Міністерство освіти і науки України
Харківський національний університет імені В.Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Спеціальність 125 «Кібербезпека»
Освітня програма «Кібербезпека»

В.о. зав. кафедрою КІСМіТ
Марина ЄСІНА
«Допущено до захисту»
« » _____ 2025р.

Пояснювальна записка

до кваліфікаційної роботи бакалавра

на тему: «Використання динамічного аналізу для виявлення витоку пам'яті та вразливостей безпеки в програмному забезпеченні»

оцінка « _____ »
Голова ЕК
Кагановська Т.Є.

Керівник: к.т.н. Олешко О.І.
Рецензент: PhD Лисицький К.Є.
Виконавець: студентка групи КБ-41
Гілязова А.І.

Харків 2025

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра містить 41 сторінку, 9 рисунків, 4 таблиці, 3 додатки, 22 посилання на використані джерела.

Мета роботи полягає у дослідженні можливостей динамічного аналізу для виявлення витоків пам’яті та вразливостей безпеки в програмному забезпеченні, написаного мовами C/C++. У рамках досягнення цілі було систематизовано типи вразливостей, пов’язаних з керуванням пам’яттю, проаналізовано методологію динамічного аналізу, проведено порівняльне дослідження сучасних інструментів (Valgrind, AddressSanitizer, Dr. Memory) та експериментально оцінено їхню ефективність на тестових прикладах. Також запропоновано шляхи інтеграції цих інструментів у процеси автоматизованої розробки ПЗ.

Об’єкт дослідження – процеси виявлення вразливостей безпеки, пов’язаних із неправильним керуванням пам’яттю, за допомогою динамічного аналізу коду.

Предмет дослідження – основні інструменти динамічного аналізу, їх функціональні можливості, точність виявлення помилок, продуктивність, а також особливості їхнього застосування у реальних проектах.

Основними методами дослідження є аналіз науково-технічної літератури, порівняльний аналіз інструментів динамічного аналізу, розробка тестового коду з контрольованими вразливостями, проведення експериментального дослідження, аналіз отриманих результатів та формулювання практичних рекомендацій.

У роботі представлено теоретичні основи динамічного аналізу, класифікацію найпоширеніших вразливостей керування пам’яттю, огляд інструментів динамічного аналізу, результати їхнього порівняльного тестування, а також пропозиції щодо оптимізації процесів тестування безпеки програмного забезпечення.

Отримані результати можуть бути використані розробниками, тестувальниками та фахівцями з кібербезпеки для підвищення якості й безпеки програмного забезпечення, а також у навчальному процесі при вивченні методів аналізу коду та безпечного програмування.

Ключові слова: ДИНАМІЧНИЙ АНАЛІЗ, ВРАЗЛИВОСТІ БЕЗПЕКИ, КЕРУВАННЯ ПАМ’ЯТТЮ, VALGRIND, ADDRESSSANITIZER, DR. MEMORY, ТЕСТУВАННЯ БЕЗПЕКИ, CI/CD, БЕЗПЕЧНЕ ПРОГРАМУВАННЯ, OWASP, CWE, СТАТИЧНИЙ АНАЛІЗ, ІНСТРУМЕНТИ АНАЛІЗУ, АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ.

ABSTRACT

The explanatory note to the bachelor's qualification work contains 41 pages, 9 figures, 4 tables, 3 appendices, and 22 references to the sources used.

The objective of this work is to investigate the possibilities of dynamic analysis for detecting memory leaks and security vulnerabilities in software written in C/C++. Within the framework of achieving this goal, types of memory-related vulnerabilities were systematized, the methodology of dynamic analysis was analyzed, a comparative study of modern tools (Valgrind, AddressSanitizer, Dr. Memory) was conducted, and their effectiveness was experimentally evaluated using test examples. Additionally, ways to integrate these tools into automated software development processes are proposed.

The subject of the research is the processes of identifying security vulnerabilities associated with improper memory management through dynamic code analysis.

The object of the research includes major dynamic analysis tools, their functional capabilities, accuracy in detecting errors, performance characteristics, and peculiarities of their application in real-world projects.

The main research methods include analysis of scientific and technical literature, comparative analysis of dynamic analysis tools, development of test code with controlled vulnerabilities, conducting experimental studies, analyzing the obtained results, and formulating practical recommendations.

The work presents theoretical foundations of dynamic analysis, classification of the most common memory-related vulnerabilities, an overview of dynamic analysis tools, results of their comparative testing, as well as suggestions for optimizing software security testing processes.

The obtained results can be used by developers, testers, and cybersecurity specialists to improve the quality and security of software products, as well as in the educational process when studying code analysis methods and secure programming practices.

Keywords: DYNAMIC ANALYSIS, SECURITY VULNERABILITIES, MEMORY MANAGEMENT, VALGRIND, ADDRESSSANITIZER, DR. MEMORY, SECURITY TESTING, CI/CD, SECURE PROGRAMMING, OWASP, CWE, STATIC ANALYSIS, ANALYSIS TOOLS, TEST AUTOMATION.

ЗМІСТ

ПЕРЕЛІК ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

ПЗ	–	Програмне забезпечення
CWE	–	Common Weakness Enumeration
NVD	–	National Vulnerability Database
ОС	–	Операційна система
ASan	–	AddressSanitizer
MSan	–	MemorySanitizer
TSan	–	ThreadSanitizer
CI	–	Continuos Integration
CD	–	Continuos Delivery

ВСТУП

Одним із найпоширеніших висловів щодо інформації в глобальній мережі є: «Уся інформація, що потрапила в мережу, залишається там назавжди». Це ствердження особливо актуальне сьогодні, коли кожна людина, що користується цифровими технологіями, хоче мати можливість безпечно та ефективно використовувати власну персональну інформацію лише в межах обраного кола.

Проте, забезпечення конфіденційності, цілісності та доступності даних – це не лише відповідальність окремого користувача, який має використовувати складні паролі, двофакторну автентифікацію чи шифрування, а й пряма відповідальність розробників, які повинні забезпечити високий рівень безпеки на етапі проектування та реалізації програмного забезпечення (ПЗ).

Однією з найкритичніших проблем у сфері безпеки є вразливості, пов’язані з помилками керування пам’яттю. Згідно зі звітами MITRE (CWE/SANS Top 25), більшість критичних вразливостей пов’язані з некоректною роботою з динамічною пам’яттю. Найпоширенішими серед них є buffer overflow, use-after-free, double free, memory leak тощо. Ці типи помилок можуть призводити до витоку конфіденційних даних, втрати контролю над системою або повного компрометування інфраструктури [1].

Провідні технологічні компанії, зокрема Google, Microsoft, Intel, активно впроваджують автоматизовані методи аналізу коду для виявлення та усунення таких помилок на ранніх етапах розробки. Наукові дослідження (наприклад, роботи К. Коуен, Д. Вагнера) підкреслюють важливість комбінації статичного, напівдинамічного та динамічного аналізу для підвищення якості ПЗ [2][3].

Динамічний аналіз набирає особливого значення завдяки можливості виявлення вразливостей у процесі виконання програми, що забезпечує високу точність результатів для конкретних сценаріїв використання. До найпопулярніших інструментів динамічного аналізу належать Valgrind, AddressSanitizer, Dr. Memory, LeakSanitizer, які широко використовуються як у відкритих проектах, так і в комерційному ПЗ.

Незважаючи на широке поширення зазначених інструментів, існують невирішені проблеми їхнього застосування. Серед них – обмежена підтримка деякими з них сучасних операційних систем, недостатня ефективність при аналізі масштабних проектів, а також слабка інтеграція з CI/CD-пайплайнами для регулярного тестування безпеки на етапі розробки. Це зумовлює актуальність подальших досліджень у даному напрямку.

Результати дослідження можуть бути використані в інженерній практиці розробників, тестувальників та фахівців з кібербезпеки для підвищення якості та безпеки ПЗ, а також у навчальному процесі при вивченні методів аналізу коду та безпечного програмування.

1 АНАЛІЗ ПРОБЛЕМАТИКИ ВРАЗЛИВОСТЕЙ У ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННІ

1.1 Поняття вразливостей безпеки

Вразливість безпеки – це слабке місце в програмній логіці (наприклад, коді), апаратних компонентах або інфраструктурі, яке при експлуатації може порушити конфіденційність, цілісність або доступність системи. Усунення виявлених вразливостей зазвичай передбачає модифікацію вихідного коду або зміну конфігурації системи. У деяких випадках, особливо коли вразливість пов'язана з застарілим або небезпечним функціоналом, необхідним може бути повне вилучення окремих компонентів, оновлення специфікацій або перегляд архітектурних рішень [4].

1.2 Класифікація вразливостей безпеки

Класифікація вразливостей є важливим етапом у процесі їх аналізу та усунення. Однак через велику різноманітність факторів, таких як тип операційної системи, використовуваний технологічний стек, специфіка інфраструктури та пріоритети застосування, ця задача є досить складною. Для стандартизації підходів до класифікації використовують кілька загальноприйнятих систем, серед яких можна виділили – OWASP Top Ten Project, Common Weakness Enumeration (CWE), National Vulnerability Database (NVD).

OWASP Top Ten Project – є загальновизнаним галузевим стандартом для ідентифікації найбільш критичних вразливостей веб-додатків. Згідно з результатами звіту OWASP Top Ten за 2021 рік, серед ключових загроз виділяються порушення контролю доступу (Broken Access Control), проблеми, пов’язані з ін’єкціями (Injection), а також помилки при обробці даних (Insecure Deserialization, XSS) [5]. Документ широко використовується як практичне керівництво для розробників, тестувальників та фахівців з кібербезпеки.

CWE – це формалізований перелік поширених помилок проєктування та реалізації програмного і апаратного забезпечення, які можуть призвести до вразливостей безпеки. Розроблений і підтримуваний організацією MITRE, CWE служить основою для опису причин вразливостей, а не їх наслідків. Кожна одиниця у списку CWE має унікальний ідентифікатор (наприклад, CWE-787: Out-of-bounds Write) та супроводжується описом, прикладами прояву, потенційними наслідками та методами запобігання [6]. CWE також є основою для більш високорівневих класифікацій, таких як CWE/SANS Top 25, які вказують на найнебезпечніші помилки програмування. Серед них, зокрема, виділяються:

- Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting');
- Out-of-bounds Write;
- Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

NVD – це урядове сховище даних управління вразливостями на основі стандартів, представлених за допомогою протоколу автоматизації безпечного контенту (SCAP). NVD забезпечує автоматизацію процесів управління вразливостями, вимірювання рівня безпеки та дотримання нормативних вимог. База включає дані про посилання на контрольні списки безпеки, недоліки ПЗ, продукти, які впливають на безпеку, а також показники їхнього впливу. Проєкт був започаткований у 1999 році як ICAT, а згодом перетворився на NVD. На сьогодні NVD вважається одним із найавторитетніших джерел інформації для фахівців у галузі кібербезпеки [7].

1.3 Актуальні загрози, пов’язані з витоками пам’яті

Витік пам’яті є типовою помилкою керування динамічною пам’яттю, яка виникає у разі, коли програма не звільняє блок пам’яті, який більше не використовується. Це призводить до поступового накопичення невикористаної пам’яті, що спричиняє зростання споживання системних ресурсів, зниження продуктивності або навіть аварійне завершення роботи програми.

Хоча сам по собі витік пам'яті не завжди становить прямий ризик для безпеки даних, він може бути симптомом глибших проблем у коді, серед яких – вразливості, що дають зловмисникам змогу порушити цілісність чи доступність системи. Найчастіше такі проблеми виникають через неправильне використання функцій виділення пам’яті, недбале звільнення ресурсів або наявність циклічних посилань у об'єктно-орієнтованих мовах програмування.

Згідно з рейтингом ця вразливість виникає, коли програма намагається записати дані за межами виділеного буфера пам’яті. Це може призвести до пошкодження важливих даних, аварійного завершення роботи програми або навіть виконання довільного коду зловмисником. Особливо небезпечна у мовах низького рівня, таких як C/C++, де відсутня автоматична перевірка меж масивів [9]. CWE Top 25 за 2024 рік, який наведено у Додатку А, існує кілька ключових категорій вразливостей, тісно пов’язаних з порушенням цілісності пам’яті. Серед них особливе місце займають такі типи вразливостей:

Out-of-Bounds Write (CWE-787). Ця вразливість виникає, коли програма намагається записати дані за межами виділеного буфера пам’яті. Це може призвести до пошкодження важливих даних, аварійного завершення роботи програми або навіть виконання довільного коду зловмисником. Особливо небезпечна у мовах низького рівня, таких як C/C++, де відсутня автоматична перевірка меж масивів [8].

Приклад:

```
int id_sequence[3];
/* Populate the id array. */
id_sequence[0] = 123;
id_sequence[1] = 234;
id_sequence[2] = 345;
id_sequence[3] = 456;
```

Оскільки масив виділено лише для зберігання трьох елементів, допустимі індекси від 0 до 2; отже, присвоєння до id_sequence[3] є недопустимим.

На рисунку 1.1 представлена візуалізація, що демонструє, як графічно виглядає процес переповнення буфера пам’яті.

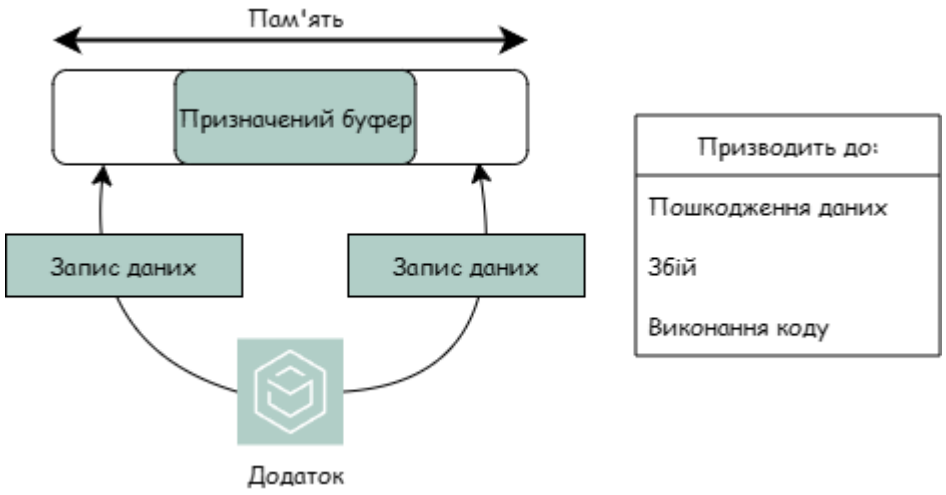

Рисунок 1.1 – Візуалізація вразливості Out-of-Bounds Write

Out-of-Bounds Read (CWE-125). Ця вразливість виникає, коли програма намагається прочитати дані за межами виділеного буфера. Це може призвести до витоку конфіденційної інформації, такої як ключі шифрування, паролі або внутрішні структури даних. Одним із найвідоміших прикладів є вразливість Heartbleed у бібліотеці OpenSSL, яка дозволяла атакувальникам зчитувати вміст оперативної пам’яті сервера [9].

Приклад:

```
#include <stdio.h>
int main() {
     int array[5] = {1, 2, 3, 4, 5};
     int index;
     printf("Enter an index: ");
     scanf("%d", &index);
     // не перевіряється, чи index знаходиться в межах масиву
     printf("Value at index %d: %d\n", index, array[index]);
     return 0;
}
```

У цьому прикладі користувач вводить індекс, за яким програма звертається до елемента масиву array. Якщо користувач введе значення, що виходить за межі допустимого діапазону (наприклад, 10 або -1), відбудеться спроба зчитування даних із області пам’яті, яка не належить цьому масиву.

На рисунку 1.2 показано, як виглядає графічне представлення виходу за межі буфера пам’яті при читанні – вказівник звертається до комірки пам’яті, яка не входить у діапазон виділеного буфера.

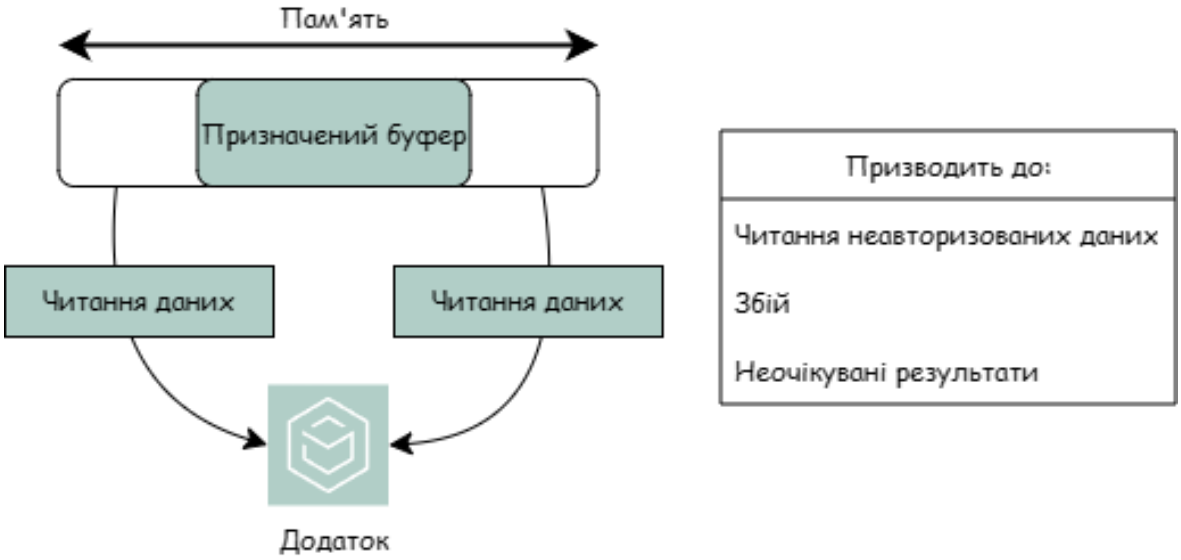

Рисунок 1.2 – Візуалізація вразливості Out-of-Bounds Read

Use After Free (CWE-416). Вразливість виникає, коли програма намагається отримати доступ до пам’яті, яка вже була звільнена. У цьому випадку пам’ять може бути повторно використана іншим процесом, і будь-яке звернення до неї може призвести до непередбачуваного стану програми або виконання довільного коду [10].

Приклад:

```
char* ptr = (char*)malloc (SIZE);
if (err) {
     abrt = 1;
     free(ptr);
}
if (abrt) {
     logError("operation aborted before commit", ptr);
}
```

При виникненні помилки вказівник негайно звільняється. Однак пізніше цей покажчик буде некоректно використано у функції logError.

На рисунку 1.3 подано візуалізацію, що демонструє, як вказівник продовжує звертатися до пам’яті, яка вже була звільнена, що може призвести до критичних помилок або витоків даних.

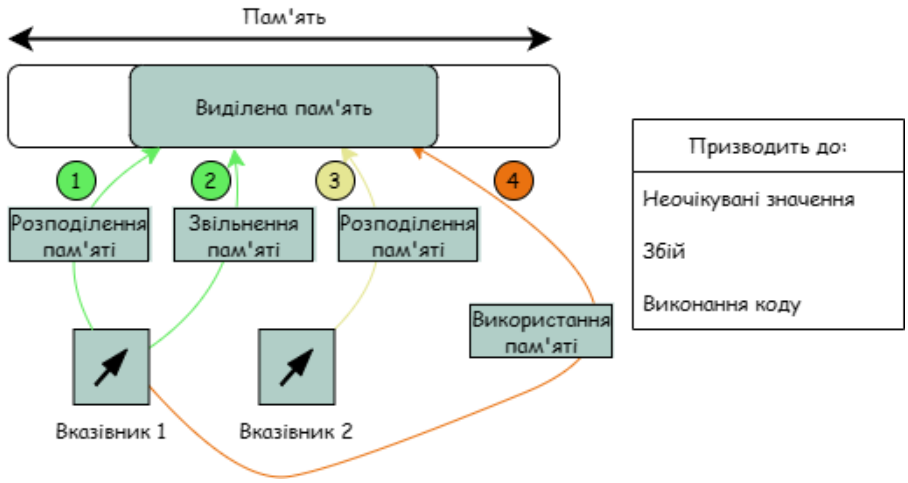

Рисунок 1.3 – Візуалізація вразливості Use After Free

Improper Restriction of Operations within the Bounds of a Memory Buffer (CWE-119). Це узагальнена категорія помилок, що охоплює як читання, так і запис за межами буфера. Вона включає попередні дві категорії (CWE-787, CWE-125) як окремі підвиди. Вразливості цього типу часто використовуються у складних експлойтах для підвищення привілеїв [11].

На рисунку 1.4 представлена візуалізація, що демонструє, як програма виходить за межі виділеного буфера пам’яті.

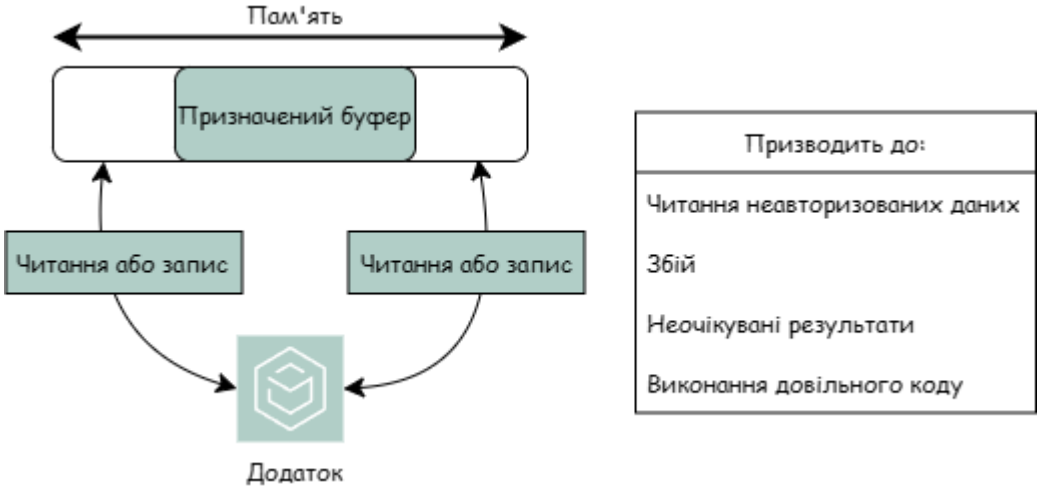

Рисунок 1.4 – Візуалізація вразливості Improper Restriction of Operations within the Bounds of a Memory Buffer

NULL Pointer Dereference (CWE-476). Дана вразливість виникає при спробі звернення до вказівника, що не вказує на жодну дійсну область пам’яті. Це може призвести до краху програми або DoS-атаки. Вона часто зустрічається у низькорівневому ПЗ та може бути наслідком інших помилок у логіці роботи з пам’яттю [12].

Приклад:

```
void host_lookup(char *user_supplied_addr) {
    struct hostent *hp;
    in_addr_t *addr;
    char hostname[64];
    in_addr_t inet_addr(const char *cp);

    // Перевірка формату IP-адреси
    validate_addr_form(user_supplied_addr);

    addr = inet_addr(user_supplied_addr);
    hp = gethostbyaddr(addr, sizeof(struct in_addr), AF_INET);

    // Помилка: не перевіряється чи hp != NULL
    strcpy(hostname, hp->h_name); // Вразливість CWE-476
}
```

У наведеному прикладі функція gethostbyaddr() може повернути значення NULL, якщо хост не знайдено. Якщо цей результат не перевірити перед використанням, то спроба звернення до hp->h_name призведе до розіменування нульового вказівника – тобто до вразливості CWE-476.

Крім того, цей код також вразливий до переповнення буфера (CWE-119) через використання небезпечного виклику strcpy(), який не перевіряє довжину рядка перед копіюванням.

На рисунку 1.5 показано графічне представлення процесу виконання програми, коли вказівник має значення NULL, але все одно використовується для доступу до даних.

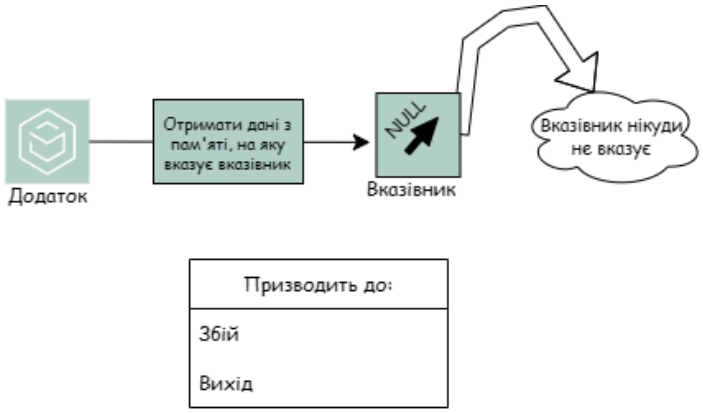

Рисунок 1.5 – Візуалізація вразливості NULL Pointer Dereference

2 ТЕОРЕТИЧНІ ОСНОВИ ДИНАМІЧНОГО АНАЛІЗУ

2.1 Поняття та цілі динамічного аналізу

Динамічний аналіз – це спосіб оцінювання комп'ютерної системи, програмого забезпечення чи додатку через запуск та спостереження за його діями й результатами під час виконання. На відміну від статичного аналізу, який аналізує код без його запуску, динамічний аналіз дозволяє перевірити реальну поведінку системи, включаючи управління пам’яттю, взаємодію з мережею або зовнішніми ресурсами [13].

Основними цілями динамічного аналізу є виявлення:

- Витоків пам’яті;
- Проблем продуктивності;
- Несумісностей із операційним середовищем;
- Нестабільної поведінки програми

Процес зазвичай включає наступні етапи:

1) Запуск програми в ізольованому тестовому середовищі, що моделює реальні умови.
2) Подача різноманітних вхідних даних, включаючи граничні значення та потенційно шкідливі запити.
3) Спостереження за поведінкою програми, використанням ресурсів та взаємодією з системою.
4) Аналіз зібраних даних для виявлення аномалій, вразливостей чи інших проблем.

2.2 Порівняння динамічного і статичного аналізу

Динамічний та статичний аналіз є двома взаємодоповнюючими підходами до тестування ПЗ, кожен з яких має свої цілі, методи та переваги. Нижче в таблиці 2.1 наведено ключові відмінності між ними:

Таблиця 2.1 – Порівняння статичного та динамічного аналізу

Критерій	**Статичний аналіз**	**Динамічний аналіз**
Виконання	Без запуску програми	Під час виконання програми
Об’єкт аналізу	Вихідний код, документація, архітектура	Реальна поведінка програми у різних сценаріях
Приклади методів	– Огляд коду – Аналіз потоків даних, – Інструменти (SonarQube, ESLint)	– Модульне тестування – Інтеграційні тести – Фаззинг
Етапи	Ранні етапи розробки (наприклад, проектування, написання коду).	Пізні етапи (наприклад, тестування перед релізом)
Цілі	– Запобігання помилкам до їх появи в продакшені – Перевірка відповідності стандартам коду	– Виявлення помилок часу виконання(витоків пам’яті, переповнень буфера) – Тестування продуктивності
Переваги	– Швидкість – Виявлення дефектів до компіляції	– Тестування в реальному середовищі – Можливість виявити динамічні помилки
Недоліки	– Не виявляє помилок, що залежать від вхідних даних або стану системи – Ризик «хибних спрацьовувань»	– Вимагає часу на підготовку тестових сценаріїв. – Не охоплює всі можливі шляхи виконання коду.
Приклади використання	– Перевірка відповідності стандарту MISRA C – Сканування вразливостей	– Автоматизація тестування інтерфейсу (Selenium), – Навантажувальне тестування API (JMeter).

Жоден з методів не є універсальним. Статичний аналіз ефективний для раннього виявлення дефектів і покращення якості коду, тоді як динамічний аналіз дозволяє перевірити програму в умовах, максимально наближених до реальних. Оптимальний підхід передбачає їх комбінацію.

2.3 Методи виявлення витоку пам’яті

Для ефективного пошуку витоків пам’яті найчастіше застосовують методи динамічного аналізу. Вони дають можливість вивчати поведінку програми безпосередньо під час її виконання, що особливо корисно для виявлення помилок, які проявляються лише при певних умовах.

Одним із таких підходів є фаззинг – техніка, яка передбачає подачу на вхід програми некоректних, нестандартних або випадкових даних. Такий метод дозволяє перевірити, як система реагує на неочікувані ситуації та чи здатна вона протистояти потенційним атакам. Завдяки цьому можна знайти слабкі місця, які можуть бути використані зловмисниками, і вчасно їх усунути [14].

Ще одним важливим способом є моніторинг виконання програми. Він полягає в постійному спостереженні за характеристиками роботи додатка – наприклад, за обсягом використаної пам’яті, кількістю викликів функцій або доступом до системних ресурсів. Аналіз цих показників допомагає виявити відхилення від нормального режиму, які можуть свідчити про наявність помилок, зокрема витоків пам’яті [14].

Також широко застосовується інструментування коду – процес додавання спеціальної службової логіки всередину програми. Це дає можливість отримувати детальну інформацію про те, як використовуються ресурси, як відбувається розподіл і звільнення пам’яті, а також які функції активно задіяні. Отримані дані полегшують аналіз внутрішніх процесів програми й допомагають виявляти навіть складні, приховані проблеми, які важко помітити звичайними способами [14].

2.4 Взаємозв’язок між динамічним аналізом і тестуванням безпеки

Тестування безпеки ПЗ відіграє ключову роль у процесі розробки. Воно спрямоване на пошук і усунення вразливостей, які можуть стати причиною порушення конфіденційності, цілісності або доступності системи. Серед усіх методів тестування особливе місце займає динамічний аналіз, оскільки він дозволяє перевіряти роботу програми безпосередньо під час її виконання.

Саме завдяки динамічному аналізу можна виявити серйозні помилки керування пам’яттю, які проявляються лише в конкретних сценаріях роботи додатка. Наприклад, це може бути переповнення буфера, використання вже звільненої пам’яті, витік пам’яті або спроба звернення за нульовим адресом. Усе це є потенційно небезпечним і може призвести до аварійного завершення роботи програми або навіть до реалізації шкідливого коду.

Крім того, інструменти динамічного аналізу активно використовуються на етапах тестування та автоматизованої збірки перед випуском нових версій ПЗ. Це дає змогу вчасно виявити помилки, які могли залишитися непоміченими під час статичного аналізу або при ручному перегляді коду.

Особливо популярним методом у тестуванні безпеки є фаззинг. Він передбачає генерацію великої кількості псевдовипадкових або модифікованих вхідних даних, метою яких є викликання несподіваної реакції з боку програми. Інструменти, такі як AFL++, libFuzzer, Honggfuzz або OWASP ZAP для веб-додатків, намагаються знайти комбінації вхідних значень, що призводять до збоїв. У цьому процесі динамічний аналіз виступає в ролі спостерігача: він фіксує моменти аварійного завершення, помилки доступу до пам’яті та інші аномалії.

Не менш важливим є також використання песочниць (sandboxing). Цей підхід передбачає запуск програми в ізольованому середовищі, що дозволяє безпечно виконувати сумнівний або недостатньо перевірений код. Така ізоляція значно зменшує ризики, пов’язані з виконанням потенційно небезпечних операцій.

Отже, динамічний аналіз є невід’ємною частиною сучасного тестування безпеки, особливо коли мова йде про помилки управління пам’яттю, які виявляються лише під час роботи програми. Поєднання фаззингу, песочниць та

інших методів динамічного аналізу забезпечує комплексний підхід до пошуку слабких місць, підвищує точність результатів і скорочує час на виправлення помилок. Разом ці технології не лише покращують загальну якість ПЗ, але й суттєво знижують ризики, пов’язані з виявленими вразливостями.

3 ІНСТРУМЕНТИ ДИНАМІЧНОГО АНАЛІЗУ

3.1 Огляд сучасних інтрументів

Для найточнішого виявлення вразливостей безпеки та помилок керування пам’яттю в ПЗ розроблено спеціалізовані інструменти динамічного аналізу. Фактично, існує цілий спектр таких автоматизованих засобів. Вони перевіряють роботу програми під час її виконання, щоб точно виявляти такі недоліки, як переповнення буфера, витоки пам’яті, використання звільненої пам’яті тощо. Наступні сім поточних інструментів динамічного аналізу практично мають значення для наукових і прикладних досліджень [15].

Valgrind – це фреймворк для створення інструментів динамічного аналізу, який надає широкий спектр можливостей для аналізу виконання програм. Найвідоміший інструмент у складі Valgrind – Memcheck – виявляє витоки пам’яті, некоректне використання вказівників та інші проблеми керування пам’яттю. Valgrind підтримує кілька архітектур процесорів і може використовуватися для тестування на різноманітних платформах[16].

AddressSanitizer (ASan) – це швидкий детектор помилок керування пам'яттю для мов C та C++. Інструмент реалізується як частина компіляторів Clang або GCC і виявляє такі типи вразливостей, як Out-of-Bounds Read/Write, Use After Free, Double Free, Heap Use After Return тощо [17].

Dr. Memory – це інструмент моніторингу пам'яті, здатний виявляти помилки програмування, пов'язані з пам'яттю, такі як доступ до неініціалізованої пам'яті, доступ до неадресованої пам'яті (в тому числі за межами виділених блоків купи, недоповнення та переповнення купи), доступ до звільненої пам'яті, подвійне звільнення, витоки пам'яті, помилки використання GDI API, а також доступ до незарезервованих слотів локальної пам'яті потоку [18].

KLEE – це символьна віртуальна машина, побудована на основі інфраструктури LLVM. Вона використовує символічне виконання для автоматичного пошуку помилок у програмах на мовах C та C++. KLEE дозволяє

генерувати тести, що покривають всі можливі шляхи виконання, і виявляє такі вразливості, як переповнення буфера, ділення на нуль, використання невиділеної пам’яті тощо [19].

angr – це міждисциплінарний фреймворк для аналізу бінарних файлів, розроблений у Каліфорнійському університеті в Санта-Барбарі (UCSB). Він підтримує символічне виконання, трасування, динамічний аналіз та інші методи аналізу без наявності вихідного коду. angr використовується для виявлення вразливостей у компільованих додатках, проведення реверс-інженерії та автоматичного генерування експлойтів. Цей інструмент широко застосовується в наукових дослідженнях та кіберрозвідці [20].

MemorySanitizer (Msan) – це інструмент для виявлення витоків пам’яті, інтегрований у LLVM/Clang. Він є частиною сімейства Sanitizer-інструментів і працює разом із ASan. MSan виявляє блоки пам’яті, які не були звільнені під час завершення програми, і надає точні звіти про місце виділення цих блоків [21].

ThreadSanitizer (Tsan) – це детектор гонки даних (data races) у багатопотокових додатках на мовах C/C++. TSan виявляє ситуації, коли два потоки одночасно здійснюють доступ до однієї комірки пам’яті без відповідної синхронізації. Це може призводити до непередбачуваної поведінки програми, включаючи порушення конфіденційності чи цілісності даних. TSan інтегрується з компіляторами Clang і GCC, забезпечуючи ефективний аналіз багатопотокових програм [22].

3.2 Порівняння інструментів

Для розумного оцінювання та вибору найкращих інструментів динамічного аналізу необхідно проаналізувати та порівняти їхню функціональність, технічні можливості, типи вразливостей, що виявляються, продуктивність, мови програмування, що підтримуються, та обмеження. У Додатку Б наведено таблицю порівняння різних інструментів динамічного аналізу.

По-перше, за здатністю їх виконання інструменти можна умовно розбити на:

- Санітайзери, інтеграція з якими відбувається на етапі компіляції і які дають можливість швидко виявити типові помилки в коді.
- Фреймворки для динамічного трасування, такі як Valgrind, Dr.Memory, які пропонують більш точне відстеження, але вносять значне падіння продуктивності.
- Інструменти символьного виконання коду, такі як KLEE, angr, які дають можливість розробнику навіть вивчати недетерміновану поведінку коду, що означає, що їх складно застосовувати на практиці.

По-друге, якщо розглядати продуктивність, з цією метою найшвидший санітайзер – ASan, а отже, він підходить для використання в безперервній інтеграції. На відміну від нього Valgrind і KLEE істотно уповільнюють виконання програми, тому рекомендовано використовувати ці інструменти для тестування вручну на етапі розробки.

По-третє, тільки angr дає змогу працювати з бінарними файлами, не маючи доступу до вихідного коду програм, що дуже важливо в разі реверс-інжинірингу та аудиту стороннього ПЗ.

Нарешті, самі види помилок, які можуть бути знайдені, можуть відрізнятися: наприклад, TSan ідеально підходить для знаходження проблем, пов'язаних із перегонами за даними в пам'яті, а MSan сконцентрується на виявленні таких помилок, як доступ до неініціалізованих змінних.

3.3 Вибір інструментів для експерименту: обґрунтування

Для реалізації експериментальної складової дослідження були задіяні три інструменти динамічного аналізу – Valgrind, ASan та Dr. Memory. Вибір визначався прагненням охопити різні стратегії виявлення помилок у керуванні пам'яттю, забезпечуючи точність, продуктивність і адаптивність під час тестування ПЗ.

Valgrind представляє собою традиційний інструмент аналізу пам’яті, що гарантує всебічний контроль за виділенням та вивільненням пам'яті. Його було відібрано для експерименту через його широку застосовуваність, наукове

супроводження та високу чутливість до помилок, пов'язаних із керуванням пам’яттю.

ASan є одним з лідерів серед інструментів, що застосовуються у комерційній та відкритій розробці, завдяки своїй швидкості та точності. Він забезпечує оптимальне співвідношення між продуктивністю та ефективністю аналізу, що робить його ідеальним вибором для перевірки безпеки на етапі автоматизованого тестування.

Dr. Memory є єдиним інструментом з аналогічним функціоналом, що працює на платформі Windows. Враховуючи, що значна частина комерційних застосунків розробляється саме для цієї платформи, включення Dr. Memory до експерименту дає змогу охопити весь спектр потенційних середовищ виконання та отримати порівняльні дані для різних операційних систем.

4 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

4.1 Підготовка тестового середовища

Для перевірки ефективності інструментів було створено ізольоване тестове середовище, до основних вимог якого належали: стабільність, повторювальність запусків, підтримку необхідних інструментів і можливість вимірювання продуктивності.

Усі експерименти виконувалися у двох окремих середовищах, щоб забезпечити підтримку всіх необхідних інструментів:

Linux-середовище

- ОС: Ubuntu 24.04.2 LTS (x86_64)
- Ядро: 6.11.0-21-generic
- Гіпервізор: KVM
- Компілятори:
  - GCC: 13.3.0
  - Clang: 18.1.3
- Інструменти
  - Valgrind: 3.22.0
  - AddressSanitizer
- Процесор: 4 ядра

Додатково: окреме Windows-середовище 10 x64 для тестування Dr. Memory.

Використання двох середовищ забезпечило комплексний підхід до тестування, підтримку всіх необхідних інструментів, а також дозволило тестувати як Linux-орієнтовані, так і Windows-орієнтовані рішення.

У кожному середовищі були встановлені інструменти, обрані на основі порівняльного аналізу в розділі 3.3:

- Valgrind 3.22.09
- AddressSanitizer (інтегрований у Clang)
- Dr. Memory 2.6.0 – у середовищі Windows

Кожен інструмент мав конфігурації згідно з рекомендаціями виробника, а також був налаштований за потребами експерименту. Можна виділити ключові моменти відповідно методики застосування:

- Valgrind запускався з флагами --leak-check=full, що забезпечують детальний аналіз витоків пам’яті.
- Asan активувався компіляторними прапорцями -fsanitize=address
- Dr. Memory запускали в інтерактивному режимі з аналізом логів. Використовувалася стандартна конфігурація, що автоматично виявляє витоки, некоректні операції з пам’яттю та інші проблеми.

Для перевірки інструментів було розроблено набір тестових програм мовою C, які містили контрольовані вразливості, такі як:

- Витік пам’яті
- Переповнення буфера
- Використання звільненої пам’яті
- Гонки даних у багатопотоковому середовищі

Тести були реалізовані з можливістю керування різноманітністю сценаріїв, щоб випробувати як складні, так і прості приклади, що моделюють реальні ситуації в коді. Кожен окремий тест компілювався окремо з відповідними флагами для інструментів-санітайзерів, щоб забезпечити коректну роботу інструментів.

Параметри, які були зафіксовані під час тестування:

- Типи виявлених вразливостей
- Повнота звітів
- Точність локалізації помилок
- Продуктивність

4.2 Розробка тестових програм з контрольованими вразливостями

Для проведення експерименту з використанням інструментів динамічного аналізу у виявленні витоків даних та інших вразливостей безпеки було розроблено набір з 9 тестів мовою C, які охоплюють різни типи помилок керування памяттю, згрупованих у три категорії:

1) Базові помилки керування пам’яттю
    a. Витік пам’яті через незвільнений malloc()

       Ціль: виявлення невиділеної пам’яті у фінальному стані програми.
    b. Подвійне звільнення пам’яті

       Ціль: аналіз на некоректне повторне звільнення пам’яті.
    c. Доступ до звільненої ділянки пам’яті

       Ціль: виявлення ситуацій, коли звліьнена пам’ять залишається доступною для читання/запису.
2) Помилки роботи з буферами
    a. Переповнення стеку

       Ціль: виявлення виходу за межі виділеного простору.
    b. Переповнення динамічного буфера

       Ціль: виявлення виходу за межі виділеного простору.
    c. Переповнення глобального масиву

       Ціль: виявлення порушення меж у різних сегментах пам’яті.
3) Складні сценарії:
    a. Використання пам’яті після повернення з функції

       Ціль: виявлення використання недійсної пам’яті.
    b. Робота з некоректним вказівником

       Ціль: аналіз реакції інструментів на спроби розіменування нульового вказівника
    c. Гонка даних у багатопотоковому додатку

       Ціль: виявлення гонок даних

Кожен тест був розроблений як окремий .c файл, який містить вразливість, коментарі з посиланням на відповідну CWE, а також мінімальну кількість коду для уникнення зайвого впливу на результати.

Тести використовувалися для проведення подальшого експериментального дослідження, яке наведено у розділі 4.3.

Реалізація тестових прикладів наведена у Додатку В.

4.3 Застосування інструментів динамічного аналізу

Після створення тестового середовища та розробки сценаріїв із закладеними вразливостями було проведено серію експериментів з використанням трьох популярних інструментів динамічного аналізу – Dr. Memory, Valgrind та AddressSanitizer (ASan). Кожен із них застосовувався до дев'яти тестових програм, розроблених раніше, які моделювали реальні помилки керування пам'яттю, такі як витік пам'яті, подвійне звільнення або доступ до звільненої області пам'яті.

Експеримент проводився в два етапи. Спочатку перевірка виконувалася в Windows-середовищі за допомогою Dr. Memory. Тестові програми компілювалися в DevC++ і запускалися через командний рядок з використанням наступної команди: `"C:\Program Files (x86)\Dr. Memory\bin64\drmemory.exe" -show_reachable -- test.exe`

Другий етап проходив у Linux-середовищі. Для Valgrind компіляція здійснювалася за допомогою стандартної команди (`gcc -o test test.c`), а виконання з параметрами `valgrind --leak-check=full ./test`. Для Asan програми компілювалися з спеціальним прапором `-fsanitize=address`, наприклад, `gcc -fsanitize=address -o test test.c` Виконання здійснювалося звичайним чином: `./test1b_asan`.

Кожен інструмент генерував звіт про результати аналізу, які можна оцінювати як наступні параметри:

1) Виявлення – чи було знайдено вразливість (Так / Ні).
2) Детальність – обсяг і повнота наданої інформації (Детальна / Коротка).
3) Точність локалізації – наскільки точно вказано місце помилки (рядок коду, адреса, розмір блоку тощо).
4) Швидкість – час виконання тесту з урахуванням накладних витрат (Низька / Середня / Висока).

Результати тестування інструмента Dr. Memory наведено в таблиці 4.1. Інструмент успішно виявив усі типи вразливостей, окрім гонки даних, де виявилася менша чутливість, хоча система й повідомила про конфлікт у роботі з пам'яттю. Звіти містили тип помилки, адреси пам'яті та номери рядків коду. Швидкість

виконання може бути оцінена як середня, що робить цей інструмент придатним для комплексного аналізу.

Таблиця 4.1 – Результати тестування інструмента Dr. Memory

Тест	**Виявлення**	**Детальність**	**Точність локалізації**	**Швидкість**
Витік пам’яті	Так	Детальна	Розмір витоку	Середня
Подвійне звільнення пам’яті	Так	Детальна	Адреса	Середня
Доступ до звільненої пам'яті	Так	Детальна	Рядок	Середня
Переповнення стеку	Так	Детальна	Розмір	Середня
Переповнення динамічного буфера	Так	Детальна	Рядок	Середня
Переповнення глобального масиву	Так	Детальна	Розмір	Середня
Використання пам'яті після повернення з функції	Так	Детальна	Рядок	Середня
Розіменування NULL вказівника	Так	Детальна	Рядок	Середня
Гонка даних	Так	Детальна	Поток	Низька

Таблиця 4.2 демонструє результати для Valgrind. Він також виявив більшість проблем і надав докладні звіти, які включали розмір витоків та точне посилання на рядки коду. Проте інструмент не виявив переповнення глобального масиву і гонки даних, що вказує на певні обмеження. Незважаючи на це, точність і повнота даних залишилися на задовільному рівні. Швидкість аналізу в більшості випадків була нижча порівняно з іншими інструментами.

Таблиця 4.2 – Результати тестування інструмента Valgrind

Тест	**Виявлення**	**Детальність**	**Точність локалізації**	**Швидкість**
Витік пам’яті	Так	Детальна	Розмір витоку та рядок	Середня
Подвійне звільнення пам’яті	Так	Детальна	Рядок та адреса пам’яті	Середня

Продовження таблиці 4.2

Тест	**Виявлення**	**Детальність**	**Точність локалізації**	**Швидкість**
Доступ до звільненої пам'яті	Так	Детальна	Рядок та тип помилки	Середня
Переповнення стеку	Так	Коротка	Рядок та тип помилки	Низька
Переповнення динамічного буфера	Так	Детальна	Рядок та розмір блоку	Середня
Переповнення глобального масиву	Ні	--	--	Середня
Використання пам'яті після повернення з функції	Так	Детальна	Рядок	Низька
Розіменування NULL вказівника	Так	Детальна	Рядок	Низька
Гонка даних	Ні	--	--	Середня

У таблиці 4.3 наведено результати для ASan. Цей інструмент показав найвищі показники серед усіх тестованих засобів. Він швидко виявляв помилки, точно вказував їхнє джерело в коді та формував добре структуровані звіти. Як і Valgrind, ASan не підтримує виявлення гонок даних, однак у всіх інших категоріях він перевершував конкурентів. Цей інструмент забезпечує оптимальний баланс між продуктивністю та якістю виявлення помилок, що робить його особливо корисним для регулярного використання в системах безперервної інтеграції (CI/CD).

Таблиця 4.3 – Результати тестування інструмента ASanitizer

Тест	**Виявлення**	**Детальність**	**Точність локалізації**	**Швидкість**
Витік пам’яті	Так	Детальна	Розмір витоку та рядок	Висока
Подвійне звільнення пам’яті	Так	Детальна	Рядок та адреса пам’яті	Висока
Доступ до звільненої пам'яті	Так	Детальна	Рядок та тип помилки	Висока
Переповнення стеку	Так	Детальна	Рядок та розмір буфера	Висока
Переповнення динамічного буфера	Так	Детальна	Рядок та розмір блоку	Висока
Переповнення глобального масиву	Так	Детальна	Рядок та глобальна змінна	Висока
Використання пам'яті після повернення з функції	Так	Коротка	Рядок	Низька
Розіменування NULL вказівника	Так	Корокта	Рядок	Низька
Гонка даних	Ні	--	--	Висока

4.4 Аналіз отриманих результатів

Результати експериментального дослідження показали, що сучасні інструменти динамічного аналізу коду мають різноманітну ефективність у виявленні помилок, пов’язаних із керуванням пам’яттю. Ця різниця пояснюється специфікою технічних механізмів, які використовують кожен з них. Детальний аналіз отриманих даних дозволяє зрозуміти переваги та обмеження окремих засобів.

У категорії витоків пам’яті (CWE-401) найвищого рівня точності досяг Valgrind. Його методика базується на трасуванні всіх операцій виділення та звільнення пам’яті, що дає можливість точно визначати моменти, коли пам’ять не була коректно звільнена. Такий піхід реалізований через емуляцію процесора, що забезпечує глибоке логування стану пам’яті та контексту викликів. ASan також може виявляти витоки, однак це не є його основним функціоналом – він орієнтований на виявлення порушень доступу до пам’яті. Dr. Memory, хоча й менш деталізований, демонструє гарні результати на платформі Windows, що робить його привабливим для використання в цьому середовищі.

Щодо переповнень буфера (CWE-787, CWE-125) найбільш ефективним виявився ASan. Він використовує технологію тіньової пам’яті, яка дозволяє швидко та точно виявляти виходи за межі виділених блоків. ASan успішно визначає різні типи переповнень – стекові, heap-переповнення та проблеми з глобальними масивами. У порівнянні з ним Valgrind має обмежену чутливість до переповнень глобальних структур, що пояснюється його фокусом на динамічну пам’ять. Dr. Memory також виявляє цей тип вразливостей, але поступається за швидкодією.

При аналізі ситуацій «використання пам’яті після звільнення» (Use After Free, CWE-416) кращі результати продемонстрував ASan завдяки механізму «отруєння» звільнених областей. Це дозволяє швидко виявити небезпечні звернення до пам’яті, а також значно спрощує локалізацію помилки через трасування викликів. Valgrind також виявляє подібні помилки, але потребує більше ресурсів і видає менш структуровані звіти. Dr. Memory справився з тестом коректно, хоча рівень деталізації в звітах був нижчим.

Щодо розіменування нульового вказівника (NULL Pointer Dereference, CWE-476), усі три інструменти показали задовільну ефективність. Проте найвищу точність та швидкість виявлення забезпечив ASan. Це пояснюється його тісною інтеграцією з компіляторами Clang і GCC, а також невеликими накладними витратами.

Що стосується гонок даних (Data Race, CWE-362), жоден із розглянутих інструментів – ні ASan, ні Valgrind, ні Dr. Memory – не показав достатньої

чутливості. Це пояснюється відсутністю спеціалізованих механізмів для аналізу багатопотокових взаємодій. Для виявлення таких помилок доцільно використовувати спеціалізовані засоби, наприклад, TSan.

На рисунку 4.1 представлено порівняльний аналіз ефективності цих інструментів, який дозволяє наочно оцінити їхню продуктивність та чутливість.

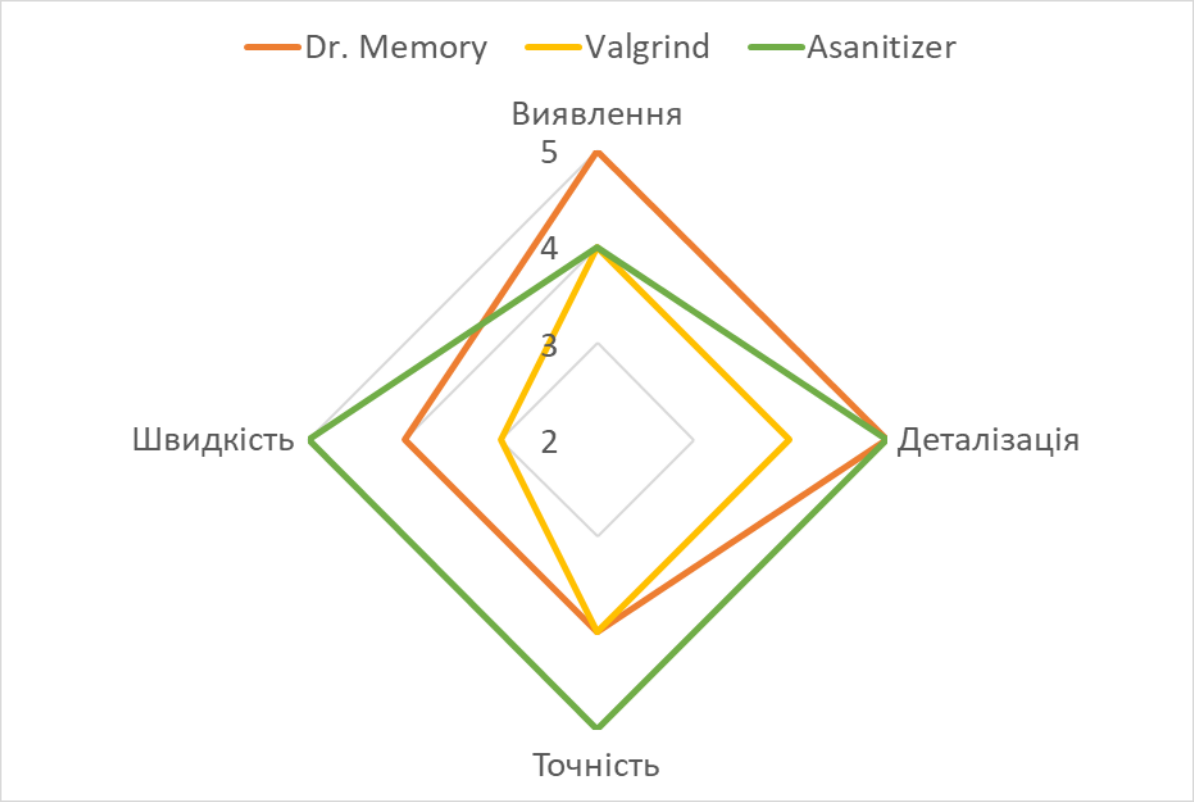

Рисунок 4.1 – Порівняння інструментів динамічного аналізу

Загалом, усі три інструменти добре справляються з виявленням базових вразливостей керування пам’яттю. ASan, проте, виявився найуніверсальнішим – він виявив усі вісім тестових вразливостей, окрім гонки даних, яка не входить до його функціоналу. Valgrind виявив більшість помилок, але не впіймав переповнення глобального масиву, що пов’язано з особливостями механізму Memcheck. Dr. Memory також виявив усі тестові вразливості, але у разі гонки даних звіт виявився менш точним, що свідчить про обмежену підтримку багатопотокового аналізу.

Детальність звітів має ключове значення для ефективного виправлення знайдених проблем. Найповніші звіти генерує ASan. Він надає точне вказівку на рядок коду, де виникла проблема, розмір витоку або переповнення, стан тіньової

пам’яті та історію виділення і звільнення пам’яті. Приклад такого звіту наведено на рисунку 4.2.

Valgrind також видає докладні звіти, але в окремих випадках, наприклад при аналізі переповнень стеку, їхня форма може бути важкою для інтерпретації. Зразок звіту Valgrind подано на рисунку 4.3.

Dr. Memory забезпечує добре збалансовані звіти – вони досить деталізовані, але одночасно зручні для розуміння, що робить цей інструмент придатним для повсякденного використання. Приклад звіту наведено на рисунку 4.4.

```
anna@anna-VirtualBox:~/memory_leak_tests$ ./test2c_asan
=================================================================
==62180==ERROR: AddressSanitizer: global-buffer-overflow on address 0x6183999c90ea at pc
0x799db90fb303 bp 0x7ffff720e180 sp 0x7ffff720d928
WRITE of size 24 at 0x6183999c90ea thread T0
    #0 0x799db90fb302 in memcpy
../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors_memintrinsic
s.inc:115
    #1 0x6183999c61ee in main /home/anna/memory_leak_tests/test2c.c:6
    #2 0x799db8c2a1c9 in __libc_start_call_main
../sysdeps/nptl/libc_start_call_main.h:58
    #3 0x799db8c2a28a in __libc_start_main_impl ../csu/libc-start.c:360
    #4 0x6183999c6104 in _start (/home/anna/memory_leak_tests/test2c_asan+0x1104)
(BuildId: 39b3bd928b822de18d80f73b5f0927d32112470f)

0x6183999c90ea is located 0 bytes after global variable 'global' defined in
'test2c.c:3:6' (0x6183999c90e0) of size 10
SUMMARY: AddressSanitizer: global-buffer-overflow
../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors_memintrinsic
s.inc:115 in memcpy
Shadow bytes around the buggy address:
  0x6183999c8e00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x6183999c8e80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x6183999c8f00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x6183999c8f80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x6183999c9000: 00 00 00 00 00 00 00 00 f9 f9 f9 f9 f9 f9 f9 f9
=>0x6183999c9080: f9 f9 f9 f9 f9 f9 f9 f9 00 00 00 00 00[02]f9 f9
  0x6183999c9100: f9 f9 f9 f9 00 00 00 00 00 00 00 00 00 00 00 00
  0x6183999c9180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x6183999c9200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x6183999c9280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x6183999c9300: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
```

Рисунок 4.2 – Приклад сгенерованого звіту ASanitizer

```
anna@anna-VirtualBox:~/memory_leak_tests$ valgrind --leak-check=full ./test2c_valgrind
==60857== Memcheck, a memory error detector
==60857== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==60857== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==60857== Command: ./test2c_valgrind
==60857==
==60857==
==60857== HEAP SUMMARY:
==60857==     in use at exit: 0 bytes in 0 blocks
==60857==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==60857==
==60857== All heap blocks were freed -- no leaks are possible
==60857==
==60857== For lists of detected and suppressed errors, rerun with: -s
==60857== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Рисунок 4.3 – Приклад сгенерованого звіту Valgrind

```
Error #1: UNADDRESSABLE ACCESS beyond top of stack: reading 0x000000000062fbe0-0x000000000062fbe8
8 byte(s)
# 0 .text                                   [../../../../../src/gcc-
4.9.2/libgcc/config/i386/cygwin.S:152]
# 1 _pei386_runtime_relocator               [C:/crossdev/src/mingw-w64-v3-git/mingw-w64-
crt/crt/pseudo-reloc.c:476]
# 2 __tmainCRTStartup                       [C:/crossdev/src/mingw-w64-v3-git/mingw-w64-
crt/crt/crtexe.c:280]
# 3 .l_start                                [C:/crossdev/src/mingw-w64-v3-git/mingw-w64-
crt/crt/crtexe.c:212]
# 4 KERNEL32.dll!BaseThreadInitThunk
Note: @0:00:00.066 in thread 2180
Note: 0x000000000062fbe0 refers to 408 byte(s) beyond the top of the stack 0x000000000062fd78
Note: instruction: or     $0x0000000000000000 (%rcx) -> (%rcx)

Error #2: REACHABLE LEAK 36 direct bytes 0x00000000016b01c0-0x00000000016b01e4 + 0 indirect bytes
<memory was allocated before tool took control>

Error #3: REACHABLE LEAK 16 direct bytes 0x00000000016b0210-0x00000000016b0220 + 0 indirect bytes
<memory was allocated before tool took control>

Error #4: REACHABLE LEAK 20 direct bytes 0x00000000016b0240-0x00000000016b0254 + 0 indirect bytes
<memory was allocated before tool took control>
```

Рисунок 4.4 – Приклад сгенерованого звіту Dr. Memory

Якщо говорити про точність локалізації, то ASan виходить на перше місце – він чітко вказує на конкретний рядок, змінну та стек викликів. Dr. Memory також показує високу точність, хоча вказівки можуть бути менш зрозумілими у великих проєктах. Valgrind забезпечує прийнятний рівень точності, але для деяких типів помилок, зокрема переповнень буфера, місце проблеми визначається менш чітко.

Продуктивність інструментів оцінювалася за часом виконання тестів та накладними витратами. ASan має найвищу швидкодію, що робить його оптимальним вибором для регулярного використання в CI/CD-системах. Valgrind, навпаки, характеризується значним уповільненням, що обмежує його застосування

у великих проєктах. Dr. Memory займає проміжне положення – його продуктивність є допустимою для більшості тестувальних сценаріїв.

Аналіз ефективності інструментів динамічного аналізу показав, що їх результати суттєво залежать від типу вразливості, умов тестування та операційного середовища. Разом із тим, максимальна користь від їх використання досягається через інтеграцію в стандартні процеси розробки ПЗ. Це дозволяє не лише підвищити загальну якість та безпеку коду, але й скоротити час на виявлення та виправлення критичних помилок. У наступному розділі буде розглянуто способи автоматизації динамічного аналізу та практичні рекомендації щодо його впровадження в робочі процеси розробників.

5 ІНТЕГРАЦІЯ ДИНАМІЧНОГО АНАЛІЗУ В ПРОЦЕСИ РОЗРОБКИ

5.1 Автоматизація аналізу

Одним із ключових факторів підвищення ефективності виявлення вразливостей у ПЗ є інтеграція динамічного аналізу на етапах автоматизованої розробки. Такий підхід дозволяє не тільки оперативно виявляти критичні помилки, пов’язані з управлінням пам’яттю (наприклад, витоки, подвійне звільнення, доступ до вже вивільненої пам’яті), але й значно скоротити витрати часу та ресурсів на їх усунення.

Традиційно аналіз безпеки проводився після завершення основного етапу розробки, що призводило до затримок у впровадженні та підвищення витрат на усунення виявлених дефектів. Сучасний підхід, особливо в рамках методології DevOps, впроваджує безпеку на кожному етапі життєвого циклу програмного продукту. У цьому контексті автоматизоване впровадження динамічного аналізу в процесах CI/CD (Continuous Integration / Continuous Delivery) відіграє ключову роль у забезпеченні постійного контролю за якістю та безпечністю коду.

Інструменти, такі як ASan, Valgrind та Dr. Memory, у процесі автоматизованого тестування дозволяють досягти кількох важливих результатів:

- оперативне виявлення вразливостей на ранніх етапах життєвого циклу програмного забезпечення;
- узгодженість результатів аналізу незалежно від гілки або середовища виконання;
- формування звітів про проблеми безпосередньо в системах контролю версій (наприклад, GitLab або GitHub);
- підтримання стабільної якості коду завдяки постійному моніторингу змін.

Автоматизація динамічного аналізу реалізується через платформи для безперервної інтеграції, такі як GitHub Actions, GitLab CI, Jenkins або Travis CI. У цих системах налаштовуються pipelines, які компілюють код із параметрами (наприклад, -fsanitize=address для ASan), виконують тестування в ізольованому

середовищі та створюють звіти у форматах, зручних для аналізу (XML, HTML тощо).

Важливим компонентом автоматизації є механізми блокування merge-запитів у разі наявності критичних помилок. Завдяки цьому забезпечується висока надійність основної гілки розробки, що особливо актуально для проєктів із посиленими вимогами до інформаційної безпеки.

Для масштабних чи довготривалих проєктів доцільно використовувати комплексні системи контролю якості коду, такі як SonarQube. Вони дозволяють об’єднати результати статичного і динамічного аналізу в одному середовищі, спрощуючи інтерпретацію даних та забезпечуючи централізоване управління якістю.

Практичний досвід і дослідження демонструють численні переваги автоматизації динамічного аналізу:

- раннє виявлення дефектів завдяки перевірці кожного коміту або збірки;
- стабільність результатів через гарантовану відтворюваність умов тестування;
- скорочення загального часу тестування за рахунок регулярних перевірок;
- стимулювання розробників до створення якісного і прогнозованого коду.

5.2 Практичні рекомендації

На основі результатів експериментального дослідження та аналізу можливостей сучасних інструментів динамічного аналізу було сформульовано практичні рекомендації щодо їхнього ефективного використання в процесах розробки ПЗ. Ці рекомендації адресовані як розробникам, тестувальникам і фахівцям з кібербезпеки, так і навчальним закладам, які займаються підготовкою кваліфікованих кадрів у галузі безпечного програмування.

Ефективність методів динамічного аналізу значною мірою залежить від правильного вибору інструментарію. Важливо враховувати специфіку проєкту, технологічну платформу та умови цільового середовища. Проведений порівняльний аналіз дозволив зробити висновки щодо оптимального застосування кожного з інструментів.

ASan продемонстрував найвищу ефективність при роботі в Linux-середовищах. Його перевагами є швидке виконання тестів, точна локалізація помилок, пов'язаних із керуванням пам’яттю, а також формування детальних звітів, які містять тип виявленої вразливості, стек викликів і адресу пам’яті. Цей інструмент успішно виявляє такі проблеми, як вихід за межі масиву, використання пам’яті після звільнення, подвійне звільнення та інші, що робить його гарним кандидатом для регулярного використання в CI/CD-процесах.

Dr. Memory виявився одним із найбільш універсальних інструментів для Windows-платформи. Він добре справляється з виявленням проблем, пов’язаних із управлінням пам’яттю – зокрема, переповнень глобальних масивів, доступу до звільнених областей чи повторного звільнення. Разом із тим, слід враховувати обмежену чутливість цього інструменту до гонок даних у багатопотокових додатках, що може вимагати додаткового застосування спеціалізованих засобів.

Valgrind, хоча й має високу чутливість до витоків пам’яті та широкий охоп охоплення різноманітних проблем, характеризується значним уповільненням виконання програми. Це робить його менш придатним для постійного використання в великих проєктах, однак він залишається цінним інструментом для наукових досліджень та глибокого аналізу життєвого циклу пам’яті.

З урахуванням отриманих результатів, для досягнення максимальної ефективності рекомендується комбіноване використання кількох інструментів динамічного аналізу, адаптованих до потреб конкретного проєкту. Так, у Linux-системах доцільно поєднувати ASan з ThreadSanitizer (TSan) для одночасного виявлення порушень доступу до пам’яті та гонок даних. У Windows-проєктах – використовувати Dr. Memory разом із Valgrind, щоб отримати більш повний аналіз стану пам’яті.

Для забезпечення високої якості та надійності ПЗ необхідно систематично проводити тестування безпеки на всіх етапах розробки. Такий підхід передбачає регулярну перевірку коду в умовах, що моделюють реальне використання програми. Особливу увагу слід звертати на модульне тестування окремих компонентів, особливо тих, що взаємодіють із динамічною пам’яттю. Не менш

важливим є інтеграційне тестування для контролю взаємодії між модулями, а також тестування граничних умов, спрямоване на аналіз реакції програми на некоректні чи шкідливі дані. Багатопотокове тестування також має ключове значення, особливо для сучасних додатків, де існує ризик виникнення гонок даних.

Регулярне проведення таких тестів забезпечує стабільність аналізу, скорочує час на виявлення та усунення вразливостей і підвищує довіру до кінцевого продукту. Кожен звіт, отриманий внаслідок динамічного аналізу, має бути уважно проаналізований розробниками. Основна увага має зосереджуватися на критичних помилках, таких як витоки пам’яті, переповнення буферів, некоректний доступ до пам’яті, а також потенційно небезпечних ситуаціях – наприклад, розіменуванні нульових вказівників чи умовах гонки. Також важливо точно визначати місце помилки: рядок коду, потік виконання, адресу пам’яті.

Для систематизації процесу аналізу та ефективного супроводу знайдених проблем рекомендується використовувати спеціалізовані платформи, такі як SonarQube. Ці інструменти дозволяють об'єднати результати статичного та динамічного аналізу в єдиному інтерфейсі, що значно полегшує виправлення виявлених дефектів.

Не менш важливим є підвищення освіченості розробників у питаннях безпечного програмування. Для цього доцільно організовувати регулярні тренінги та семінари, присвячені типовим вразливостям, таким як CWE. Ефективним підходом є використання прикладів із реальних проєктів як навчальних матеріалів, а також впровадження загальноприйнятих стандартів безпечного програмування. Такі заходи допомагають формувати свідомий підхід до розробки, знижують імовірність допущення критичних помилок уже на етапі написання коду та підвищують професійний рівень учасників проекту.

Щоб забезпечити регулярне та відтворюване тестування безпеки, слід автоматизувати процеси динамічного аналізу в середовищах CI/CD. Це дозволяє автоматично запускати перевірки при кожному коміті, блокувати злиття змін у разі наявності критичних помилок, а також генерувати стандартизовані звіти та інтегрувати їх із системами контролю версій. Для цього можна використовувати

такі платформи, як Jenkins, GitHub Actions, GitLab CI, які підтримують інтеграцію з ASan, Valgrind, Dr. Memory та іншими інструментами.

Отримані в ході дослідження результати мають значний потенціал для використання в освітньому процесі, особливо при вивченні дисциплін, пов’язаних із безпечним програмуванням, аналізом коду та кібербезпекою. Навчальні приклади, наведені в Додатку В, можуть послужити основою для лабораторних занять, курсових проектів та наукових досліджень. Такий підхід дає студентам змогу набути практичних навичок виявлення та усунення вразливостей у реальному коді, що є критично важливим у контексті підготовки кваліфікованих фахівців у галузі інформаційної безпеки.

ВИСНОВКИ

Аналіз актуальності дослідження свідчить про те, що проблема вразливостей, пов’язаних із неправильним керуванням пам’яттю, залишається однією з найгостріших у сучасному програмному забезпеченні. Це особливо важливо в умовах зростання складності систем і широкого застосування мов C/C++ у критичних інфраструктурах. За даними рейтингів CWE/SANS Top 25 і практики таких компаній, як Google та Microsoft, більшість серйозних уразливостей виникає саме через некоректну роботу з пам’яттю. Тому дослідження можливостей динамічного аналізу для виявлення цих помилок має велике наукове й практичне значення.

Перший розділ присвячено теоретичному аналізу питань безпеки ПЗ, зокрема вразливостей, пов’язаних із керуванням пам’яттю. У ньому розглянуто основні типи помилок, таких як виходження за межі масиву, використання звільненої пам’яті або витоки пам’яті, а також їх класифікацію за системами CWE, OWASP і NVD. Проаналізовано причини виникнення цих проблем і доведено необхідність застосування спеціалізованих інструментів, здатних виявити динамічні помилки, які статичний аналіз пропускає.

У другому розділі увагу зосереджено на методології динамічного аналізу – його принципах, меті та відмінностях від статичного підходу. Докладно описано ключові способи виявлення витоків пам’яті, серед яких інструментування коду, фаззинг і моніторинг виконання програми. Також показано, що динамічний аналіз тісно пов’язаний із сучасними методами тестування безпеки, наприклад, використанням sandboxing і fuzz-тестування.

Третій розділ містить порівняльний аналіз сучасних інструментів динамічного аналізу, зокрема Valgrind, ASan і Dr. Memory. На основі їхніх функціональних можливостей та продуктивності обрано три інструменти для подальшого експериментального дослідження. Розглянуто різні підходи до аналізу – трасування, компіляторна інструменталізація, символьне виконання – і показано, як кожен із них впливає на ефективність виявлення вразливостей.

Четвертий розділ присвячено експериментальному дослідженню. Для перевірки інструментів було розроблено набір тестових програм із контрольованими вразливостями, створено ізольовані середовища на Linux і Windows. Після запуску тестів у кожному середовищі проаналізовано точність виявлення, швидкодію, деталізацію звітів та загальну ефективність. Найвищу продуктивність і зручність звітів показав ASan, Valgrind виявився чутливим до витоків пам’яті, а Dr. Memory добре себе зарекомендував у Windows.

П’ятий розділ присвячено інтеграції динамічного аналізу в процеси CI/CD та розробки ПЗ. Описано способи налаштування автоматичних перевірок, генерації звітів і блокування merge-запитів при виявленні проблем. На основі отриманих результатів сформульовано рекомендації щодо вибору інструментів, регулярного тестування, правильного розуміння результатів і підвищення кваліфікації розробників.

Крім того, у цьому розділі запропоновано шляхи практичного використання отриманих результатів. Зокрема, ASan рекомендовано впровадити в стандартні проекти з відкритим кодом, Valgrind – використовувати для глибокого аналізу критичних компонентів, а тестові програми – як навчальний матеріал. Також висунуто пропозиції щодо розвитку підтримки багатопотокового аналізу через використання таких інструментів, як ThreadSanitizer.

У ході дослідження вдалося досягти всіх поставлених цілей: було класифіковано основні помилки, вивчено доступні інструменти, проведено експерименти та сформульовано практичні рекомендації. Проте частково залишилася невирішеною проблема виявлення багатопотокових вразливостей – жоден із вибраних інструментів не забезпечує повного покриття. Перспективним напрямком дослідження є інтеграція спеціалізованих засобів, таких як ThreadSanitizer або Helgrind, а також розробка гібридних моделей аналізу.

Отримані результати узгоджуються з підходами К. Коуен (2019), що стосуються комбінованого тестування, а запропонована інтеграція ASan у CI/CD доповнює дослідження Д. Вагнера (2021), розширюючи їх у напрямку практичного впровадження в умовах відкритого коду. [2][3]

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. MITRE Corporation. CWE/SANS Top 25 Most Dangerous Software Weaknesses [Електронний ресурс]. – 2023. – Режим доступу: https://cwe.mitre.org/top25
2. *Cowan C.* StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks / C. Cowan, C. Pu, D. Maier et al. // Proceedings of the 7th USENIX Security Symposium. — Berkeley, 1998. — 14 p.
3. *Wagner D.* Intrusion Detection via Static and Dynamic Analysis / D. Wagner, D. Dean // IEEE Symposium on Security and Privacy. — Oakland, 2001. — P. 15–21.
4. National Institute of Standards and Technology (NIST). *National Vulnerability Database (NVD)* [Електронний ресурс]. – 2023. – Режим доступу: https://nvd.nist.gov/vuln
5. OWASP Foundation. *OWASP Top Ten Project: 2021* [Електронний ресурс]. – 2023. – Режим доступу:https://owasp.org/www-project-top-ten/
6. MITRE Corporation. *Common Weakness Enumeration (CWE): Introduction* [Електронний ресурс]. – 2023. – Режим доступу: https://cwe.mitre.org/about/new_to_cwe.html
7. National Institute of Standards and Technology (NIST). *NVD General Overview* [Електронний ресурс]. – 2023. – Режим доступу: https://nvd.nist.gov/general
8. MITRE Corporation. CWE-125: Out-of-bounds Read [Електронний ресурс]. – 2023. – Режим доступу:https://cwe.mitre.org/data/definitions/125.html
9. MITRE Corporation. CWE-787: Out-of-bounds Write [Електронний ресурс]. – 2023. – Режим доступу:https://cwe.mitre.org/data/definitions/787.html
10. MITRE Corporation. CWE-416: Use After Free [Електронний ресурс]. – 2023. – Режим доступу:https://cwe.mitre.org/data/definitions/416.html
11. MITRE Corporation. CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer [Електронний ресурс]. – 2023. – Режим доступу:https://cwe.mitre.org/data/definitions/119.html
12. MITRE Corporation. CWE-476: NULL Pointer Dereference [Електронний ресурс]. – 2023. – Режим доступу: https://cwe.mitre.org/data/definitions/476.html
13. SENCODE: Dynamic Analysis – визначення та базові принципи [Електронний ресурс]. – 2023. – Режим доступу: https://sencode.co.uk/glossary/dynamic-analysis/
14. VPN Unlimited. Dynamic Analysis [Електронний ресурс]. – Режим доступу:https://www.vpnunlimited.com/ua/help/cybersecurity/dynamic-analysis?srsltid=AfmBOor-xplYPjYQ6QRosbava5lHNPHgTDUiyIUzQhnjfbCG_0tXZNnl

15. GitHub. Dynamic Analysis Tools [Електронний ресурс]. – Режим доступу: https://github.com/analysis-tools-dev/dynamic-analysis?tab=readme-ov-file
16. Valgrind – Instrumentation Framework for Building Dynamic Analysis Tools [Електронний ресурс]. – Режим доступу: https://valgrind.org/
17. GitHub. AddressSanitizer [Електронний ресурс]. – Режим доступу: https://github.com/google/sanitizers/wiki/AddressSanitizer
18. GitHub. Dr. Memory – Memory Monitoring Tool [Електронний ресурс]. – Режим доступу: https://github.com/DynamoRIO/drmemory
19. GitHub. KLEE – Symbolic Virtual Machine [Електронний ресурс]. – Режим доступу: https://github.com/klee/klee
20. GitHub. angr – Binary Analysis Framework [Електронний ресурс]. – Режим доступу: https://github.com/angr/angr
21. GitHub. MemorySanitizer [Електронний ресурс]. – Режим доступу: https://github.com/google/sanitizers/wiki/MemorySanitizer
22. GitHub. ThreadSanitizer – C++ Manual [Електронний ресурс]. – Режим доступу:https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

ДОДАТОК А

ТАБЛИЦЯ CWE TOP 25 ЗА 2024 РІК

Нижче у таблиці А.1 наведено список 25 найпоширеніших типів уразливостей у програмному забезпеченні за версією MITRE за 2024 рік. Таблиця містить ідентифікатор уразливості (ID), її назву, загальний бал (Score), кількість CVE-зразків, що входять до відомого експлуатованого вектора атак (KEV), а також зміни рейтингу порівняно з попереднім 2023 роком [1]. Синім кольором виділено ті типи вразливостей, які будуть детально розглядатися в рамках цієї роботи.

Таблиця А.1 – CWE Top 25 найкритичніших програмних уразливостей за 2024 рік

Rank	ID	Name	Score	CVEs in KEV	Rank Change vs. 2023
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	56.92	3	+1
2	CWE-787	Out-of-bounds Write	45.20	18	-1
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	35.88	4	0
4	CWE-352	Cross-Site Request Forgery (CSRF)	19.57	0	+5
5	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	12.74	4	+3
6	CWE-125	Out-of-bounds Read	11.42	3	+1
7	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.30	5	-2
8	CWE-416	Use After Free	10.19	5	-4
9	CWE-862	Missing Authorization	10.11	0	+2
10	CWE-434	Unrestricted Upload of File with Dangerous Type	10.03	0	0
11	CWE-94	Improper Control of Generation of Code ('Code Injection')	7.13	7	+12

Продовження таблиці А.1

12	CWE-20	Improper Input Validation	6.78	1	-6
13	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	6.74	4	+3
14	CWE-287	Improper Authentication	5.94	4	-1
15	CWE-269	Improper Privilege Management	5.22	0	+7
16	CWE-502	Deserialization of Untrusted Data	5.07	5	-1
17	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	5.07	0	+13
18	CWE-863	Incorrect Authorization	4.05	2	+6
19	CWE-918	Server-Side Request Forgery (SSRF)	4.05	2	0
20	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	3.69	2	-3
21	CWE-476	NULL Pointer Dereference	3.58	0	-9
22	CWE-798	Use of Hard-coded Credentials	3.46	2	-4
23	CWE-190	Integer Overflow or Wraparound	3.37	3	-9
24	CWE-400	Uncontrolled Resource Consumption	3.23	0	+13
25	CWE-306	Missing Authentication for Critical Function	2.73	5	-5

ДОДАТОК Б

ПОРІВНЯЛЬНА ТАБЛИЦЯ ІНСТРУМЕНТІВ ДИНАМІЧНОГО АНАЛІЗУ

Нижче у таблиці Б.1 наведено порівняльну таблицю найпоширеніших інструментів динамічного аналізу коду, що використовуються для виявлення уразливостей та помилок часу виконання. Інструменти розрізняються за типом аналізу, підтримуваними мовами програмування, ефективністю виявлення проблем, продуктивністю, можливостями інтеграції та обмеженнями. Огляд інструментів засновано на офіційній документації та наукових дослідженнях [16][17][18][19][20][21][22].

Таблиця Б.1 – Порівняльна характеристика інструментів динамічного аналізу програмного забезпечення

Інструмент	**Тип аналізу**	**Підтримувані мови**	**Виявлені вразливості**	**Продуктивність**	**Інтеграція**	**Обмеження**
Valgrind	Динамічний (DBI), пам’ять	C, C++	Витоки пам’яті, некоректні вказівники, помилки купи	Висока накладність (20–30× уповільнення)	GDB, IDE (CLion, Eclipse)	Не підтримує Windows, обмежена підтримка C++11+
Address Sanitizer	Динамічний (компіляторний)	C, C++	Out-of-bounds, Use-after-free, Double-free	Низька накладність (2× уповільнення)	Clang, GCC, LLVM	Обмежений аналіз stack-use-after-return без додаткових прапорців

Продовження таблиці Б.1

Інструмент	**Тип аналізу**	**Підтримувані мови**	**Виявлені вразливості**	**Продуктивність**	**Інтеграція**	**Обмеження**
Dr. Memory	Динамічний (пам’ять)	C, C++	Неініціалізована пам’ять, витоки, помилки GDI API	Середня накладність (10× уповільнення)	Windows, Linux (обмежено)	Часткова підтримка C++, проблеми з багатопотоковістю
KLEE	Символічне виконання	C, C++ (LLVM)	Переповнення буфера, ділення на нуль, невизначена поведінка	Дуже висока накладність (можливість "зависання")	LLVM, Z3	Вимагає модифікації коду, обмежена підтримка складних бібліотек
angr	Символічний/динамічний	Бінарні файли	Вразливості в компільованому коді, реверс-інжиніринг	Залежить від складності програми	IDA, Ghidra, Python	Висока складність налаштування, ресурсомісткий
Memory Sanitizer	Динамічний (компіляторний)	C, C++	Неініціалізована пам’ять, витоки	Низька накладність (3× уповільнення)	Clang, GCC	Не виявляє Use-after-free, потребує інструментування всіх бібліотек
Thread Sanitizer	Динамічний (багатопотоковість)	C, C++	Гонки даних (data races), deadlocks	Середня накладність (5–10× уповільнення)	Clang, GCC, TSAN_OPTIONS	Не виявляє atomicity violations, обмежена підтримка lock-free алгоритмів

ДОДАТОК В

ТЕСТОВІ ПРОГРАМИ МОВОЮ С

У цьому додатку наведено реалізацію програм мовою С, які використовувалися у тестуванні інструментів динамічного аналізу у виявленні вразливостей керування пам’яттю.

Лістинг В.1. Витік пам'яті

```
//CWE-401: Memory Leak
#include <stdlib.h>
int main () {
     char *ptr = ( char *)malloc(100); //Виділення пам’яті
     // Програма завершується без звільнення
     return 0;
}
```

Лістинг В.2. Подвійне звільнення пам'яті

```
//CWE-415: Double Free
#include <stdlib.h>
int main () {
     char *ptr = ( char *)malloc(100); //Виділення пам’яті
     free(ptr);
     free(ptr); //Повторне звільнення
     return 0;
}
```

Лістинг В.3. Доступ до звільненої пам'яті

```
//CWE-416: Use After Free
#include <stdlib.h>
#include <string.h>
int main () {
     char *ptr = ( char *)malloc(100); //Виділення пам’яті
     free(ptr);
     strcpy(ptr, “Test”); //Запис після звільнення
```

```
    return 0;
}
```

Лістинг В.4. Переповнення стеку

```
//CWE-121: Stack-Based Buffer Overflow
#include <string.h>
int main () {
    char buffer[10];
    strcpy(buffer, “This string is too long”);
    //Переповнення стеку
    return 0;
}
```

Лістинг В.5. Переповнення динамічного буфера

```
//CWE-122: Heap-Based Buffer Overflow
#include <string.h>
#include <stdlib.h>
int main () {
    char *ptr = ( char *)malloc(10); //Виділення пам’яті
    strcpy(ptr, “This string is too long”);
    //Переповнення стеку
    free(ptr);
    return 0;
}
```

Лістинг В.6. Переповнення глобального масиву

```
//CWE-119: Improper Restriction of Operations within the Bounds
of a Memory Buffer
#include <string.h>
char global[10]
int main () {
    strcpy(global, “This string is too long”);
    //Переповнення стеку
    return 0;
}
```

Лістинг В.7. Використання пам'яті після повернення з функції

```
//CWE-562: Return of Stack Variable Address
#include <stdio.h>
char *get_buffer(){
     char local [100];
     return local; //Повертається вказівник на стек
}
int main () {
     char *ptr = get_buffer();
     ptr[0] = 'A'; //Невизначена поведінка
     return 0;
}
```

Лістинг В.8. Розіменування NULL вказівника

```
//CWE-476: NULL Pointer Dereference
#include <stdio.h>
int main () {
     char *ptr = NULL;
     *ptr = 'A'; //Розіменування NULL
     return 0;
}
```

Лістинг В.9. Гонка даних

```
//CWE-362: Concurrent Execution using Shared Resource with
Improper Synchronization
#include <pthread.h>
#include <stdio.h>
int shared = 0;
void* increment(void* arg) {
     int i;
     for (i = 0; i < 100000; i++)
          shared++;
     return NULL;
}
```

```
int main () {
     pthread_t t1,t2;
     pthread_create(&t1, NULL, increment, NULL);
     pthread_create(&t2, NULL, increment, NULL);
     pthread_join(t1, NULL);
     pthread_join(t2, NULL);
     printf ("Shared = %d\n", shared); //Результат
     непередбачуваний
     return 0;
}
```