аналізу фіксують помилку лише в разі, якщо під час тестування помилковий фрагмент коду виконується. З іншого боку, інструмент динамічного аналізу виявляють програмні помилки кінцевого працюючого процесу. А статичний аналіз бінарного коду займає достатньо багато часу, через необхідність створення проміжного внутрішнього представлення. Для мінімізації недоліків кожного з розглянутих засобів аналізу вразливостей ПЗ, в рамках даної роботи було запропоновано використовувати комбінований підхід, було сформовану схему такого аналізу та за цією схемою розроблено інструмент комбінованого аналізу. В попередньому розділі створений інструмент було проведено тестування розробленого інструменту, в рамках якого підтвердились теоретичні припущення попередніх розділів.

Як було розглянуто в першому розділі, найбільш поширеною методологією розробки ПЗ на сьогодні є S-SDLC [8]. Також було розглянуто, що аналіз виконується на вихідному чи бінарному коді програми, тому очевидно, що основним етапом для використання в циклі розробки є етап написання коду. Розглянувши матеріали даної роботи, можна сформувати приклад спільного використання двох видів інструментів:

- На початку робочого дня розробник переглядає звіт про результати нічного компонування. В даний звіт включаються як власне помилки

компоновки, так і результати комбінованого аналізу, проведеного під час компонування.

- У звіті про проведений аналіз наводяться виявлені дефекти в вигляді перетворень високого рівня, що допоможе розробнику швидко орієнтуватися між бінарним представлення програми та вихідним кодом. Після цього розробником проводиться виправлення фактично присутніх помилок.

- Розробник зберігає внесені зміни разом з будь-якими новими фрагментами коду. Ці зміни не передаються до системи контролю версій до тих пір, поки зміни не будуть проаналізовані і протестовані.

- Розробник аналізує і коригує новий код, використовуючи інструмент статичного аналізу вихідного коду на локальному робочому місці.

- Після аналізу і корегування нового коду розробник вбудовує код в локальний тестовий образ або виконуваний файл.

- Використовуючи динамічний аналіз бінарного коду, розробник запускає тести для перевірки внесених змін. Паралельно з цим програміст запускає статичний аналіз бінарного коду, який проводиться незалежно від динамічного.

- Розробник вивчає висновки засобів аналізу та виправляє помилки. Код вважається готовим до цільового тестування, коли він повністю пройшов через всі перевірки комбінованого аналізу.

- Розробник передає зміни в систему управління версіями, після цього змінений код бере участь в процесі подальшого нічного компонування.

Подібна послідовність дій добре підходить для проектів різних розмірів. Розробники мають змогу швидко виконувати комбінований аналіз ще на етапі створення вихідного коду програми та створення прототипів, не відхиляючись від типової послідовності дій. В результаті якість розроблюваного продукту зростає вже на етапі створення вихідних текстів програми.

Особливістю методології S-SDLC також є те, що написання коду програми не завершується після випуску продукту, адже в процесі активного використання

програми можуть виявлятися дефекти, для виправлення яких необхідно вносити зміни в вихідний код [8]. Зазвичай такі зміни вносяться на етапі підтримки продукту. Також було розглянуто, що інструменти динамічного аналізу дозволяють проводити розбір вже на готовому програмному забезпеченні без вихідного коду. Враховуючи всі розглянуті в даній роботі матеріали, можна створити модифіковану схему методології S-SDLC, зображену на рисунку 7.1.

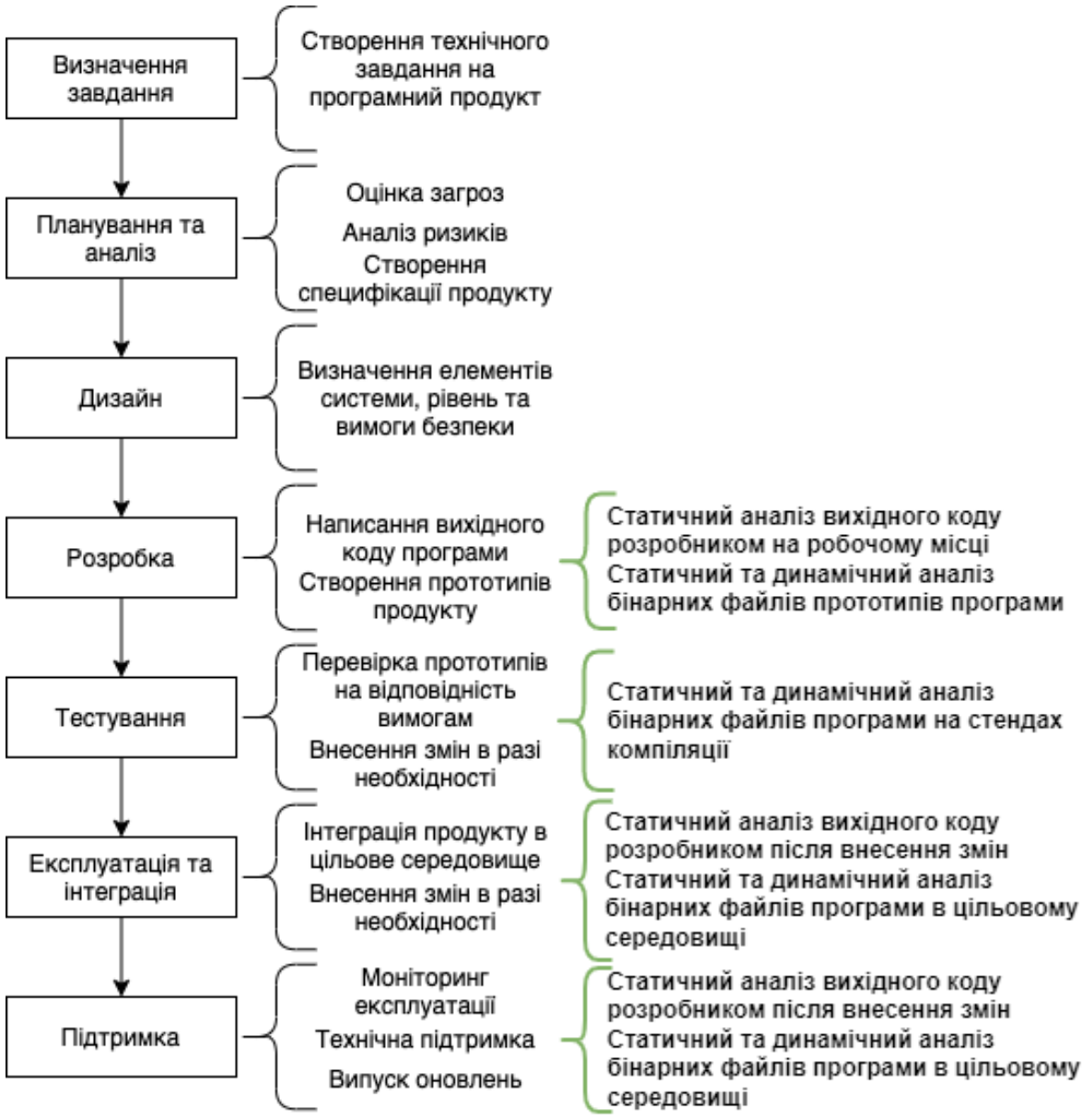

Рисунок 7.1 – Інтеграція комбінованого аналізу ПЗ в методологію розробки S-SDLC.

ВИСНОВКИ

Світовий устрій на сьогодні характеризується ростом кількості програмного забезпечення у геометричній прогресії. Разом з цим, відведений час на розробку цих продуктів зменшується. Такі реалії сьогодення створюють достаткові проблеми для компаній-розробників ПЗ, адже при зменшенні часу на розробку, зменшується також і час відведений на тестування та усунення дефектів. Прогалини в програмному забезпеченні сьогодні є серйозною проблемою: більшість успішних кібератак реалізується за допомогою використання вразливостей в програмного забезпеченні. Для вирішення даної проблеми було запропоновано розглянути комбінацію інструментів статичного аналізу вихідного коду та динамічного і статичного аналізу бінарного коду.

В першому розділі даної роботи було розглянуто поняття вразливості. Дано визначення цьому явищу та розглянуті їх основні види та класифікації. Виявлено що основним етапом на якому з’являються вразливості є етап розробки, в свою чергу найоптимальнішим етапом для виявлення та усунення дефектів є етапи розробки та тестування.

В другому розділі вивчаються сучасні підходи до пошуку вразливостей в ПЗ. Розглянуто статичний та динамічний аналіз, а також технології, що реалізують ці засоби та виявлено основні переваги та недоліки досліджуваних інструментів. Основною перевагою технології статичного аналізу є те, що вона дозволяє з високою ймовірністю знаходити помилки ще на етапі написання вихідного коду. В той самий час, перевагою засобів динамічного аналізу є те, що їх специфіка виключає ймовірність хибних висновків.

Третій розділ присвячено аналізу вимог до системи комбінованого аналізу, основними вимогами до такого засобу є:

- відсіювання хибних висновків;
- покращення покриття потенційно загрозливими сценаріями;
- можливість використовувати на ранніх етапах розробки ПЗ;

В четвертому розділі розглянуто створення техніки комбінованого аналізу ПЗ. Було створено схему аналітичної платформи, що дозволить розробнику знаходити дефекти під час своєї роботи. Створена схема передбачає інтерактивну взаємодію, але може бути легко модифікована для використання в автоматичному режимі.

П’ятий розділ присвячено розробці інструмента комбінованого аналізу ПЗ. При розробці цього засобу використовувались вимоги, завдання та ідеї комбінованого аналізу, сформовані в попередніх розділах. В розділі описано основні програмні компоненти розробленого проекту, сформовано графічний інтерфейс користувача, а також окремо розглянуто засоби статичного аналізу вихідного коду і статичного і динамічного аналізу бінарного коду.

В шостому розділі було проведено тестування розробленого в попередньому розділі інструмента комбінованого аналізу ПЗ. В рамках цього тестування було підтверджено, що використання запропонованого підходу допомагає вирішувати поставленні в рамках даної роботи завдання, це в свою чергу дозволило зробити позитивний висновок про перспективу використання комбінованого аналізу ПЗ в процесі його розробки.

В останньому розділі сформовану загальну послідовність дій для використання комбінованого аналізу ПЗ та запропоновано схему інтеграції в сучасну методологію розробки S-SDLC.

В подальших роботах планується розвивати тему комбінованого аналізу та передбачається його покращення та оптимізація. Зокрема, як було розглянуто в даній роботі, інструменти аналізу використовують внутрішнє представлення програми. Таке внутрішнє представлення може формувати дані для інших видів аналізу, наприклад, статичний аналіз бінарного коду може формувати вхідні дані для динамічного аналізу. Такий підхід допоможе збільшити покриття програми тестовими сценаріями.

ПЕРЕЛІК ПОСИЛАНЬ

1. Common Vulnerabilities and Exposures [Електронний ресурс] – Режим доступу до ресурсу: https://cve.mitre.org/.

2. Source Code Analysis Tools [Електронний ресурс] – Режим доступу до ресурсу: https://owasp.org/www-community/Source_Code_Analysis_Tools.

3. IEEE 1044-2009 Standard Classification for Software Anomalies. IEEE. 3 Park Avenue, New Yourk, NY 10016-5997, USA, 7 January 2010, ISBN 978-0-7381-6114-3.

4. About CWE [Електронний ресурс] – Режим доступу до ресурсу: https://cwe.mitre.org/about.

5. About CVE [Електронний ресурс] – Режим доступу до ресурсу: https://cve.mitre.org/about.

6. Common Vulnerability Scoring System SIG [Електронний ресурс] – Режим доступу до ресурсу: https://www.first.org/cvss/.

7. Аветисян А. Технології статичного та динамічного аналізу вразливостей програмного забезпечення / Арутюн Аветисян., 2014.

8. Steve L. The Trustworthy Computing Security Development Lifecycle / Lipner Steve., 2005.

9. Gallaher M. P. and Kropp B. M. Economic impacts of inadequate infrastructure for software testing. Technical report, RTI International, National Institute of Standards and Technology, US Dept of Commerce, May 2002.

10. Forrest Shull, Vic Basili, Barry Boehm, Winsor A. Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fi ghting defects. In International Software Metrics Symposium. Ottawa, Canada, 2002.

11. Lint [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Lint_(software).

12. Kunst F. Lint, a C Program Checker / Frans Kunst. – Amsterdam: Vrije Universiteit, 1988.

13. Iulian N. Understanding Source Code Evolution Using Abstract Syntax Tree Matching / Neamtiu Iulian. – Maryland, 2005. – 5 с.

14. CVE-2014-1266 [Електронний ресурс] – Режим доступу до ресурсу: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1266.

15. Apple's SSL/TLS bug (22 Feb 2014) [Електронний ресурс] – Режим доступу до ресурсу: https://www.imperialviolet.org/2014/02/22/applebug.html.

16. CVE-2014-1776 [Електронний ресурс] – Режим доступу до ресурсу: https://web.archive.org/web/20170430095220/http:/www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1776.

17. Yuwei Z. Automated defect identification via path analysis-based features with transfer learning / Zhang Yuwei. – 2020. – С. 166.

18. Аветисян А. Комбінований (статичний та динамічний) аналіз бінарного коду / А. Аветисян, А. Тихонов., 2012. – 20 с.

19. IDA Pro Disassembler [Електронний ресурс] – Режим доступу до ресурсу: http://www.hex-rays.com/idapro.

20. Sanjay R. Combining Static and Dynamic Analysis for Vulnerability Detection / Rawat Sanjay. – Gi`eres, France: University of Grenoble, 2013. – 15 с.

21. Ball T. The concept of dynamic analysis / Ball. – European Software Engineering Conference.: Lecture Notes in Computer Science, 1999.

22. Федюшин О., Ярещенко В. Технологія комбінованого аналізу програмного забезпечення як перспективний напрям пошуку вразливостей під час його розробки / О. Федюшин, В. Ярещенко. VIII Міжнародна науково-технічна конференція «Інформатика, управління та штучний інтелект»; ІУШІ-2021, м. Харків, 24-26 листопада 2021р. –С. 134.

23. The Qt Company. Qt Framework Documentation [Електронний ресурс] / The Qt Company – Режим доступу до ресурсу: https://doc.qt.io/.

24. Andy C. CMake Documentation [Електронний ресурс] / Cedilnik Andy – Режим доступу до ресурсу: https://cmake.org/documentation/.

25. Cppcheck Team. Cppcheck Manual [Електронний ресурс] / Cppcheck Team – Режим доступу до ресурсу: https://cppcheck.sourceforge.io/manual.pdf.

26. CWE Checker sources repository [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/fkie-cad/cwe_checker.

27. Julian S. Valgrind Documentation [Електронний ресурс] / Seward Julian – Режим доступу до ресурсу: https://valgrind.org/docs/.