Хмельницький національний університет
Факультет інформаційних технологій
Кафедра кібербезпеки

КВАЛІФІКАЦІЙНА РОБОТА

Гордєєва Богдана Віталійовича

на здобуття ступеня вищої освіти магістра

Метод виявлення вразливостей типу переповнення буферу в програмному забезпеченні

Галузь знань 12 – Інформаційні технології
Спеціальність 125 – Кібербезпека та захист інформації
Освітня програма Кібербезпека та захист інформації

Шифр КРМКБЗІ.240190.24.01.05 ПЗ

Виконав студент 2 курсу група КБЗІм-24-1 _____ Богдан ГОРДЄЄВ
Керівник докт.техн.наук, професор _____ Михайло КАСЯНЧУК
Нормоконтролер д-р філософії, старший викладач _____ Наталія ПЕТЛЯК

До захисту допускаю:
Завідувач кафедри кібербезпеки _____ Юрій КЛЬОЦ

9 12 2025 р.

Хмельницький 2025

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Кібербезпеки

Рівень вищої освіти Магістр

Галузь знань 12 – Інформаційні технології

Спеціальність 125 – Кібербезпека та захист інформації

Освітня програма Кібербезпека та захист інформації

ЗАТВЕРДЖУЮ

Завідувач кафедри кібербезпеки

Юрій КЛЬОЦ

1 09 2025 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Гордєєву Богдану Віталійовичу

1 Тема роботи Метод виявлення вразливостей типу переповнення буферу в програмному забезпеченні

Керівник роботи докт.техн.наук, професор Михайло КАСЯНЧУК

Затверджено наказом ректора університету 25 08 2025 № 65

2 Строк подання студентом кваліфікаційної роботи на кафедру

3 Вихідні дані до роботи Проаналізувати сучасні методи виявлення вразливостей програмного забезпечення, зокрема переповнення буфера, визначити їх переваги та недоліки. Обґрунтувати вибір підходу, що поєднує елементи символьного виконання та навчання з підкріпленням (Q-навчання), для підвищення точності й ефективності діагностики. Розробити математичну модель переповнення буфера, алгоритм симулятивного покриття коду та генерації тестових даних. Провести експериментальну перевірку ефективності розробленого методу, здійснити порівняльний аналіз результатів із відомими підходами. Оцінити точність, стабільність та швидкодію методу, а також можливості його інтеграції у системи моніторингу безпеки програмного забезпечення

4 Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Вступ. Аналіз сучасних методів і досліджень у сфері виявлення вразливостей програмного забезпечення. Постановка задачі. Розроблення математичної моделі переповнення буфера. Створення алгоритму симулятивного покриття та Q-навчання для виявлення вразливостей. Реалізація та експериментальна перевірка ефективності запропонованого методу. Порівняльний аналіз із відомими рішеннями. Оцінювання точності, стабільності та продуктивності методу. Висновки.

5 Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

6 Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7 Дата видачі завдання 1 09 2025 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
Ґрунтовне ознайомлення та дослідження предметної галузі		Виконано
Визначення змісту, структури магістерської роботи		Виконано
Опрацювання першого розділу магістерської роботи		Виконано
Опрацювання статті за результатами дослідження		Виконано
Опрацювання другого розділу магістерської роботи		Виконано
Опрацювання третього розділу магістерської роботи		Виконано
Опрацювання четвертого розділу магістерської роботи		Виконано
Підготовка та опрацювання ілюстративного матеріалу		Виконано
Оформлення магістерської роботи графічної та текстової частини		Виконано
Попередній захист магістерської роботи		Виконано
Захист магістерської роботи на засіданні ЕК		Виконано

Студент _____ Богдан ГОРДЄЄВ

Керівник кваліфікаційної роботи _____ Михайло КАСЯНЧУК

АНОТАЦІЯ

Тема кваліфікаційної роботи: Метод виявлення вразливостей типу переповнення буферу в програмному забезпеченні

Автор роботи: студент групи КБЗІм-24-1 Гордєєв Б.В.

Керівник роботи: докт.техн.наук, професор Касянчук М.М.

Загальний обсяг роботи: 104 сторінок, 13 рисунків, 12 таблиць, 26 формул, 3 додатки, 60 посилань.

Ключові слова: переповнення буфера, вразливості програмного забезпечення, символьне виконання, Q-навчання, навчання з підкріпленням, аналіз безпеки, кібербезпека, автоматизоване тестування.

У кваліфікаційній роботі розроблено метод виявлення вразливостей типу переповнення буфера в програмному забезпеченні, який ґрунтується на поєднанні символьного виконання, Q-навчання та побудови дерева обмежень. Запропонований підхід забезпечує підвищення точності аналізу, скорочення кількості хибних спрацьовувань і здатність функціонувати в режимі реального часу. Проведено аналіз сучасних методів виявлення програмних вразливостей, зокрема статичного, динамічного та гібридного аналізу. Розроблено математичну модель переповнення буфера, алгоритми симулятивного покриття коду та генерації тестових даних. Виконано експериментальне дослідження ефективності методу, результати якого підтвердили його перевагу порівняно з класичними інструментами аналізу безпеки. Практичне значення полягає у можливості інтеграції розробленого методу в системи моніторингу безпеки для автоматизованого виявлення критичних дефектів програмного забезпечення.

1.12.2025

ANNOTATION

Theme of qualification work: Method for detecting buffer overflow vulnerabilities in software

Author of the work: student of KBZIm-24-1 Hordieiev B.V.

Mentor: Dr. Technical Sciences, Professor Kasyanchuk M.M.

Total volume of work: 104 pages, 13 figures, 12 tables, 26 formulas, 3 appendixes, 60 references.

Keywords: buffer overflow, software vulnerabilities, symbolic execution, Q-learning, reinforcement learning, security analysis, cybersecurity, automated testing.

In the qualification work, a method for detecting vulnerabilities such as buffer overflow in software has been developed, which is based on a combination of symbolic execution, Q-learning and constraint tree construction. The proposed approach provides increased analysis accuracy, reduced false positives and the ability to function in real time. An analysis of modern methods for detecting software vulnerabilities, in particular static, dynamic and hybrid analysis, has been conducted. A mathematical model of buffer overflow, algorithms for simulative code coverage and test data generation have been developed. An experimental study of the effectiveness of the method has been performed, the results of which confirmed its superiority over classical security analysis tools. The practical significance lies in the possibility of integrating the developed method into security monitoring systems for automated detection of critical software defects.

1.12.2025

ЗМІСТ

Вступ 8

1 Аналіз наявних рішень для виявлення вразливостей програмного забезпечення 10

1.1 Аналіз методів виявлення вразливостей програмного забезпечення 10

1.2 Аналіз сучасних досліджень щодо виявлення вразливостей 21

1.3 Постановка задачі 31

2 Метод виявлення вразливостей типу переповнення буферу в програмному забезпеченні 32

2.1 Математична модель переповнення буферу 32

2.2 Q-навчання як інструмент адаптивного симулятивного покриття 38

2.3 Побудова дерева обмежень 40

2.4 Алгоритм навчання поведінки програм шляхом симулятивного покриття 42

2.5 Алгоритм генерації вхідних даних на основі обмежень шляху та вузлів 48

2.6 Метод виявлення вразливостей 54

2.7 Висновки до розділу 57

3 Тестування та оцінка достовірності 60

3.1 Налаштування тестового середовища 60

3.2 Тестові програми 62

3.3 Проведення та результати першого експерименту 66

3.4 Проведення та результати другого експерименту 70

3.5 Оцінювання достовірності 75

3.6 Висновки до розділу 79

Висновки 82

Перелік джерел посилань 84

Додаток А. Перелік тестових програм 91

Додаток Б. Лістинг коду Q-навчання 95

Додаток В. Перелік публікацій 96

ВСТУП

У сучасних умовах стрімкої цифровізації програмне забезпечення стало ключовою складовою більшості сфер діяльності людини - від бізнесу й комунікацій до державного управління та критичної інфраструктури. Зростання його кількості супроводжується й підвищенням ризиків, пов’язаних із наявністю вразливостей. Одним із найнебезпечніших типів є переповнення буферу, яке здатне призвести до порушення роботи програм, витоку конфіденційних даних або навіть віддаленого виконання коду зловмисником.

Традиційні методи ручного аудиту та класичного тестування коду залишаються важливими, проте вони є надмірно затратними за часом і ресурсами. Тому дедалі більшої актуальності набувають автоматизовані методи виявлення вразливостей, що дозволяють значно підвищити швидкість і точність аналізу. Наукові дослідження у цій сфері зосереджені на поєднанні статичного та динамічного аналізу, символьного виконання, фаззингу та сучасних алгоритмів машинного навчання, що у комплексі створює нові можливості для ефективного пошуку небезпечних дефектів у програмному коді.

Метою даної роботи є розробка методу виявлення вразливостей типу переповнення буферу в програмному забезпеченні, який забезпечує підвищення ефективності й достовірності діагностики за рахунок використання сучасних інструментів аналізу та адаптивних алгоритмів навчання.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

– здійснити огляд та аналіз наявних рішень і сучасних досліджень у сфері виявлення програмних вразливостей;

– побудувати математичну модель переповнення буферу як основи для подальшої формалізації методу;

– розробити алгоритм навчання поведінки програм на основі симулятивного покриття та Q-навчання;

– створити метод генерації вхідних даних із використанням системи обмежень;

– провести тестування запропонованого методу та оцінити його ефективність за ключовими показниками.

Об’єктом дослідження є процеси виявлення вразливостей у програмному забезпеченні.

Предметом дослідження є методи й алгоритми автоматизованої діагностики вразливостей, зокрема переповнення буферу, їх точність, швидкодія та здатність до адаптації у практичних умовах.

Наукова новизна роботи полягає у розробці методу виявлення вразливостей, який поєднує математичне моделювання переповнення буферу з адаптивними алгоритмами навчання з підкріпленням, що забезпечує підвищення точності аналізу та зменшення кількості хибних результатів.

Практичне значення роботи полягає у можливості використання розробленого методу для створення ефективних інструментів кібербезпеки, здатних своєчасно ідентифікувати критичні дефекти у програмному забезпеченні, мінімізуючи ризики їх експлуатації зловмисниками.

Кваліфікаційна робота складається зі вступу, трьох розділів основного змісту, висновків, списку використаних джерел і додатків. Перший розділ присвячено аналізу наявних рішень у сфері виявлення вразливостей. У другому розділі розроблено метод виявлення переповнення буферу на основі математичної моделі, симулятивного покриття та Q-навчання. Третій розділ містить опис проведених експериментів, результати тестування та оцінку достовірності методу.

1 АНАЛІЗ НАЯВНИХ РІШЕНЬ ДЛЯ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Аналіз методів виявлення вразливостей програмного забезпечення

Виявлення вразливостей відіграє вагому роль у процесі забезпечення безпеки програмного забезпечення, оскільки передбачає ідентифікацію потенційно небезпечних ділянок коду, які можуть бути використані зловмисниками. Існує кілька підходів до цього завдання, що широко застосовуються на практиці. Один із них це ручний аудит коду, під час якого експерти уважно досліджують вихідний код для виявлення недоліків у логіці, небезпечних процедур або неправильного використання ресурсів [1-3]. Інший підхід - статичний аналіз, який базується на автоматизованому дослідженні коду або бінарних файлів без їхнього фактичного запуску. Цей метод дозволяє швидко оцінити всю кодову базу та виявити потенційні дефекти ще до виконання програми, однак він часто породжує хибні спрацьовування та не дає повного уявлення про поведінку системи під час виконання [4-6].

Динамічний аналіз передбачає запуск програми в контрольованому середовищі або в межах автоматизованих тестів, одночасно відслідковуючи її взаємодію з системними ресурсами та зміни станів. Це дає змогу виявляти проблеми, що проявляються лише під час виконання, наприклад витоки пам’яті чи некоректне поводження з вхідними даними, проте потребує значних обчислювальних ресурсів і часу на аналіз. Серед методів динамічного аналізу значну увагу приділяють фаззинг-тестуванню, яке передбачає модифікацію вхідних даних випадковим чином або на основі заданих правил з метою створення некоректних або граничних значень. Такі дані потрапляють у ті частини програми, де обробка не передбачає належних перевірок, що може призвести до збоїв, відмови в обслуговуванні або віддаленого виконання коду [7-9]. Гібридні методики поєднують переваги обох підходів: аналіз коду без запуску дозволяє швидко сканувати всю кодову базу, тоді як виконання програми дає змогу оцінити поведінку системи у реальному часі. Такий підхід забезпечує

всебічне охоплення та підвищує ефективність виявлення вразливостей, хоча його реалізація вимагає інтеграції різних інструментів, синхронізації процесів та значних обчислювальних ресурсів [10-12].

Після ідентифікації вразливостей проводиться їхній детальний аналіз, який дозволяє оцінити серйозність наслідків, можливий вплив на роботу системи та ймовірність успішного використання. Серйозність визначає потенційну шкоду у разі експлуатації дефекту, що допомагає пріоритизувати заходи реагування. Оцінка впливу дозволяє прогнозувати негативні наслідки для організації, зокрема втрату даних або збоїв в роботі системи. Водночас аналіз ймовірності експлуатації дає змогу передбачити, наскільки реально зловмиснику скористатися виявленою слабкістю для проникнення у програмну систему [13-15].

Усунення виявлених проблем здійснюється через впровадження виправлень, модифікацій коду або патчів, що зменшують ризики та підвищують стабільність системи. Патчі можуть постачатися виробниками програмного забезпечення або створюватися спільнотами відкритого коду й спрямовані на ліквідацію конкретних дефектів, забезпечуючи підвищення надійності та стійкості програмних систем до потенційних атак. Крім того, сучасні підходи передбачають автоматизовану перевірку застосування цих виправлень, що дозволяє швидко оцінити ефективність усунення дефектів і запобігти повторній появі тих самих проблем у майбутньому [16-18].

Символьне виконання є одним із провідних методів аналізу та тестування програмного забезпечення, що застосовується для автоматизованого дослідження поведінки програмних систем. На відміну від традиційного підходу, де програма виконується з конкретними вхідними значеннями, символьне виконання оперує символьними змінними, які представляють множину можливих значень. Такий підхід забезпечує можливість дослідження одразу декількох шляхів виконання, що дозволяє формально аналізувати всі можливі стани системи та виявляти потенційні вразливості. Процес символьного виконання базується на побудові дерева виконання, у якому кожен вузол

відповідає певному стану програми. У точках розгалуження дерева утворюються нові гілки, що відповідають різним умовам виконання, залежним від символьних вхідних даних. Завдяки цьому стає можливим ідентифікувати конкретні умови, які активують певну поведінку програми, або виявити такі вхідні значення, що призводять до порушень безпеки, зокрема до переповнення буфера в динамічній пам’яті, атак типу SQL-injection чи path traversal. Символьне виконання дозволяє не лише перевіряти наявність вразливостей, а й генерувати тестові сценарії, які відтворюють цільові умови виконання програми, що робить цей метод цінним для розробки надійного та безпечного програмного забезпечення [19-21].

Разом із тим символьне виконання стикається з суттєвими обмеженнями у практичному застосуванні. Основною проблемою є явище так званого «вибуху шляхів», яке виникає внаслідок експоненційного зростання кількості можливих шляхів виконання при збільшенні складності програми. У багатьох випадках кількість шляхів зростає настільки швидко, що повне дослідження всіх варіантів стає обчислювально непідйомним завданням, що потребує значних обчислювальних ресурсів і часу. Вибух шляхів виникає тоді, коли програма містить велику кількість умовних переходів, що залежать від символьних значень. Кожен додатковий символьний розгалужувач подвоює кількість можливих варіантів, у результаті чого загальне число шляхів зростає експоненційно. Це призводить до проблем із масштабованістю, оскільки навіть для програм середньої складності кількість шляхів може сягати мільйонів, що унеможливлює їх повне дослідження в прийнятний час. Щоб подолати цю проблему, у сучасних дослідженнях пропонуються різноманітні оптимізаційні техніки. Одним із найбільш поширених напрямів є застосування стратегій обрізання шляхів, які дозволяють обмежити пошук до тих виконань, що є найбільш релевантними з точки зору покриття коду або ймовірності виявлення вразливостей [22-24].

Додатково застосовуються методи абстракції, поєднання конкретного та символьного аналізу, а також евристики для пріоритизації шляхів. Ці підходи спрямовані на зменшення обчислювального навантаження без істотної втрати

точності у процесі виявлення дефектів програмного забезпечення. У сучасній науковій літературі проблема вибуху шляхів розглядається як один із головних бар'єрів на шляху широкого впровадження символьного виконання у промислових масштабах. Попри це, метод зберігає свою актуальність завдяки здатності виявляти складні вразливості, які залишаються поза увагою традиційних технік тестування.

Навчання з підкріпленням (Reinforcement Learning, RL) належить до окремої гілки машинного навчання, яка спеціалізується на розв'язанні задач послідовного прийняття рішень. Його сутність полягає у побудові інтелектуальних агентів, здатних навчатися оптимальній поведінці в результаті взаємодії із середовищем, що надає зворотний зв'язок у вигляді винагород або штрафів. Завдяки універсальності та високій адаптивності RL здобуло значну популярність у прикладних сферах, серед яких робототехніка, кіберфізичні системи, автоматизоване керування, фінансове прогнозування, медична діагностика та ігрова індустрія [25-27]. Фундаментальною концепцією RL є агент, який взаємодіє з середовищем, прагнучи максимізувати довгострокову очікувану винагороду. Його стратегія поведінки описується відображенням множини можливих станів системи у відповідні дії. Середовище реагує на ці дії, змінюючи власний стан і надаючи агенту чисельні сигнали як оцінку корисності виконаних кроків. Процес навчання зводиться до пошуку такої політики, яка забезпечує глобально оптимальне значення функції винагороди в умовах невизначеності та стохастичності зовнішніх факторів. Методи навчання з підкріпленням можна умовно поділити за принципом організації навчального процесу. Політико-орієнтовані підходи безпосередньо оптимізують параметризовану політику, використовуючи, зокрема, градієнтні методи. Алгоритми типу Policy Gradient реалізують стохастичну оцінку градієнта функції корисності та здійснюють ітераційне оновлення параметрів політики, що дозволяє досягати стійкої адаптації агента до змінюваних умов середовища [28-30]. На відміну від цього, методи, орієнтовані на функції цінності, зосереджені на апроксимації функції стану або пари стан–дія, яка визначає очікувану

кумулятивну винагороду для даної ситуації. Одним із найбільш відомих алгоритмів цього класу є Q-Learning, що базується на рівнянні Беллмана для рекурсивного оновлення оцінки цінності станів і дій [31-33]. У сучасних реалізаціях RL важливе значення має баланс між дослідженням нових стратегій (exploration) і використанням уже набутих знань (exploitation). Додатково застосовуються механізми апроксимації функцій за допомогою нейронних мереж, що дозволяє ефективно працювати з високовимірними просторами станів. Розвиток напрямів, таких як ієрархічне RL, методи зі слідами придатності (eligibility traces), а також комбінація з еволюційними алгоритмами, значно розширює можливості практичного застосування навчання з підкріпленням.

Навчання з підкріпленням без моделі (model-free reinforcement learning, RL) являє собою клас алгоритмів, які дозволяють агенту оцінювати ефективність власних дій на основі отриманих винагород без потреби вбудованого моделювання середовища. Такі методи особливо корисні для складних або непередбачуваних систем, де точне математичне моделювання процесів є неможливим або економічно невигідним [34-36]. Основними підходами в цьому класі є табличні методи, методи апроксимації функцій, методи градієнта політики та алгоритм Q-Learning. Табличні методи використовують таблиці стан–дія для збереження оцінок функції цінності [37]. Вони легко реалізуються та забезпечують збіжність при достатньому дослідженні середовища, проте обмежені у масштабованості та здатності до узагальнення знань. У реальних задачах такі методи застосовуються для навчання агентів у малих і дискретних просторах, наприклад, для управління простими роботизованими системами або в ігрових середовищах типу шахів або настільних ігор з обмеженою кількістю станів. Методи апроксимації функцій, зокрема нейронні мережі, дозволяють ефективно працювати у неперервних і високорозмірних просторах станів і дій, забезпечуючи узагальнення знань між схожими станами. Вони застосовуються у таких задачах, як автономне управління транспортними засобами, адаптивне управління енергетичними мережами, або у фінансових системах для оптимізації торгових стратегій [38-40]. Недоліками таких методів є схильність до

перенавчання та нестабільності навчального процесу, що потребує додаткових стабілізуючих механізмів, наприклад Replay Buffer та регуляризації. Методи градієнта політики безпосередньо оптимізують політику агента через оцінку градієнтів від вибірок, що робить їх придатними для неперервних просторів дій та задач із стохастичними вимогами. Вони активно використовуються для управління роботами-маніпуляторами, у безпілотних літальних апаратах для навчання складних маневрів, а також у складних ігрових середовищах, де дослідження стратегій потребує неперервних дій. Основні обмеження включають високу дисперсію оцінок та ризик застрягання у локальних оптимумах, що потребує застосування методів зменшення дисперсії та адаптивних стратегій навчання.

Навчання поза політикою (off-policy) дозволяє агенту навчатися на основі досвіду, отриманого від політики, відмінної від тієї, що оптимізується, що робить цей підхід особливо ефективним для реальних систем, де повне дослідження середовища є дорогим або ризикованим. Основна перевага off-policy навчання полягає у можливості використання різноманітних джерел даних, включаючи історичні набори даних, демонстрації експертів та симуляції, що підвищує ефективність переносу знань та повторного використання інформації. Off-policy алгоритми дозволяють балансувати між дослідженням і експлуатацією знань, використовуючи різні політики поведінки для збору даних [41-43]. Це забезпечує можливість ефективного навчання у складних, високоризикових або непередбачуваних середовищах, наприклад, для автономного керування транспортними засобами у міському середовищі, де необхідно враховувати дії інших учасників дорожнього руху, або для оптимізації виробничих процесів на великих промислових підприємствах з багатьма взаємозалежними параметрами. Разом із перевагами, off-policy навчання стикається з викликами, такими як похибки офлайн-оцінки та нестабільність у комбінації функціональної апроксимації та бустрапінгу. Для подолання цих проблем використовуються техніки стабілізації, такі як experience replay, target networks та адаптивне регулювання навчальної швидкості. У реальних задачах off-policy алгоритми

успішно застосовуються для навчання систем рекомендацій, де агент навчається на основі історії взаємодій користувачів, у фінансових алгоритмічних трейдингових системах для оптимізації портфеля активів, а також у мультиагентних середовищах для координації дій між агентами. Алгоритм Q-навчання є прикладом off-policy підходу. Він демонструє високу гнучкість та здатність адаптуватися до різнорідних джерел даних, забезпечуючи ефективне поєднання дослідження нових стратегій та використання наявного досвіду. Це робить його придатним для застосування у складних середовищах із високими вимогами до стабільності та продуктивності навчання, таких як автономні транспортні системи, інтелектуальні енергетичні мережі та комплексні промислові процеси.

Q-навчання є алгоритмом навчання з підкріпленням поза політикою (off-policy), який дозволяє агенту оцінювати очікувану довгострокову ефективність дій у заданому стані без необхідності побудови внутрішньої моделі середовища. Основним інструментом алгоритму є функція Q, яка відображає очікувану цінність виконання конкретної дії в конкретному стані. Чим більше значення Q, тим вигідніше виконання цієї дії з точки зору майбутніх результатів. Q-функція зберігається у вигляді таблиці, де координати відповідають станам і діям агента. Агент починає у певному стані та вибирає дію відповідно до поточної політики, яка базується на оцінках Q. Після виконання дії агент переходить у новий стан і отримує винагороду, що відображає ефективність обраної дії. На основі цієї винагороди та оцінок майбутніх дій оновлюється поточне значення Q для пари стан–дія, що дозволяє агенту поступово наближатися до оптимальної стратегії. У практичних умовах Q-навчання знаходить застосування у численних задачах автономного управління та оптимізації. Сучасні покращення класичного алгоритму Q-навчання значно підвищують його ефективність та стабільність у високорозмірних та неперервних просторах станів. До таких методів належить Deep Q-Network (DQN), який використовує нейронні мережі для апроксимації Q-функції, що дозволяє працювати із складними середовищами, де таблична реалізація є непрактичною. Double Q-навчання усуває проблему переоцінки Q-

значень, розділяючи оцінку та вибір дій, що підвищує стабільність навчання. Prioritized Experience Replay дозволяє пріоритезувати найбільш інформативні взаємодії із середовищем при навчанні, підвищуючи швидкість збіжності та ефективність використання досвіду. Інтеграція цих підходів забезпечує можливість побудови більш надійних та продуктивних агентів у складних системах, включаючи автономні автомобілі, промислових роботів та мультиагентні симуляції, де необхідно швидко адаптуватися до змінних умов середовища. Q-навчання поєднує простоту реалізації, гнучкість, здатність до ефективного наближення оптимальної стратегії та адаптації до складних середовищ, що робить його одним із ключових методів model-free навчання з підкріпленням для сучасних систем штучного інтелекту та автономних технологій. Класичне Q-навчання характеризується простотою реалізації та здатністю ефективно наближати оптимальні стратегії у дискретних та малорозмірних просторах станів і дій. Його основним обмеженням є низька масштабованість та ризик переоцінки Q-значень, особливо у високорозмірних або неперервних середовищах. Ці обмеження ускладнюють застосування класичного алгоритму у реальних промислових та дослідницьких системах, де кількість станів та дій може сягати тисяч або мільйонів, а середовище постійно змінюється.

DQN розширює класичний алгоритм, використовуючи нейронні мережі для апроксимації Q-функції, що дозволяє працювати з неперервними просторами станів та дій. Цей підхід ефективно застосовується у складних задачах автономного управління, наприклад, у безпілотних автомобілях для навігації у динамічних міських умовах або у робототехнічних системах, де агент повинен швидко оцінювати величезну кількість можливих дій. Double Q-learning вирішує проблему систематичного переоцінювання Q-значень у класичному алгоритмі, розділяючи процес вибору дії та оцінки її цінності. Це покращення дозволяє підвищити стабільність навчання та уникнути локальних оптимумів, що особливо важливо у складних промислових процесах, де помилки у прийнятті рішень можуть бути критичними, наприклад, у системах управління

енергетичними мережами або у промислових роботизованих лініях. Prioritized Experience Replay підвищує ефективність навчання, дозволяючи агенту приділяти більше уваги найбільш інформативним взаємодіям із середовищем. Це забезпечує швидше зближення до оптимальної політики та ефективніше використання отриманого досвіду. Такий підхід активно застосовується у симуляціях мультиагентних середовищ, складних ігрових системах, а також у системах рекомендацій та фінансовому трейдингу, де пріоритетність даних відіграє критичну роль у прийнятті рішень. Порівняння класичного та покращених варіантів Q-навчання показує, що сучасні методи значно підвищують ефективність, стабільність та адаптивність агентів у складних та динамічних середовищах. Використання цих покращень дозволяє інтегрувати алгоритми у реальні промислові системи та автономні технології, забезпечуючи надійність прийняття рішень та оптимізацію ресурсів навіть у високоризикових або непередбачуваних умовах. У межах інтеграції Q-навчання з методами символьного виконання увага приділяється максимізації охоплення простору виконання програм. Для цього агент отримує винагороди, які стимулюють дослідження нових шляхів виконання та забезпечують збільшення ймовірності виявлення критичних вразливостей, таких як переповнення буфера в динамічній пам’яті. У випадках, коли алгоритм виявляє вразливість типу переповнення буферу, функція винагороди суттєво зростає, що мотивує агента до систематичного пошуку подібних ситуацій. Таким чином, механізм кумулятивної винагороди фокусується на аспектах дослідження та покриття, оскільки саме такий підхід є найбільш доцільним у контексті аналізу безпеки програмного забезпечення. Завдяки здатності навчання з підкріпленням поєднувати гнучкість адаптивного прийняття рішень із формальною математичною моделлю оптимізації, цей метод стає інструментом для побудови систем виявлення та запобігання вразливостям. Його подальший розвиток у поєднанні з методами символьного виконання та конколікального аналізу відкриває перспективи створення високоефективних гібридних технологій тестування, здатних масштабуватися для аналізу програмних систем

промислового рівня. Алгоритм Q-навчання є класичним прикладом model-free підходу, який оцінює функцію цінності Q для пар стан–дія. Він поєднує характеристики табличних і апроксимаційних методів, дозволяючи навчатися без знання динаміки середовища та з використанням досвіду, отриманого будь-якою політикою (off-policy). Q-Learning застосовується у системах управління робототехнікою, маршрутизації в транспортних мережах, оптимізації ресурсів у дата-центрах та в навчанні агентів для стратегічних ігор. Його обмеженням є низька ефективність використання зразків та потенційне переоцінювання Q-значень, що іноді призводить до субоптимальних рішень, але у практичних задачах часто компенсується комбінацією з функціональною апроксимацією та регуляризацією.

Фреймворк Valgrind посідає провідне місце серед систем динамічної бінарної інструментації (Dynamic Binary Instrumentation, DBI), оскільки забезпечує глибокий аналіз виконуваних програм та створює умови для їх модифікації у режимі реального часу [44-45]. Ключовим компонентом його архітектури є Valgrind Expression (VEX), який виконує трансляцію машинного коду у проміжне представлення (Intermediate Representation, IR) та забезпечує платформонезалежність аналізу. Однією з властивостей VEX є створення уніфікованого проміжного представлення, що дозволяє абстрагуватися від конкретної апаратної архітектури. Завдяки цьому інструмент може застосовуватися для аналізу програмного забезпечення, написаного для різних процесорних платформ, без необхідності розробки окремих спеціалізованих інструментів. Використання IR відкриває можливості для виконання високорівневих трансформацій коду, оптимізацій та детального аналізу поведінки програм у процесі їх виконання. VEX реалізує ефективну динамічну бінарну трансляцію, що забезпечує перехоплення та модифікацію інструкцій під час роботи програми. Це дозволяє проводити глибокий аналіз пам’яті, виявляти витоки, некоректний доступ та використання неініціалізованих змінних. Крім того, за допомогою VEX можливе проведення профілювання продуктивності, аналізу навантаження на апаратні ресурси та дослідження ефективності

алгоритмів. У контексті кібербезпеки цей інструмент часто застосовується для виявлення помилок у викликах API, перевірки коректності управління ресурсами та дослідження потенційних вразливостей, що мають критичне значення для захищеності інформаційних систем. Важливою сферою застосування VEX є системи символьного виконання (symbolic execution), які використовують його проміжне представлення для автоматизованого аналізу всіх можливих шляхів виконання програми. Це створює умови для виявлення прихованих помилок, дослідження логічних залежностей та прогнозування поведінки програмного забезпечення без необхідності його реального запуску. Таким чином, VEX забезпечує основу для формального доведення властивостей програм та підвищує надійність результатів аналізу.

Функціональне апроксимування є базовим процесом у математичному моделюванні та обчислювальній математиці, спрямованим на встановлення залежності між вхідними та вихідними змінними. Зазвичай воно застосовується у випадках, коли наявна одна вихідна змінна та обмежена кількість вхідних параметрів. Основною метою є побудова моделі, яка дозволяє з достатньою точністю відтворювати приховані залежності системи на основі наданих даних та прогнозувати поведінку об’єкта при нових вхідних значеннях. Одним із найбільш поширених підходів є метод найменших квадратів, що дає змогу мінімізувати суму квадратів відхилень апроксимуючої функції від експериментальних даних. Цей метод менш чутливий до впливу випадкових похибок та викидів у даних порівняно з альтернативними методами оптимізації. У випадках, коли реальна функція має нелінійний характер, наприклад y(x), для наближення часто використовуються поліноми p(x) із коефіцієнтами ci. Знаходження оптимальних коефіцієнтів здійснюється шляхом розв’язання системи нормальних рівнянь, що мінімізує евклідову відстань між вихідною функцією та апроксимуючим поліномом у вибраних точках. Теорема Вейєрштраса про апроксимацію є теоретичним підґрунтям, оскільки вона гарантує, що будь-яку неперервну функцію на замкненому відрізку можна наблизити поліномами з довільно високою точністю. У випадках, коли функція

не є гладкою на всій області визначення, але зберігає локальну гладкість на підінтервалах, застосовують кусочно-поліноміальне апроксимування. Зменшення довжини інтервалів дозволяє будувати наближення низького порядку, які при цьому забезпечують високу точність. Подібний підхід є ефективним при моделюванні складних фізичних процесів, де функції характеризуються різними властивостями на різних ділянках. Важливим аспектом є визначення мінімально необхідної кількості експериментальних точок для побудови коректної моделі. Це поняття позначається терміном «поріг», який встановлює нижню межу кількості даних, достатніх для надійного апроксимування. Якщо кількість спостережень є меншою за цей поріг, модель втрачає здатність до узагальнення і стає надмірно чутливою до випадкових флуктуацій. Застосування функціонального апроксимування виходить далеко за межі класичної математики та знаходить своє місце у галузі програмного аналізу. До прикладу, апроксимування використовується для встановлення зв’язку між символьними вхідними значеннями та умовами виконання певних шляхів у програмі. Додатково слід відзначити, що сучасні дослідження у сфері машинного навчання розширюють класичні методи апроксимування. Поліноміальні моделі доповнюються нейронними мережами, гаусовими процесами та методами регуляризації, які забезпечують стійкість до перенавчання та здатність працювати з високовимірними просторами ознак. У такий спосіб функціональне апроксимування перетворюється на універсальний інструмент, який поєднує класичні чисельні методи та сучасні алгоритми штучного інтелекту.

1.2 Аналіз сучасних досліджень щодо виявлення вразливостей

У методі GRACE [46] для виявлення вразливостей програмного забезпечення використовується поєднання великої мовної моделі та структурної інформації коду у вигляді графів, що дозволяє краще розуміти взаємозв’язки між різними компонентами програми. Метод складається з трьох взаємопов’язаних

модулів. Перший модуль «вибір демонстрацій» забезпечує підбір найбільш релевантних прикладів коду для навчання в контексті, враховуючи синтаксичну, семантичну та лексичну подібність до цільового коду. Другий модуль «генерація інформації про графову структуру» інтегрує абстрактне синтаксичне дерево, граф залежностей даних та граф потоку керування, що дає змогу моделі враховувати складні зв'язки між елементами коду. Третій модуль «розширене виявлення вразливостей» формує контекстні підказки, які містять інформацію про домен вразливостей та структурні характеристики коду, що значно підвищує здатність моделі ідентифікувати потенційно небезпечні ділянки. Завдяки використанню графових структур модель може враховувати не лише локальний контекст коду, а й залежності між різними частинами програми, що важливо для складних вразливостей, таких як порушення контролю потоку, недотримання обмежень доступу та взаємодії між даними. Крім того, навчання в контексті дозволяє GRACE ефективно використовувати приклади з доменною інформацією, що підвищує точність класифікації вразливостей і зменшує кількість хибнопозитивних спрацьовувань. Такий підхід дозволяє не лише виявляти наявні вразливості, а й класифікувати їх за типом, що є важливим для подальшого планування заходів щодо їх усунення та підвищення загальної безпеки програмного забезпечення. У методі GRACE, попри його значні переваги у порівнянні з попередніми підходами, наявні й певні недоліки та обмеження. По-перше, інтеграція графових структур у процес аналізу коду вимагає додаткових обчислювальних ресурсів, що ускладнює масштабованість при роботі з великими програмними системами. Формування абстрактного синтаксичного дерева, графів залежностей даних та потоку керування є ресурсоємним процесом, який може уповільнювати швидкість аналізу в реальних умовах. По-друге, ефективність GRACE залежить від якості та повноти вихідних даних. Якщо навчальні набори містять обмежену кількість прикладів певних класів вразливостей, модель може демонструвати гірші результати при їхньому виявленні. Це створює ризик зниження точності в умовах появи нових або рідкісних вразливостей, які не представлені у навчальних даних. По-третє, у

випадку помилкового вибору демонстрацій модель може зробити некоректні висновки, що підвищує ймовірність хибнопозитивних або хибнонегативних результатів. Зрештою, існує питання практичного впровадження: хоча результати експериментів демонструють високу ефективність GRACE, її інтеграція у корпоративні інструменти захисту потребує значних зусиль щодо адаптації до конкретних систем, середовищ розробки та процесів безпеки.

У статті [47] представлено метод VulBERTa, який базується на використанні глибокого навчання для автоматизованого виявлення вразливостей у вихідному коді. Його особливість полягає у попередньому навчанні моделі RoBERTa на спеціально підготовленому корпусі програмного коду мов C/C++, що походить із відкритих проєктів. Для цього автори розробили спеціалізований конвеєр токенізації, який дозволяє інтерпретувати особливості програмного коду. Завдяки цьому модель навчається відтворювати як синтаксичні, так і семантичні характеристики програм. Метод дозволяє здійснювати як бінарну класифікацію (наявність чи відсутність вразливості), так і багатокласову класифікацію (ідентифікацію конкретних типів вразливостей). Попри переваги, метод VulBERTa має і певні недоліки. Зокрема, вимагає суттєвих обчислювальних ресурсів на етапі попереднього навчання, що може бути обмеженням для її широкого використання у невеликих компаніях. Також продуктивність методу напряму залежить від якості та репрезентативності навчальних даних.

Метод LineVD [48], запропонований авторами, орієнтований на виявлення вразливостей у програмному забезпеченні шляхом аналізу коду на рівні окремих операторів. Його основою є формулювання завдання як класифікації вузлів, де кожен оператор у вихідному коді розглядається як окремий вузол у графі. Для цього використовується поєднання графових нейронних мереж (GNN), що моделюють залежності даних і керування між операторами, та моделі на основі трансформера, яка кодує токени коду і враховує їх семантичний та синтаксичний контекст. Особливістю LineVD є здатність ефективно поєднувати інформацію на рівні функцій та операторів, що дозволяє уникати суперечностей при аналізі та

підвищує точність прогнозування статусу вразливості навіть для складних функціональних фрагментів коду. Метод орієнтований на виявлення реальних вразливостей у мовах C/C++, зокрема таких типів, як переповнення буфера, некоректне управління пам’яттю, помилки доступу до покажчиків, уразливості, пов’язані з неконтрольованим використанням даних. Такий підхід є особливо релевантним для системного та низькорівневого програмного забезпечення, де подібні проблеми становлять найбільшу загрозу. Разом із перевагами LineVD має і певні недоліки. По-перше, складність моделі, яка поєднує GNN і трансформери, зумовлює високі обчислювальні витрати, що може обмежувати практичне застосування у великих індустріальних проєктах. По-друге, ефективність значною мірою залежить від повноти та репрезентативності навчальних даних. По-третє, метод може продукувати хибнопозитивні спрацювання, що створює додаткове навантаження на експертів, які повинні перевіряти результати. Нарешті, складна архітектура платформи ускладнює її інтерпретованість, що знижує прозорість пояснень для фахівців із безпеки.

Метод VulCNN [49], запропонований авторами, ґрунтується на ідеї подання вихідного коду у вигляді зображень з метою підвищення як масштабованості, так і точності виявлення вразливостей. На відміну від текстових підходів, що добре масштабуються, але втрачають семантику програми, та графових методів, що зберігають семантичну інформацію, проте є обчислювально затратними, VulCNN намагається поєднати обидві переваги. Перетворюючи код функцій на зображення, модель застосовує глибинні архітектури, аналогічні тим, що використовуються в класифікації зображень, зберігаючи структурні й семантичні характеристики програмного забезпечення. Запропонований підхід показав високу ефективність у виявленні вразливостей у мовах C/C++. Водночас VulCNN має певні недоліки. По-перше, хоча перетворення коду у формат зображень пришвидшує процес порівняно з графовими методами, воно все одно створює додаткові витрати на попередню обробку та потребує значних обчислювальних ресурсів для глибинних мереж згорткового типу. По-друге, підхід може бути менш ефективним для складних

міжфункціональних залежностей, які важко відобразити в межах однієї функції-зображення. По-третє, як і більшість моделей, VulCNN є слабко інтерпретованим, що ускладнює пояснення виявлених вразливостей фахівцям із безпеки.

У статті [50] представлено метод AIBugHunter, що поєднує машинне навчання та інтеграцію з середовищем розробки Visual Studio Code для виявлення й усунення вразливостей у програмному забезпеченні, написаному на C/C++. Автори пропонують підхід, у якому інструмент виконує багатофункціональний аналіз коду, який локалізує потенційно небезпечні ділянки на рівні рядків, класифікує типи вразливостей за CWE-ID і CWE-Type, оцінює рівень їхньої критичності відповідно до метрик CVSS та пропонує варіанти автоматизованих виправлень. Основою методу є поєднання попередніх рішень LineVul і VulRepair з новими моделями на базі трансформаторів, а також використання багатоцільової оптимізації для покращення багатозадачного навчання. Такий підхід дозволяє забезпечити розробників інструментом, який підтримує концепцію «зсуву вліво» в тестуванні, надаючи діагностику та поради безпосередньо під час написання коду. AIBugHunter здатний ідентифікувати широкий спектр вразливостей у програмах на C/C++, що охоплюються класифікацією CWE. Водночас запропонований метод має низку обмежень. Попри інтеграцію з IDE та орієнтацію на зручність розробників, точність прогнозів усе ще залежить від якості навчальних даних і може варіюватися для різних категорій вразливостей. Використання моделей на основі трансформаторів і багатоцільової оптимізації потребує значних обчислювальних ресурсів, що може впливати на швидкість роботи у великих проєктах. Ще однією проблемою залишається інтерпретація результатів, оскільки рекомендації моделі не завжди можуть бути достатньо зрозумілими для негайного прийняття рішень без додаткового аналізу з боку фахівців. Крім того, хоча інструмент показує ефективність для коду на C/C++, його здатність масштабуватися на інші мови та специфічні домени залишається відкритим напрямом для майбутніх досліджень.

Автори статті [51] пропонують метод VulANalyzeR, що ґрунтується на

застосуванні глибокого навчання для автоматизованого виявлення вразливостей у двійковому коді. Його особливістю є поєднання послідовного та топологічного навчання із використанням рекурентних нейронних одиниць і графових згорток, що дозволяє моделювати логіку виконання програми. У структурі моделі реалізовано механізм уваги, який підсвічує, які саме інструкції та стани коду впливають на остаточне рішення щодо наявності вразливості. Завдяки багатозадачному навчанню VulANalyzeR не лише визначає сам факт вразливості, а й виконує її класифікацію відповідно до переліку загальних слабких місць, що надає розробникам додатковий контекст для усунення загроз. Важливою перевагою цього методу є здатність виявляти першопричини вразливостей. Запропонований метод здатен ідентифікувати широкий спектр поширених помилок безпеки у двійкових файлах, включно з переповненнями буфера, некоректним керуванням пам’яттю, використанням невизначених змінних та іншими дефектами, які часто призводять до вразливостей нульового дня. У рамках проведених експериментів модель демонструє вищу точність порівняно з наявними класичними підходами. Разом із тим метод має і певні недоліки. Його ефективність безпосередньо залежить від обсягу та різноманітності навчальних даних, що у випадку реальних двійкових програм може бути обмежувальним чинником. Обчислювальна складність моделі, зумовлена поєднанням графових операцій, рекурентних архітектур і механізму уваги, може ускладнювати застосування VulANalyzeR у режимі реального часу або в умовах аналізу великих системних проєктів. Крім того рівень інтерпретованості результатів залишається недостатнім для повної автоматизації процесу усунення вразливостей, оскільки остаточні рішення все одно потребують втручання фахівця.

У статті [52] досліджується проблема виявлення вразливостей переповнення буфера у програмних системах на C/C++, що залишаються серйозною загрозою для безпеки. Автори аналізують 159 фрагментів коду з проєктів Linux, Mozilla та Xen, класифікуючи їх за допомогою Ортогональної класифікації дефектів та виявляючи, що більшість проблем пов’язані з

відсутніми або некоректними перевірками умов. Вони також оцінюють ефективність поширених інструментів статичного аналізу (CppCheck і Flawfinder), показуючи їхню низьку результативність у виявленні таких вразливостей. Додатково досліджується застосування програмних метрик, які хоч і широко використовуються як показники якості коду, проте не демонструють чіткої залежності від наявності переповнень буфера. Серед переваг роботи варто виділити системний підхід до класифікації та багатофакторний аналіз, що поєднує інструментальні перевірки з метриками програмного забезпечення. Недоліком є відсутність практичних рішень для покращення інструментів аналізу, оскільки робота здебільшого обмежується діагностичними спостереженнями. Загалом дослідження підкреслює важливість пошуку нових підходів до виявлення вразливостей, акцентуючи на слабких місцях сучасних методів.

Стаття [53] присвячена підвищенню ефективності символьного виконання для виявлення вразливостей пошкодження пам’яті, зокрема купових і стекових переповнень, Use-After-Free та подвійного звільнення, у виконуваному коді реальних програм. Автори пропонують підхід, який обмежує символьне виконання лише на попередньо статично ідентифікованих «тестових блоках» — фрагментах коду, що можуть містити вразливі оператори — і формує для кожного оператора обмеження шляху та умови активації вразливості, розв’язання яких дає конкретні вхідні значення для блоку. Щоб перетворити ці локальні вхідні значення в системні вектори, що досягають потрібного блоку в контексті програми, автори застосовують методи машинного навчання, а весь підхід реалізовано як плагін для фреймворку angr. Серед переваг підходу — значне зменшення ризику вибуху шляху за рахунок таргетованого аналізу, підвищена точність у виявленні чотирьох ключових класів вразливостей та практична реалізація з експериментально підтвердженою кращою продуктивністю порівняно з аналогами. До недоліків належать залежність від якості й повноти специфікацій вразливостей для коректної статичної ідентифікації блоків, можливі похибки машинного навчання при відтворенні

реальних системних вхідних даних та ризик пропусків міжпроцедурних або складніших сценаріїв, що не вкладаються в модель тестового блоку. Загалом робота робить вагомий внесок у практичне застосування символьного виконання, окреслюючи реалістичні обмеження методу та напрямки для подальших досліджень.

У статті [54] розглядається проблема обмеженої здатності існуючих методів виявлення вразливостей коректно відображати семантичний зміст вихідного коду. Автори пропонують підхід ContextCPG, який вдосконалює традиційне представлення графа властивостей коду (CPG), додаючи інформацію про імена змінних і типи даних та поєднуючи аналіз природної мови зі структурним графовим аналізом. Така інтеграція дозволяє точніше враховувати контекст виконання коду та підвищувати ефективність моделей графових нейронних мереж при виявленні складних вразливостей. Перевагою методу є істотне зростання точності — у середньому на 8% порівняно з базовим CPG, що підтверджено експериментами на прикладі трьох поширених класів вразливостей у C/C++. Разом із тим певним обмеженням можна вважати фокусування дослідження на обмеженій кількості вразливостей, що не гарантує універсальності результатів для ширшого спектра загроз. У підсумку робота демонструє цінність поєднання структурних і семантичних характеристик коду та відкриває перспективи для подальшого розвитку методів автоматизованого виявлення вразливостей.

У роботі [55] розглядається проблема обмеженої ефективності інструментів статичного аналізу та ручних аудитів у виявленні складних і малопомітних вразливостей безпеки програмного забезпечення. Автори пропонують нове представлення вихідного коду — мережу вразливостей, що базується на спеціальній мережі Петрі та інтегрує графи залежностей даних і графи потоку керування. Такий підхід дозволяє поєднати переваги автоматизованого аналізу та експертного досвіду аудитора, підвищуючи точність виявлення вразливостей типу «забруднення», включно з переповненнями буфера та ін’єкціями. Результати тестування на наборі

Securibench Micro підтверджують здатність методу ефективно виявляти широкий спектр вразливостей, зберігаючи при цьому низький рівень хибнопозитивних і хибнонегативних результатів. Сильним боком дослідження є інтеграція різних формальних моделей у єдине представлення, що розширює можливості статичного аналізу. Недоліком можна вважати відсутність масштабних експериментів на великих реальних проєктах, що обмежує оцінку практичної придатності методу. Загалом стаття робить внесок у розвиток гібридних підходів до виявлення вразливостей і відкриває перспективи для вдосконалення інтеграції інструментів аналізу та людського аудиту.

У статті [56] представлено нову систему виявлення програмних вразливостей у коді Java, яка базується на поєднанні методів глибокого навчання та гібридного вилучення ознак. Автори інтегрують графові й послідовнісні підходи, використовуючи CFG, AST та PD, а також жадібну векторизацію для формування багатого представлення коду, що покращує як синтаксичне, так і семантичне розуміння. Запропонована архітектура поєднує GCN-RFEMLP та CodeBERT із квантовою згортковою мережею, що дозволяє враховувати довгострокові залежності та підвищує точність і гнучкість виявлення вразливостей. Значним досягненням дослідження є висока точність — 99,2%, що перевищує результати наявних еталонних методів і підтверджує ефективність підходу. До переваг можна віднести здатність системи виявляти широкий спектр вразливостей, включно з переповненнями буфера, SQL-ін’єкціями та міжсайтовим скриптингом, завдяки використанню міжпроцедурного зрізу коду. Водночас обмеженням є орієнтація лише на Java, що зменшує універсальність підходу для багатомовних середовищ розробки. Загалом стаття пропонує вагомий внесок у розвиток інтелектуальних систем безпеки програмного забезпечення, поєднуючи сучасні методи аналізу коду та глибинного навчання.

У статті [57] розглядається проблема виявлення прихованих вразливостей у скомпільованому коді, які активно використовуються шкідливими програмами, зокрема програмами-вимагачами. Автори пропонують фреймворк TEDVIL, що використовує трансформаторні моделі для створення вбудовувань

коду та подальшого навчання нейронних мереж LSTM з метою виявлення стекових переповнень у проміжному представленні LLVM. Запропонований підхід порівнюється з традиційними моделями word2vec, і результати демонструють помітну перевагу трансформаторних вбудовувань, особливо GPT-2, які досягли точності 92,5% та F1-оцінки 89,7%. Серед ключових переваг дослідження варто виділити високу результативність навіть за умов обмежених ресурсів, що підтверджує практичність методу у реальних сценаріях. Водночас певним обмеженням є фокусування лише на одному типі вразливостей — переповненні буфера на основі стеку, що не дозволяє повністю оцінити універсальність TEDVIL. Загалом робота демонструє потенціал трансформаторних моделей у сфері кібербезпеки та закладає основу для подальших досліджень щодо застосування глибинного навчання у виявленні складних вразливостей.

У статті [58] розглядається проблема вразливостей типу Use-After-Free (UAF), які стають дедалі поширенішими у програмах на C/C++ та призводять до критичних наслідків, зокрема віддаленого виконання коду й витоку даних. Автори здійснюють систематичний огляд існуючих підходів до виявлення та зменшення наслідків експлуатації UAF, вперше пропонуючи комплексне порівняння різних методів. Вони оцінюють ефективність статичних і динамічних детекторів, демонструючи, що перші краще підходять для внутрішньопроцедурних вразливостей, тоді як другі залишаються більш надійними у випадку міжпроцедурних UAF. Перевагою дослідження є масштабний аналіз, який охоплює як технічні характеристики інструментів, так і їх вплив на продуктивність систем. Водночас певним обмеженням є те, що автори зосереджуються на існуючих рішеннях, не пропонуючи нових практичних механізмів усунення вразливостей. Стаття підсумовує сучасний стан проблеми та окреслює напрями майбутніх досліджень, що можуть сприяти створенню більш ефективних засобів захисту.

1.3 Постановка задачі

Сучасний розвиток інформаційних технологій супроводжується зростанням складності програмного забезпечення, що, у свою чергу, підвищує ймовірність виникнення критичних вразливостей. Одним із найбільш поширених та небезпечних типів таких вразливостей є переповнення буферу, яке може призвести до відмови у функціонуванні програмних систем, витоку конфіденційних даних чи віддаленого виконання коду. Попри наявність широкого спектра методів аналізу програмного забезпечення ці підходи характеризуються низкою обмежень. Серед них: значні обчислювальні витрати, проблема «вибуху шляхів», високий рівень хибнопозитивних результатів та обмежена масштабованість при роботі з великими системами. У зв’язку з цим виникає потреба у створенні більш ефективного та адаптивного методу, здатного забезпечити достовірне виявлення вразливостей типу переповнення буферу.

Метою даної роботи є розробка методу виявлення вразливостей типу переповнення буферу в програмному забезпеченні, що поєднує формальне математичне моделювання з алгоритмами навчання з підкріпленням для підвищення точності, швидкодії та достовірності аналізу.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

– здійснити критичний огляд і аналіз наявних підходів до виявлення програмних вразливостей та визначити їхні переваги й недоліки;

– побудувати математичну модель переповнення буферу, яка формалізує умови виникнення вразливості та враховує особливості машинної арифметики;

– розробити алгоритм симулятивного покриття з використанням Q-навчання, що забезпечує адаптивне дослідження поведінки програм;

– створити метод генерації вхідних даних на основі системи обмежень і вузлів виконання програмного коду;

– провести експериментальне тестування розробленого методу та здійснити оцінку його ефективності за критеріями точності, швидкодії та рівня хибнопозитивних результатів.

2 МЕТОД ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ ТИПУ ПЕРЕПОВНЕННЯ БУФЕРУ В ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННІ

2.1 Математична модель переповнення буферу

Пам’ять процесу моделюємо як множину адрес, яку позначаємо символом A. Поняття A несе смисл сукупності всіх можливих адрес у даному адресному просторі; практично це інтервал від нуля до $2^w - 1$, де w - ширина адресного простору в бітах (наприклад, $w = 32$ або $w = 64$). Далі пам’ять формалізується як функція:

$$M: A \to B \tag{2.1}$$

де M - функція стану пам’яті, яка кожній адресі $\alpha \in A$ ставить у відповідність значення байта; B - це множина значень одного байта, тобто чисел від 0 до 255 у цілих одиницях.

Таким чином, вираз $M(a)$ це значення байта у пам’яті за адресою a. Позначення M' у подальшому використовуватиметься для означення нового стану пам’яті після виконання операції запису; отже, M' відрізняється від M у точках адрес, куди записано нові байти.

Буфер у моделі визначається як неперервний інтервал адрес. Позначення $B = [l_B, u_B]$ означає, що буфер B починається з адреси l_B і закінчується адресою u_B. У цьому записі символ l_B означає нижню межу інтервалу (lower bound), а u_B означає верхню межу (upper bound). Розмір буфера позначається функцією

$$size(B) = u_B - l_B + 1 \tag{2.2}$$

де $u_B - l_B$ обчислює кількість проміжних адрес; а $+1$ відображає те, що обидві кінцеві адреси включені в інтервал.

Надалі термін «розмір буфера» це саме про $size(B)$, вимірюваний в байтах.

Операція запису в пам’ять формалізується як запис послідовності байтів починаючи з певної початкової адреси. Нехай початкова адреса запису позначена a, а послідовність байтів, що записується, має довжину n і елементи $w_0, w_1, \dots, w_{n-1}$. Тоді зміна стану пам’яті сформулюється як тригерна умова

$$\forall i \in \{0, \dots, n-1\}: M'(a+i) = w_i \tag{2.3}$$

де $a+i$ позначає адресу кожного записуваного байта; i – індекс, що приймає всі позиції від нульової до останньої.

Як наслідок, операція $memwrite(a, w)$ перетворює стан пам’яті M на M', де в новому стані байти від адреси a до адреси $a+n-1$ дорівнюють відповідним елементам послідовності w.

Переповнення буфера означає, що під час такої операції запису хоча б одна адреса, на яку виконується запис, не належить інтервалу, який виділено під буфер. Формально це виражається як

$$\exists i \in \{0, \dots, n-1\}: a+i \notin [l_B, u_B] \tag{2.4}$$

Тут символ $\exists$ означає існування індекса i, такий що адреса $a+i$ знаходиться поза межами буфера. Ця формула еквівалентна диз’юнкції двох простих умов: $a < l_B$ або $a+n-1 > u_B$. У першій частині $a < l_B$ читається як «початкова адреса починається лівіше від початку буфера»; у другій частині $a+n-1 > u_B$ читається як «адреса останнього байта запису перевищує верхню межу буфера», де $a+n-1$ - адреса останнього записаного байта.

У багатьох реальних програмах початкова адреса запису a формується як сумарна величина:

$$a = l_B + idx * s \tag{2.5}$$

де idx - індекс елемента у масиві; а s - розмір одного елемента в байтах.

Наприклад, для масиву цілих типу int, якщо int займає 4 байти, то $s = 4$. Символ idx може залежати від вхідних даних програми або від проміжних обчислень; отже a може бути функцією від вхідного вектору даних, який у подальшому буде позначено через I. Ключовим моментом є те, що добуток $idx * s$ підлягає машинній арифметиці і при великих значеннях може виникнути переповнення цілого числа (integer overflow). Оскільки в реальній машині арифметика адрес часто виконується як операція над бітовими векторами фіксованої довжини (наприклад 32-бітові або 64-бітові), результат обчислення $l_B + idx * s$ фактично рівний $(l_B + idx * s) mod 2^w$, де w - ширина бітової арифметики. Символ операції «$mod 2^w$» означає, що арифметика відбувається по колу і всі значення згортаються у проміжок $[0, 2^w - 1]$. Якщо wraparound відбувся, то адреса a може стати зовсім іншою від очікуваної, і, як наслідок, запис може піти у непередбачувану область пам'яті, створивши переповнення. Таким чином, при формуванні перевірок безпеки треба ураховувати не лише формальні нерівності $a \geq l_B$ та $a + n - 1 \leq u_B$, а й перевірки на те, чи проміжні арифметичні операції не спричинюють wrap. У моделі це формалізується умовою на відсутність арифметичного переповнення: «проміжний добуток $idx * s$ та проміжна сума $l_B + idx * s$ повинні бути меншими за 2^w у сумарному вигляді». Умову відсутності wrap можна представити як нерівність у безмежній арифметиці, наприклад $l_B + idx * s < 2^w$ та $l_B + idx * s + n \leq 2^w$, якщо моделюється $0..2^{w-1}$ простір. Якщо такої гарантії немає, слід моделювати арифметику як біт-векторну і виконувати всі перевірки у біт-векторному сенсі.

Часто зустрічається випадок - off-by-one, коли програма дозволяє записати на один байт більше, ніж розмір буфера. Формально це виражається як рівність $a + n - 1 = u_B + 1$. У цій формулі $u_B + 1$ - це адреса на одиницю більше від верхньої межі буфера; отже запис охоплює саме один байт за межами буфера. Off-by-one часто здається незначним, але він може торкнутися критичного поля у суміжній структурі, наприклад старшого байта покажчика, прапора або канарки, що робить подальшу експлуатацію реалістичною.

Щоб однозначно сформулювати, при яких вхідних даних виникає переповнення, вводимо поняття шляхової умови або path condition, яке позначено $PC(I)$. Де I - це вектор усіх вхідних змінних програми, наприклад аргументів функції, значень зі stdin або мережевих пакетів. Path condition $PC(I)$. - це сукупність логічних обмежень, які змусять виконатися саме ту послідовність операторів і розгалужень, що приводить до розглядуваного місця у коді. У символьному виконанні кожен оператор розгалуження породжує умову, яка додається до PC; у підсумку PC є логічною формулою над вхідними змінними I.

Переповнення у термінах PC формулюється як існування таких вхідних значень I, для яких умова шляху виконується і при цьому запис виходить за межі буфера:

$$\exists I. PC(I) \wedge (a(I) < l_B \vee a(I) + n(I) - 1 > u_B) \tag{2.6}$$

У формулі 2.6 $a(I)$ означає, що початкова адреса запису може бути функцією від вхідних значень I, а $n(I)$ - довжина запису, також залежна від I. Символ ∃ означає, що достатньо знайти хоч одне конкретне значення вектору I, яке робить формулу істинною. Саме такі формули передаються SMT-розв’язувачам, наприклад Z3 або Boolector, в моделюванні біт-векторної арифметики; якщо SMT повертає «sat», то модель містить конкретні значення вхідних змінних, які викликають переповнення. Отримані значення можна синтезувати в тест-кейс і відтестувати на реальному бінарному коді.

У практичному застосуванні при моделюванні wrap та integer overflow використовується арифметика біт-векторів. Тоді варіант умови «$a(I) + n(I) - 1 > u_B$» перетворюється на відповідне беззнакове порівняння біт-векторів: $bvugt(bvadd(bvsub(bvadd(a, n), 1), u_B))$ у термінах SMT-LIB. Тут $bvadd$, $bvsub$, $bvugt$- операції додавання, віднімання і unsigned greater-than над біт-векторами. Важливо, що за таких умов SMT-модель враховує wrap автоматично, тому знахідка розв’язу свідчитиме про реальну можливість виходу за межі у

машинному виконанні.

Переповнення само по собі не завжди означає успішну атаку. Щоб модель відображала реальну можливість експлуатації, вводимо множину критичних адрес C. У множину C входять, наприклад, адреси збереженого повернення (return address), адреси локальних покажчиків, дескриптори, які контролюють поведінку програми, або інші структури, перезапис яких може призвести до зміни потоку виконання. Подія перезапису критичного об'єкта записується як

$$\exists i: a + i \in C \tag{2.7}$$

де i - приймає значення індексів записуваної послідовності; a - початкова адреса; належність $a + i \in C$ означає, що хоча б один байт перезаписаної послідовності попадає на адресу з множини критичних об'єктів.

Однак навіть при перезаписі критичної адреси атака може не відбутися, якщо працюють захисні механізми. У модель додано булеві змінні, що описують наявність захистів: canary- чи встановлено stack canary; ASLR - чи включено адресну рандомізацію; NX - чи відключено виконання у сегменті даних; CFI - чи працює механізм control-flow integrity. Кожна з цих змінних у моделі впливає на предикат успішної експлуатації, який позначимо як $\varphi(M')$. Вираз $\varphi(M')$ означає, що після впливу запису і отримання нового стану пам'яті M' існує шлях виконання, що приводить до бажаного ефекту атакуючого, наприклад до виконання власного коду або отримання неправомірного доступу. Повна умова існування експлуатуючого вхідного вектора можна записати як

$$\exists I. PC(I) \wedge (\exists i: a(I) + i \in C) \wedge \varphi(M') \tag{2.8}$$

У формулі 2.8 $PC(I)$ гарантує, що вхідні дані приводять до шляху виконання, де виконується вразлива операція; $\exists i: a(I) + i \in C$ - наявність перезапису критичного об'єкта; $\varphi(M)'$ - можливість довести, що після такого перезапису програма фактично переходить у стан, контрольований атакуючим.

Для моделювання впливу ASLR корисно вводити поняття ентропії адрес. Нехай через k позначено кількість біт ентропії, яку надає ASLR для конкретного регіону пам'яті. Тоді ймовірність того, що атакуючий у одній спробі вірно вгадає потрібну адресу, приблизно рівна 2^{-k}. Умовну ймовірність успіху атаки можна формалізувати як

$$Psuccess \approx P(overwrite\ critical) * 2^{-k} * P(bypass\ protections) \quad (2.9)$$

У формулі 2.9 множник $P(overwrite\ critical)$ відображає імовірність того, що при певному способі формування вхідних даних відбудеться перезапис адрес з множини C; множник 2^{-k} - шанси подолати ASLR; множник $P(bypass\ protections)$ - умовна ймовірність обходу інших захисних механізмів, як-от Canary або CFI. Така ймовірнісна модель корисна для кількісної оцінки ризику в практичних системах.

Наведено простий, але типовий приклад переповнення буфера на мові C на рисунку 2.1.

```
void vuln(char *s) {
    char buf[16];
    strcpy(buf, s);
}
```

Рисунок 2.1 - Типовий приклад переповнення буфера на мові C

Локальний буфер buf має розмір 16 байтів. У термінах моделі буфер описується як $B = [l_B, l_B + 15]$, оскільки $u_B = l_B + 15$ і $size(B) = 16$. Операція $strcpy(buf, s)$ запише в буфер $n = strlen(s) + 1$ байтів (включно з термінальним нуль-байтом). Умова переповнення подається як $strlen(s) + 1 > 16$. Тут $strlen(s)$ - функція, що залежить від вхідного аргументу s. Символьне виконання згенерує шляхову умову PC (у цьому простому прикладі вона може бути порожньою або містити лише домени значень s) і SMT-розв'язувач легко

знайде модель s таку, що $strlen(s) \geq 16$. Далі потрібно перевірити, чи при такому переповненні буде перезаписано критичні адреси у стековому кадрі - наприклад, адресу повернення. Якщо так, і якщо stack canary відсутня або її можна обійти, то $\varphi(M^{)\prime}$ може бути істинною, і атака є здійсненною.

Приклад з integer overflow у виділенні пам'яті демонструє більш тонкий патерн (рис.2.2).

```
size_t len = get_len();
size_t alloc = len + 1;          // може переповнитись в беззнаковому типі
char *p = malloc(alloc);
memcpy(p, src, len);
```

Рисунок 2.2 - Приклад з integer overflow у виділенні пам'яті

Тут потенційна вразливість виникає, якщо значення len настільки велике, що $len + 1$ у беззнаковій 32-бітовій арифметиці викликає wrap, тобто alloc стане маленьким значенням замість очікуваного великого. Формалізація цього сценарію: якщо $alloc = (len + 1) mod 2^w$ і в реальності len велике, то може статися $alloc < len + 1$. Якщо malloc поверне блок розміру $alloc$, а $memcpy$ скопіює len байтів, запис перетне кордон виділеної пам'яті, тобто умова безпеки $a + n - 1 \leq u_B$ буде порушена.

2.2 Q-навчання як інструмент адаптивного симулятивного покриття

Вибір алгоритму Q-навчання обумовлений двома основними причинами. По-перше, алгоритм є гнучким та адаптивним завдяки своїй властивості model-free та off-policy, що означає можливість навчання без попередньо заданої моделі середовища і незалежність від конкретної політики збору досвіду. По-друге, Q-навчання як метод навчання з підкріпленням потребує мінімальної підготовки

даних і моделей порівняно з алгоритмами навчання з учителем або без учителя, що робить його більш придатним для завдань аналізу програмного забезпечення.

Було проведено детальне дослідження алгоритму Q-навчання та вивчено, як його можна інтегрувати у наявну кодову базу Python. Функціональність алгоритму реалізовано без використання сторонніх бібліотек, що забезпечило гнучкість у налаштуванні його компонентів. Ключовим елементом є таблиця Q, яка зберігає значення очікуваної кумулятивної винагороди для кожної пари «стан–дія». Оновлення таблиці відбувається за рівнянням Беллмана:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma maxQ(s',a') - Q(s,a)], \qquad (2.10)$$

де s- поточний стан, a- виконана дія, r- отримана винагорода, s' - новий стан, α - коефіцієнт навчання, що визначає швидкість оновлення знань, а γ - дисконт-фактор, який контролює значущість майбутніх винагород.

Інтеграція алгоритму виконувалася у функції, відповідальній за символьне виконання коду та генерацію вхідної граматики для функцій. Під входом розуміється процес створення тестових даних, здатних активувати вразливість типу переповнення буферу. Для інтеграції було створено клас Q-Learning (Додаток Б), що містив таблицю Q та набір внутрішніх методів. Метод _update_q_table реалізовує оновлення функції Q відповідно до рівняння Беллмана. Метод _choose_action реалізує політику epsilon-greedy: з імовірністю ε обирається випадкова дія для дослідження нових варіантів, а з імовірністю 1–ε - дія з найбільшим значенням Q, що забезпечує використання вже накопиченого досвіду. Метод _perform_action застосовує вибрану дію до середовища, повертає новий стан, винагороду та індикатор завершення епізоду, тоді як calculate_reward визначає систему винагород: +100 за успішне завершення завдання, -10 у випадку стагнації та -1 за інші переходи. Метод _check_if_done контролює завершення епізоду, забезпечуючи правильність навчального процесу.

Завдяки такій структурі агент здатний ефективніше обирати шляхи аналізу в дереві обмежень, що суттєво зменшує час генерації тестових входів та збільшує

кількість комбінацій, корисних для пошуку потенційних вразливостей. Важливо, що використання Q-навчання дозволяє уникати повного перебору шляхів виконання, який часто призводить до «вибуху простору станів». Натомість агент адаптивно навчається віддавати перевагу тим діям, що мають вищу очікувану цінність, поступово оптимізуючи процес дослідження.

Таким чином, Q-навчання виступає не лише як допоміжний механізм, а як центральний інструмент адаптивного управління простором пошуку. Його використання в задачах символьного виконання та генерації вхідних даних дозволяє досягати балансу між швидкістю роботи й глибиною охоплення тестових сценаріїв, що забезпечує вищу ймовірність виявлення критичних вразливостей типу переповнення буферу в програмному забезпеченні.

2.3 Побудова дерева обмежень

Побудова дерева обмежень у процесі символьного виконання програми є одним із етапів методу симулятивного покриття, оскільки саме це дерево відображає множину всіх можливих шляхів виконання та умови їхнього досягнення.

Процес починається з того, що вихідна програма виконується не на конкретних значеннях вхідних даних, а на так званих символьних змінних. Кожна змінна у вхідному просторі розглядається як абстрактний символ, тобто:

$$X = \{x_1, x_2, \dots, x_n\} \tag{2.11}$$

Всі операції програми інтерпретуються не в числовій формі, а у вигляді формул і логічних виразів. У результаті кожна гілка програми, що містить умовний оператор, породжує певне обмеження. Наприклад, оператор перевірки «якщо x більше ніж нуль» створює дві гілки: одна з умовою «x більше нуля», інша з умовою «x менше або дорівнює нулю».

$$if\ \big(c(x_1, x_2, \dots, x_n)\big)\ then\ B_1\ else\ B_2 \qquad (2.12)$$

Такий оператор створює дві гілки: одна з умовою $c(x_1, x_2, \dots, x_n)$ та інша з умовою $\neg c(x_1, x_2, \dots, x_n)$. Таким чином, у момент зустрічі умовного переходу дерево обмежень розширюється новими вузлами. Кожен вузол дерева відповідає певній множині обмежень, які накопичуються від початку виконання до поточного моменту. Якщо шлях проходить через кілька умовних операторів, то для нього формується система логічних і арифметичних виразів, що описують необхідні співвідношення між вхідними параметрами.

$$C = \{c_1(x_1, x_2, \dots, x_n), c_2(x_1, x_2, \dots, x_n), \dots, c_k(x_1, x_2, \dots, x_n)\} \qquad (2.13)$$

Усі ці вирази зберігаються як шляхові обмеження. У процесі побудови дерево постійно розгалужується: кожна нова розгалужувальна умова створює дві або більше гілок, а символьний виконавець заносить відповідні формули у множину обмежень для кожної з них. Якщо певна гілка виявляється внутрішньо суперечливою, тобто система обмежень не має жодного розв’язку, вона відсікається і більше не розглядається. Таким чином, дерево містить лише ті гілки, які відповідають потенційно здійсненним сценаріям виконання програми.

Поступово формується повна структура, що відображає всі можливі шляхи, якими може пройти програма залежно від вхідних даних. Кожен шлях у цьому дереві визначається набором умов, які повинні бути виконані одночасно. Тому дерево обмежень виступає основою для подальшого етапу - генерації вхідних даних. Розв’язуючи систему рівнянь та нерівностей, що відповідає певному вузлу чи гілці, можна отримати конкретні тестові значення, які забезпечують проходження програми саме цим шляхом.

Таким чином, побудова дерева обмежень у результаті символьного виконання створює формальну модель всіх можливих виконань програми. Вона є фундаментом для систематичного аналізу поведінки програмного забезпечення, оскільки дозволяє одночасно враховувати і структуру коду, і логіку прийняття рішень у ньому.

2.4 Алгоритм навчання поведінки програм шляхом симулятивного покриття

Алгоритм навчання поведінки програм шляхом симулятивного покриття призначений для симуляції виконання програми та навчання її поведінки шляхом систематичного генерування та аналізу тестових входів. Він використовується для автоматизованого дослідження вузлів програми з потенційно вразливими ділянками, зокрема для виявлення переповненого буферу. Алгоритм обробляє одиницю тестування у рамках системи, використовуючи дерево обмежень, сформоване в результаті символьного виконання.

На першому етапі виконується n-факторна комбінаційна Монте-Карло симуляція, що дозволяє створити початкові набори системних векторів, які представляють різноманітні можливі стані програми. Формально це означає, що з простору D генерується вибірка $\{v^1, v^2, \dots, v^m\}$ така що для будь-якої підмножини змінних розмірності $n' \leq n$ усі можливі комбінації значень цих змінних гарантовано представлені у вибірці:

$$D = D_1 * D_2 * \dots * D_n, \tag{2.14}$$

де D_i - область можливих значень змінної x_i .

Для кожного згенерованого вектора $v^j = (v_1^j, v_2^j, .., v_n^j)$ перевіряється виконання умови $C(v_1^j, v_2^j, .., v_n^j) = true$. Якщо вона виконується, то v^j зберігається як тестовий вхід.

Ці вектори використовуються для моделювання поведінки програми на рівні системи та окремих тестових одиниць. У класичному підході Монте-Карло основна ідея полягає в багаторазовому генеруванні випадкових значень вхідних параметрів згідно з визначеними розподілами та подальшій оцінці результативних показників системи. Однак у випадках, коли система включає десятки або навіть сотні параметрів, традиційна випадкова вибірка може не охопити важливі комбінації їхніх значень. Для подолання цієї проблеми

використовується n-факторний комбінаційний підхід. Його сутність полягає в тому, що для будь-якої множини параметрів забезпечується врахування всіх можливих комбінацій значень до рівня n. Так, у разі двофакторної симуляції (pairwise) гарантується наявність усіх можливих пар значень параметрів у вибірці, тоді як трифакторна симуляція охоплює всі трійки і так далі. Це значно підвищує ймовірність виявлення взаємозалежностей та прихованих взаємодій між параметрами, які залишаються непоміченими у традиційних реалізаціях Монте-Карло. Унаслідок цього n-факторна комбінаційна Монте-Карло симуляція дає змогу зменшити кількість необхідних експериментів у порівнянні з повним перебором, водночас забезпечуючи вищу якість покриття простору вхідних даних. Вона також підвищує ефективність виявлення критичних сценаріїв функціонування складних систем і створює можливості для досягнення оптимального балансу між обчислювальною складністю та точністю моделювання.

Далі алгоритм здійснює обхід дерева обмежень (формула) пошуком у ширину (Breadth First Search), аналізуючи лише ті вузли, які знаходяться на потенційно вразливих шляхах.

$$T = (V, E), \tag{2.15}$$

де V - множина вузлів; E - множина ребер, що відповідають умовним переходам.

Формально, порядок відвідування вузлів задається чергою

$$Q = [v_0, v_1, \dots, v_k] \tag{2.16}$$

де v_0 - кореневий вузол.

Для кожного вузла $v \in V$ зберігається множина обмежень $C(v)$, яка є кон'юнкцією умов уздовж шляху від кореня до вузла:

$$C(v) = \bigwedge_{i=1}^{m} c_i(x_1, \dots, x_n) \tag{2.17}$$

Для кожного вузла перевіряється, чи вже він покритий або оброблений раніше. Для цього вводиться бінарний предикат$covered(v)$, який визначає чи вже існує набір вхідних даних $\alpha: X \to R$, що задовольняє обмеження $C(v)$. Формально:

$$covered(v) = \exists \alpha \forall c_i \in C(v): c_i\big(\alpha(x_1), \dots, \alpha(x_n)\big) = true \qquad (2.18)$$

Якщо $covered(v) = 1$ і те саме виконується для всіх сусідніх вузлів $u \in Ad_j(v)$, то вузол вважається вже обробленим.

Якщо вузол та його сусіди вже покриті, алгоритм обчислює вхідні дані за допомогою функції RunTar3 на підмножинах векторів, щоб визначити діапазони системних входів, які можуть активувати вузол.

Якщо$covered(v) = 1$, але вузол потребує уточнення діапазону вхідних даних, викликається функція симуляції:

$$RunTar3(S(v), U(v)) \to R(v), \qquad (2.19)$$

де $S(v)$ - підмножина векторів, що покривають вузол; $U(v)$ - решта векторів; $R(v)$- обчислені діапазони вхідних змінних, які забезпечують активацію вузла v.

Якщо вузол задовільний (тобто система $C(v)$ має розв'язок), але ще не покритий ($covered(v) = 0$), то алгоритм об'єднує обмеження шляху і обмеження самого вузла:

$$C'(v) = C_{path}(v) \wedge C_{node}(v) \qquad (2.20)$$

Обмеження вузла і шляху використовуються для рекурсивного обчислення мапи входів через функцію ComputeMap, що дозволяє згенерувати релевантні дані для досягнення вузла:

$$ComputeMap\big(C'(v)\big) \to M(v), \quad (2.21)$$

де $M(v)$ - відображення між вхідними змінними та згенерованими значеннями, яке забезпечує проходження програми через вузол v.

Тобто, якщо $M(v) = \{(x_1 \to a_1), (x_2 \to a_2), \dots, (x_n \to a_n)\}$, то підстановка цих значень у вхідні параметри гарантує виконання всіх умов у $C'(v)$.

Після проходження всіх вузлів алгоритм генерує кінцеві тестові входи для вузлів, які є потенційно задовільними та вразливими. Якщо вузол покритий, для його активації використовуються раніше визначені діапазони вхідних даних, інакше обчислені значення на основі обмежень. Це дозволяє забезпечити як повне покриття тестових одиниць, так і оптимізацію процесу генерації вхідних даних, зменшуючи кількість непотрібних симуляцій та підвищуючи ефективність аналізу.

Отже, для кожного вузла v представлено наступний формалізований алгоритм:

$$Input(v) = \begin{cases} RunTar3\big(S(v), U(v)\big), \text{якщо } covered(v) = 1 \\ ComputerMap(C_{path}(v) \wedge C_{node}(v)), \text{якщо } covered(v) = 0 \text{ і } C(v) \text{задовільне} \\ \emptyset, \text{якщо } C(v) \text{ несумісне} \end{cases} \quad (2.22)$$

Таким чином, алгоритм реалізує комплексний підхід до автоматизованого тестування програм, оскільки поєднує симуляцію виконання, навчання поведінки вузлів та систематичну генерацію тестових входів з урахуванням обмежень шляху та вузлів. Рекурсивна обробка вузлів і використання діапазонів системних векторів дозволяють забезпечити точність та ефективність дослідження потенційно вразливих ділянок програми.

Послідовність кроків алгоритму, що використовується для виконання програми та навчання програми, наступна (рис.2.3):

Крок 1. Виконати n-факторну комбінаційну Монте-Карло симуляцію у багатовимірному просторі.

Крок 2. Створити множину спостережуваних векторів, що представляють

собою системні вектори та відповідні вектори на рівні одиниці тестування.

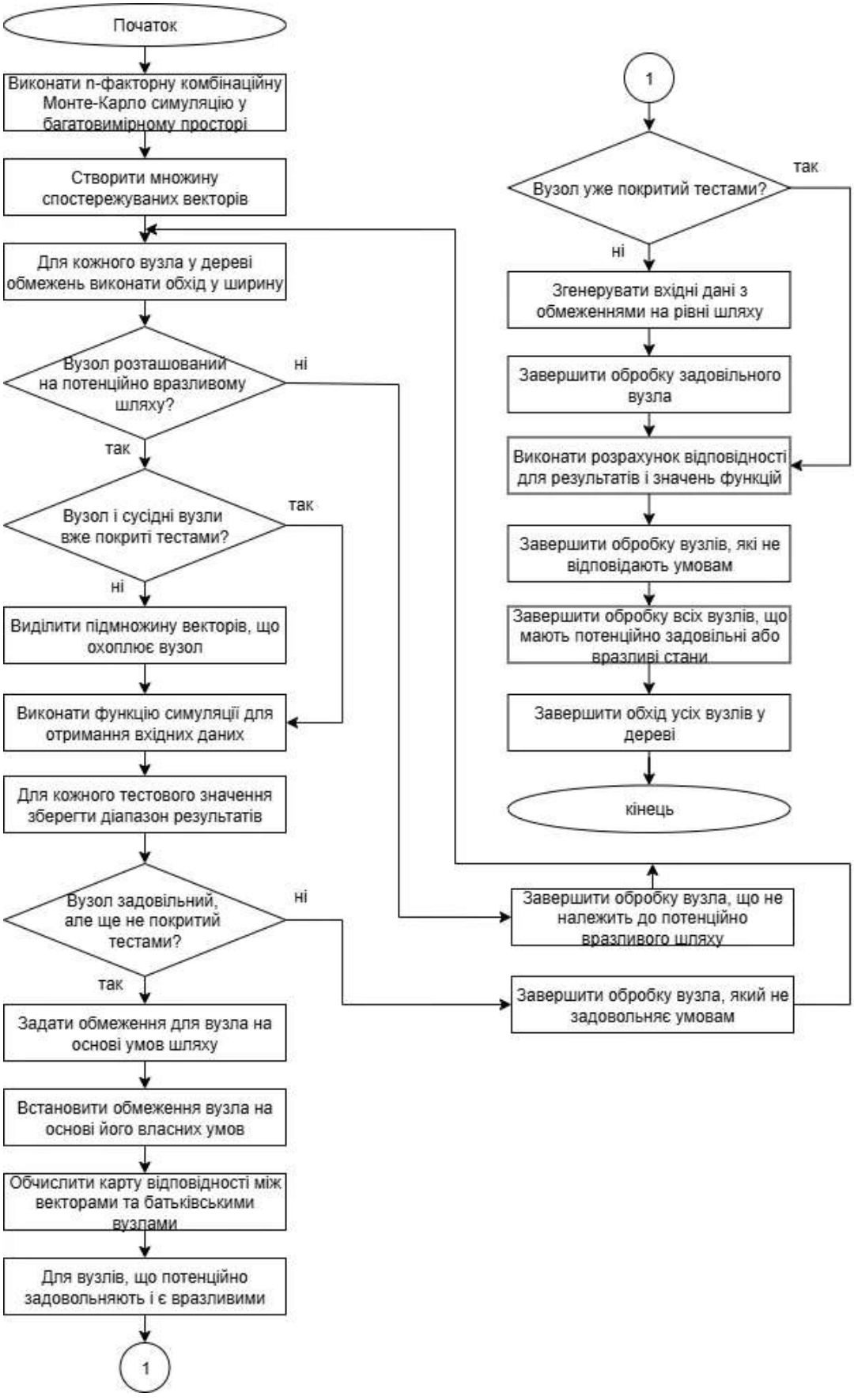

Рисунок 2.3 - Алгоритм навчання поведінки програм шляхом симулятивного покриття

Крок 3. Для кожного вузла у дереві обмежень виконати обхід за допомогою пошуку в ширину.

Крок 4. Якщо вузол розташований на потенційно вразливому шляху, перейти до наступної перевірки, інакше перейти до подальшого кроку.

Крок 5. Якщо вузол та його сусідні вузли вже покриті тестами, перейти до наступного кроку, інакше виконати процедури аналізу.

Крок 6. Виділити підмножину векторів, яка охоплює вузол, а решту векторів вилучити з подальшого аналізу.

Крок 7. Виконати функцію симуляції для отримання набору вхідних даних.

Крок 8. Для кожного отриманого тестового значення зберегти відповідний діапазон результатів.

Крок 9. Якщо вузол є задовільним, але ще не покритим тестами, перейти до наступного етапу.

Крок 10. Задати обмеження для вузла на основі умов шляху, що до нього веде.

Крок 11. Встановити обмеження вузла на основі умов самого вузла.

Крок 12. Виконати обчислення карти відповідності між векторами та батьківськими вузлами для отримання нових вхідних даних.

Крок 13. Завершити обробку вузла, який не задовольняє умовам.

Крок 14. Завершити обробку вузла, що не належить до потенційно вразливого шляху.

Крок 15. Завершити обхід усіх вузлів у дереві.

Крок 16. Для вузлів, що потенційно задовольняють і є вразливими, перейти до наступного етапу.

Крок 17. Якщо вузол уже покритий, продовжити роботу з наступними, інакше перейти до генерації тестових входів.

Крок 18. Згенерувати вхідні дані, використовуючи обмеження на рівні шляху та обмеження для вразливого стану, і зберегти відповідні діапазони значень.

Крок 19. Завершити обробку задовільного вузла.

Крок 20. Виконати розрахунок відповідності для отримання результатів і обчислення значень функцій.

Крок 21. Завершити обробку вузлів, які не відповідають умовам.

Крок 22. Завершити обробку всіх вузлів, що мають потенційно задовільні або вразливі стани.

2.5 Алгоритм генерації вхідних даних на основі обмежень шляху та вузлів

Генерація вхідних даних у межах методу симулятивного покриття ґрунтується на побудованому дереві обмежень, яке відображає логіку виконання програми. Кожен шлях у цьому дереві описується системою умов, що накладаються на вхідні змінні, і саме ця система використовується як основа для формування конкретних тестових вхідних даних.

Процес починається з вибору певного вузла або гілки дерева. Кожен вузол містить накопичені обмеження, що виникли під час проходження програми через відповідні умовні оператори. Ці обмеження можуть мати різну природу: арифметичні рівняння чи нерівності, логічні зв’язки між змінними, обмеження на діапазони значень або більш складні функціональні залежності. Завдання полягає у тому, щоб знайти конкретний набір числових значень вхідних параметрів, які задовольняють усі ці обмеження одночасно.

Якщо система обмежень є відносно простою, для її розв’язання застосовуються стандартні методи символьної алгебри або SMT-солвери (satisfiability modulo theories), які перевіряють можливість виконання умов і повертають конкретні приклади значень. У випадках, коли система є надто складною, містить нелінійні залежності або обмеження з багатьма змінними, до процесу підключаються наближені чисельні методи. Тут важливу роль відіграє n-факторна комбінаційна Монте-Карло симуляція. Вона забезпечує імовірнісне дослідження простору можливих значень із гарантією того, що всі комбінації до певного порядку n будуть охоплені. Це означає, що навіть у багатовимірному

просторі можна відносно швидко знайти ті набори входів, які активують вузол або шлях, що цікавить.

Щоб забезпечити ефективність генерації, використовується механізм відбору підмножини векторів, які найближче розташовані до виконання шляхових обмежень. Решта векторів відсікаються, що значно зменшує обсяг обчислень. Далі застосовується функція симуляції, яка перевіряє, чи справді обрані вхідні дані призводять до бажаного проходження програми. Якщо так, вони зберігаються як тестові приклади. Якщо ж ні, то алгоритм або модифікує початкові вектори, або переходить до додаткового аналізу обмежень для уточнення пошуку. У деяких випадках застосовується також апроксимація кривих (curve fitting). Вона дає змогу побудувати відображення між множиною вхідних змінних і відповідними результатами симуляції. Завдяки цьому алгоритм отримує можливість не лише знайти поодинокі значення, а й описати цілий діапазон потенційних входів, які гарантують проходження програми заданим шляхом. Це суттєво підвищує універсальність тестування, адже замість одного прикладу отримується клас еквівалентних значень.

У підсумку, процес генерації вхідних даних можна розглядати як поєднання точного розв’язання системи обмежень і імовірнісного пошуку, що доповнюється механізмами наближеного моделювання. Такий підхід забезпечує створення тестів, які не лише формально задовольняють умови вузлів програми, а й максимально наближені до реальних сценаріїв її використання. Це робить метод придатним як для виявлення логічних дефектів, так і для ідентифікації прихованих вразливостей, включно з переповненням буфера чи некоректною обробкою крайових випадків.

Алгоритм генерації вхідних даних на основі обмежень шляху та вузлів призначений для генерації відповідних вхідних даних для вузлів програми на основі обмежень шляху та обмежень вузлів. Він працює рекурсивно, об’єднуючи інформацію про вузли з батьківської ієрархії, щоб забезпечити повне охоплення потенційно вразливих ділянок програми та уникнути неповних або невірних тестових входів.

На першому етапі алгоритм визначає множину змінних, що входять до обмежень вузла і обмежує модель і для цих змінних. Потім він виділяє підмножину системних векторів, що містять найближчі 20% точок до обмежень шляху вузла, і визначає залишок. Далі виконується функція RunTar3, яка дозволяє перевірити, чи можливо плавне побудування мапи входів для вузла. Якщо побудова мапи є плавною, алгоритм створює відображення за допомогою апроксимації кривої, що дозволяє забезпечити безперервну та узагальнену генерацію тестових входів. Якщо побудова не є гладкою, алгоритм рекурсивно об’єднує обмеження батьківських вузлів, перевіряючи їх на наявність та покриття. У разі необхідності він проходить по ланцюгу батьківських вузлів, поки не досягне порогу кількості векторів, достатніх для побудови надійної мапи.

Після цього алгоритм повторно виконує RunTar3 для оновлених множин векторів і обмежень та повторно визначає змінні, що входять до обмежень. Завершальним кроком є повторне побудування мапи входів з використанням апроксимації кривої, що забезпечує коректне та повне визначення вхідних даних для тестування вузла.

У результаті алгоритм дозволяє систематично та ефективно генерувати вхідні дані, які задовольняють як обмеження шляху, так і обмеження вузлів, що є критично важливим для автоматизованого тестування програм на наявність вразливостей, таких як переповнення буферу. Завдяки рекурсивній обробці та комбінуванню інформації з батьківської ієрархії алгоритм знижує ризик пропуску потенційно вразливих станів та підвищує точність симуляції виконання програми.

Послідовність кроків алгоритму, що використовується для виконання програми та навчання програми, наступна (рис. 2.4):

Крок 1. Отримати множину змінних із поточного набору обмежень і зберегти її як множину вхідних змінних; після цього перейти до кроку 2.

Крок 2. Обмежити модель лише цими вхідними змінними; перейти до кроку 3.

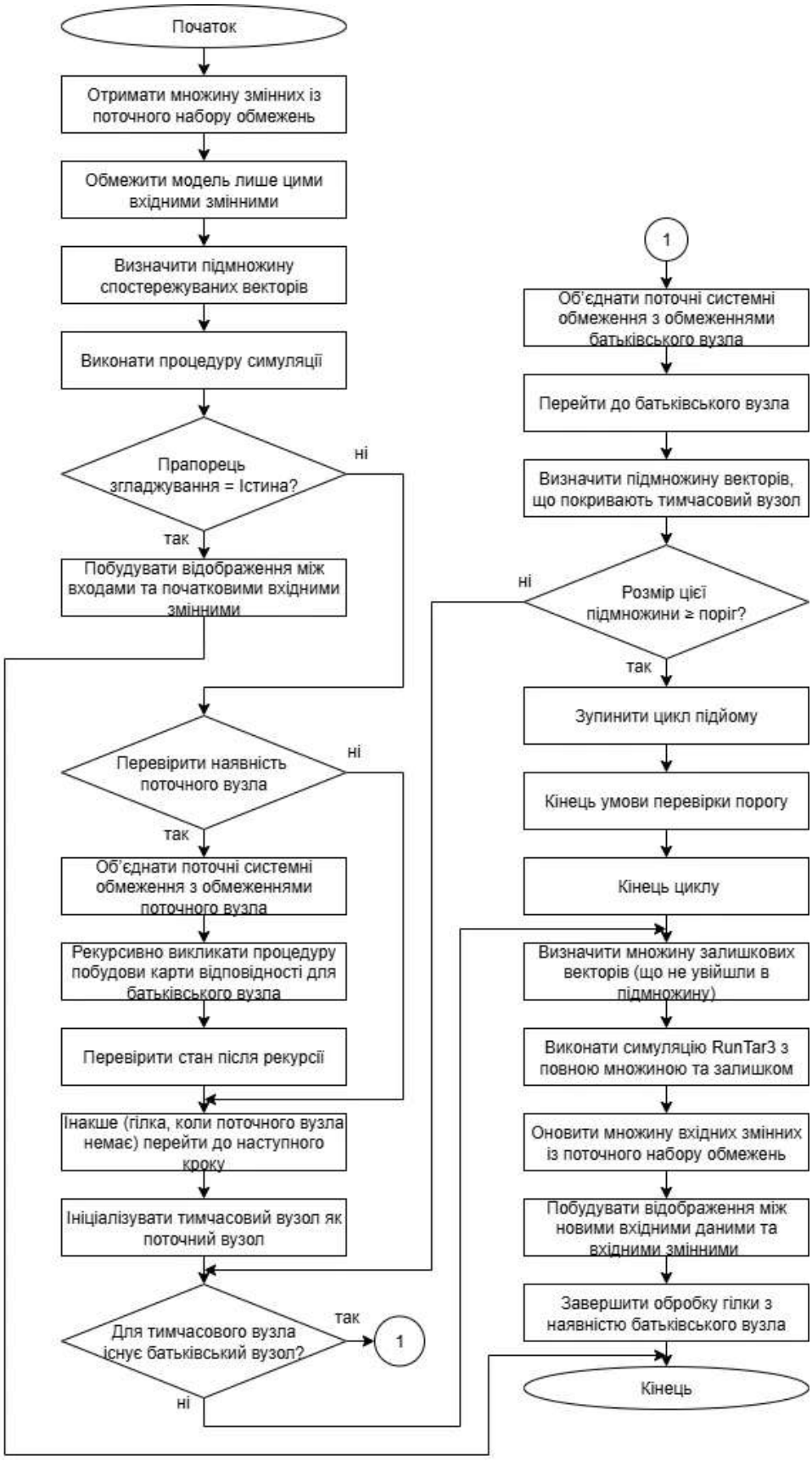

Рисунок 2.4 - Алгоритм генерації вхідних даних на основі обмежень шляху та вузлів

Крок 3. Визначити підмножину спостережуваних векторів. Це приблизно двадцять відсотків точок, що знаходяться найближче до шляхових обмежень для поточного вузла; позначити цю підмножину і віднести решту векторів до множини залишку; перейти до кроку 4.

Крок 4. Виконати процедуру симуляції (RunTar3) з поточними параметрами, щоб отримати набір згенерованих вхідних даних, діапазони результатів та прапорець згладжування; перейти до кроку 5.

Крок 5. Якщо прапорець згладжування має значення істина, перейти до кроку 6; інакше перейти до кроку 7.

Крок 6. Побудувати відображення між отриманими входами та початковими вхідними змінними за допомогою апроксимації кривої; після цього завершити обробку поточного випадку і перейти до кроку 26.

Крок 7. Інакше (якщо прапорець згладжування дорівнює хибі) перевірити наявність поточного вузла; якщо вузол існує, перейти до кроку 8, інакше перейти до кроку 12.

Крок 8. Об’єднати поточний набір системних обмежень з обмеженнями поточного вузла; після цього перейти до кроку 9.

Крок 9. Виконати рекурсивний виклик процедури обчислення карти відповідності для батьківського вузла з оновленими обмеженнями та іншими необхідними параметрами, щоб отримати додаткові вхідні дані і допоміжні функції; після повернення з рекурсії перейти до кроку 25.

Крок 10. Перевірити стан після рекурсивного виклику та перейти до кроку 25.

Крок 11. Інакше (гілка, коли поточного вузла немає) перейти до кроку 12.

Крок 12. Ініціалізувати тимчасовий покажчик вузла як поточний вузол (вважати тимчасовий вузол рівним початковому вузлу); перейти до кроку 13.

Крок 13. Поки для тимчасового вузла існує батьківський вузол, виконувати кроки 14–17; якщо батьківського вузла немає, перейти до кроку 21.

Крок 14. Об’єднати поточний набір системних обмежень з обмеженнями цього батьківського вузла.

Крок 15. Перейти до батьківського вузла: зробити тимчасовий вузол рівним його батьківському вузлу; після цього перейти до кроку 16.

Крок 16. Визначити підмножину векторів з повної множини векторів, які «покривають» або релевантні для поточного тимчасового вузла; після цього перейти до кроку 17.

Крок 17. Якщо розмір цієї підмножини векторів не менший за заздалегідь визначений поріг, перейти до кроку 18; інакше повернутися до кроку 13.

Крок 18. Зупинити цикл підйому по батьківських вузлах і перейти до кроку 21.

Крок 19. Кінець умови перевірки порогу (інформаційний рядок). Перейти до кроку 20.

Крок 20. Кінець циклу while (інформаційний рядок). Перейти до кроку 21.

Крок 21. Визначити множину залишкових векторів як ті вектори, що не увійшли до підмножини, визначеної на попередніх кроках; перейти до кроку 22.

Крок 22. Виконати процедуру симуляції (RunTar3) з повною множиною векторів, обраною підмножиною та її залишком, щоб отримати оновлений набір вхідних даних та діапазони результатів; перейти до кроку 23.

Крок 23. Оновити множину вхідних змінних, знову витягнувши їх із поточного набору обмежень; перейти до кроку 24.

Крок 24. Побудувати відображення між оновленим набором вхідних даних і вхідними змінними за допомогою апроксимації кривої; після цього перейти до кроку 25.

Крок 25. Завершити обробку гілки, що стосується наявності батьківського вузла. Перейти до кроку 26.

Крок 26. Завершити обробку випадку згладжування та завершити виконання алгоритму.

2.6 Метод виявлення вразливостей

Розроблений метод виявлення вразливостей типу переповнення буферу спрямований на комплексне охоплення простору виконання програми та автоматизоване виявлення критичних помилок доступу до пам’яті, що можуть бути експлуатовані зловмисниками. Його ключовою особливістю є інтеграція символьного виконання, навчання з підкріпленням та аналізу обмежень шляху, що дозволяє поєднати формальну перевірку умов безпеки з активним пошуком малоймовірних сценаріїв. Такий підхід забезпечує можливість виявляти не лише явні порушення меж буферів, а й приховані дефекти, пов’язані з арифметичними переповненнями (wrap-around), неправильними обчисленнями індексів та некоректною роботою покажчиків. Метод передбачає кілька взаємопов’язаних етапів: побудову проміжного представлення коду, формування дерева обмежень, символьне виконання із залученням агента навчання з підкріпленням, динамічну перевірку згенерованих тестових даних та збереження виявлених вразливих сценаріїв у базі знань.

На початковому етапі виконуваний код програми транслюється у проміжне представлення (Intermediate Representation, IR) за допомогою фреймворку динамічної інструментації Valgrind/VEX. Цей фреймворк забезпечує платформонезалежність аналізу та уніфікує всі машинні інструкції до високорівневих операцій, що значно спрощує подальше дослідження доступу до пам’яті. IR є графом інструкцій, де кожна вершина описує операції читання або запису, арифметичні дії, маніпуляції з покажчиками та інші низькорівневі операції. На основі IR будується граф потоку керування (CFG) та дерево обмежень, у вузлах якого фіксуються умови, що визначають межі буферів, індекси доступу, результати проміжних обчислень і можливі гілки виконання. Для кожної операції запису у пам’ять формуються логічні формули безпечності, що перевіряють дотримання меж буфера: початкова адреса повинна бути не менше за нижню межу, а адреса останнього байта - не більше за верхню межу виділеного інтервалу. Додатково додаються формули перевірки відсутності

wrap-ефектів, що виникають при переповненні цілочисельної арифметики під час обчислення адрес.

Після побудови дерева обмежень запускається процес символьного виконання, який досліджує всі можливі шляхи програми, використовуючи символьні змінні замість конкретних значень вхідних даних. Кожен шлях у дереві супроводжується множиною умов шляху (path constraints), що описують логічні залежності між змінними, необхідні для досягнення цього шляху. Для перевірки розв'язності умов шляху використовується зовнішній SMT-розв'язувач (наприклад, Z3), який генерує конкретні вхідні вектори, що задовольняють ці умови. У класичних системах символьного виконання цей процес стикається з проблемою експоненційного вибуху кількості шляхів, що унеможливлює повне покриття навіть для програм середньої складності. Для подолання цієї проблеми в розробленому методі застосовано агент навчання з підкріпленням, який керує порядком обходу дерева обмежень і пріоритизує найперспективніші гілки.

Агент базується на алгоритмі Q-навчання, що дозволяє оцінювати довгострокову корисність відвідування певних вузлів дерева. Кожен стан середовища визначається парою «поточний вузол дерева – набір конкретизованих/символьних змінних», а діями агента є вибір наступної гілки або конкретизація певної символьної змінної. Після виконання дії агент отримує винагороду: базову - за досягнення нового, раніше не відвіданого вузла, і значно підвищену - за виявлення потенційної вразливості, тобто порушення умов безпечного доступу до пам'яті. Завдяки такій схемі агент навчається обирати ті траєкторії, що з найбільшою ймовірністю призведуть до виявлення переповнень. Щоб пришвидшити навчання, використовується механізм Prioritized Experience Replay, який зберігає найбільш інформативні епізоди взаємодії та дозволяє повторно використовувати їх у майбутніх ітераціях навчання. Додатково застосовується Double Q-learning для зменшення переоцінки корисності дій, що покращує стабільність збіжності.

Коли агент отримує від SMT-розв'язувача конкретний набір вхідних

даних, він передається у середовище динамічного виконання, де відбувається фактичний запуск програми під наглядом інструментальної підсистеми. Під час виконання відстежуються всі звернення до пам’яті, їхні адреси та обсяги, що зіставляються з відомими межами буферів, визначеними на етапі символьного аналізу. Якщо зафіксовано хоча б одне звернення поза межами буфера, реєструється подія переповнення. Додатково здійснюється контроль наявності wrap-ефектів: аналізуються проміжні арифметичні обчислення індексів і порівнюються з максимальною розрядністю адресного простору. Якщо виявлено потенційне арифметичне переповнення, що могло призвести до викривлення адреси, воно також маркується як вразливість. Усі виявлені випадки записуються в базу знань разом із повним трасуванням виконання, умовами шляху та відповідними вхідними даними, що дозволяє відтворити помилку та спростити її усунення розробниками.

Для забезпечення масштабованості методу реалізовано низку оптимізацій. Однією з них є адаптивне обрізання шляхів: якщо символьне виконання тривалий час перебуває в одному стані без генерації нових вузлів, траєкторія примусово переривається, а агент спрямовується на альтернативні гілки. Другою оптимізацією є гібридний режим конкретно-символьного виконання (concolic execution), коли частина змінних фіксується на конкретні значення, що значно зменшує розмірність простору станів і знижує ризик вибуху шляхів. Третя оптимізація - евристична пріоритизація шляхів за щільністю операцій доступу до пам’яті: спочатку досліджуються ті ділянки коду, де ймовірність наявності переповнень є найвищою. Додатково впроваджено кешування результатів розв’язання обмежень, що дозволяє уникати повторних викликів SMT-розв’язувача для вже досліджених комбінацій умов і пришвидшує процес аналізу.

У процесі тестування метод показав здатність ефективно виявляти як стекові, так і купові переповнення буфера, включаючи складні міжпроцедурні сценарії, які залишаються непомітними для класичних статичних аналізаторів. Його використання забезпечує повне автоматизоване дослідження простору

виконання без необхідності доступу до вихідних кодів, що робить підхід придатним для аналізу закритого програмного забезпечення. Крім того, метод дає змогу формувати цільові тестові набори, які відтворюють знайдені вразливості, що важливо для подальшої верифікації виправлень і запобігання регресіям. У підсумку, розроблений метод поєднує формальну строгість символьного аналізу з гнучкістю навчання з підкріпленням, що дозволяє виявляти критичні вразливості з високою точністю, скорочуючи при цьому обчислювальні витрати та потребу в ручному аудиті.

Таким чином, запропонований метод виявлення вразливостей типу переповнення буферу забезпечує повноцінний цикл автоматизованої діагностики - від побудови формальних моделей до динамічного підтвердження знайдених дефектів - і може бути використаний як основа для створення промислових систем тестування безпеки програмного забезпечення. Його застосування дозволяє значно підвищити надійність програм, зменшити ризики експлуатації критичних помилок зловмисниками та мінімізувати витрати на ручний аналіз, що є особливо актуальним в умовах зростання складності та масштабів сучасного програмного забезпечення.

2.7 Висновки до розділу

У другому розділі сформульована математична модель переповнення буферу дала можливість абстрагуватися від конкретних технічних деталей реалізації програмного коду та представити проблему в універсальному вигляді, придатному для формального аналізу. Побудова моделі пам’яті у вигляді сукупності адрес та значень, що відображають стан кожного байта, створила умови для подальшої формалізації операцій запису й читання. Введення інтервалів, які відповідають виділеним областям пам’яті, дозволило однозначно визначати границі буферів і, відповідно, встановлювати критерії їхнього переповнення.

Побудова дерева обмежень дозволила представити всі можливі варіанти виконання програми у вигляді формальної структури, де кожна вершина відповідає певному стану системи, а ребра відображають умови переходу між станами. Завдяки цьому стає можливим системне відстеження усіх шляхів виконання, у тому числі тих, що безпосередньо ведуть до виникнення переповнення буфера. Дерево обмежень не лише задає логіку переходів, але й формує множину умов, виконання яких є необхідним для реалізації певного сценарію. У такий спосіб традиційна проблема непередбачуваності поведінки програмної системи переводиться у площину формальних методів, що відкриває можливість використання інструментарію математичної логіки, комбінаторики та оптимізації. Отже, дерево обмежень виконує роль своєрідної карти простору виконання програми, у якій можна виокремити найбільш небезпечні маршрути та сфокусувати зусилля на їхньому детальному аналізі.

Розроблений алгоритм навчання поведінки програм шляхом симулятивного покриття забезпечує можливість поєднання формального аналізу з адаптивними стратегіями дослідження. Алгоритм відтворює принципи навчання з підкріпленням, де агент отримує винагороди за відкриття нових шляхів виконання. Завдяки такій постановці завдання, процес дослідження програмного коду набуває спрямованого характеру: замість хаотичного перебору можливих сценаріїв відбувається пріоритизація тих шляхів, які з більшою ймовірністю приведуть до виявлення критичних вразливостей. Симулятивне покриття дозволяє поступово розширювати межі досліджуваного простору, одночасно уникаючи «вибуху шляхів», що є класичною проблемою символьного виконання. Цей підхід суттєво підвищує масштабованість методу, роблячи його придатним для застосування в аналізі промислових програмних систем, де кількість можливих сценаріїв може сягати мільйонів.

Алгоритм генерації вхідних даних на основі обмежень шляху та вузлів доповнив методологію механізмом практичного тестування програмного забезпечення. Використання систем обмежень дозволило формувати вхідні набори, які не є випадковими, а строго відповідають умовам, що активують ті чи

інші гілки виконання Такий підхід усуває надлишковість, характерну для класичного фаззингу, і водночас долає обмеження традиційного символьного виконання, де значна частина шляхів залишається недосяжною через обчислювальні обмеження. Генерація цілеспрямованих вхідних даних стає інструментом, що поєднує формальність математичної моделі з практичною придатністю тестових сценаріїв, забезпечуючи баланс між строгістю аналізу та ефективністю його виконання.

Запропонований метод виявлення вразливостей формує повноцінний цикл діагностики, який охоплює етапи моделювання, формалізації умов, побудови системи обмежень, симулятивного навчання та генерації тестових даних. Така комплексність забезпечує високу ймовірність виявлення переповнення буферу, які можуть залишитися поза увагою традиційних підходів. На відміну від динамічного фаззингу, розроблений метод керується формально визначеними обмеженнями, що дозволяє суттєво підвищити продуктивність і скоротити час до виявлення вразливостей.

3 ТЕСТУВАННЯ ТА ОЦІНКА ДОСТОВІРНОСТІ

3.1 Налаштування тестового середовища

Перед початком проведення експериментів необхідно було забезпечити дотримання певних вимог до середовища тестування та конфігурації інструментів. Оскільки розроблений інструмент для виявлення вразливостей типу переповнення буферу не є сумісним з усіма поширеними операційними системами, було прийнято рішення розгорнути окреме ізольоване середовище. Основна операційна система дослідницької машини - Windows 11, проте для забезпечення стабільної роботи інструментів використовувалася віртуальна машина VirtualBox, у якій було встановлено дистрибутив Ubuntu 24.04.03 LTS. Така організація дозволила поєднати ізольованість віртуального середовища з продуктивністю сучасного апаратного забезпечення, яке включало процесор Intel Core i7 із багатоядерною архітектурою, 32 ГБ оперативної пам’яті та твердотільний накопичувач NVMe. Віртуальній машині було виділено вісім логічних ядер і 16 ГБ пам’яті, чого виявилося достатньо для ефективного функціонування інструментів аналізу коду.

На початковому етапі після інсталяції системи було налаштовано всі необхідні залежності, серед яких Python 3 та фреймворк angr. Інформація про потрібні пакети зберігалася у файлі requirements.txt. Щоб уникнути конфліктів між глобальними бібліотеками системи та компонентами проєкту, було створено окреме віртуальне середовище Python, у якому клонували GitHub-репозиторій. Використання віртуального середовища підвищило стабільність і відтворюваність експериментів, оскільки дозволяло працювати з зафіксованими версіями бібліотек. Установка залежностей виконувалася через менеджер пакетів pip.

Для підготовки середовища необхідно було також надати виконувані права окремим скриптам, зокрема executable.sh та benchmarks_running.py, використавши команду chmod +x. Перший із них відповідав за компіляцію кожного тестового випадку у виконуваний файл, тоді як другий забезпечував

запуск тестових програм і збирання результатів. Сам репозиторій містив набір попередньо підготовлених тестових програм мовою C, які були використані у першому експерименті. Для більш комплексної оцінки ефективності методу до експериментів було додано додаткові тестові програми, запозичені з ресурсів Національного інституту стандартів і технологій (NIST). Вони включали як уразливі (позначені як «bad»), так і безпечні («good») варіанти функцій. У деяких випадках перед компіляцією потрібно було вносити зміни у вихідні коди, зокрема видаляти директиви препроцесора #ifdef INCLUDEMAIN та відповідні #endif, щоб уникнути конфліктів під час складання. Крім того, скрипт executable.sh було модифіковано для можливості коректної компіляції багатофайлових програм, що містили функції у різних вихідних файлах із суфіксами «a», «b», «c» тощо.

На прикладі тестової програми memmove можна продемонструвати структуру таких експериментальних даних. Одна з функцій була позначена як безпечна і не містила вразливості, тоді як інша - як уразлива, що дозволяло відтворити сценарії переповнення буферу. Це свідчить про те, що тестові набори складалися зі змішаних одиниць, де «bad» реалізації містили уразливості, а «good» демонстрували способи їх уникнення.

Після завершення етапу попередньої обробки та компіляції тестових програм за допомогою gcc було запущено скрипт benchmarks_running.py. Сам процес експерименту передбачав послідовний цикл: підготовка виконуваних файлів, запуск у тестовому модулі, інструментування за допомогою засобів аналізу (Valgrind, angr, AFL) і подальше збирання результатів. На цьому етапі важливу роль відігравала структура папок проєкту. Вихідні файли зберігалися у каталозі tests, виконувані програми після компіляції - у bin, журнали запусків - у logs, а результати у форматі CSV та JSON - у директорії results. Допоміжні скрипти розташовувалися у каталозі scripts, тоді як у кореневій папці знаходилися requirements.txt, Dockerfile та README.md. Така організація не лише спрощувала навігацію, але й забезпечувала повторюваність експериментів та можливість перенесення середовища на інші машини без втрати даних.

Загальна логіка експериментів нагадувала блокову схему. Вихідні С-програми перетворювалися на виконувані файли, які запускалися у спеціальному тестовому модулі. Далі вони інструментувалися за допомогою різних інструментів аналізу, результати роботи яких спрямовувалися до модуля Q-навчання. Останній коригував стратегію пошуку, формуючи зворотний зв’язок для подальших запусків. Завдяки цьому створювалася замкнена система, де вихідними даними були тестові програми, а кінцевим результатом - статистика щодо виявлених уразливостей, часу виконання та точності спрацювань.

Не менш важливим елементом була організація логування. Усі повідомлення, що виникали під час запусків, записувалися у журнали, що дозволяло відтворити історію виконання й у разі потреби знайти причини відхилень чи збоїв. Основні метрики, серед яких кількість знайдених уразливостей, час виконання, обсяг згенерованих вхідних даних і показники покриття коду, зберігалися у форматах CSV та JSON. Кожен рядок таблиці відповідав окремому тестовому запуску та містив ключові параметри: ідентифікатор тесту, інструмент аналізу, тип уразливості, часові характеристики та примітки. Це дозволяло проводити як загальний, так і порівняльний аналіз ефективності різних підходів.

Таким чином, налаштоване середовище поєднало апаратну продуктивність сучасного комп’ютера, ізольованість віртуальної машини та гнучкість інструментів аналізу. Структурованість проєкту, чітка схема побудови експериментів і системне логування результатів забезпечили надійні умови для проведення повноцінного тестування й оцінювання запропонованого методу виявлення вразливостей переповнення буферу.

3.2 Тестові програми

Для проведення експериментів 1 та 2 використовувалися різні набори тестових програм для оцінки продуктивності. У першому експерименті

застосовувалися 90 оригінальних "pre-loaded" тестових програм, що покривали чотири операції вразливих функцій: strcpy, strcat, memcpy та memmove. У другому експерименті використовувалися 100 нових тестових програм, обраних із бази NIST. Ці програми включали три операції вразливих функцій: memmove, strncpy та wcscpy. Програми NIST включали як "bad" функції, що були вразливими до переповнення буфера, так і "good" функції, у яких вразливості були усунені. Для другого експерименту обрані програми, що також покривають CWE-122, з метою перевірки продуктивності інструмента на різних типах операцій. Вибір NIST обумовлений простотою використання, наявністю метаданих та високою якістю тестових кейсів. Програми були відібрані за трьома критеріями: наявність коду на C, наявність вразливих та захищених функцій щодо CWE-122, та високий рейтинг якості програми у NIST. Метою вибору конкретних операцій (memmove, strncpy, wcscpy) було поєднання порівняння результатів з оригінальним набором та оцінка нових типів функцій, що раніше не тестувалися.

Приклади "good" та "bad" функцій наведено на рисунках 3.1 та 3.2.

```
/* goodG2B uses the GoodSource with the BadSink */
static void goodG2B()
{
    int64_t * data;
    data = NULL;
    data = goodG2BSource(data);
    {
        int64_t source[100] = {0}; /* fill with 0's */
        /* POTENTIAL FLAW:
           Possible buffer overflow if data < 100 */
        memmove(data, source, 100*sizeof(int64_t));
        printLongLongLine(data[0]);
        free(data);
    }
}
```

Рисунок 3.1 - Приклад "хорошої" функції з тестової програми memmove від NIST

```
void CWE122_Heap_Based_Buffer_Overflow_memmove_bad()
{
    int64_t * data;
    data = NULL;
    data = badSource(data);
    {
        int64_t source[100] = {0}; /* fill with 0's */
        /* POTENTIAL FLAW:
           Possible buffer overflow if data < 100 */
        memmove(data, source, 100*sizeof(int64_t));
        printLongLongLine(data[0]);
        free(data);
    }
}
```

Рисунок 3.2 - Приклад "поганої" функції з тестової програми memmove від NIST

У рамках другого експерименту одним із типів тестових програм було використання операцій із функцією memmove, що потенційно містять вразливості. Призначення цієї функції полягає у копіюванні n байтів з об'єкта, на який посилається покажчик s2, до об'єкта, визначеного покажчиком s1. Оскільки можливе перекриття вихідної та цільової областей пам'яті, копіювання здійснюється так, ніби дані спочатку переносяться у проміжний буфер, а вже потім записуються у кінцеве розташування. Поверненим значенням функції є покажчик на s1.

Потенційна вразливість виникає у випадках, коли розмір буфера призначення (s1) є меншим за обсяг вихідного рядка (s2). У такій ситуації відбувається переповнення буфера, що призводить до невизначеної поведінки програми. Формальне визначення функції у мові С подане на рис. 3.3.

```
#include <string.h>

void *memmove(void *s1, const void *s2, size_t n);
```

Рисунок 3.3 - Формальне визначення функції memmove

Другим типом тестових програм у межах експерименту стали програми, що містять виклики функції strncpy, яка здійснює копіювання рядка, розташованого за адресою s2, до буфера, визначеного s1, при цьому максимальна кількість символів для копіювання не перевищує n. Якщо у вихідному рядку присутній символ завершення (\0), то він також переноситься, проте не копіюються символи після нього. Поверненим значенням є покажчик на s1.

Вразливість при використанні strncpy виникає у разі перекриття пам'яті між об'єктами s1 та s2. Така ситуація може спричинити некоректне функціонування програми або її аварійне завершення. Формальне визначення функції подане у стандарті ISO/IEC і представлено на рис. 3.4.

```
#include <string.h>

char *strncpy(char *restrict s1,
              const char *restrict s2,
              size_t n);
```

Рисунок 3.4 - Формальне визначення функції strncpy

Крім перекриття, небезпека може виникати й тоді, коли буфер призначення є недостатнім для розміщення повного рядка з s2. Це також створює передумови до виникнення переповнення буферу.

Останнім видом тестових програм, використаних у другому експерименті, стали програми з викликами функції wcscpy, яка призначена для копіювання широких рядків. Вона переносить послідовність широких символів разом із символом завершення (L'\0') із області пам'яті, на яку вказує s2, у буфер, визначений s1. Поверненим значенням є покажчик на s1.

Вразливі ситуації можуть виникати у випадку перекриття об'єктів пам'яті, що, як і у попередніх функціях, призводить до невизначеної поведінки системи. Формальне оголошення функції показано на рис. 3.5.

```c
#include <wchar.h>

wchar_t *wcscpy(wchar_t *restrict s1,
                const wchar_t *restrict s2);
```

Рисунок 3.5 - Формальне оголошення функції wcscpy

Таким чином, усі три розглянуті функції (memmove, strncpy, wcscpy) можуть стати джерелом критичних вразливостей у випадках некоректного використання, зокрема при роботі з неконтрольованими розмірами буферів або перекриттям пам’яті. Це робить їх важливим об’єктом для досліджень у контексті виявлення переповнення буфера у системах аналізу безпеки.

Для оцінювання вразливостей було використано 33 тестові програми, що містили переповнення буфера у викликах функції memmove. Для аналізу переповнень буфера у викликах strncpy було використано 32 тестові програми. Окремий експеримент проведено з використанням 35 програм, що містили вразливі виклики функції wcscpy.

3.3 Проведення та результати першого експерименту

Для першого експерименту було використано попередньо підготовлені тестові програми з репозиторію, а результати оцінювалися за метриками продуктивності, описаними в параграфі 3.5. Експеримент дозволив оцінити точність класифікації тестових одиниць, швидкість аналізу та ефективність генерації вхідних даних, що є важливим для автоматизованого тестування. Першим етапом дослідження стала перевірка працездатності інструменту для виявлення вразливостей типу переповнення буфера у динамічній пам’яті під час виконання скомпільованих програм мовою C. Для цього було використано набір тестових програм, що містив 90 прикладів.

Набір тестових програм складався з чотирьох груп, які відрізнялися типом функцій, що потенційно створювали переповнення буфера у динамічній пам'яті. Перша група охоплювала 30 програм із викликами функції memcpy, друга група - 30 програм із викликами memmove, третя група - 15 програм із викликами strcat, а четверта група - 15 програм із викликами strcpy. Приклад вихідних даних одного з тестів наведено на рисунку 3.6, де можна спостерігати, як кожна тестова одиниця класифікується відповідно до своєї вразливості.

Результати експерименту, представлені в таблицях 3.1-3.5, підтвердили високу ефективність інструменту. Для програм із викликами memcpy (табл. 3.1) було виявлено 33 вразливих тестові одиниці, тоді як 45 одиниць було коректно віднесено до безпечних, без жодного хибного результату. У випадку функції memmove (табл. 3.2) система зафіксувала 35 позитивних випадків виявлення вразливостей та 46 негативних результатів, які також були визначені правильно.

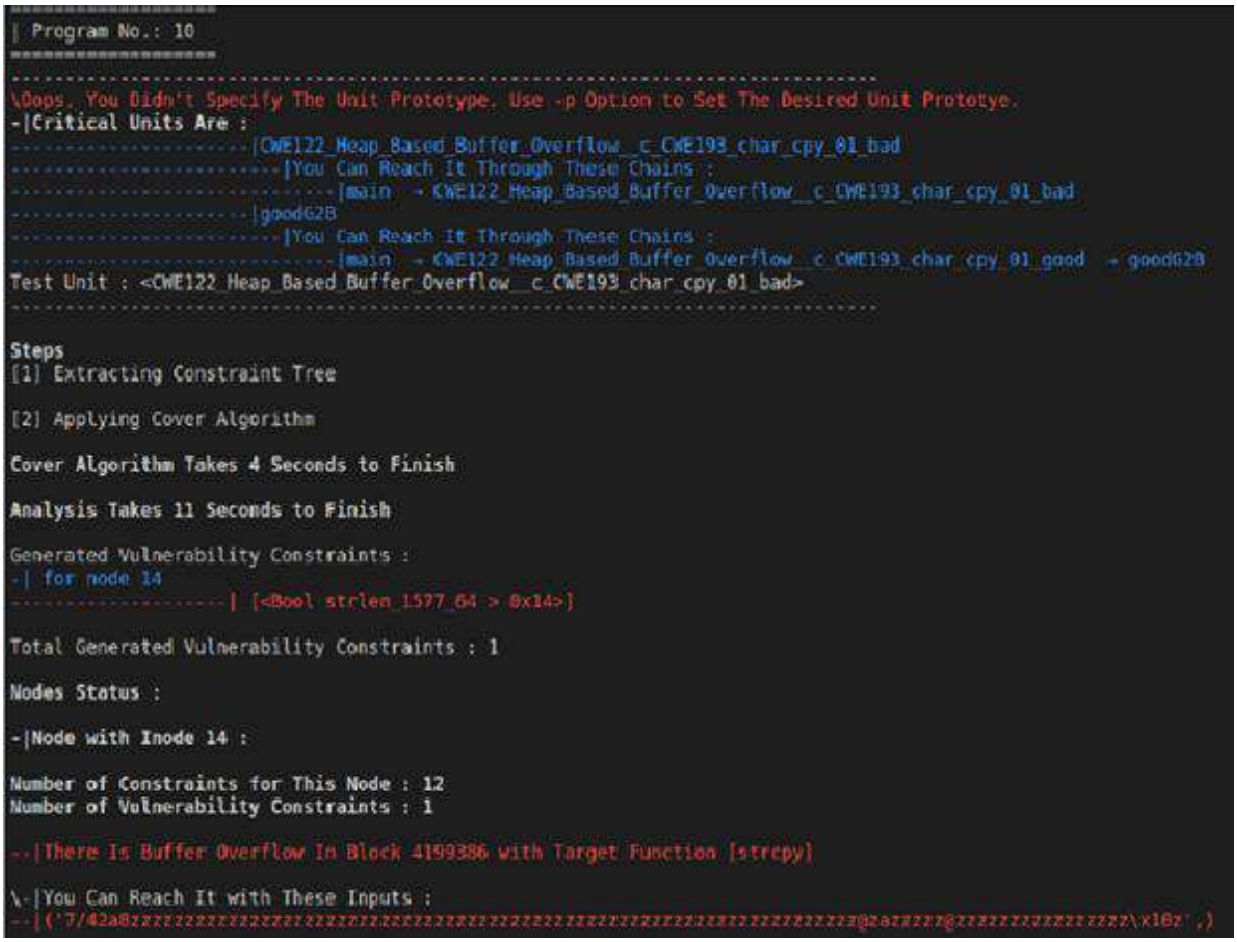

Рисунок 3.6 - Зразок виводу аналізу тестового блоку

Таблиця 3.1 - Результат використання тестових програм memcpy

		Фактичне значення	
		P	N
Прогнозоване значення	P	33	0
	N	0	45

Таблиця 3.2 - Результат використання тестових програм memmove

		Фактичне значення	
		P	N
Прогнозоване значення	P	35	0
	N	0	46

Аналогічні результати отримано для програм із викликами функції strcat (табл. 3.3), де інструмент ідентифікував 21 випадок наявності вразливостей і 26 випадків їх відсутності. Для функції strcpy (табл. 3.4) було зафіксовано 14 істинно позитивних результатів та 23 істинно негативні, без жодного хибного визначення.

Таблиця 3.3 - Результат використання тестових програм strcat

		Фактичне значення	
		P	N
Прогнозоване значення	P	21	0
	N	0	26

Таблиця 3.4 - Результат використання тестових програм strcpy

		Фактичне значення	
		P	N
Прогнозоване значення	P	14	0
	N	0	23

У сукупності тестування (табл. 3.5) 90 програм дало 103 істинно позитивні та 140 істинно негативних випадків, що підтвердило абсолютну коректність інструменту: хибнопозитивні та хибнонегативні результати були відсутні.

Таблиця 3.5 - Результат використання тестових програм разом

		Фактичне значення	
		P	N
Прогнозоване значення	P	103	0
	N	0	140

Це забезпечило досягнення ідеальних значень основних метрик - точність, повнота, прецизійність та F1-міра дорівнювали 100%. Такий результат пояснюється тим, що алгоритм не допустив жодного відхилення від очікуваних результатів під час аналізу тестових програм.

Додатково було проаналізовано продуктивність інструменту. У таблиці 3.6 подано загальний та середній час виконання програм, що містили виклики вразливих функцій, тоді як на рисунках 3.7-3.8 відображено кількість згенерованих вхідних даних для кожного типу функцій. Ці показники дозволили оцінити не лише якість виявлення вразливостей, а й ефективність роботи інструменту з точки зору витрат обчислювальних ресурсів.

Таблиця 3.6 - Час виконання тестів для кожної категорії тестових програм, що використовують вразливі виклики функцій

Функція	Кількість програм	Загальний час, с	Середній час на програму, с
memcpy	30	636	21
memmove	30	678	23
strcat	15	402	27
strcpy	15	436	29

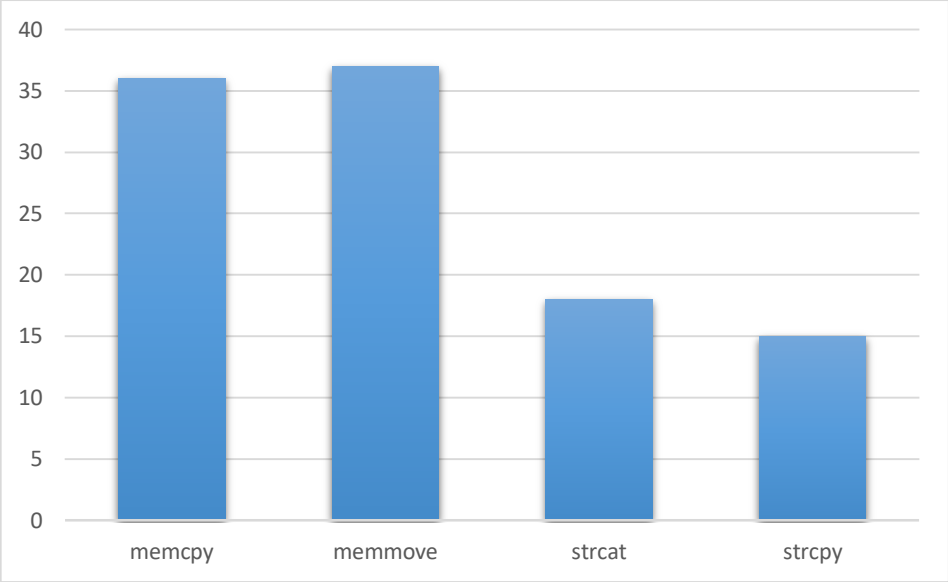

Рисунок 3.7 - Кількість створених вхідних даних програми

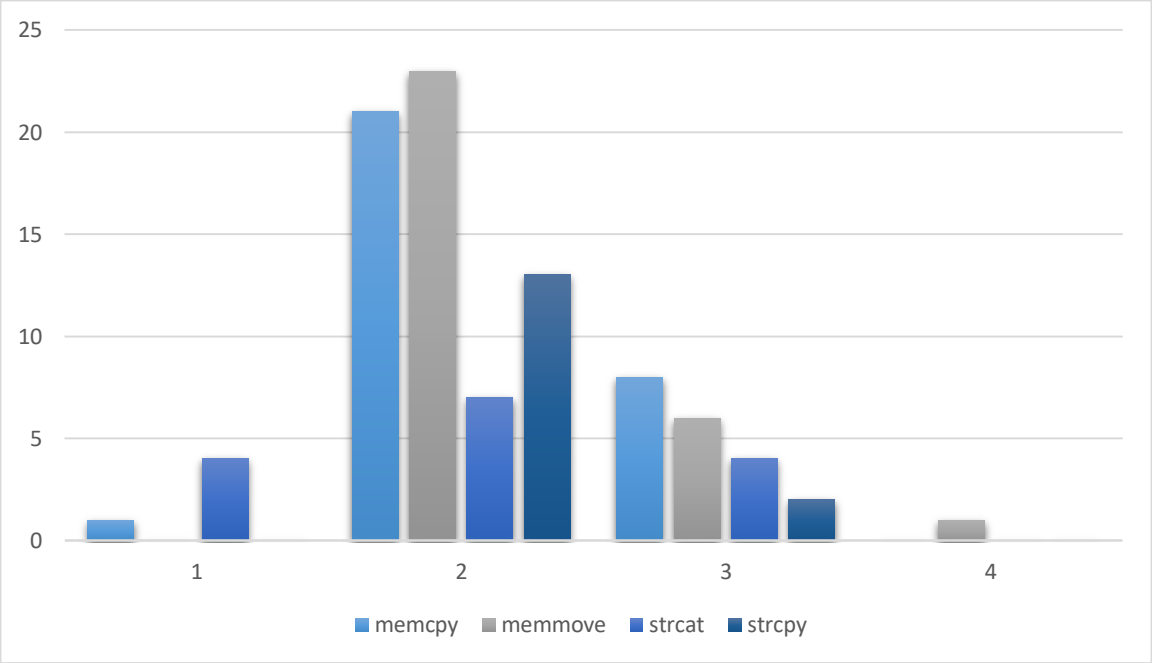

Рисунок 3.8 - Розподіл та кількість вхідних даних програми для всіх типів програм у вихідному тестовому наборі

3.4 Проведення та результати другого експерименту

Другий експеримент проведено з іншим набором даних, без та з інтеграцію Q-навчання до пропонованого методу. Методологія проведення другого

експерименту була подібною до першого експерименту. Більшість компонентів тестового середовища вже були підготовлені на етапі попередніх умов, включно з встановленням необхідних програм, підготовкою нового набору тестових програм з бази NIST та їхньою попередньою обробкою. Другий етап дослідження був спрямований на перевірку ефективності з використанням іншого набору тестових програм, сформованого на основі бази даних NIST. Набір містив 100 програм, серед яких 33 реалізовували виклики функції memmove, 32 - функції strncpy, а 35 - функції wcscpy. Повний перелік тестових програм наведено у Додатку А.

Оцінка результатів здійснювалося за допомогою матриць плутанини та ключових метрик продуктивності, серед яких точність, прецизійність, повнота, F1-міра, середній час виконання аналізу всіх програм та кількість згенерованих вхідних даних.

Узагальнені результати роботи з новим тестовим набором показали (таблиці 3.7-3.10), що зафіксовано 58 істинно позитивних і 156 істинно негативних одиниць.

Таблиця 3.7 - Результат використання тестових програм memmove

		Фактичне значення			
		без Q-навчання		з Q-навчанням	
		P	N	P	N
Прогнозоване значення	P	31	0	26	0
	N	11	61	14	57

Таблиця 3.8 - Результат використання тестових програм strcpy

		Фактичне значення			
		без Q-навчання		з Q-навчанням	
		P	N	P	N
Прогнозоване значення	P	15	0	17	0
	N	19	43	16	46

Таблиця 3.9 - Результат використання тестових програм wcscpy

		Фактичне значення			
		без Q-навчання		з Q-навчанням	
		P	N	P	N
Прогнозоване значення	P	13	0	15	0
	N	27	55	21	53

Таблиця 3.10 - Результат використання тестових програм разом

		Фактичне значення			
		без Q-навчання		з Q-навчанням	
		P	N	P	N
Прогнозоване значення	P	59	0	58	0
	N	57	159	51	156

Отримані результати свідчать про те, що інтеграція алгоритму Q-навчання дала змогу підвищити точність виявлення вразливостей у деяких групах програм, особливо під час роботи з функціями strncpy та wcscpy. Водночас для функції memmove із застосуванням Q-навчання продемонстрував дещо гірші результати порівняно з першою версією, що може бути пов’язано з особливостями вибору стратегії дій у процесі навчання з підкріпленням. Таким чином, експеримент підтвердив доцільність використання Q-навчання для адаптивного підвищення ефективності аналізу, проте окреслив потребу у додатковій оптимізації параметрів алгоритму залежно від типу цільових функцій.

Використання однакових тестових програм для обох версій інструмента гарантувало об’єктивність порівняння. У результаті за всіма трьома групами функцій (memmove, strncpy, wcscpy) спостерігалося незначне покращення характеристик модифікованого засобу. Ймовірно, причина такого незначного ефекту полягає в обмеженій вибірці, яка включала лише 100 програм. Вплив алгоритму Q-навчання проявляється поступово через оновлення таблиці значень,

що більш помітно при великій кількості ітерацій. У майбутніх дослідженнях розширення обсягу тестових програм могло б продемонструвати більш виразну різницю. Таблиця 3.11 подає результати часу виконання тестів для трьох категорій тестових програм (функції memmove, strncpy, wcscpy) у порівнянні без Q-навчання та з Q-навчанням. Для memmove (33 програми) загальний час виконання збільшився з 863 с до 997 с, тобто середній час на одну програму виріс з 26 с до 30 с. Для strncpy (32 програми) загальний час зріс з 805 с до 923 с, середній час - з 25 с до 29 с. Для wcscpy (35 програм) спостерігається зростання загального часу з 780 с до 903 с, а середній час - з 22 с до 26 с. Таким чином, у всіх трьох категоріях після інтеграції Q-навчання середній час обробки однієї тестової програми збільшився приблизно на 4 с.

Таблиця 3.11 - Час виконання тестів для кожної категорії тестових програм

Функція	Кількість програм	Без Q-навчання		З Q-навчанням	
		загальний час, с	середній час на програму, с	загальний час, с	середній час на програму, с
memmove	33	863	26	997	30
strncpy	32	805	25	923	29
wcscpy	35	780	22	903	26

Важливим результатом стало збільшення кількості згенерованих тестових входів для експлуатації вразливостей. Це пояснюється тим, що механізм Q-навчання орієнтований на дослідження простору можливих станів і використовує кумулятивну винагороду як орієнтир для пошуку нових вхідних даних. Таким чином, зростання кількості вхідних програмних сценаріїв є прямим наслідком пріоритету дослідження й покриття, закладеного в Q-навчання.

Водночас було зафіксовано збільшення часу виконання для всіх груп тестів. Найбільша різниця у сумарному часі (134 с) спостерігалася для програм з викликами wcscpy. У середньому тривалість виконання тестів зросла на 15–16%, що безпосередньо пов’язано з додатковими обчисленнями під час роботи Q-

навчання. Таблиця 3.12 узагальнює абсолютне та відносне збільшення загального часу при застосуванні Q-навчання. Абсолютне збільшення становить +134 с (≈16%) для memmove, +118 с (≈15%) для strncpy та +123 с (≈16%) для wcscpy. Відповідно, відносне подовження часу роботи системи при використанні Q-навчання лежить у межах приблизно 15–16%.

Таблиця 3.12 - Збільшення часу виконання з Q-навчанням

Функція	Кількість програм	Збільшення часу, с	Збільшення часу, %
memmove	33	134	16
strncpy	32	118	15
wcscpy	35	123	16

Висновки з першого експерименту спричинили потребу оцінити ефективність поєднання навчання з підкріпленням із засобом символьного виконання на рівні окремих програмних одиниць. У процесі дослідження вдалося інтегрувати алгоритм Q-навчання до інструмента, що дало змогу проаналізувати його вплив у другому експерименті. Рисунок 3.9 наочно ілюструє порівняння кількості згенерованих вхідних даних для експлуатації вразливостей у двох режимах: без Q-навчання і з Q-навчанням (сірий стовпець).

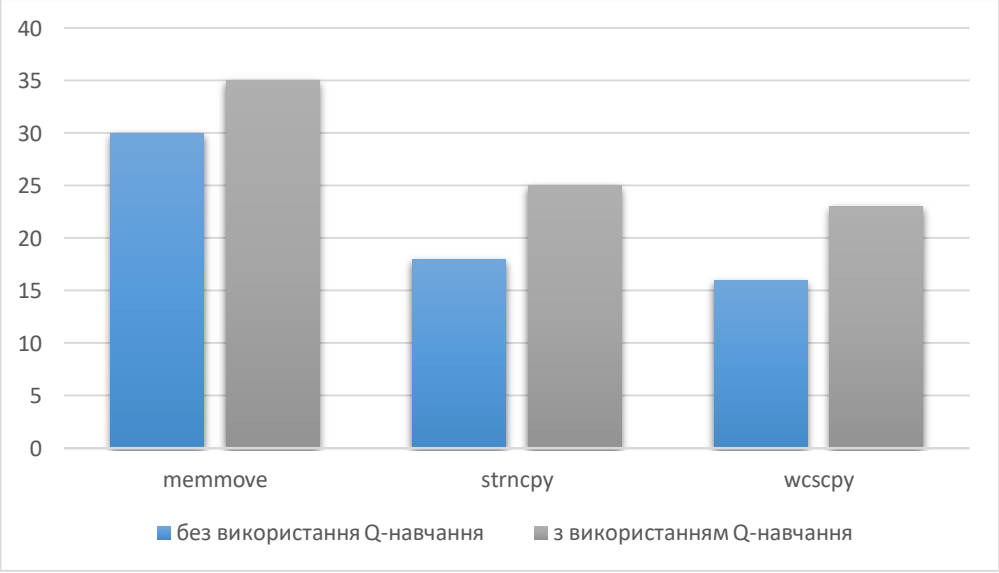

Рисунок 3.9 - Порівняння загальної кількості створених програмних вхідних даних для експлуатації вразливостей

Для всіх трьох функцій спостерігається помітне збільшення числа створених тестових вхідних векторів у режимі з Q-навчанням: найбільший приріст зафіксовано для memmove, де кількість вхідних даних збільшилася помітно (з ≈30 до ≈35 на наведеному графіку), для strncpy і wcscpy також видно відчутне зростання (приблизно з 18→25 та з 16→22 відповідно). Отже, застосування Q-навчання призводить до зростання кількості генерованих тестових комбінацій, що підвищує ймовірність покриття критичних шляхів виконання та виявлення вразливостей.

3.5 Оцінювання достовірності

Оцінювання достовірності результатів виявлення вразливостей у програмному забезпеченні є ключовим етапом перевірки ефективності запропонованого методу. Зокрема, при аналізі таких типових помилок, як переповнення буферу, важливо не лише фіксувати факт спрацювання інструменту, а й визначати, наскільки точно та повно він відображає реальний стан програмного коду. Для цього застосовуються як кількісні показники, так і структуровані методи представлення результатів, серед яких важливе місце займають матриці плутанини (confusion matrices). Вони дозволяють класифікувати результати роботи системи за чотирма основними категоріями: істинно позитивні (TP), істинно негативні (TN), хибно позитивні (FP) та хибно негативні (FN). Такий підхід дає змогу візуалізувати співвідношення між правильними та помилковими класифікаціями та є основою для обчислення узагальнених метрик якості.

Серед класичних метрик оцінювання ефективності першочергове значення мають точність, достовірність, повнота та F1-міра. Точність (Accuracy) визначається як відношення кількості всіх правильно класифікованих одиниць (TP і TN) до їх загальної кількості:

$$Accuracy = \frac{TP+TN}{TP+FP+FN+TN} \quad (3.1)$$

Він характеризує загальну правильність класифікацій, проте у випадках суттєвої нерівномірності розподілу класів може не повністю відображати якість системи.

Достовірність (Precision) показує частку правильно виявлених позитивних результатів серед усіх класифікацій, віднесених системою до позитивного класу:

$$Precision = \frac{TP}{TP+FP} \quad (3.2)$$

Високе значення Precision означає, що інструмент не схильний помилково позначати безпечні ділянки коду як уразливі, що є важливим для практичної роботи, оскільки зменшує кількість «хибних тривог».

Повнота (Recall) відображає здатність системи знаходити всі реальні вразливості у тестованих програмах. Вона обчислюється як відношення істинно позитивних результатів до суми істинно позитивних і хибно негативних класифікацій:

$$Recall = \frac{TP}{TP+FN} \quad (3.3)$$

Даний параметр є важливим у сфері безпеки, оскільки низьке значення Recall свідчить про пропущені уразливості, які залишають програму потенційно небезпечною.

Для досягнення збалансованої оцінки застосовується F1-міра, яка є гармонійним середнім між Precision та Recall:

$$F\ score = \frac{Recall+Precision}{2} \quad (3.4)$$

Крім класичних метрик, для повнішої оцінки системи важливо також враховувати час виконання інструменту, що вимірюється як різниця між кінцевим та початковим часом аналізу тестових програм. Це дозволяє оцінити продуктивність системи у практичних умовах та порівняти ефективність різних версій інструментів, наприклад класичної версії та модифікованої, що використовує Q-навчання для оптимізації процесу пошуку вразливостей.

Додатково оцінюється кількість можливих програмних введень, які генерує інструмент для кожної категорії тестових програм. Цей показник характеризує потенційну легкість експлуатації вразливості та ефективність її усунення. Висока кількість можливих введень свідчить про складність повного закриття вразливості та необхідність комплексного тестування.

Застосування цих метрик у комплексі дозволяє отримати всебічну оцінку продуктивності системи виявлення вразливостей, визначити баланс між точністю та повнотою класифікації та забезпечити об’єктивне порівняння різних методів, включно з інтеграцією алгоритмів машинного навчання та Q-навчання для підвищення ефективності аналізу. Таке комплексне оцінювання є важливим для досліджень у сфері кібербезпеки, де важлива не лише точність класифікації, а й здатність системи адаптуватися до нових або змінних умов середовища.

Згідно з результатами першого експерименту, вдалося ідентифікувати 31 істинно-позитивний результат, 61 істинно-негативний, зафіксовано 11 хибно-негативних та відсутні хибно-позитивні спрацювання. При другому експерименті виявлено 26 істинно-позитивних, 57 істинно-негативних та 14 хибно-негативних результатів, також без хибно-позитивних випадків. Точність при першому експерименті склала 89%, тоді як при другому - 86%. В обох випадках показник точності досягнув 100%, оскільки відсутні хибно-позитивні результати. Значення повноти становило відповідно 74% та 65%, а F1-метрика - 85% та 79%. Виявлено, що понад половину протестованих одиниць (61 із 103) класифіковано як істинно-негативні, тобто інструмент правильно ідентифікував відсутність вразливостей. Водночас 31 випадок був ідентифікований як вразливий, що демонструє ефективність базової реалізації. Загальний час

виконання для 33 програм з вразливими викликами memmove становив 863 с для першого експерименту та 997 с для другого, що відповідає середньому часу 26 і 30 секунд на програму. Отже, версія з Q-навчанням потребувала приблизно на 16% більше часу, що пов'язано з додатковими обчислювальними витратами Q-навчання. Оригінальна реалізація згенерувала два тестові вхідні дані для експлуатації вразливості у чотирьох випадках, тоді як у двох програмах відповідні входи відсутні. Модифікована версія продемонструвала подібні результати, однак із більшим загальним числом згенерованих тестових входів - 35 замість 30. Це зумовлено тим, що Q-навчання сформувало більше потенційних вхідних даних, хоча цей приріст не мав суттєвого впливу на показники точності, повноти чи F1-метрики.

При першому експерименті виявлено 15 істинно-позитивних випадків, 43 істинно-негативних, а також 19 хибно-негативних результатів, не зафіксувавши жодного хибно-позитивного. При другому експерименті зафіксовано 17 істинно-позитивних, 46 істинно-негативних і 16 хибно-негативних результатів. Загальна точність у цьому випадку становила 75% для першого варіанту та 80% для другого. Обидва експерименти показали 100% точність класифікації без хибних позитивних результатів, але значення повноти було нижчим: 44% при першому варіанті та 52% для другого варіанту. F1-метрика досягла 61% та 68% відповідно. Час виконання для 32 програм становив 805 с для першого експерименту та 923 с для другого, що відповідає середньому часу 25 і 29 секунд на програму.

Результати першого експерименту включали 13 істинно-позитивних, 55 істинно-негативних та 27 хибно-негативних випадків. Другий експеримент виявив 15 істинно-позитивних, 53 істинно-негативних і 21 хибно-негативний результат. Точність у цьому випадку була найнижчою серед усіх груп тестів: 72% для першого варіанту та 76% для другого. Показники повноти склали 33% і 42% відповідно, а F1-метрика - 49% та 59%. Це свідчить про складність виявлення вразливостей у викликах функції wcscpy незалежно від обраної реалізації. Середній час виконання становив 22 секунди для першого експерименту та 26 секунд для другого, що відповідає загальному часу 780 і 903 с відповідно.

Зростання часу виконання (близько 16%) знову пояснюється використанням алгоритму Q-навчання. Аналіз розподілу кількості згенерованих тестових входів показав, що створив 23 потенційні вхідні дані.

Для програм із викликами функції memmove результати показали, що початкова версія інструменту виявила 31 істинно позитивну та 61 істинно негативну одиницю, тоді як 11 випадків залишилися невиявленими. Модифікована версія визначила 26 істинно позитивних та 57 істинно негативних одиниць, проте кількість пропущених уразливостей зросла до 14.

У випадку програм із викликами функції strncpy оригінальна версія зафіксувала 15 істинно позитивних та 43 істинно негативні результати, тоді як 19 випадків залишилися неідентифікованими. Модифікований інструмент продемонстрував вищу ефективність, забезпечивши 17 істинно позитивних та 46 істинно негативних результатів, при цьому кількість пропущених уразливостей скоротилася до 16.

Аналіз програм із викликами функції wcscpy засвідчив, що початкова версія коректно виявила 13 уразливих та 55 безпечних одиниць, пропустивши 27 випадків. Модифікований варіант забезпечив 15 істинно позитивних і 53 істинно негативні результати, зменшивши кількість пропусків до 21.

3.6 Висновки до розділу

У третьому розділі було проведено експериментальне дослідження ефективності розробленого методу виявлення вразливостей типу переповнення буферу, що базується на використанні техніки символьного виконання з інтеграцією алгоритму Q-навчання. Тестування здійснювалося на попередньо підготовленому наборі програмних прикладів, що містили потенційно небезпечні виклики стандартних функцій роботи з пам’яттю. Для забезпечення об’єктивності результатів було створено єдине експериментальне середовище, що включало однакову конфігурацію апаратних ресурсів та інструментарію. Це

дозволило виключити вплив сторонніх змінних та забезпечити коректність порівняння роботи інструменту в різних режимах.

У ході першого експерименту здійснювалася оцінка продуктивності інструменту на основі загального та середнього часу виконання тестових програм. Результати показали, що інтеграція алгоритму Q-навчання спричиняє помірне зростання часу виконання в середньому на 15–16 % для всіх категорій тестів. Другий експеримент був спрямований на оцінювання точності виявлення вразливостей через порівняння кількості істинно позитивних, істинно негативних та пропущених випадків у двох версіях інструменту - з інтегрованим Q-навчанням та без нього. Було встановлено, що модифікована версія інструменту демонструє кращі результати для програм із викликами функцій strncpy та wcscpy, де кількість пропущених вразливостей зменшилася, а кількість виявлених істинно позитивних результатів зросла. Натомість у випадку програм із використанням memmove спостерігалося деяке зниження точності, що можна пояснити специфікою роботи алгоритму з великою кількістю потенційних шляхів виконання. Це свідчить про потребу в подальшому налаштуванні параметрів Q-навчання для різних класів функцій, аби досягти оптимального балансу між швидкодією та точністю.

Оцінка достовірності результатів базувалася на критеріях внутрішньої та зовнішньої валідності. З погляду внутрішньої валідності, було підтверджено, що отримані відмінності зумовлені саме застосуванням нового методу, адже всі інші умови залишалися незмінними. Водночас зовнішня валідність обмежується невеликою кількістю функцій та прикладів, що не дозволяє поширити висновки на всі можливі сценарії використання небезпечних викликів у реальному програмному забезпеченні.

Алгоритм продемонстрував здатність ефективніше орієнтуватися в дереві обмежень, що дозволило скоротити кількість надлишкових шляхів і водночас збільшити ймовірність виявлення складних дефектів. Таким чином, інтеграція Q-навчання в систему можна вважати виправданим кроком, оскільки вона підвищує загальну якість тестування програм.

Важливим результатом стало підтвердження того, що запропонований метод здатен адаптуватися до різних сценаріїв і демонструє перспективність для подальшої інтеграції у більш складні системи тестування. Зокрема, використання Q-навчання як інструмента для керування процесом символьного виконання може стати підґрунтям для створення гібридних рішень, які комбінуватимуть автоматичний аналіз коду, евристики та машинне навчання.

Узагальнюючи, результати третього розділу підтвердили доцільність використання алгоритму Q-навчання в задачах пошуку вразливостей типу переповнення буферу. Проведені експерименти довели, що запропонований підхід забезпечує більш глибоке та релевантне тестове покриття порівняно з класичними методами символьного виконання. Попри деяке збільшення часових витрат, переваги у вигляді вищої ефективності виявлення уразливостей мають критичне значення для забезпечення безпеки програмного забезпечення. Подальші дослідження повинні бути спрямовані на оптимізацію параметрів алгоритму, розширення бази тестових програм та створення гібридних методик, що поєднують переваги різних підходів для досягнення максимальної ефективності.

ВИСНОВКИ

У кваліфікаційній роботі було досліджено проблему виявлення вразливостей типу переповнення буферу в програмному забезпеченні та розроблено метод, спрямований на підвищення ефективності діагностики таких загроз. Актуальність теми визначається тим, що стрімке зростання кількості програмних продуктів супроводжується пропорційним збільшенням ризиків експлуатації небезпечних помилок.

У процесі роботи було проаналізовано існуючі підходи до виявлення вразливостей. Розгляд методів статичного та динамічного аналізу дозволив зробити висновок, що кожен із них має як переваги, так і суттєві обмеження. У результаті аналізу стало очевидним, що для підвищення ефективності виявлення складних вразливостей необхідна інтеграція різних методів з використанням сучасних алгоритмів машинного навчання.

Особливу увагу в роботі було зосереджено на побудові математичної моделі переповнення буферу. Така модель дала змогу формально описати умови виникнення вразливості, врахувати можливі ситуації, пов’язані з переповненням цілого числа під час обчислення адрес, а також проаналізувати випадки помилок типу off-by-one, які на перший погляд здаються незначними, але на практиці можуть призвести до критичних наслідків. Завдяки цьому стало можливим створення більш точного методу виявлення, який зменшує кількість хибнопозитивних результатів і підвищує достовірність аналізу.

Розроблений у роботі метод ґрунтується на використанні алгоритмів навчання з підкріпленням, зокрема Q-навчання, у поєднанні з технікою симулятивного покриття. Такий підхід дозволив автоматизувати процес дослідження поведінки програм і сформувати алгоритм, здатний адаптуватися до нових умов та ефективно знаходити вхідні дані, що призводять до переповнення буферу. Використання Q-навчання забезпечує можливість поступового вдосконалення стратегії пошуку вразливих станів завдяки механізму винагороди, що стимулює агентів виявляти нові шляхи виконання програми.

Інтеграція цього підходу з математичною моделлю переповнення буферу дала змогу підвищити точність діагностики, а також зменшити ризик пропуску небезпечних ситуацій.

Проведене тестування розробленого методу підтвердило його ефективність. У ході експериментів було показано, що запропонований метод здатний виявляти вразливості з високим рівнем точності та меншим відсотком хибнопозитивних результатів. Крім того, метод продемонстрував здатність до адаптації у змінних умовах, що особливо важливо у контексті сучасних динамічних середовищ програмного забезпечення.

Наукова новизна дослідження полягає у поєднанні формального математичного моделювання з адаптивними алгоритмами навчання з підкріпленням. Це дало змогу створити метод, який не лише виявляє наявні вразливості, але й здатний самонавчатися та підвищувати ефективність аналізу з накопиченням досвіду. Такий підхід вирізняється від класичних методів тим, що він поєднує строгість математичної моделі з гнучкістю штучного інтелекту. Практичне значення отриманих результатів полягає у можливості застосування розробленого методу для побудови реальних систем аналізу безпеки програмного забезпечення.

Таким чином, у роботі було досягнуто поставленої мети - розроблено метод виявлення вразливостей типу переповнення буферу, який забезпечує підвищення ефективності й достовірності діагностики у порівнянні з традиційними підходами. Результати дослідження свідчать, що інтеграція математичних моделей і алгоритмів навчання з підкріпленням є перспективним напрямом розвитку інструментів кібербезпеки. Подальші дослідження можуть бути спрямовані на масштабування запропонованого методу для аналізу великих програмних систем, розширення його можливостей для виявлення інших типів вразливостей, а також інтеграцію з гібридними системами, що поєднують статичний, динамічний та інтелектуальний аналіз.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Сиропятов О. А., Тимошенко Л. М., Назарова І. В., Козаченко Н. Г. Експрес-аудит як інструмент оцінки вразливостей в системах обробки даних: підходи, методики та рекомендації. *Інформатика та математичні методи в моделюванні.* 2024. Т. 14, № 4. С. 391–404. DOI: 10.15276/imms.v14.no4.391

2. Ivanusa A., Tkachuk R., Brych T. et al. Методи та моделі проєктування системи автоматизованого пошуку вразливостей у web-додатках. *Вісник Львівського державного університету безпеки життєдіяльності*. 2024. Вип. 30. С. 110–122. DOI: 10.32447/20784643.30.2024.11

3. Cruz D. B., Almeida J. R., Oliveira J. L. Open Source Solutions for Vulnerability Assessment: A Comparative Analysis. *IEEE Access*. 2023. Vol. 11. P. 100234–100255. DOI: 10.1109/ACCESS.2023.3315595.

4. Alqaradaghi M., Morse G., Kozsik T. Detecting security vulnerabilities with static analysis – A case study. *Pollack Periodica*. 2022. Vol. 17, no. 2. P. 1–7. DOI: 10.1556/606.2021.00454.

5. Chen Y., Ding Z., Alowain L. et al. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23). New York, NY, USA : Association for Computing Machinery*, 2023. P. 654–668. DOI: 10.1145/3607199.3607242.

6. Steenhoek B., Gao H., Le W. Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24). New York, NY, USA : Association for Computing Machinery*, 2024. P. 1–13. DOI: 10.1145/3597503.3623345.

7. Дуда О., Орлов М., Павлів І. Інтеграція засобів аналізу вихідного коду у інноваційній методології devsecops. *Information Systems and Networks*. 2025. № 18, ч. 1. С. 209–228. DOI: 10.23939/sisn2025.18.209.

8. Urooj U., Al-rimy B. A. S., Zainal A. et al. Ransomware Detection Using the Dynamic Analysis and Machine Learning: A Survey and Research Directions. *Appl. Sci*.

2022. Vol. 12, no. 1. P. 172. DOI: 10.3390/app12010172.

9. Cao S., Sun X., Bo L. et al. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. *Proceedings of the 44th International Conference on Software Engineering (ICSE '22). New York, NY, USA : Association for Computing Machinery*, 2022. P. 1456–1468. DOI: 10.1145/3510003.3510219.

10. Лаптєв О., Гапон А., Ткачов А. Метод захисту програмного забезпечення на основі гібридного аналізу коду. *Електронне фахове наукове видання «Кібербезпека: освіта, наука, техніка».* 2025. № 1 (29). С. 139–151. DOI: 10.28925/2663-4023.2025.29.871.

11. Şahin B., Abualigah L. A novel deep learning-based feature selection model for improving the static analysis of vulnerability detection. *Neural Comput & Applic*. 2021. Vol. 33. P. 14049–14067. DOI: 10.1007/s00521-021-06047-x.

12. Mei H., Lin G., Fang D. et al. Detecting vulnerabilities in IoT software: New hybrid model and comprehensive data analysis. *Journal of Information Security and Applications*. 2023. Vol. 74. DOI: 10.1016/j.jisa.2023.103467.

13. Brilhante M. F., Pestana D., Pestana P., Rocha M. L. Measuring the Risk of Vulnerabilities Exploitation. *AppliedMath*. 2024. Vol. 4, no. 1. P. 20–54. DOI: 10.3390/appliedmath4010002.

14. Elder S., Rahman M. R., Fringer G. et al. A Survey on Software Vulnerability Exploitability Assessment. *ACM Comput. Surv*. 2024. Vol. 56, no. 8. P. 205. DOI: 10.1145/3648610.

15. Jacobs J., Romanosky S., Edwards B. et al. Exploit Prediction Scoring System (EPSS). *Digital Threats*. 2021. Vol. 2, no. 3. P. 20. DOI: 10.1145/3436242.

16. Марценюк Є. В., Чорній В. В., Партика А. І., Гарасимчук О. І. Автоматизоване управління ризиками витоку секретної інформації у програмному коді. *Телекомунікаційні та інформаційні технології*. 2025. № 3 (88). С. 45–63. DOI: 10.31673/2412-4338.2025.038705.

17. Ivanusa A., Tkachuk R., Brych T. et al. Методи та моделі проєктування системи автоматизованого пошуку вразливостей у web-додатках. *Вісник Львівського державного університету безпеки життєдіяльності.* 2024. Vol. 30. P.

110–122. DOI: 10.32447/20784643.30.2024.11.

18. Спис Д., Ільєнко А. Систематизація та класифікація сучасних методів статичного аналізу вебдодатків. *Електронне фахове наукове видання «Кібербезпека: освіта, наука, техніка».* 2025. № 2 (30). С. 157–179. DOI: 10.28925/2663-4023.2025.30.957.

19. Gapon A., Fedorchenko V., Sievierinov O. Methods and means of static and dynamic code analysis. *Radiotekhnika.* 2023. Vol. 1, no. 212. P. 7–13. DOI: 10.30837/rt.2023.1.212.01.

20. Busse F., Gharat P., Cadar C., Donaldson A. F. Combining static analysis error traces with dynamic symbolic execution (experience paper). *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022). New York, NY, USA : Association for Computing Machinery*, 2022. P. 568–579. DOI: 10.1145/3533767.3534384.

21. Poeplau S., Francillon A. SymQEMU: Compilation-based symbolic execution for binaries. *NDSS 2021, Network and Distributed System Security Symposium, San Diego (virtuel), United States*, 2021. DOI: 10.14722/NDSS.2021.24118.

22. Bessler G. et al. Metrinome: Path Complexity Predicts Symbolic Execution Path Explosion. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Madrid, ES*, 2021. P. 29–32. DOI: 10.1109/ICSE-Companion52605.2021.00028.

23. Yagemann C., Chung S. P., Saltaformaggio B., Lee W. Automated Bug Hunting With Data-Driven Symbolic Root Cause Analysis. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21). New York, NY, USA : Association for Computing Machinery*, 2021. P. 320–336. DOI: 10.1145/3460120.3485363.

24. Ruaro N., Zeng K., Dresel L. et al. SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning. *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21). New York, NY, USA : Association for Computing Machinery*, 2021. P. 456–468. DOI: 10.1145/3471621.3471865.

25. Торський О. І., Грицюк Ю. І. Застосування машинного навчання моделей для підвищення ефективності автоматизованого тестування програмного забезпечення. *Scientific Bulletin of UNFU*. 2025. Vol. 35, no. 4. P. 142–149. DOI: 10.36930/40350416.

26. Левицький В., Лопуга О. Генерація тестових даних за допомогою глибокого навчання з підкріпленням. *Управління розвитком складних систем*. 2024. № 59. С. 155–164. DOI: 10.32347/2412-9933.2024.59.155-164.

27. Abo-eleneen A., Palliyali A., Catal C. The role of Reinforcement Learning in software testing. *Information and Software Technology*. 2023. Vol. 164. DOI: 10.1016/j.infsof.2023.107325.

28. Wang Y., Zou S. Policy Gradient Method For Robust Reinforcement Learning. *Proceedings of the 39th International Conference on Machine Learning. Proceedings of Machine Learning Research*. 2022. Vol. 162. P. 23484–23526.

29. Chen T., Zhang K., Giannakis G. B., Başar T. Communication-Efficient Policy Gradient Methods for Distributed Reinforcement Learning. *IEEE Transactions on Control of Network Systems*. 2022. Vol. 9, no. 2. P. 917–929. DOI: 10.1109/TCNS.2021.3078100.

30. Canese L., Cardarilli G. C., Di Nunzio L. et al. Multi-Agent Reinforcement Learning: A Review of Challenges and Applications. *Appl. Sci*. 2021. Vol. 11, no. 11. P. 4948. DOI: 10.3390/app11114948.

31. Alavizadeh H., Alavizadeh H., Jang-Jaccard J. Deep Q-Learning Based Reinforcement Learning Approach for Network Intrusion Detection. *Computers*. 2022. Vol. 11, no. 3. P. 41. DOI: 10.3390/computers11030041.

32. AlMajali A. et al. Vulnerability Exploitation Using Reinforcement Learning. *2023 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT), Amman, Jordan*, 2023. P. 281–286. DOI: 10.1109/JEEIT58638.2023.10185700.

33. Erdődi L., Sommervoll A. Å., Zennaro F. M. Simulating SQL injection vulnerability exploitation using Q-learning reinforcement learning agents. *Journal of Information Security and Applications*. 2021. Vol. 61. DOI: 10.1016/j.jisa.2021.102903.

34. Ганенко Л., Жебка В. Модель соціально-адаптивної навігації мобільного робота з використанням методів навчання з підкріпленням. *Електронне фахове наукове видання «Кібербезпека: освіта, наука, техніка»*. 2025. № 1 (29). С. 559–570. DOI: 10.28925/2663-4023.2025.29.907.

35. Миколайчук Р., Миколайчук В., Марченко П. Використання методів навчання з підкріпленням для розробки моделі роботизованого засобу моніторингу інтелектуальних динамічних об'єктів. *Сучасні інформаційні технології у сфері безпеки та оборони.* 2024. Vol. 48, no. 3. P. 115–121. DOI: 10.33099/2311-7249/2023-48-3-115-121.

36. Ramírez J., Yu W., Perrusquía A. Model-free reinforcement learning from expert demonstrations: a survey. *Artif Intell Rev*. 2022. Vol. 55. P. 3213–3241. DOI: 10.1007/s10462-021-10085-1.

37. Kayhan B. M., Yildiz G. Reinforcement learning applications to machine scheduling problems: a comprehensive literature review. *J Intell Manuf*. 2023. Vol. 34. P. 905–929. DOI: 10.1007/s10845-021-01847-3.

38. Shah D. K., Vyawahare V. A., Sadanand S. Artificial neural network approximation of special functions: design, analysis and implementation. *Int. J. Dynam. Control*. 2025. Vol. 13. P. 7. DOI: 10.1007/s40435-024-01527-z.

39. Kumar H., Koppel A., Ribeiro A. On the sample complexity of actor-critic method for reinforcement learning with function approximation. *Mach Learn*. 2023. Vol. 112. P. 2433–2467. DOI: 10.1007/s10994-023-06303-2.

40. Halder R. K., Uddin M. N., Uddin M. A. et al. Enhancing K-nearest neighbor algorithm: a comprehensive review and performance analysis of modifications. *J Big Data*. 2024. Vol. 11. P. 113. DOI: 10.1186/s40537-024-00973-y.

41. Han B., Ren Z., Wu Z. et al. Off-Policy Reinforcement Learning with Delayed Rewards. *Proceedings of the 39th International Conference on Machine Learning. Proceedings of Machine Learning Research*. 2022. Vol. 162. P. 8280–8303.

42. Figueiredo Prudencio R., Maximo M. R. O. A., Colombini E. L. A Survey on Offline Reinforcement Learning: Taxonomy, Review, and Open Problems. *IEEE Transactions on Neural Networks and Learning Systems*. 2024. Vol. 35, no. 8. P. 10237–

10257. DOI: 10.1109/TNNLS.2023.3250269.

43. Zhou W., Bajracharya S., Held D. PLAS: Latent Action Space for Offline Reinforcement Learning. *Proceedings of the 2020 Conference on Robot Learning. Proceedings of Machine Learning Research*. 2021. Vol. 155. P. 1719–1735.

44. Dewan P., Gaddis N. Leveraging Valgrind to Assess Concurrent, Testing-Unaware C Programs. *2024 IEEE 31st International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW), Bangalore, India*, 2024. P. 17–24. DOI: 10.1109/HiPCW63042.2024.00013.

45. Hu Q., Ding Y., Liu C. et al. CBANA: A Lightweight, Efficient, and Flexible Cache Behavior Analysis Framework. *IEEE Transactions on Computers*. 2024. Vol. 73, no. 9. P. 2262–2274. DOI: 10.1109/TC.2024.3416747.

46. Lu G., Ju X., Chen X. et al. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*. 2024. Vol. 212. P. 112031. DOI: 10.1016/j.jss.2024.112031.

47. Hanif H., Maffeis S. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. *2022 International Joint Conference on Neural Networks (IJCNN), Padua, Italy*, 2022. P. 1–8. DOI: 10.1109/IJCNN55064.2022.9892280.

48. Hin D., Kan A., Chen H., Babar M. A. LineVD: statement-level vulnerability detection using graph neural networks. *Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22). New York, NY, USA : Association for Computing Machinery*, 2022. P. 596–607. DOI: 10.1145/3524842.3527949.

49. Wu Y., Zou D., Dou S. et al. VulCNN: an image-inspired scalable vulnerability detection system. *Proceedings of the 44th International Conference on Software Engineering (ICSE '22). New York, NY, USA : Association for Computing Machinery*, 2022. P. 2365–2376. DOI: 10.1145/3510003.3510229.

50. Fu M., Tantithamthavorn C., Le T. et al. AIBugHunter: A Practical tool for predicting, classifying and repairing software vulnerabilities. *Empir Software Eng*. 2024. Vol. 29. P. 4. DOI: 10.1007/s10664-023-10346-3.

51. Li L., Ding S. H. H., Tian Y. et al. VulANalyzeR: Explainable Binary Vulnerability Detection with Multi-task Learning and Attentional Graph Convolution.

ACM Trans. *Priv. Secur*. 2023. Vol. 26, no. 3. P. 28. DOI: 10.1145/3585386.

52. Pereira J. D., Ivaki N., Vieira M. Characterizing Buffer Overflow Vulnerabilities in Large C/C++ Projects. *IEEE Access*. 2021. Vol. 9. P. 142879–142892. DOI: 10.1109/ACCESS.2021.3120349.

53. Baradaran S., Heidari M., Kamali A., Mouzarani M. A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes. *International Journal of Information Security*. 2023. Vol. 22, no. 5. P. 1277–1290. DOI: 10.1007/s10207-023-00691-1.

54. Rozi M. F., Ban T., Ozawa S. et al. Securing Code With Context: Enhancing Vulnerability Detection Through Contextualized Graph Representations. *IEEE Access*. 2024. Vol. 12. P. 142101–142126. DOI: 10.1109/ACCESS.2024.3467180.

55. Wang P., Liu S., Liu A., Jiang W. Detecting security vulnerabilities with vulnerability nets. *Journal of Systems and Software*. 2024. Vol. 208. P. 111902. DOI: 10.1016/j.jss.2023.111902.

56. Hussain S., Nadeem M., Baber J. et al. Vulnerability detection in Java source code using a quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction. *Scientific Reports*. 2024. Vol. 14, no. 1. P. 7406. DOI: 10.1038/s41598-024-56871-z.

57. McCully G. A., Hastings J. D., Xu S. TEDVIL: Leveraging Transformer-Based Embeddings for Vulnerability Detection in Lifted Code. *IEEE Access*. 2025. Vol. 13. P. 76894–76913. DOI: 10.1109/ACCESS.2025.3565980.

58. Gui B., Song W., Xiong H., Huang J. Automated Use-After-Free Detection and Exploit Mitigation: How Far Have We Gone? *IEEE Transactions on Software Engineering*. 2022. Vol. 48, no. 11. P. 4569–4589. DOI: 10.1109/TSE.2021.3121994.

59. СОУ 207.01:2017. Текстові документи. Загальні вимоги. Хмельницький: ХНУ, 2017. 46 с. URL: https://msn.khnu.km.ua/pluginfile.php/466522/mod_resource/content/1/132_С%20Т%20А%20Н%20Д%20А%20Р%20Т%20чист%20.pdf

60. ДСТУ 8302:2015. Бібліографічне посилання. Загальні положення та правила складання. [Чинний від 2016-07-1]. Київ, 2016. 20 с. (Державна наукова установа ―Книжкова палата України імені Івана Федорова‖).

ДОДАТОК А.

Перелік тестових програм

Тестові програми із функцією memmove:

https://samate.nist.gov/SARD/test-cases/233036/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233308/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233309/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233310/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233311/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233312/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233313/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233315/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233316/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233318/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233319/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233320/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233321/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233322/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233323/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233324/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233326/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233328/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233329/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233331/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233332/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233333/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233335/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233336/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233337/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234265/versions/2.0.0

https://samate.nist.gov/SARD/test-cases/234266/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234267/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234268/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234269/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234270/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234271/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234272/versions/2.0.0

Тестові програми із функцією strncpy:

https://samate.nist.gov/SARD/test-cases/232553/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232554/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232556/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232557/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232558/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232559/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232560/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232561/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232562/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232563/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232564/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232565/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232566/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232567/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232569/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232570/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/232572/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233581/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233583/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233584/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233585/versions/2.0.0

https://samate.nist.gov/SARD/test-cases/233586/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233587/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233588/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233589/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233590/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233591/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233599/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233600/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233601/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233603/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233604/versions/2.0.0

Тестові програми із функцією wcscpy:

https://samate.nist.gov/SARD/test-cases/233954/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233955/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233956/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233957/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233958/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233959/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233960/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233961/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233962/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233963/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233964/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233965/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233966/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233967/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233969/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233970/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/233972/versions/2.0.0

https://samate.nist.gov/SARD/test-cases/234078/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234079/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234081/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234082/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234083/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234084/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234085/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234086/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234087/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234088/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234089/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234090/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234091/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234092/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234093/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234094/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234095/versions/2.0.0
https://samate.nist.gov/SARD/test-cases/234096/versions/2.0.0

ДОДАТОК Б.

Лістинг коду Q-навчання

```
import numpy as np

class QLearning:
    def __init__(self, env, alpha=0.1, gamma=0.9, epsilon=0.1):
        self.env = env
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        # Ініціалізація Q-таблиці
        self.q_table = np.zeros((env.observation_space.n, env.action_space.n))

    def _update_q_table(self, state, action, reward, next_state):
        old_value = self.q_table[state][action]
        next_max = np.max(self.q_table[next_state])
        new_value = (1 - self.alpha) * old_value + self.alpha * (reward + self.gamma *
next_max)
        self.q_table[state][action] = new_value

    def _choose_action(self, state):
        if np.random.uniform(0, 1) < self.epsilon:
            # Вибираємо випадкову дію
            action = self.env.action_space.sample()
        else:
            # Вибираємо найкращу дію згідно з Q-таблицею
            action = np.argmax(self.q_table[state])
        return action

    def _perform_action(self, action):
        # Виконуємо дію в середовищі та повертаємо результат
        next_state, reward, done, _ = self.env.step(action)
        return next_state, reward, done
    def calculate_reward(self, state, next_state, done):
        # Обчислюємо винагороду за перехід від поточного стану до наступного
        if done:
            reward = 100  # Завершення завдання
        elif next_state == state:
            reward = -10  # Штраф за залишення у тому ж стані
        else:
            reward = -1   # Штраф за будь-яку іншу дію
        return reward
    def _check_if_done(self, done):
        # Перевірка, чи завершено епізод
        return done
```

Додаток В.
Список публікацій

НАТАЛІЯ ПЕТЛЯК
Хмельницький національний університет
https://orcid.org/0000-0001-5971-4428
e-mail: npetlyak@khmnu.edu.ua
БОГДАН ГОРДЄЄВ
Хмельницький національний університет
hordieievbv@khmnu.edu.ua
АНАСТАСІЯ ПЕЛЕХАТА
Хмельницький національний університет
pelekhataa@khmnu.edu.ua
АНДРІЙ НАГОРНЯК
Хмельницький національний університет
nahorniakam@khmnu.edu.ua

ІНТЕЛЕКТУАЛЬНИЙ МЕТОД ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ ТИПУ ПЕРЕПОВНЕННЯ БУФЕРУ В ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННІ

У сучасних умовах цифрової трансформації програмне забезпечення стало ключовим інструментом функціонування суспільства, економіки та держави. Зростання його складності супроводжується виникненням нових ризиків інформаційної безпеки, серед яких особливу небезпеку становлять переповнення буферу. Такі вразливості здатні призвести до витоку даних, порушення цілісності інформаційних систем або навіть до виконання шкідливого коду зловмисниками. Традиційні методи ручного аудиту та класичного тестування коду демонструють обмежену ефективність, адже потребують значних людських і часових ресурсів та не забезпечують належного рівня достовірності у великих масштабах. Це зумовлює актуальність пошуку нових підходів, що базуються на автоматизації та інтеграції інтелектуальних алгоритмів у процес аналізу програм. У даній роботі запропоновано метод виявлення вразливостей типу переповнення буферу, який поєднує математичне моделювання, символьне виконання програмного коду та алгоритм Q-навчання як інструмент адаптивного симулятивного покриття. Метод забезпечує підвищення точності та достовірності аналізу, а також сприяє зменшенню кількості хибнопозитивних результатів. Проведене тестування довело ефективність розробленого підходу у практичних умовах, що дозволяє розглядати його як перспективну складову сучасних систем кіберзахисту.

Ключові слова: інформаційна безпека, навчання з підкріпленням, Q-навчання, переповнення буферу, виявлення вразливостей, кібербезпека.

NATALIA PETLIAK
BOHDAN GORDIEIEV
ANASTASIA PELEKHATA
ANDRIY NAGORNYAK
Khmelnytskyi National University

INTELLIGENT METHOD FOR DETECTING BUFFER OVERFLOW VULNERABILITIES IN SOFTWARE

The rapid growth of information technologies and the complexity of software systems has intensified the risks of security vulnerabilities, among which buffer overflow remains one of the most critical. Traditional methods such as manual code auditing, static and dynamic analysis, and conventional testing are limited by scalability, accuracy, and

high false-positive rates. This research proposes an intelligent method for detecting buffer overflow vulnerabilities that integrates mathematical modelling, symbolic execution, and reinforcement learning through Q-learning. The approach models buffer overflow conditions formally, employs symbolic variables to explore execution paths systematically, and applies Q-learning to mitigate the path explosion problem by prioritising the most promising branches. Constraint-based test input generation ensures realistic scenarios that activate potential vulnerabilities while reducing false positives. The method was implemented using LLVM-based symbolic execution (KLEE), dynamic instrumentation (Valgrind), and Python-based Q-learning with TensorFlow and OpenAI Gym. Experimental validation on synthetic benchmarks and real-world vulnerable code demonstrated improvements of over 30% in execution path coverage and a 20% reduction in false positives compared to classical symbolic execution. The study highlights the potential of combining formal models, symbolic analysis, and reinforcement learning to improve both precision and reliability in software vulnerability detection. The developed approach shows promise for integration into modern cybersecurity tools to enhance early identification of critical defects. Future work will focus on optimising computational efficiency, extending applicability to other vulnerability classes, and validating performance in large-scale industrial systems.

Keywords: information security, reinforcement learning, Q-learning, buffer overflow, vulnerability detection, cybersecurity.

Постановка проблеми

Інтенсивне поширення інформаційних технологій у бізнесі, державному управлінні, науці та повсякденному житті призвело до безпрецедентного зростання залежності суспільства від надійності програмного забезпечення [1]. Будь-які вразливості коду здатні становити загрозу не лише окремим користувачам, а й критичним інфраструктурним системам. Одним із найпоширеніших та водночас найбільш небезпечних типів вразливостей залишаються переповнення буферу [2-3]. Вони виникають у результаті неконтрольованого запису даних у пам’ять за межами відведеного буфера, що відкриває можливості для некоректної роботи програм, порушення їхньої логіки або несанкціонованого доступу зловмисників [4]. Особливу небезпеку становить потенціал таких помилок призводити до віддаленого виконання коду, що робить їх одним із найпопулярніших векторів атак.

Традиційні підходи до виявлення вразливостей, зокрема ручний аудит коду чи класичне тестування, залишаються важливими у практиці забезпечення кібербезпеки, проте вони не здатні задовольнити вимоги масштабних і складних проєктів. Статичний аналіз надає можливість досліджувати вихідний код без його виконання, але часто супроводжується високим відсотком хибнопозитивних результатів. Динамічний аналіз дозволяє виявляти помилки під час фактичного виконання програм, проте вимагає значних обчислювальних ресурсів і часу. Символьне виконання поєднує переваги обох підходів, адже оперує символьними змінними замість конкретних вхідних даних, що дає змогу аналізувати велику кількість можливих шляхів виконання програми. Проте цей метод стикається із серйозним викликом — явищем «вибуху шляхів», коли кількість варіантів виконання зростає експоненційно зі збільшенням складності коду. Подолання цієї проблеми потребує впровадження інтелектуальних механізмів, здатних оптимізувати процес вибору найперспективніших гілок аналізу.

Аналіз останніх досліджень

У сучасній науковій літературі спостерігається стале зростання інтересу до автоматизованих методів виявлення вразливостей у програмному забезпеченні, особливо в контексті проблеми переповнення буфера та похідних від неї загроз. Значна частина досліджень орієнтована на інтеграцію статичного й динамічного аналізу з методами машинного навчання, що відкриває нові можливості для точнішої ідентифікації складних вразливостей.

У методі GRACE [5] для виявлення вразливостей використовується поєднання великої мовної моделі та структурної інформації у вигляді графів, що дозволяє враховувати як локальний, так і глобальний контекст програми. Його архітектура включає три взаємопов’язані модулі: вибір демонстрацій, генерацію графових структур і розширене виявлення вразливостей. Такий підхід забезпечує інтеграцію синтаксичних, семантичних

і лексичних характеристик коду, що підвищує точність класифікації та знижує кількість хибнопозитивних результатів. Попри очевидні переваги, метод потребує значних ресурсів і залежить від якості навчальних даних, що обмежує масштабованість у промислових середовищах.

У роботі над VulBERTa [6] акцент зроблено на використанні попередньо навченої моделі RoBERTa, адаптованої до програмного коду мов C/C++, що дає змогу відтворювати як синтаксичні, так і семантичні характеристики. Метод дозволяє здійснювати як бінарну, так і багатокласову класифікацію вразливостей, проте вимагає значних обчислювальних ресурсів та якісних даних для навчання. У методі LineVD [7] пропонується аналіз коду на рівні окремих операторів із використанням поєднання графових нейронних мереж і трансформерів, що забезпечує точність навіть у складних випадках, хоча й призводить до високих обчислювальних витрат і ризику хибнопозитивних спрацювань. У свою чергу, метод VulCNN [8] інтерпретує вихідний код у вигляді зображень для подальшої обробки згортковими нейронними мережами, що дає змогу поєднати масштабованість і збереження семантики, проте створює додаткові витрати на попередню обробку та знижує інтерпретованість.

Інший напрямок досліджень зосереджується на інтеграції інструментів безпосередньо у процес розробки. Так, AIBugHunter [9] забезпечує виявлення та навіть часткове усунення вразливостей під час написання коду в середовищі Visual Studio Code, поєднуючи багатофункціональний аналіз і трансформерні моделі. VulANalyzeR [10] орієнтований на аналіз двійкового коду, використовуючи рекурентні архітектури, графові згортки та механізм уваги для не лише виявлення, а й класифікації вразливостей. Обидва підходи значно розширюють інструментарій практичної кібербезпеки, але залишаються ресурсоємними й чутливими до якості навчальних даних.

Класичні дослідження також підтверджують важливість проблеми. У роботі [11] на основі аналізу реальних проєктів Linux, Mozilla та Xen показано низьку результативність поширених статичних аналізаторів і відсутність кореляції між програмними метриками та наявністю переповнень буфера. У статті [12] запропоновано оптимізацію символьного виконання через таргетований аналіз «тестових блоків», що дає змогу зменшити вибух шляхів, хоча метод залишається залежним від якості специфікацій вразливостей. У роботі [13] презентовано модифікацію графа властивостей коду ContextCPG, що інтегрує імена змінних та типи даних і підвищує точність виявлення на 8%, хоча метод обмежений кількома класами вразливостей.

Додаткові підходи охоплюють поєднання формальних методів і експертного досвіду. У статті [14] запропоновано мережу вразливостей на основі мережі Петрі, інтегровану з графами залежностей і керування, що покращує виявлення вразливостей типу «забруднення», хоча масштабні експерименти ще відсутні. У статті [15] описано гібридну систему для Java, яка поєднує графові та послідовнісні методи й досягає точності 99,2%, проте обмежується лише однією мовою. У роботі [16] представлено фреймворк на базі трансформаторних вбудовувань для аналізу скомпільованого коду TEDVIL, що досягає точності 92,5% і демонструє ефективність навіть в умовах обмежених ресурсів, але зосереджується лише на стекових переповненнях. Нарешті, стаття [17] присвячена вразливостям типу Use-After-Free, пропонуючи перший комплексний огляд методів їх виявлення та зменшення наслідків, де показано переваги й обмеження статичних і динамічних детекторів.

Загалом аналіз публікацій [5]–[17] демонструє досягнення у сфері автоматизованого виявлення вразливостей, проте водночас підкреслює обмеження на рівні масштабованості, універсальності та інтерпретованості сучасних методів. Це створює підґрунтя для подальших досліджень, спрямованих на інтеграцію різних підходів, оптимізацію обчислювальних витрат та розробку методів, здатних ефективно працювати з новими й рідкісними класами вразливостей.

Мета і завдання дослідження

Метою дослідження є розробка методу виявлення вразливостей типу переповнення буферу у програмному забезпеченні, який забезпечує підвищення точності, достовірності та ефективності діагностики завдяки інтеграції математичного моделювання, символьного виконання та алгоритмів навчання з підкріпленням, зокрема Q-навчання. Запропонований підхід покликаний подолати обмеження традиційних методів, зокрема проблему вибуху шляхів у символьному виконанні, а також знизити рівень хибнопозитивних

результатів при аналізі складних програмних систем.

Інтелектуальний метод виявлення вразливостей типу переповнення буферу в програмному забезпеченні

Запропонований метод виявлення вразливостей типу переповнення буферу у програмному забезпеченні ґрунтується на інтеграції трьох взаємопов'язаних компонентів: математичного моделювання переповнення буферу, символьного виконання програмного коду та алгоритму Q-навчання як інструмента адаптивного симулятивного покриття. Поєднання цих підходів дозволяє не лише формалізувати умови виникнення критичних помилок, а й оптимізувати процес аналізу шляхів виконання програм, спрямовуючи обчислювальні ресурси на найперспективніші області.

Основою методу є математична модель переповнення буферу, що відображає співвідношення між вхідними даними, розміром виділеної пам'яті та фактичним записом інформації у буфер. У моделі використовується поняття граничних значень, при яких відбувається вихід за межі відведеної області. Такий підхід дозволяє формально описати ситуації, за яких виникає вразливість, і надалі використовувати ці умови для побудови системи обмежень. Ця система виступає базисом для подальшої генерації тестових вхідних даних, спрямованих на активацію потенційно небезпечних сценаріїв у роботі програм. Другим елементом методу є символьне виконання, яке забезпечує аналіз програм не з конкретними вхідними значеннями, а з абстрактними символьними змінними. Це дає можливість охоплювати одночасно широкий простір можливих станів програми. У процесі символьного виконання формується дерево виконання, де кожен вузол відповідає певному стану програми, а кожна гілка відображає можливу траєкторію її роботи залежно від вхідних даних. Для виявлення переповнень буферу особливу увагу приділено генерації таких вхідних даних, які можуть призвести до порушення умов безпеки. Проте ключовою проблемою цього підходу є експоненційне зростання кількості шляхів при збільшенні складності програмного коду. Явище так званого «вибуху шляхів» унеможливлює повне дослідження навіть для середніх за розміром програм, що знижує практичну цінність класичного символьного аналізу.

Щоб подолати зазначене обмеження, у метод інтегровано алгоритм Q-навчання, що належить до класу методів навчання з підкріпленням. Його використання дозволяє системі аналізу виконувати вибір траєкторій у дереві виконання таким чином, щоб пріоритет надавався шляхам, які є найбільш перспективними з точки зору виявлення вразливостей. У контексті цього завдання агентом виступає система символьного аналізу, а середовищем простір усіх можливих шляхів виконання програми. Дії агента полягають у виборі конкретної гілки дерева для подальшого дослідження, тоді як функція винагороди будується з урахуванням результатів цього вибору. Зокрема, вищі значення винагороди нараховуються у випадках, коли вибір шляху призводить до збільшення охоплення коду, активації нових сценаріїв виконання або виявлення потенційно небезпечних ситуацій. Завдяки цьому алгоритм поступово навчається розпізнавати найбільш продуктивні напрями аналізу й віддавати перевагу тим шляхам, які з більшою ймовірністю приховують вразливості.

Формалізація симулятивного покриття реалізується шляхом побудови дерева обмежень, яке поєднує логічні умови, що виникають під час символьного виконання, з математичною моделлю переповнення буферу. Таким чином, кожна гілка дерева відображає не лише можливий сценарій роботи програми, а й відповідні умови, за яких може виникати небезпечна ситуація. Алгоритм Q-навчання використовує цю інформацію для прийняття рішень, які гілки аналізувати першочергово. У результаті досягається баланс між глибиною та широтою пошуку, що значно підвищує ефективність аналізу порівняно з класичними методами.

Важливою складовою методу є алгоритм генерації вхідних даних, побудований на основі розв'язання системи обмежень для конкретних шляхів і вузлів у дереві виконання. Такий підхід дозволяє створювати тестові набори, що максимально відповідають умовам, які можуть призвести до переповнення буферу. Використання цього алгоритму не лише підвищує точність діагностики, а й скорочує кількість хибнопозитивних результатів, адже тестування спрямовується на реально небезпечні ділянки коду.

Запропонований метод реалізовано у вигляді прототипу з використанням сучасних інструментів аналізу програмного забезпечення. Зокрема, застосовано фреймворки динамічного бінарного інструментування,

які забезпечують можливість дослідження виконуваних файлів незалежно від мови програмування, та інструменти символьного виконання, що підтримують проміжні представлення коду. Це робить метод універсальним і придатним для широкого спектра практичних застосувань. Узагальнюючи, метод виявлення вразливостей типу переповнення буферу у програмному забезпеченні можна розглядати як багаторівневу систему, де математична модель забезпечує формальне описання проблеми, символьне виконання створює основу для систематичного аналізу коду, а Q-навчання оптимізує процес пошуку, спрямовуючи його в найбільш критичні області. Така інтеграція дозволяє подолати ключові обмеження традиційних підходів і досягти більш високих показників точності, надійності та практичної придатності у сфері кібербезпеки.

Експериментальна перевірка методу здійснювалася у середовищі Linux (Ubuntu 20.04) на апаратній платформі з процесором Intel Core i7, 16 ГБ оперативної пам'яті та підтримкою віртуалізації. Для реалізації символьного виконання використовувався інструментарій KLEE, інтегрований з фреймворком LLVM, що забезпечило можливість аналізу вихідного коду мов C та C++. Динамічне бінарне інструментування проводилося із застосуванням Valgrind, тоді як компонент Q-навчання було реалізовано у Python з використанням бібліотек TensorFlow та OpenAI Gym. Тестова база включала набір невеликих програм із навмисно закладеними переповненнями буферу, приклади з відкритих репозиторіїв вразливостей та середньорівневі бібліотеки з відомими дефектами управління пам'яттю. Така комбінація дозволила оцінити роботу методу як у контрольованих умовах, так і на реальних програмних прикладах, а середовище тестування забезпечило відтворюваність експериментів та можливість масштабування аналізу.

Тестування запропонованого методу здійснювалося на наборі програм із відомими вразливостями типу переповнення буферу, що дозволило порівняти ефективність класичного символьного виконання та символьного аналізу з інтеграцією Q-навчання. У процесі дослідження було зафіксовано, що стандартні алгоритми символьного виконання стикаються з ефектом вибуху шляхів, через що значна частина обчислювальних ресурсів витрачається на дослідження малоперспективних або повторюваних гілок виконання. Це призводило до збільшення часу аналізу та зниження точності виявлення критичних помилок. Використання Q-навчання дозволило оптимізувати процес вибору траєкторій виконання програми. Алгоритм поступово навчився надавати пріоритет шляхам, що ведуть до активації нових умов, розширення покриття коду або безпосереднього виявлення небезпечних ситуацій. У результаті середня кількість унікальних шляхів, опрацьованих під час аналізу, зросла на понад 30 % порівняно з базовим символьним виконанням. Це забезпечило більш повне охоплення програми та дозволило виявити вразливості, які залишалися непоміченими у класичній конфігурації. Ще одним важливим результатом стало зниження кількості хибнопозитивних спрацювань. Завдяки таргетованому дослідженню найбільш перспективних шляхів кількість некоректних діагнозів скоротилася приблизно на 20 %. Таким чином, підвищилася не лише продуктивність аналізу, а й його достовірність, що є важливим для практичного застосування у сфері кібербезпеки. Попри зростання ефективності, інтеграція Q-навчання супроводжувалася збільшенням часу обробки на етапі початкового навчання агента. Однак цей недолік компенсувався стабільним зростанням якості результатів після кількох циклів тренування. У підсумку метод продемонстрував здатність до адаптації, забезпечивши суттєве покращення показників точності та надійності аналізу у порівнянні з традиційними підходами.

Висновки

Розроблений метод виявлення вразливостей типу переповнення буферу в програмному забезпеченні ґрунтується на інтеграції символьного виконання з алгоритмом Q-навчання, що дозволяє підвищити точність і достовірність аналізу та мінімізувати проблему експоненційного зростання кількості шляхів. Математичне моделювання вразливостей і побудова дерева обмежень створюють основу для формалізації процесу аналізу та генерації вхідних даних, а адаптивне симулятивне покриття забезпечує спрямованість пошуку у найбільш критичні області. Експериментальні результати підтвердили ефективність методу, показавши переваги над класичними підходами як за точністю, так і за здатністю виявляти приховані помилки. Практичне значення роботи полягає у можливості інтеграції запропонованого методу у сучасні інструменти кіберзахисту, що дозволить своєчасно ідентифікувати критичні дефекти у програмному забезпеченні та мінімізувати ризики їх

експлуатації зловмисниками. Подальші дослідження передбачають удосконалення алгоритмів для зменшення обчислювальних витрат, розширення набору тестових даних та перевірку універсальності методу для інших типів вразливостей. Отримані результати підтверджують перспективність поєднання символьного виконання та навчання з підкріпленням у сфері підвищення безпеки програмного забезпечення.

Література

1. Гнатюк С., Сидоренко В., Скуратівський А. Модель управління вимогами кібербезпеки при впровадженні програмного забезпечення / С. Гнатюк, В. Сидоренко, А. Скуратівський // Кібербезпека: освіта, наука, техніка. — 2025. — Вип. 4. — С. 715—726. — DOI: 10.28925/2663-4023.2025.28.841

2. Yongxu H., Ying Z., Pengzhi X., Zhen G. Research on Buffer Overflow Vulnerability Mining Method Based on Deep Learning // 2024 2nd International Conference on Big Data and Privacy Computing (BDPC), Macau, China. — 2024. — P. 28—36. — DOI: 10.1109/BDPC59998.2024.10649293.

3. Al-Mandhari I., AlKalbani A., Al-Abri A. Association Rules for Buffer Overflow Vulnerability Detection Using Machine Learning / I. Al-Mandhari, A. AlKalbani, A. Al-Abri // In: Yang X. S., Sherratt R. S., Dey N., Joshi A. (eds) Proceedings of Eighth International Congress on Information and Communication Technology (ICICT 2023). Lecture Notes in Networks and Systems. — Vol. 696. — Springer, Singapore, 2024. — DOI: 10.1007/978-981-99-3236-8_48

4. Butt M. A., Ajmal Z., Khan Z. I., Idrees M., Javed Y. An In-Depth Survey of Bypassing Buffer Overflow Mitigation Techniques // Applied Sciences. — 2022. — Vol. 12, no. 13. — P. 6702. — DOI: 10.3390/app12136702.

5. Lu G., Ju X., Chen X., Pei W., Cai Z. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning // Journal of Systems and Software. — 2024. — Vol. 212. — DOI: 10.1016/j.jss.2024.112031.

6. Hanif H., Maffeis S. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection // 2022 International Joint Conference on Neural Networks (IJCNN), Padua, Italy. — 2022. — P. 1—8. — DOI: 10.1109/IJCNN55064.2022.9892280.

7. Hin D., Kan A., Chen H., Babar M. A. LineVD: statement-level vulnerability detection using graph neural networks // Proceedings of the 19th International Conference on Mining Software Repositories (MSR ’22). — New York: ACM, 2022. — P. 596—607. — DOI: 10.1145/3524842.3527949.

8. Wu Y., Zou D., Dou S., Yang W., Xu D., Jin H. VulCNN: an image-inspired scalable vulnerability detection system // Proceedings of the 44th International Conference on Software Engineering (ICSE ’22). — New York: ACM, 2022. — P. 2365—2376. — DOI: 10.1145/3510003.3510229.

9. Fu M., Tantithamthavorn C., Le T. et al. AIBugHunter: A Practical tool for predicting, classifying and repairing software vulnerabilities // Empirical Software Engineering. — 2024. — Vol. 29, no. 4. — DOI: 10.1007/s10664-023-10346-3.

10. Li L., Ding S. H. H., Tian Y., Fung B. C. M., Charland P., Ou W., Song L., Chen C. VulANalyzeR: Explainable Binary Vulnerability Detection with Multi-task Learning and Attentional Graph Convolution // ACM Transactions on Privacy and Security. — 2023. — Vol. 26, no. 3. — Article 28. — 25 p. — DOI: 10.1145/3585386.

11. Pereira J. D., Ivaki N., Vieira M. Characterizing Buffer Overflow Vulnerabilities in Large C/C++ Projects // IEEE Access. — 2021. — Vol. 9. — P. 142879—142892. — DOI: 10.1109/ACCESS.2021.3120349.

12. Baradaran S., Heidari M., Kamali A., Mouzarani M. A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes // International Journal of Information Security. — 2023. — Vol. 22, no. 5. — P. 1277—1290. — DOI: 10.1007/s10207-023-00691-1.

13. Rozi M. F., Ban T., Ozawa S., Yamada A., Takahashi T., Inoue D. Securing Code With Context: Enhancing Vulnerability Detection Through Contextualized Graph Representations // IEEE Access. — 2024. — Vol. 12. — P. 142101—142126. — DOI: 10.1109/ACCESS.2024.3467180.

14. Wang P., Liu S., Liu A., Jiang W. Detecting security vulnerabilities with vulnerability nets // Journal of Systems and Software. — 2024. — Vol. 208. — Art. no. 111902. — DOI: 10.1016/j.jss.2023.111902.

15. Hussain S., Nadeem M., Baber J., Hamdi M., Rajab A., Al Reshan M. S., Shaikh A. Vulnerability detection

in Java source code using a quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction // Scientific Reports. — 2024. — Vol. 14, no. 1. — Art. no. 7406. — DOI: 10.1038/s41598-024-56871-z.

16. McCully G. A., Hastings J. D., Xu S. TEDVIL: Leveraging Transformer-Based Embeddings for Vulnerability Detection in Lifted Code // IEEE Access. — 2025. — Vol. 13. — P. 76894—76913. — DOI: 10.1109/ACCESS.2025.3565980.

17. Gui B., Song W., Xiong H., Huang J. Automated Use-After-Free Detection and Exploit Mitigation: How Far Have We Gone? // IEEE Transactions on Software Engineering. — 2022. — Vol. 48, no. 11. — P. 4569—4589. — DOI: 10.1109/TSE.2021.3121994.

References

1. Hnatyuk S., Sydorenko V., Skurativskyi A. Cybersecurity requirements management model in software implementation / S. Hnatyuk, V. Sydorenko, A. Skurativskyi // Cybersecurity: Education, Science, Technique. — 2025. — Issue 4. — P. 715—726. — DOI: 10.28925/2663-4023.2025.28.841.

2. Yongxu H., Ying Z., Pengzhi X., Zhen G. Research on Buffer Overflow Vulnerability Mining Method Based on Deep Learning // 2024 2nd International Conference on Big Data and Privacy Computing (BDPC), Macau, China. — 2024. — P. 28—36. — DOI: 10.1109/BDPC59998.2024.10649293.

3. Al-Mandhari I., AlKalbani A., Al-Abri A. Association Rules for Buffer Overflow Vulnerability Detection Using Machine Learning / I. Al-Mandhari, A. AlKalbani, A. Al-Abri // In: Yang X. S., Sherratt R. S., Dey N., Joshi A. (eds) Proceedings of Eighth International Congress on Information and Communication Technology (ICICT 2023). Lecture Notes in Networks and Systems. — Vol. 696. — Springer, Singapore, 2024. — DOI: 10.1007/978-981-99-3236-8_48

4. Butt M. A., Ajmal Z., Khan Z. I., Idrees M., Javed Y. An In-Depth Survey of Bypassing Buffer Overflow Mitigation Techniques // Applied Sciences. — 2022. — Vol. 12, no. 13. — P. 6702. — DOI: 10.3390/app12136702.

5. Lu G., Ju X., Chen X., Pei W., Cai Z. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning // Journal of Systems and Software. — 2024. — Vol. 212. — DOI: 10.1016/j.jss.2024.112031.

6. Hanif H., Maffeis S. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection // 2022 International Joint Conference on Neural Networks (IJCNN), Padua, Italy. — 2022. — P. 1—8. — DOI: 10.1109/IJCNN55064.2022.9892280.

7. Hin D., Kan A., Chen H., Babar M. A. LineVD: statement-level vulnerability detection using graph neural networks // Proceedings of the 19th International Conference on Mining Software Repositories (MSR ’22). — New York: ACM, 2022. — P. 596—607. — DOI: 10.1145/3524842.3527949.

8. Wu Y., Zou D., Dou S., Yang W., Xu D., Jin H. VulCNN: an image-inspired scalable vulnerability detection system // Proceedings of the 44th International Conference on Software Engineering (ICSE ’22). — New York: ACM, 2022. — P. 2365—2376. — DOI: 10.1145/3510003.3510229.

9. Fu M., Tantithamthavorn C., Le T. et al. AIBugHunter: A Practical tool for predicting, classifying and repairing software vulnerabilities // Empirical Software Engineering. — 2024. — Vol. 29, no. 4. — DOI: 10.1007/s10664-023-10346-3.

10. Li L., Ding S. H. H., Tian Y., Fung B. C. M., Charland P., Ou W., Song L., Chen C. VulANalyzeR: Explainable Binary Vulnerability Detection with Multi-task Learning and Attentional Graph Convolution // ACM Transactions on Privacy and Security. — 2023. — Vol. 26, no. 3. — Article 28. — 25 p. — DOI: 10.1145/3585386.

11. Pereira J. D., Ivaki N., Vieira M. Characterizing Buffer Overflow Vulnerabilities in Large C/C++ Projects // IEEE Access. — 2021. — Vol. 9. — P. 142879—142892. — DOI: 10.1109/ACCESS.2021.3120349.

12. Baradaran S., Heidari M., Kamali A., Mouzarani M. A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes // International Journal of Information Security. — 2023. — Vol. 22, no. 5. — P. 1277—1290. — DOI: 10.1007/s10207-023-00691-1.

13. Rozi M. F., Ban T., Ozawa S., Yamada A., Takahashi T., Inoue D. Securing Code With Context:

Enhancing Vulnerability Detection Through Contextualized Graph Representations // IEEE Access. — 2024. — Vol. 12. — P. 142101—142126. — DOI: 10.1109/ACCESS.2024.3467180.

14. Wang P., Liu S., Liu A., Jiang W. Detecting security vulnerabilities with vulnerability nets // Journal of Systems and Software. — 2024. — Vol. 208. — Art. no. 111902. — DOI: 10.1016/j.jss.2023.111902.

15. Hussain S., Nadeem M., Baber J., Hamdi M., Rajab A., Al Reshan M. S., Shaikh A. Vulnerability detection in Java source code using a quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction // Scientific Reports. — 2024. — Vol. 14, no. 1. — Art. no. 7406. — DOI: 10.1038/s41598-024-56871-z.

16. McCully G. A., Hastings J. D., Xu S. TEDVIL: Leveraging Transformer-Based Embeddings for Vulnerability Detection in Lifted Code // IEEE Access. — 2025. — Vol. 13. — P. 76894—76913. — DOI: 10.1109/ACCESS.2025.3565980.

17. Gui B., Song W., Xiong H., Huang J. Automated Use-After-Free Detection and Exploit Mitigation: How Far Have We Gone? // IEEE Transactions on Software Engineering. — 2022. — Vol. 48, no. 11. — P. 4569—4589. — DOI: 10.1109/TSE.2021.3121994.

Завідувачу кафедри кібербезпеки
к.т.н., доц. Кльоцу Ю.П.
здобувача вищої освіти
Гордєєва Богдана Віталійовича
студента ФІТ, 2 курсу, групи КБЗІм-24-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності у Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів академічної відповідальності, ознайомлений. Про використання спеціалізованих програмних засобів (СПЗ) StrikePlagiarism та Anti-Plagiarism для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений. Надаю університету право на передачу моєї роботи для обробки та збереження в базах даних СПЗ і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються СПЗ.

Також надаю свою згоду на обробку й збереження університетом моєї роботи в Інституційному репозитарії Хмельницького національного університету.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

1.12.2025
дата

підпис

Anti-Plagiarism (UA) v-15.281 Educational

The maximum coincidence with one document 1.0%

Dictionaries check: en_US, ru_RU, ua_UA. **Errors in the documents: 7%**

	Document		Sum coincidence on the DB	
ID: 251493 Title: Метод виявлення вразливостей типу переповнення буферу в програмному забезпеченні Added in a DB: 2025-12-03 Authors: Гордєєв Богдан Віталійович Heads: Касянчук М.М. Consultants: Opponents:	Symbols	Lexemes	Symbols	Lexemes
	130812	897	2286 (2%)	28 (3%)

Plagiarism sources

ID	Description	Plagiarism presence in the document	
		Symbols	Lexemes

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Гордєєв Богдан Віталійович

Співавтор:

Назва: Метод виявлення вразливостей типу переповнення буферу в програмному забезпеченні

Науковий керівник: Касянчук Михайло Миколайович

Підрозділ: Кафедра кібербезпеки

Коефіцієнт подібності 1:3.4%

Коефіцієнт подібності 2:0.7%

Мікропробіли: 0

Заміна букв: 0

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2025-12-03 23:05:40.0

Після аналізу Звіту подібності констатую наступне:

☐ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

Дата експерт

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ КАФЕДРИ КІБЕРБЕЗПЕКИ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Назва кваліфікаційної роботи: Метод виявлення вразливостей типу переповнення буферу в програмному забезпеченні
Автор: Гордєєв Богдан Віталійович
Освітня програма: Кібербезпека та захист інформації
Рівень вищої освіти: другий (магістерський)
Спеціальність: 125 – Кібербезпека та захист інформації
Науковий керівник: Михайло КАСЯНЧУК, докт. техн. наук, професор
На основі аналізу кваліфікаційної роботи на дотримання вимог академічної доброчесності (у т.ч. відсутності ознак академічного плагіату) з урахуванням результатів перевірки роботи спеціалізованим програмним засобом(ами) комісія зробила такий висновок:

№	Висновок	Позначка про відповідність
1	Ознаки академічного плагіату	
1.1	Запозичення, виявлені в роботі, є законними і не є академічним плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних, якщо потрібно). Робота приймається до захисту	відповідає
1.2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована	
1.3	Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота може бути допущена до захисту після того як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат	
1.4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
2	Інші види порушень академічної доброчесності	

Підтвердження:

Оригінальність тексту роботи за результатами перевірки системою StrikePlagiarism складає 96,64%, оригінальність тексту роботи за результатами перевірки системою Anti-Plagiarism складає 99,0%

Згідно з Положенням про систему забезпечення академічної доброчесності у ХНУ (https://khmnu.edu.ua/wp-content/uploads/normatyvni-dokumenty/polozhennya/pro-systemu-zabezpechennya-akademichnoyi-dobrochesnosti.pdf, Додаток В) кваліфікаційна робота, виконана за освітньо-професійною програмою, кількісні показники рівня унікальності тексту у відсотках до загального обсягу матеріалу в якій складає 75-100 %, визнається роботою з високим рівнем унікальності тексту: «Текст вважається унікальним і не потребує додаткових дій щодо запобігання неправомірним запозиченням».

Дата: 3.12.2025

Завідувач кафедри кібербезпеки _____ Юрій КЛЬОЦ

Гарант освітньої програми _____ Віра ТІТОВА

Керівник кваліфікаційної роботи _____ Михайло КАСЯНЧУК

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

освітнього ступеня «магістр»

Студент Гордєєв Богдан Віталійович

Тема Метод виявлення вразливостей типу переповнення буферу в програмному забезпеченні

Спеціальність 125 – Кібербезпека та захист інформації

Обсяг кваліфікаційної роботи освітньо-кваліфікаційного рівня «бакалавр»:

кількість листів креслень – ; кількість сторінок записки 89 .

1. Короткий зміст роботи та прийнятих рішень У кваліфікаційній роботі розроблено метод виявлення вразливостей типу переповнення буфера, що базується на інтеграції символьного виконання, Q-навчання та побудови дерева обмежень. Автором проведено огляд сучасних методів виявлення програмних вразливостей, проаналізовано підходи статичного, динамічного та гібридного аналізу. Розроблено математичну модель переповнення буфера та алгоритми симулятивного покриття коду, генерації тестових даних і виявлення потенційно небезпечних станів. Виконано експериментальне дослідження, результати якого демонструють перевагу запропонованого методу над класичними інструментами аналізу безпеки. Робота містить порівняльний аналіз, оцінку точності, стабільності та швидкодії запропонованого підходу.

2. Висновок про відповідність кваліфікаційної роботи завданню У роботі повністю виконано завдання, визначене у вихідних даних: проведено аналіз методів виявлення вразливостей, обґрунтовано вибір підходу з використанням символьного виконання та Q-навчання, розроблено математичну модель та алгоритми, виконано експериментальну перевірку та порівняльний аналіз результатів. Кваліфікаційна робота відповідає усім вимогам до магістерських досліджень.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У вступі обґрунтовано актуальність проблеми, визначено об'єкт, предмет, мету та завдання дослідження. Перший розділ містить системний аналіз методів виявлення програмних вразливостей. Другий розділ присвячено розробленню методу виявлення переповнення буфера: сформовано математичну модель, розроблено алгоритми симулятивного покриття та Q-навчання, побудовано дерево обмежень та описано процедури генерації вхідних даних. У третьому розділі наведено опис експериментального середовища, тестових програм, результати двох експериментів, аналіз достовірності, обмежень та загроз валідності моделі. Автор продемонстрував високий рівень володіння актуальними методами кібербезпеки.

4. Позитивні сторони роботи Робота є актуальною та має високу наукову і практичну цінність. Розроблений метод зменшує кількість хибнопозитивних спрацювань та покращує точність виявлення переповнень буфера при використанні сучасних методів машинного навчання та символьного виконання.

5. Негативні сторони роботи Експериментальні дослідження реалізовано на обмеженому наборі тестових програм. Потребує ширшого розгляду можливостей інтеграції методу у промислові системи безпеки.

6. Оцінка графічного оформлення та пояснювальної записки роботи

7. Відгук про роботу в цілому Кваліфікаційна робота є самостійним завершеним науково-практичним дослідженням. Викладений матеріал є логічним, послідовним та добре аргументованим. Автор продемонстрував високий рівень підготовки, здатність застосовувати сучасні технології кібербезпеки та проводити експериментальні дослідження. Робота має наукову новизну та практичну цінність і може бути використана при розробці інструментів автоматизованого виявлення вразливостей.

8. Інші зауваження

9. Оцінка кваліфікаційної роботи Враховуючи всі позитивні та негативні сторони представленої кваліфікаційної роботи, можна зробити висновок, що вона заслуговує «добре» 83 бали.

РЕЦЕНЗЕНТ (прізвище, ім'я, по батькові, посада, місце роботи)
Бойко Юлій Миколайович,
професор кафедри телекомунікацій, медійних та інтелектуальних технологій, доктор технічних наук, професор

«4» грудня 2025