Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Міністерство освіти і науки

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Міністерство освіти і науки

Кваліфікаційна наукова
праця на умовах рукопису

КУБ'ЮК ЄВГЕНІЙ ЮРІЙОВИЧ

УДК 004.021:004.023:004.93

ДИСЕРТАЦІЯ

АНАЛІЗ ПРОГРАМНОГО КОДУ З ВИКОРИСТАННЯМ ГІБРИДНОГО МЕТОДУ ПОШУКУ ТА КЛАСИФІКАЦІЇ ВРАЗЛИВОСТЕЙ

122 – Комп'ютерні науки
Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії
Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело
_____Є. Ю. Куб'юк

Науковий керівник: Кисельов Геннадій Дмитрович, кандидат технічних наук, доцент

Київ – 2024

АНОТАЦІЯ

Куб'юк Є.Ю. Аналіз програмного коду з використанням гібридного методу пошуку та класифікації вразливостей. - Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 «Комп'ютерні науки». – Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», 2024.

Актуальність теми. У контексті стрімкого розвитку інформаційних технологій та цифровізації суспільства, проблематика кібербезпеки набуває особливої гостроти. Згідно зі статистикою, у 2022 році кількість кібератак зросла на 42% порівняно з попереднім роком, а фінансові збитки сягнули $6 трлн.

Основною причиною такої ситуації є зростання кількості вразливостей у програмному забезпеченні, які активно використовуються зловмисниками. Зокрема, у Національній базі даних вразливостей США (NVD) за 2022 було зареєстровано понад 25 000 нових вразливостей, що на 20% більше ніж у 2021. Більше 75% успішних кібератак базувалися на експлуатації вже відомих вразливостей.

Це свідчить про гостру необхідність розробки ефективних заходів щодо своєчасного виявлення та усунення вразливостей у програмному коді. Одним з найбільш перспективних напрямів вирішення цієї проблеми є створення автоматизованих систем аналізу коду на основі технологій штучного інтелекту.

Такі системи здатні ефективно виявляти приховані дефекти безпеки на етапах розробки ПЗ, що дозволяє знизити ризики та підвищити захищеність. Їх інтеграція у процеси розробки поліпшить дотримання практик безпечного програмування та скоротить час на тестування безпеки.

Отже, створення інноваційних методів та засобів автоматизації аналізу безпеки програмного коду є вкрай актуальним завданням для підвищення рівня кіберзахисту сучасних програмних систем.

Мета і завдання дослідження. Метою дисертаційного дослідження є розробка та удосконалення методів виявлення та класифікації вразливостей в програмному коді з використанням нейронних мереж та методів виявлення подібності коду.

Основні завдання дослідження включають:

- Дослідити методи виявлення вразливостей в програмному коді.
- Дослідити ефективність методів виявлення подібності коду в задачі класифікації вразливостей.
- Дослідити моделі представлення програмного коду.
- Розробити метод побудови проміжного представлення програмного коду для подальшого аналізу на предмет вразливостей.
- Розробити метод виявлення вразливостей на основі нейронних мереж.
- Розробити метод класифікації вразливостей на основі методу виявлення подібності коду.
- Виконати програмну реалізацію системи аналізу програмного коду, з використанням гібридного методу пошуку та класифікації вразливостей програмного коду.
- Провести обчислювальні експерименти та оцінку розробленої системи.
- Продемонструвати практичну цінність розробленої системи на прикладі впровадження в процеси життєвого циклу розробки програмного забезпечення.

Об’єктом дослідження є процес пошуку та класифікації вразливостей безпеки в програмному коді.

Предметом дослідження виступають методи, алгоритми та структури даних для ефективного аналізу програмного коду на предмет вразливостей безпеки.

Методи дослідження. В роботі застосовуються наступні методи дослідження: абстрактно-логічний аналіз проблеми, експерименти, моделювання, аналіз даних, порівняння. Зазначені методи були обрані з огляду на поставлені мету та задачі дослідження, а також на практичність їх застосування при вирішенні проблеми автоматичного аналізу програмного коду на предмет вразливостей безпеки.

Наукова новизна отриманих результатів.

Вперше запропоновано гібридний метод аналізу програмного коду, що поєднує методи глибокого навчання та методи виявлення подібності коду для пошуку та класифікації вразливості в коді, який дозволяє ефективно виконувати пошук вразливостей в коді, а також класифікувати з високою точністю знайдені вразливості.

Отримав подальший розвиток метод побудови проміжного представлення програмного коду у вигляді кодового гаджету, який відрізняється від існуючих методів наявністю обмеження по розміру локального контексту відносно ключової точки, що дозволило зменшити результуючий розмір кодових гаджетів та підвищити точність класифікації при подальшому аналізі нейронною мережею.

Вперше запропоновано метод класифікації вразливостей в програмному коді з використанням ковзного хешування абстрактного синтаксичного дерева, який відрізняється від існуючих методів тим що використовує метод виявлення подібності коду для ефективної класифікації вразливостей без необхідності використання навчальної вибірки великого об’єму.

Практичне значення отриманих результатів. Розроблений метод дозволяє знаходити вразливості в програмному коді, написаному на мові

C/C++, а також класифікувати тип знайденої вразливості, що дозволяє спростити процес пріоретизації та виправлення знайдених вразливостей для розробників програмних продуктів.

Технологія пошуку та класифікації вразливостей в програмному коді успішно впроваджена в процеси розробки IT компанії та використовується для аналізу програмних продуктів на предмет вразливостей у якості сервісу аналізу коду. Також, командний інтерфейс системи дозволяє використовувати її у якості утиліти для аналізу коду, без необхідного розгортання у вигляді сервісу, що дозволяє спеціалістам з кібербезпеки використовувати даний інструмент в якості рекомендаційної системи в рамках процесів оцінки безпеки програмного коду.

Розроблений метод було протестовано на реальних проектах з відкритим вихідним кодом. Зокрема, за допомогою розробленої системи вдалося знайти вразливість в проекті з відкритим вихідним кодом - Microsoft Terminal.

Публікації. За результатами дисертаційного дослідження опубліковано 5 наукових праць, з яких 4 статті у наукових фахових виданнях України (3 з яких входять до міжнародних наукометричних баз) та 1 публікація в збірнику матеріалів конференції.

Результати дослідження. Розроблено програмне забезпечення системи аналізу коду, що реалізує запропонований гібридний метод. Експериментально підтверджено ефективність системи у задачах пошуку та класифікації вразливостей в програмному коді.

Структура та обсяг роботи. Дисертація складається зі вступу, 4 розділів, висновків, списку використаних джерел. Повний обсяг дисертації становить 140 сторінок. Робота містить 27 рисунків та 9 таблиць. Список використаних джерел налічує 81 найменувань.

Основний зміст роботи. У вступі обґрунтовано актуальність теми, сформульовано мету та завдання дослідження, визначено об'єкт, предмет та

методи дослідження, розкрито наукову новизну та практичне значення отриманих результатів.

У **першому розділі** проведено аналіз сучасного стану у сфері кібербезпеки та тенденцій зростання кількості вразливостей у програмному забезпеченні. Розглянуто ключові стандарти та методології забезпечення безпеки ПЗ, зокрема Microsoft SDL, OWASP та ISO 27034.

Проаналізовано сучасні виклики, пов’язані зі зростанням складності систем та появою нових технологій. Розглянуто існуючі стандарти та методології забезпечення безпеки ПЗ. Досліджено можливості методів штучного інтелекту, зокрема машинного навчання, для автоматизації процесів аналізу безпеки коду.

Розглянуто різні архітектури нейронних мереж та їх застосування для детекції вразливостей. Проаналізовано переваги та недоліки підходів.

У **другому розділі** сформульовано постановку задачі дослідження та побудовано її математичну модель. Формалізовано функції перетворення даних на вході та виході системи. Розглянуто різні моделі подання програмного коду та обґрунтовано доцільність використання абстрактних синтаксичних дерев.

Запропоновано власний гібридний підхід аналізу коду, що поєднує переваги глибокого навчання та алгоритмів пошуку подібності. Наведено детальний опис архітектури та математичних моделей складових компонент системи, зокрема нейромережевої моделі та модуля ковзного хешування AST.

У **третьому розділі** представлено розробку ключових модулів запропонованої системи аналізу програмного коду з використанням гібридного методу пошуку та класифікації вразливостей.

Описано удосконалений метод формування проміжних представлень коду на базі кодових гаджетів та їх подальша векторизація. Наведено

архітектуру та алгоритми функціонування моделі глибокого навчання на основі BLSTM та модуля ковзного хешування AST.

Також деталізовано розроблений метод класифікації вразливостей на основі ковзного хешування вузлів AST та порівняння з базою еталонів. Описано процедуру формування бази знань хешів вразливого коду та метод визначення оптимального вікна хешування AST.

Четвертий розділ присвячено експериментальним дослідженням розробленої системи з використанням спеціалізованих наборів даних та кодових баз реальних проектів. Представлено аналіз отриманих результатів та порівняння з існуючими аналогами за критеріями якості та продуктивності. Обґрунтовано практичну цінність системи та можливості інтеграції в процеси безперервної інтеграції та доставки ПЗ.

Модель пошуку вразливостей з використанням нейронних мереж продемонструвала 94.1% точності, а модель класифікації вразливостей з використанням ковзного хешування AST для 40 класів вразливостей - 51.1% точності. Обґрунтовано можливі шляхи вдосконалення системи.

Висновки. У дисертаційній роботі вирішено актуальне науково-прикладне завдання розробки методів та програмних засобів для автоматизації процесів аналізу програмного коду на предмет вразливостей безпеки:

1. Запропоновано гібридний підхід до аналізу коду на основі поєднання методів глибокого навчання та алгоритмів пошуку подібності коду. Розроблено відповідну систему, що реалізує даний підхід.
2. Розроблено метод побудови проміжного представлення програмного коду у вигляді кодового гаджету, що дозволило підвищити ефективність подальшого аналізу за допомогою нейромереж.
3. Запропоновано метод класифікації вразливостей на основі ковзного хешування AST, що демонструє переваги за швидкодією у порівнянні з існуючими RNN-моделями.

4. Експериментально підтверджено ефективність розробленої системи, зокрема 94.1% точність пошуку вразливостей та 51.1% багатокласової точності їх класифікації.

Ключові слова: аналіз програмного коду, пошук вразливостей, класифікація вразливостей, нейронні мережі, ковзне хешування, машинне навчання, глибоке навчання, кібербезпека, виявлення аномалій, абстрактне синтаксичне дерево, база знань.

ABSTRACT

Yevhenii Kubiuk. Source code analysis using a hybrid method of detecting and classifying vulnerabilities. Qualifying scientific work in manuscript copyright.

Thesis for the degree of Doctor of Philosophy in specialty 122 "Computer Science". – National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", 2024.

Relevance of the topic. Relevance of the topic. In the context of rapid advancements in information technology and the digitization of society, cybersecurity issues become particularly acute. According to statistics, in 2022, the number of cyber attacks increased by 42% compared to the previous year, resulting in financial losses reaching $6 trillion.

The main reason for this situation is the growing number of vulnerabilities in software actively exploited by malicious actors. Specifically, the U.S. National Vulnerability Database (NVD) recorded over 25,000 new vulnerabilities in 2022, a 20% increase compared to 2021. More than 75% of successful cyber attacks were based on exploiting already known vulnerabilities.

This highlights the urgent need for the development of effective measures for timely detection and elimination of vulnerabilities in software code. One of the most promising solutions to this problem is the creation of automated code analysis systems based on artificial intelligence technologies.

Such systems are capable of efficiently detecting hidden security defects during the software development stages, reducing risks, and enhancing overall security. Integrating them into development processes will improve adherence to secure programming practices and reduce security testing time.

Therefore, the creation of innovative methods and tools for automating the analysis of security in software code is an extremely relevant task for enhancing the cybersecurity of modern software systems.

Purpose and objectives. The purpose of the dissertation research is to develop and improve methods for detecting and classifying vulnerabilities in source code using neural networks and code similarity detection methods.

The main objectives of the research include:

- Investigate methods for detecting vulnerabilities in source code.
- Investigate the effectiveness of code similarity detection methods in the task of vulnerability classification.
- Investigate code representation models.
- Develop a method for constructing an intermediate representation of source code for further analysis for vulnerabilities.
- Develop a method for detecting vulnerabilities based on neural networks.
- Develop a method for classifying vulnerabilities based on the code similarity detection method.
- Implement software for a source code analysis system using a hybrid method for searching and classifying source code vulnerabilities.
- Conduct computational experiments and evaluate the developed system.
- Demonstrate the practical value of the developed system using the example of implementation in software development life cycle processes.

The object of the research is the process of searching for and classifying security vulnerabilities in software code.

The subject of the research is methods, algorithms and data structures for efficient analysis of source code for security vulnerabilities.

Research methods. The following research methods are used: abstract-logical analysis of the problem, experiments, modeling, data analysis, comparison. These methods were chosen in view of the set purpose and objectives of the study, as well as the practicality of their application in solving the problem of automatic analysis of source code for security vulnerabilities.

Scientific novelty of the results. For the first time, a hybrid method for analyzing program code is proposed, combining deep learning methods and code similarity detection methods to search for and classify vulnerabilities in code. This method allows for efficient vulnerability detection in code and high-precision classification of the discovered vulnerabilities.

The method of constructing an intermediate representation of program code in the form of a code gadget has been further developed. It differs from existing methods by the presence of a restriction on the size of the local context relative to the key point, which allowed reducing the resulting size of code gadgets and improving classification accuracy during subsequent analysis by a neural network.

For the first time, a method for classifying vulnerabilities in program code using sliding hashing of the abstract syntax tree is proposed. It differs from existing methods in that it uses a code similarity detection method for efficient vulnerability classification without the need for a large training dataset.

Practical value of the results obtained. The proposed method allows detecting vulnerabilities in C/C++ code, as well as classifying the type of vulnerability found, which simplifies the process of prioritizing and fixing discovered vulnerabilities for software developers.

The technology has been successfully integrated into the software development processes of an IT company and is used to analyze software products for vulnerabilities as a code analysis service. Also, the command-line interface of the system allows using it as a utility to analyze code without deploying it as a service, which allows cybersecurity professionals to use this tool as a recommendation system in software code security assessment processes.

The proposed method has been tested on real open source projects. In particular, the developed system helped to find a vulnerability in the open source Microsoft Terminal project.

Publications. The results of the dissertation research have led to the publication of 5 scientific works, including 4 articles in specialized scientific journals in Ukraine (3 of which are indexed in international scientific databases) and 1 publication in a conference proceedings compilation.

Research Results. The developed software for code analysis implements the proposed hybrid method. The effectiveness of the system in tasks such as vulnerability detection and classification in software code has been experimentally confirmed.

Structure and Scope of the Work. The dissertation comprises an introduction, 4 chapters, conclusions, and a list of references. The total volume of the dissertation is 140 pages, including 27 figures and 9 tables. The list of references includes 81 sources.

Main Content of the Work. The introduction justifies the relevance of the topic, formulates the purpose and tasks of the research, defines the object, subject, and research methods, and highlights the scientific novelty and practical significance of the obtained results.

In the first chapter, an analysis of the current state of cybersecurity and the trends in the increasing number of vulnerabilities in software is conducted. Key standards and methodologies for software security, including Microsoft SDL, OWASP, and ISO 27034, are discussed.

Modern challenges related to the complexity of systems and emerging technologies are examined, along with existing standards and methodologies for software security. The possibilities of artificial intelligence methods, particularly machine learning, for automating code security analysis processes are investigated.

Various neural network architectures and their application for vulnerability detection are explored, along with an analysis of the advantages and disadvantages of these approaches.

The second chapter formulates the research problem, builds its mathematical model, and formalizes the data transformation functions at the input and output of the system. Different models for representing software code are considered, justifying the use of abstract syntax trees.

A hybrid code analysis approach that combines the benefits of deep learning and similarity search algorithms is proposed. The architecture and mathematical models of the components of the system, including the neural network model and the abstract syntax tree sliding hash module, are detailed.

The third chapter presents the development of key modules of the proposed code analysis system using the hybrid method for vulnerability search and classification.

An improved method for generating intermediate code representations based on code gadgets and their subsequent vectorization is described. The architecture and functioning algorithms of the deep learning model based on BLSTM and the abstract syntax tree sliding hash module are provided.

The developed vulnerability classification method based on sliding hash of AST nodes and comparison with a reference database is detailed. The procedure for creating a knowledge base of vulnerable code hashes and the method for determining the optimal sliding hash window for AST are also described.

The fourth chapter is dedicated to experimental research of the developed system using specialized datasets and codebases of real projects. The analysis of

the obtained results is presented, comparing them with existing analogs in terms of quality and performance criteria. The practical value of the system and its integration possibilities into continuous integration and software delivery processes are justified.

The vulnerability search model using neural networks demonstrated an accuracy of 94.1%, while the vulnerability classification model using AST sliding hash for 40 vulnerability classes showed an accuracy of 51.1%. Possible ways to improve the system are discussed.

Conclusions. The dissertation successfully addresses the relevant scientific and applied task of developing methods and software tools for automating the processes of analysing software code for security vulnerabilities:

1. A hybrid approach to code analysis, combining deep learning methods and code similarity search algorithms, has been proposed. A corresponding system implementing this approach has been developed.
2. A method for constructing an intermediate representation of the software code in the form of a code gadget has been developed, enhancing the efficiency of subsequent analysis using neural networks.
3. A vulnerability classification method based on sliding hash of Abstract Syntax Trees (AST) has been proposed, demonstrating advantages in terms of speed compared to existing RNN models.
4. The effectiveness of the developed system has been experimentally confirmed, achieving a 94.1% accuracy in vulnerability detection and a 51.1% multi-class accuracy in their classification.

Keywords: code analysis, vulnerability detection, vulnerability classification, neural networks, sliding hashing, machine learning, deep learning, cybersecurity, anomaly detection, abstract syntax tree, knowledge base.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Статті у наукових фахових виданнях України

[1] Kubiuk, Y., Chernousov, A., Savchenko, A., Osadchyi, S., Kostenko, Y., & Likhomanov, D. (2019). Deep learning based automatic software defects detection framework. (Здобувачем запропоновано архітектуру системи, розроблено нейромережеву модель пошуку вразливостей, проведено експерименти)

[2] Kubiuk, Y., & Kyselov, G. (2021). Comparative analysis of approaches to source code vulnerability detection based on deep learning methods. Technology audit and production reserves, 3(2), 59. (Здобувачем проведено аналіз існуючих методів детекції вразливостей, сформульовано вимоги до удосконалення підходів, запропоновано концепцію поєднання методів)

[3] Kaliuzhna, T., & Kubiuk, Y. (2022). Analysis of machine learning methods in the task of searching duplicates in the software code. Technology audit and production reserves, 4(2 (66)), 6-13. (Здобувач приймав участь у проведенні аналізу ефективності методів машинного навчання для пошуку дублікатів коду, а також визначав експерименти, що мають бути проведені при вирішення задач пошуку подібності коду та пошуку вразливостей)

[4] Kubiuk, Y., & Kyselov, G. (2023). Development of an algorithm for code clone detection in source code based on abstract syntax tree. Technology audit and production reserves, 4(2 (72)), 33-36. (Здобувачем запропоновано метод детекції клонів коду на основі ковзного хешування AST та алгоритм його роботи, проведено експерименти та проаналізовано результати)

Статті у неперіодичний збірниках наукових праць:

[5] Черноусов, А. В., Савченко, А. Ю., & Куб’юк, Є. Ю. Методи виявлення помилок безпеки в програмному забезпеченні на основі глибинного навчання, XVII Всеукраїнська науково-практична конференція студентів, аспірантів та молодих вчених «Теоретичні і прикладні проблеми фізики, математики та інформатики» (Україна, м. Київ, 25-26 квітня 2019 р.) : матеріали конференції. – Київ : КПІ ім. Ігоря Сікорського, 2019. (Здобувачем досліджено ефективність методів глибинного навчання для детекції дефектів ПЗ, проаналізовано архітектури нейромереж, розроблено рекомендації щодо застосування)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AST – Abstract syntax tree (абстрактне синтаксичне дерево)

RNN – Recurrent neural network (рекурентна нейронна мережа)

CNN – Convolutional neural network (згорткова нейронна мережа)

GRU – Gated recurrent unit (вентильний рекурентний вузол)

LSTM – Long short-term memory (довга короткочасна пам’ять)

BLSTM – Bidirectional long short-term memory (двонаправлена довга короткочасна пам’ять)

DBN – Deep belief networks (глибока мережа вірувань)

NVD – National vulnerability database (національна база даних вразливостей)

SARD – Software assurance reference dataset (набір даних для забезпечення якості програмного забезпечення)

NIST – National institute of standards and technology (національний інститут стандартів та технологій)

CVE – Common vulnerabilities and exposures (загальні вразливості та викриття)

CWE – Common weakness enumeration (загальна таксономія вразливостей)

SDL – Security development lifecycle (Життєвий цикл розробки заходів безпеки)

OWASP – Open worldwide application security project (всесвітньо відкритий проект безпеки додатків)

CFG – Control flow graph (граф потоку керування)

PDG – Program dependency graph (граф залежностей програми)

JSON – Java script object notation (об'єктна нотація JavaScript)

XML – Extended markup language (розширена мова розмітки)

UML – Unified modelling language (уніфікована мова моделювання)

TF-IDF – Term frequency – inverse document frequency (частота термінів – інвертована частота документів)

DFS – Depth-first search (пошук в глибину)

GNN – Graph neural network (графова нейронна мережа)

CI – Continuous integration (безперервна інтеграція)

CD - Continuous delivery (безперервна доставка)

BoW – Bag of words (мішок слів)

САПК – Система аналізу програмного коду

ШІ – Штучний інтелект

ПЗ – Програмне забезпечення

СПИСОК РИСУНКІВ

ВСТУП

Актуальність дослідження. У контексті стрімкого розвитку інформаційних технологій, питання кібербезпеки набуває все більшої гостроти. Згідно зі статистикою, у 2022 році кількість кібератак зросла на 42% порівняно з попереднім роком, а фінансові збитки сягнули $6 трлн [1]. Основною причиною такої ситуації є зростання кількості вразливостей у програмному забезпеченні, які активно використовуються зловмисниками.

Зокрема, як продемонстровано на рис 1.1**Помилка! Джерело посилання не знайдено.**, в Національній базі даних вразливостей США (NVD) за 2022 було зареєстровано понад 25 000 вразливостей, що на 20% більше ніж у 2021[2]. Причому більше 75% успішних кібератак базувалися на експлуатації відомих вразливостей [3]. Це свідчить про гостру необхідність розробки ефективних заходів щодо їх своєчасного виявлення та усунення.

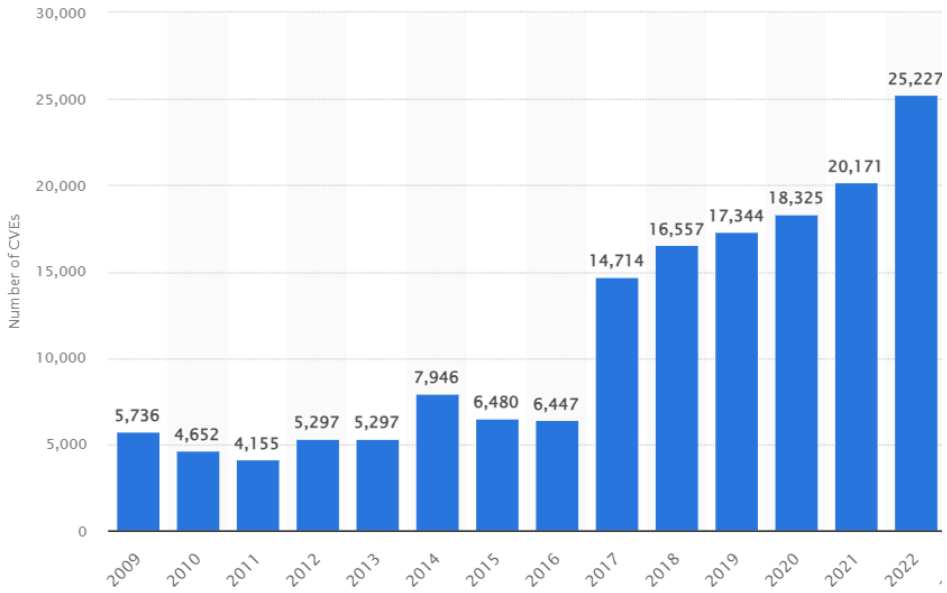

Рис. 1.1 - Динаміка кількості зареєстрованих вразливостей за роками

Одним з найбільш перспективних напрямів вирішення цієї проблеми є створення автоматизованих систем аналізу програмного коду. Завдяки можливостям технологій штучного інтелекту, такі системи здатні ефективно виявляти приховані вразливості на ранніх стадіях розробки ПЗ, що дозволяє суттєво знизити ризики та підвищити захищеність.

Мета та завдання дослідження. Метою даного дисертаційного дослідження є розробка та удосконалення методів виявлення та класифікації вразливостей в програмному коді програмних продуктів з використанням нейронних мереж та методів виявлення подібності коду.

Для досягнення поставленої мети необхідно вирішити наступні **задачі дослідження:**

- Дослідити методи виявлення вразливостей в програмному коді.
- Дослідити ефективність методів виявлення подібності коду в задачі класифікації вразливостей.
- Дослідити моделі представлення програмного коду.
- Розробити метод побудови проміжного представлення програмного коду для подальшого аналізу на предмет вразливостей.
- Розробити метод виявлення вразливостей на основі нейронних мереж.
- Розробити метод класифікації вразливостей на основі методу виявлення подібності коду.
- Виконати програмну реалізацію системи аналізу програмного коду, з використанням гібридного методу пошуку та класифікації вразливостей програмного коду.
- Провести обчислювальні експерименти та оцінку розробленої системи.

- Продемонструвати практичну цінність розробленої системи на прикладі впровадження в процеси життєвого циклу розробки програмного забезпечення.

Об’єктом дослідження є процес пошуку та класифікації вразливостей безпеки в програмному коді.

Предметом дослідження виступають методи, алгоритми та структури даних для ефективного аналізу програмного коду на предмет вразливостей безпеки.

Методи дослідження. В роботі застосовуються наступні методи дослідження: абстрактно-логічний аналіз проблеми, експерименти, моделювання, аналіз даних, порівняння. Зазначені методи були обрані з огляду на поставлені мету та задачі дослідження, а також на практичність їх застосування при вирішенні проблеми автоматичного аналізу програмного коду на предмет вразливостей безпеки.

Наукова новизна отриманих результатів.

Вперше запропоновано гібридний метод аналізу програмного коду, що поєднує методи глибокого навчання та методи виявлення подібності коду для пошуку та класифікації вразливості в коді, який дозволяє ефективно виконувати пошук вразливостей в коді, а також класифікувати з високою точністю знайдені вразливості.

Отримав подальший розвиток метод побудови проміжного представлення програмного коду у вигляді кодового гаджету, який відрізняється від існуючих методів наявністю обмеження по розміру локального контексту відносно ключової точки, що дозволило зменшити результуючий розмір кодових гаджетів та підвищити точність класифікації при подальшому аналізі нейронною мережею.

Вперше запропоновано метод класифікації вразливостей в програмному коді з використанням ковзного хешування абстрактного

синтаксичного дерева, який відрізняється від існуючих методів тим що використовує метод виявлення подібності коду для ефективної класифікації вразливостей без необхідності використання навчальної вибірки великого об’єму.

Практичне значення отриманих результатів. Розроблений метод дозволяє знаходити вразливості в програмному коді, написаному на мові C/C++, а також класифікувати тип знайденої вразливості, що дозволяє спростити процес пріоретизації та виправлення знайдених вразливостей для розробників програмних продуктів.

Технологія успішно впроваджена в процеси розробки IT компанії та використовується для аналізу програмних продуктів на предмет вразливостей у якості сервісу аналізу коду. Також, командний інтерфейс системи дозволяє використовувати її у якості утиліти для аналізу коду, без необхідного розгортання у вигляді сервісу, що дозволяє спеціалістам з кібербезпеки використовувати даний інструмент в якості рекомендаційної системи в рамках процесів оцінки безпеки програмного коду.

Розроблений метод було протестовано на реальних проектах з відкритим вихідним кодом. Зокрема, за допомогою розробленої системи вдалося знайти вразливість в проекті з відкритим вихідним кодом - Microsoft Terminal.

Особистий внесок здобувача. Усі основні результати дисертаційного дослідження, представлені до захисту, одержані автором особисто. У публікаціях у співавторстві здобувачеві належать: дослідження існуючих методів, проектування системи, розробка системи автоматизації процедури оцінювання безпеки програмного коду за допомогою гібридного методу пошуку вразливостей на основі аналізу абстрактного синтаксичого дерева, експериментальне підтвердження ефективності запропонованого методу.

Апробація матеріалів дисертації. Основні положення та отримані наукові результати, що викладені в даній дисертації, пройшли апробацію через публікації в фахових наукових журналах.

Публікації у наукових фахових виданнях України.

[1] Kubiuk, Y., Chernousov, A., Savchenko, A., Osadchyi, S., Kostenko, Y., & Likhomanov, D. (2019). Deep learning based automatic software defects detection framework.

[2] Kubiuk, Y., & Kyselov, G. (2021). Comparative analysis of approaches to source code vulnerability detection based on deep learning methods. Technology audit and production reserves, 3(2), 59.

[3] Kaliuzhna, T., & Kubiuk, Y. (2022). Analysis of machine learning methods in the task of searching duplicates in the software code. Technology audit and production reserves, 4(2 (66)), 6-13.

[4] Kubiuk, Y., & Kyselov, G. (2023). Development of an algorithm for code clone detection in source code based on abstract syntax tree. Technology audit and production reserves, 4(2 (72)), 33-36.

Публікації у неперіодичний збірниках наукових праць.

[5] Черноусов, А. В., Савченко, А. Ю., & Куб'юк, Є. Ю. Методи виявлення помилок безпеки в программному забезпеченні на основі глибинного навчання, XVII Всеукраїнська науково-практична конференція студентів, аспірантів та молодих вчених «Теоретичні і прикладні проблеми фізики, математики та інформатики» (Україна, м. Київ, 25-26 квітня 2019 р.) : матеріали конференції. – Київ : КПІ ім. Ігоря Сікорського, 2019

1 Аналіз сучасного стану та визначення напрямків розвитку автоматизації в процесах аналізу програмного коду

1.1 Вступ

У контексті стрімкого розвитку інформаційних технологій, питання кібербезпеки набуває все більшої гостроти. Згідно зі статистикою [1], у 2022 році кількість кібератак зросла на 42% порівняно з попереднім роком, а фінансові збитки сягнули $6 трлн. Основною причиною такої ситуації є зростання кількості вразливостей у програмному забезпеченні, які активно використовуються зловмисниками.

Зокрема, як продемонстровано в дослідженні [2], у Національній базі даних вразливостей США (NVD) за 2022 було зареєстровано понад 25 000 вразливостей, що на 20% більше ніж у 2021 [3]. Причому більше 75% успішних кібератак базувалися на експлуатації відомих вразливостей [4].

Це свідчить про гостру необхідність розробки ефективних заходів щодо їх своєчасного виявлення та усунення. Одним з найбільш перспективних напрямів вирішення цієї проблеми є створення автоматизованих систем аналізу програмного коду на предмет вразливостей.

Завдяки можливостям технологій штучного інтелекту, такі системи здатні ефективно виявляти приховані вразливості на ранніх стадіях розробки ПЗ, що дозволяє суттєво знизити ризики та підвищити захищеність. Їх інтеграція у процеси розробки дозволить поліпшити дотримання практик безпечного програмування та скоротити час на тестування безпеки.

Автоматизація також надасть можливість швидше реагувати на нові загрози, адаптуючи системи під змінювані умови. Таким чином, розробка

таких рішень є надзвичайно актуальною для підвищення рівня кібербезпеки та стійкості сучасних програмних систем.

У даній главі розглядаються сучасні тенденції та виклики у сфері кібербезпеки. Одним із основних викликів є швидке збільшення кількості нових технологій та засобів програмування, які вносять нові типи вразливостей та загроз. Це вимагає від розробників та фахівців з кібербезпеки постійного оновлення своїх знань та навичок, а також застосування новітніх методів та інструментів для оцінки та забезпечення безпеки.

Крім того, у главі висвітлюються існуючі методології та підходи до автоматизації процесів оцінки безпеки, оцінюються їх переваги та недоліки, а також розглядаються потенційні напрямки подальшого розвитку та вдосконалення.

Важливим аспектом є також інтеграція автоматизованих систем аналізу програмного коду у загальний процес розробки програмного забезпечення, що дозволить забезпечити високий рівень безпеки програмних продуктів на всіх етапах їх життєвого циклу.

Таким чином, дана глава представляє собою аналіз поточного стану автоматизації в оцінці безпеки програмного забезпечення та визначає стратегічні напрямки її розвитку, що є вкрай важливим для розуміння сучасних викликів та можливостей у сфері кібербезпеки.

1.2 Актуальність дослідження

У контексті цифрової епохи стабільність та надійність програмних систем набувають критичного значення внаслідок інтенсифікації загроз, обумовлених кібернетичними вразливостями. Вразливості у цьому аспекті представляють собою дефекти чи слабкі точки у процедурах забезпечення безпеки, дизайні, впровадженні, або внутрішніх контрольних механізмах,

які можуть бути потенційно експлуатовані з метою порушення політики безпеки системи.

Аналітичний звіт, опублікований Comparitech [2], акцентує увагу на тривожній статистиці, що стосується кібернетичних вразливостей, підкреслюючи посилення проблематики в даній сфері. Згідно з даними Національної бази даних вразливостей (NVD) уряду США, загальна кількість зареєстрованих вразливостей перевищує 176,000, що свідчить про множинність ризиків, інтегрованих у кіберсистеми.

Протягом першого кварталу 2022 року, NVD зареєструвала понад 8,051 нових вразливостей, що відображає 25-відсоткове зростання порівняно з аналогічним періодом минулого року. Ця тенденція свідчить про можливе збільшення кількості нових вразливостей, яке може перевищити показники 2021 року, коли було зареєстровано приблизно 22,000 випадків, вказуючи на наростаючу хвилю загроз, якими мають керувати як організації, так і індивідуальні користувачі.

Звіт також акцентує увагу на серйозності цих вразливостей. Наприклад, майже половина вразливостей, що виявлені в системах внутрішньо орієнтованих веб-застосунків, класифікується як високоризикові. Крім того, організації з чисельністю персоналу понад 100 осіб мають більш високу ймовірність зіткнення з високими або критичними ризиками вразливостей, що підтверджує кореляцію між розмірами організації та рівнем ризику вразливості.

Однією з найбільш критичних вразливостей, зареєстрованих у період, була CVE-2021-44228 [5], що мала вплив на утиліту Log4j, активно використовувану в різноманітних програмних проектах. Ця специфічна вразливість надавала можливість зловмисникам виконувати довільний код на скомпрометованих системах, представляючи собою серйозну загрозу безпеці.

Варто зауважити, що деякі вразливості залишаються невирішеними протягом тривалого часу. Наприклад, вразливість CVE-1999-0517 [6], виявлена ще у 1999 році, продовжує бути загрозою високого рівня. Такі обставини акцентують необхідність безперервного управління вразливостями та систематичних оновлень системи.

Звіт [2] також вказує, що понад 11% вразливостей отримали критичний статус, і, що особливо тривожно, 75% кібератак у 2020 році були здійснені шляхом експлуатації вразливостей, відомих протягом принаймні двох років. Ця статистика служить суворим нагадуванням про важливість оперативного виявлення вразливостей та їх своєчасного усунення.

Наукове дослідження наслідків вразливостей в кібербезпеці [10] свідчить про серйозні наслідки для індивідів і підприємств. Ці наслідки можуть бути різними, від незначних незручностей до катастрофічних аварій. Серед них виділяється порушення конфіденційності даних, яке може викликати витік конфіденційної інформації і призвести до фінансових втрат та викрадення ідентичності для фізичних осіб та компаній.

Іншим серйозним наслідком є втрата довіри, що може призвести до негативних наслідків для підприємств. Клієнти можуть сумніватися в проведенні бізнесу з компанією, яка має історію проблем у сфері кібербезпеки, що може призвести до зниження доходу та шкоди репутації.

Порушення вимог щодо кібербезпеки може призвести до правових і регуляторних санкцій, таких як великі штрафи, витрати на юридичні послуги та навіть можливість судового переслідування. Крадіжка інтелектуальної власності та кібершпигунство також можуть призвести до серйозних наслідків для підприємства.

Важливо пам'ятати, що навіть один випадок порушення конфіденційності даних може мати серйозні наслідки, впливаючи на репутацію організації та її фінансовий стан. Довіра набувається складно, але

легко втрачається, і вразливості в кібербезпеці можуть бути фактором, який спричиняє її втрату.

Загалом, вразливості в кібербезпеці мають серйозні наслідки. Тому дуже важливо серйозно ставитися до кібербезпеки і вживати заходів для запобігання вразливостям ще до їх виникнення.

Існують практики та стандарти, такі як Microsoft Security Development Lifecycle (SDL) [7], OWASP [8] або ISO/IEC 27034 [9], які визначають комплексний підхід до управління безпекою на всіх етапах життєвого циклу розробки ПЗ. Вони включають заходи з аналізу вимог щодо безпеки, архітектурного проектування, безпеки написання коду, пенетраційне тестування, а також процеси реагування на вразливості та їх усунення.

Дотримання таких рекомендацій дозволяє значно знизити ризики появи вразливостей та підвищити загальний рівень захищеності систем. Однак їх ефективна реалізація вимагає залучення фахівців з безпеки на всіх етапах розробки та інтеграції процесів аналізу вразливостей у загальний цикл створення ПЗ. Тому актуальним залишається питання автоматизації та оптимізації процедур аналізу безпеки програмного коду для досягнення необхідного рівня захисту.

Зростання кількості програмних вразливостей підкреслює критичну потребу в розробці та впровадженні міцних стратегій кібербезпеки. Організаціям слід надавати абсолютний пріоритет процесам безперервного моніторингу, оперативного реагування на інциденти, а також запровадженню профілактичних заходів для захисту від непередбачувано змінюваного спектру кіберзагроз. Подана статистика не лише акцентує увагу на посиленні викликів у сфері кібербезпеки, але й виступає закликом до дій щодо підвищення рівня кіберстійкості в умовах всеосяжної цифровізації сучасного світу.

Одночасно з цим, існує потреба оптимізувати процеси оцінки безпеки, щоб вони були більш гнучкими, швидкими та інтегрованими.

Сучасний швидкий темп розробки програмного забезпечення вимагає, щоб методи оцінки безпеки були такими ж динамічними, як і загрози, з якими вони борються. Це означає впровадження автоматизованих систем, які можуть проводити постійні перевірки, адаптуватися до нових вразливостей та негайно реагувати на проблеми, що виникають, тим самим зменшуючи час на реагування та відновлення.

Штучний інтелект (ШІ) відкриває нові можливості для автоматизації та оптимізації процесів у сфері кібербезпеки. Завдяки своїм алгоритмам машинного навчання та аналітичним здібностям, ШІ може ефективно виявляти, аналізувати та класифікувати загрози в реальному часі. Це не тільки сприяє швидкому виявленню вразливостей та атак, але й дозволяє системам безпеки адаптуватися до нових та розвиваючихся викликів, тим самим підвищуючи загальний рівень захищеності.

Використання ШІ також може оптимізувати процеси моніторингу та відповіді на інциденти. Автоматизовані системи можуть неперервно моніторити мережевий трафік та поведінку систем, відстежуючи будь-які відхилення від норми, що може свідчити про потенційні загрози. Це значно зменшує навантаження на фахівців кібербезпеки, дозволяючи їм зосередитися на більш складних завданнях та стратегічному плануванні.

Крім того, інтеграція ШІ у системи безпеки сприяє розвитку так званих "самонавчаючихся" систем, здатних аналізувати дані про попередні атаки та вразливості, щоб краще протистояти майбутнім загрозам. Така постійна адаптація та вдосконалення систем безпеки є ключовим фактором у боротьбі зі зростаючою складністю кіберзагроз.

Таким чином, інтенсивний розвиток у сфері кібербезпеки не тільки підкреслює значущість ролі фахівців у цій області, але й акцентує увагу на необхідності постійного вдосконалення процесів, інструментів та підходів, які використовуються для оцінки та забезпечення безпеки в цифровому просторі.

1.3 Опис проблеми

На сьогоднішній день еталоном практик забезпечення безпеки розробки та роботи програмного забезпечення можна вважати методологію "Security Development Lifecycle" (SDL) [7], яку розробила корпорація Microsoft. Ця методологія включає комплекс підходів та методик, які застосовуються для створення програмних продуктів та онлайн-сервісів в умовах високих ризиків у сфері безпеки. Особливістю SDL є інтеграція принципів безпеки та конфіденційності на кожному етапі процесу розробки.

Впровадження концепцій безпечної розробки в уже існуючі процеси може бути складним і витратним, особливо у разі некоректного підходу. Ефективний контроль над впливом цих складнощів можливий завдяки розумінню ключових елементів практик безпечної розробки та приоритетизації заходів залежно від рівня зрілості команди розробників.

Методологія SDL Microsoft передбачає наступні етапи: тренінг, вимоги, проектування, розробка, верифікація, реліз, реагування. Дана методологія включає обов'язкові заходи безпеки, які виконуються на кожному етапі розробки. Схематичне зображення етапів життєвого циклу розробки та практик згідно Microsoft SDL зображено на Рис. 1.2:

Рис. 1.2 - Етапи життєвого циклу розробки ПЗ, згідно MSDL

Однією з ключових фаз методології SDL є фаза розробки. Цей етап має вирішальне значення у визначенні ступеня вразливості розроблюваного програмного продукту. Під час фази розробки, розробники втілюють у життя попередньо сплановані вимоги безпеки та конфіденційності, втілюючи їх у коді програмного продукту. Цей процес включає не лише написання коду, а і його тестування, перевірку на наявність помилок, вразливостей та інших потенційних ризиків.

Саме на етапі розробки реалізуються основні механізми безпеки майбутнього програмного продукту. Недоліки в реалізації можуть призвести до серйозних вразливостей в програмному продукті, які стануть відкритими для зловмисників та можуть бути використані для атак.

Тому, особливу увагу на фазі розробки потрібно приділяти точному дотриманню специфікацій безпеки, а також проведенню ретельних тестувань і перевірок коду, таких як статичний аналіз коду. Статичний аналіз коду включає в себе перевірку коду на відповідність стандартам програмування, пошук шаблонів, які можуть вказувати на помилки або вразливості, та аналіз коду без його виконання. Цей підхід дозволяє ідентифікувати проблеми ще до тестування чи випуску продукту, значно підвищуючи рівень безпеки та надійності програмного забезпечення.

На фазі випуску програмного продукту, статичний аналіз також відіграє важливу роль. Він допомагає впевнитись, що всі виправлення та оновлення, внесені під час розробки, не ввели нових вразливостей та що загальний рівень безпеки програмного продукту відповідає встановленим вимогам.

Для статичного аналізу коду існує значна кількість інструментів, які в більшості базуються на принципі пошуку певних шаблонів у коді. Цей метод дозволяє використовувати статичні аналізатори коду без необхідності додаткових налаштувань, що робить їх зручними у

використанні та ефективними для швидкої ідентифікації типових помилок та вразливостей.

Пошук шаблонів у коді зазвичай зосереджується на виявленні відомих патернів, які можуть вказувати на потенційні проблеми – такі як використання застарілих функцій, відомих вразливостей або небезпечних практик програмування.

Наприклад, інструменти, такі як Checkmarx [12], Google's Code Search[13], Coverity Scan[14], Micro Focus's Static Code Analyzer [15] та Flawfinder[16], використовують методи пошуку за шаблоном для виявлення потенційних проблем з безпекою в програмному забезпеченні. Ці інструменти дуже ефективні у швидкому скануванні великих обсягів коду для виявлення шаблонів, які схожі на відомі вразливості. Цей підхід особливо корисний для початкових сканувань або для інтеграції безпеки на ранні етапи життєвого циклу розробки програмного забезпечення, де швидкий зворотний зв'язок є критичним.

Однак основним обмеженням методів пошуку за шаблоном є їхня тенденція до великої кількості помилкових позитивних результатів[11] [17]. Ця проблема виникає через те, що дані методи часто ідентифікують шаблони коду як вразливі, не враховуючи загальний контекст або конкретні використання коду. В результаті розробники можуть витрачати значний час на ручний перегляд та підтвердження відзначених проблем, що може бути не продуктивно і призводити до затримок у процесі розробки.

Тому, хоча методи пошуку за шаблоном і є швидкими для попередніх сканувань, їхня недостатня точність і висока кількість помилкових позитивних результатів робить їх менш ефективними для комплексного і точного виявлення вразливостей. Саме тому виникає потреба в рішенні, яке дозволить проводити аналіз програмного коду не тільки швидко, але й з більшою точністю та мінімальною кількістю хибних спрацювань. Це стає

особливо важливим для підвищення безпеки процесів розробки та верифікації програмного забезпечення.

Розвиток технологій, особливо у сфері штучного інтелекту та машинного навчання, може пропонувати рішення, здатні виявляти більш складні і менш очевидні вразливості, не обмежуючись лише стандартними шаблонами. Такі інструменти можуть аналізувати код на глибшому рівні, враховуючи контекст, логіку програми та потенційні шляхи виконання коду.

Використання розширених методів аналізу дозволить не лише виявляти вразливості, але й забезпечувати більш точні прогнози щодо потенційних проблем, мінімізуючи при цьому хибно позитивні та хибно негативні результати. Це, в свою чергу, зробить процеси розробки та верифікації програмного забезпечення значно ефективнішими, знижуючи ризики та підвищуючи якість кінцевого продукту.

1.4 Постановка задачі дослідження

Оскільки галузь штучного інтелекту стрімко розвивається, окрім вже частково розглянутих методів на основі пошуку патернів, існують численні рішення та моделі, які мають потенціал внести суттєві поліпшення в існуючі процеси розробки та верифікації програмного забезпечення.

Методи аналізу програмного коду на основі подібності стають все більш актуальними, особливо при виявленні вразливостей у великих кодових базах. Ці методи працюють, порівнюючи фрагменти коду, щоб виявити "клони коду", які в сутності є частинами коду, що були репліковані всередині чи між програмними системами. Такі клони часто поширюють вразливості і помилки, тому виявлення їх є важливим для забезпечення безпеки програмного забезпечення. Ефективність цих методів полягає в їхній здатності виявляти не лише точні копії коду, але й варіанти, які можуть містити незначні зміни або бути перероблеными. Інструменти, такі

як VulPecker[20] і VUDDY[19], ілюструють досягнення у цій галузі, використовуючи складні алгоритми та техніки, такі як гранулярність на рівні функцій та абстракція з підвищеною увагою до безпеки для покращення точності виявлення.

Проте ці методи мають деякі обмеження. Однією з основних проблем є їхня висока обчислювальна складність, що призводить до високих вимог до часу, а також до об'єму пам'яті, особливо при аналізі великих та складних програмних продуктів [18]. Це робить процес витратним на ресурси та іноді непрактичним для швидких потреб у аналізі. Крім того, хоча ці методи вміло виявляють вразливості, вони часто стикаються з високою кількістю помилкових позитивних результатів. Це означає, що вони можуть неправильно ідентифікувати не-вразливий код як вразливий, що призводить до додаткової ручної перевірки і можливих затримок у процесі розробки. Збалансування точності виявлення вразливостей з мінімізацією помилкових позитивних результатів залишається критичною областю для покращення методів аналізу програмного коду на основі подібності.

Застосування рекурентних нейронних мереж (RNN) в області виявлення вразливостей програмного забезпечення показало перспективні результати, як це підтверджується численними дослідженнями [21-26]. Ці мережі, відомі своєю здатністю обробляти послідовності даних, ефективно використовувалися для виявлення паттернів і аномалій у коді, які можуть вказувати на вразливості.

В роботах [21] [22] автори досліджували використання технік глибокого навчання, включаючи RNNs, для підвищення відстежування помилок в програмах та виявлення клонів коду. Вони продемонстрували, що глибоке навчання може семантично обробляти великі обсяги коду та ідентифікувати схожі або клоновані фрагменти коду з високою точністю.

Ця здатність є критичною для виявлення потенційних вразливостей, які можуть бути присутніми в реплікованих фрагментах коду.

Автори [23] використали новаторський підхід, застосовуючи нейронні мережі для розпізнавання функцій в бінарних файлах. Ця техніка особливо важлива для виявлення вразливостей у скомпільованому коді, де вихідний код не завжди доступний. їх робота підкреслює потенціал RNNs у розумінні та аналізі більш абстрактних представлень програмного забезпечення.

Дослідження авторів [24] продемонструвало можливість RNNs навчатися представленням функцій з нерозмічених програмних проектів, що допомагає в виявленні вразливостей. Цей підхід важливий, оскільки він зменшує залежність від великих розмічених наборів даних, які часто відсутні в області вразливостей програмного забезпечення.

В роботах [25] [26] автори представили системи, такі як VulDeePecker і SySeVR, які використовують глибоке навчання для виявлення вразливостей. Ці системи демонструють, як BLSTM (бідерикційні рекурентні нейронні мережі) можуть ефективно вивчати представлення коду та виявляти складні паттерни, пов'язані з різними вразливостями програмного забезпечення. Їх робота також підкреслює важливість вибору ознак та виклики, пов'язані з балансуванням виявлення справжніх вразливостей проти помилкових позитивних результатів.

Ефективність RNNs в цих дослідженнях свідчить про їхній великий потенціал у виявленні вразливостей програмного забезпечення. Ці нейронні мережі відмінно розпізнають складні паттерни у послідовностях даних, що робить їх ідеально підходящими для аналізу рядків коду, виявлення аномалій і визначення подібностей.

Отже, дослідження показують, що RNN демонструють значний потенціал у аналізі вразливостей програмного забезпечення з таких причин:

1. Здатність ефективно обробляти послідовності – код складається з послідовностей рядків, інструкцій, викликів функцій.
2. Можливість виявляти складні шаблони вразливостей, які складно помітити іншими методами.
3. Здатність працювати з абстрактними представленнями коду, такими як бінарні файли та функції.
4. Потенціал навчатися безпосередньо з нерозмічених програмних проектів, уникаючи залежності від великих наборів розмічених даних.
5. Гнучкість щодо поєднання з іншими методами глибокого навчання для покращення ефективності.

Однак існують й певні виклики:

1. Складність налаштування архітектури та гіперпараметрів мережі.
2. Необхідність великих обчислювальних ресурсів для навчання глибоких RNN.
3. Складність інтерпретації внутрішніх представлень мережі, набутих під час навчання.
4. Ризик перенавчання та підлаштування до конкретних наборів даних.

Подальші дослідження мають фокусуватися на подоланні цих обмежень, а також на реалізації RNN-моделей у практичних інструментах аналізу безпеки ПЗ. Це дозволить реалізувати повною мірою їх потенціал у виявленні вразливостей.

Використання згорткових нейронних мереж (Convolutional Neural Networks, CNN) для виявлення вразливостей у програмному коді показало значущі результати, але супроводжується певними викликами.

Згорткові нейронні мережі, які в основному відомі своєю успішністю у завданнях обробки зображень і розпізнавання шаблонів, були адаптовані для аналізу і прогнозування дефектів і вразливостей програмного коду. Їх перевагою є здатність ефективно виявляти локальні паттерни в даних, що робить їх придатними для пошуку потенційно небезпечних конструкцій у коді.

Дослідження Лі та ін. [27] демонструє застосування згорткових нейронних мереж у прогнозуванні дефектів програмного коду. Розглядаючи код як форму даних, яку можна обробляти аналогічно зображенням, вони показали, що згорткові нейронні мережі можуть ефективно ідентифікувати шаблони, які вказують на наявність дефектів.

Цей підхід вказує на здатність згорткових нейронних мереж захоплювати та вивчати складні особливості програмного коду, які часто свідчать про потенційні вразливості. Наприклад, певні комбінації функцій чи структур даних можуть бути індикаторами уразливих ділянок коду. Згорткові мережі добре підходять для виявлення таких комбінацій.

Дослідники Генг та ін. [28] та Ванг та ін. [29] у своїх роботах детально аналізують застосування згорткових нейронних мереж для семантичної сегментації зображень. Вони розглядають як сильні, так і слабкі сторони згорткових нейронних мереж у цій задачі.

Зокрема, автори зазначають, що згорткові нейронні мережі демонструють високу ефективність у класифікації та розпізнаванні локальних візуальних патернів на зображеннях. Це пояснюється наявністю згорткових шарів, які виділяють ознаки з вхідних даних, та використанням операції пулінгу, котра підвищує стійкість до невеликих зсувів і поворотів.

Проте дослідники зазначають, що чисто згорткові архітектури можуть неправильно інтерпретувати загальний контекст зображення та семантичні зв'язки між об'єктами. Це відбувається тому, що згорткові нейронні мережі обробляють зображення послідовно, шар за шаром,

фокусуючись лише на локальному оточенні кожного пікселя. Така особливість ускладнює моделювання глобального контексту сцени.

Аналогічні труднощі виникають при застосуванні згорткових нейронних мереж для аналізу програмного коду. З одного боку, вони добре розпізнають локальні синтаксичні конструкції, але можуть неправильно інтерпретувати їх роль та взаємозв’язки у загальній програмі. Це ускладнює точне виявлення вразливостей, оскільки потрібен аналіз як окремих фрагментів коду, так і їх взаємодії.

Отже, незважаючи на потенціал згорткових нейронних мереж, подальші дослідження мають фокусуватися на вирішенні проблеми врахування контекстних зв’язків як у зображеннях, так і у програмному коді. Це дозволить повністю розкрити переваги даних нейромереж для складних задач комп’ютерного зору та аналізу безпеки ПЗ.

Глибокі мережі вірувань (Deep Belief Networks, DBNs), як одна з форм архітектури глибокого навчання, демонструють обіцяні результати у сфері виявлення вразливостей програмного забезпечення, як це видно з робіт дослідників [30] [31]. Ці мережі проявляють особливу ефективність у виділенні високорівневих та складних абстракцій з даних, що вкрай важливо для аналізу та передбачення дефектів та вразливостей програмного коду.

В роботі [30] було продемонстровано використання мереж глибоких вірувань для автоматичного навчання семантичних ознак, які вказують на наявність дефектів у програмному коді. Цей підхід набуває великого значення, оскільки він виходить за межі поверхневого аналізу коду та досліджує глибинну семантику коду, яка часто містить важливі вказівки на можливі вразливості. Шляхом навчання цих глибоких семантичних ознак глибокі мережі вірувань можуть прогнозувати дефекти з вищою точністю, ніж деякі традиційні методи.

Дослідження авторів [31] розглядає використання глибокого навчання, включаючи глибокі мережі вірувань, для прогнозування дефектів в режимі реального часу. Цей підхід особливо важливий для виявлення вразливостей на початкових етапах розробки програмного забезпечення, що дозволяє вчасно виправляти помилки та зменшує ризик впровадження вразливого коду.

Незважаючи на обіцяні результати, існують помітні обмеження в застосуванні глибоких мереж вірувань для виявлення вразливостей програмного коду. глибокі мережі вірувань є складними моделями, які вимагають значних обчислювальних ресурсів для тренування та інференсу, що може ускладнювати їх практичну використаність у менших проектах чи командах з обмеженими ресурсами. Ефективність глибоких мереж вірувань в значній мірі залежить від якості та обсягу навчальних даних, що може бути складним завданням в контексті виявлення вразливостей програмного коду [32]. Крім того, глибокі мережі вірувань, подібно до багатьох моделей глибокого навчання, стикаються з проблемами інтерпретованості, що може ускладнювати розуміння причини класифікації певного фрагменту коду як вразливого, що є проблематичним, коли розробники повинні перевіряти та виправляти ці прогнози. Здатність глибоких мереж вірувань до загальнізації результатів між різними програмними проектами та мовами програмування не завжди гарантується, особливо якщо існують значні відмінності в стилі програмування, архітектурі чи мові.

Великі мовні моделі (Large language model, LLM) є перспективним напрямкрм розвитку технологій штучного інтелекту. Їхня унікальна архітектура та потужні обчислювальні можливості дозволяють великим мовним моделям ефективно опрацьовувати природну мову, розуміти контекст та генерувати зміст.

Останні дослідження демонструють, що великі мовні моделі також можуть успішно застосовуватися для аналізу програмного коду з метою

виявлення прихованих вразливостей [78]. По суті, програмний код може розглядатися як особлива форма природної мови з власним синтаксисом та семантикою.

Це дозволяє адаптувати архітектуру та підходи, розроблені для обробки текстів, для цілей аналізу коду. Завдяки величезній кількості параметрів (до десятків мільярдів), великі мовні моделі здатні вловлювати складні закономірності та залежності між різними фрагментами коду.

Наприклад, модель CodeBERT [63], попередньо навчена на великих масивах даних, демонструє здатність до семантичного аналізу коду та пошуку логічних помилок, які можуть вказувати на наявність вразливостей. Подібні моделі також ефективно застосовуються для визначення структурної та функціональної подібності між фрагментами коду різних програм [79].

Використання великих мовних моделей надає можливості "переносу знань", коли модель, попередньо навчена на одних мовах та проектах, успішно застосовує набуті навички для аналізу інших програм, написаних в інших мовах та для інших цілей. Це суттєво розширює можливості систем детекції вразливостей та спрощує їх адаптацію.

Втім, великі мовні моделі стикаються з певними обмеженнями, такими як складність і ресурсоємність тренування, чутливість до якості вхідних даних та проблеми тлумачення отриманих результатів. Тому наразі великі мовні моделі доцільно комбінувати з іншими легшими та прозорішими методами аналізу коду для створення найоптимальніших систем пошуку вразливостей.

Графові нейронні мережі (GNN) є порівняно новим, але вкрай перспективним типом архітектур штучних нейронних мереж, орієнтованих на аналіз топологічних графових структур даних [80]. Їхня унікальна особливість полягає у здатності обробляти не регулярні вектори чи тензори,

а саме графи зі складною топологією, такі як молекулярні графи, графи знань, мережі зв'язків тощо.

Ця властивість робить графові нейронні мережі особливо релевантними і придатними для аналізу структурованих даних на кшталт програмного коду, який також має складну графову природу. Зокрема, різноманітні залежності, взаємодії та потоки даних між функціями, модулями, змінними у програмах можуть бути представлені у вигляді складних графів.

На відміну від традиційних нейромереж, які обробляють лінійні послідовності даних, графові нейронні мережі спеціально адаптовані для моделювання таких топологічних залежностей. Завдяки ітераційним графовим алгоритмам, вони здатні поширювати інформацію по ребрах графа, вивчаючи зв'язки як між сусідніми, так і віддаленими вузлами [81].

Така здатність GNN до глибокого контекстного аналізу топологічних структур даних робить їх надзвичайно корисним інструментом для моделювання складних залежностей та потоків даних у програмному забезпеченні. Графове подання коду дає GNN можливість одночасно аналізувати як локальні шаблони (окремі фрагменти коду), так і глобальний контекст (загальна архітектура та логіка програми).

Така комбінація локальних та глобальних ознак є критично важливою для ефективного виявлення комплексних, контекстно-залежних вразливостей, які вимагають розуміння як конкретних фрагментів коду, так і їх ролі та взаємодій в загальній системі.

Тому GNN є вкрай перспективним напрямком для розвитку інтелектуальних систем аналізу безпеки коду, здатних оперувати складними структурами даних та залежностями у програмах. Інтеграція графових мереж з іншими підходами (LLM, пошук подібності тощо) може істотно розширити можливості автоматизованої верифікації та підвищити стійкість систем до нових типів кіберзагроз.

Розглянуті методи детекції вразливостей демонструють, що пошук за патернами не є єдиним ефективним способом аналізу програмного коду. Існує потенціал для створення більш комплексної моделі, яка б об'єднувала точність нейронних мереж, швидкість аналізу на основі пошуку за патерном, та гнучкість моделей, які аналізують подібності коду.

Така інтегрована модель могла б значно автоматизувати процес верифікації коду, ефективно виявляючи вразливості та помилки, які можуть бути неочевидними для традиційних методів. Це б не лише підвищило б швидкість та якість аналізу, але й зробило б кінцевий програмний продукт значно безпечнішим.

Можливі шляхи створення такої комплексної моделі включають:

1. Поєднання методів машинного навчання (нейронні мережі) та евристичних підходів (пошук за патернами). Це дозволить використовувати переваги обох підходів - автоматичне навчання та швидкість евристик.
2. Застосування ансамблів моделей, що комбінують прогнози декількох алгоритмів. Такі моделі часто демонструють кращу точність та узагальнюваність, ніж окремі алгоритми.
3. Використання ієрархічних систем, де на першому рівні застосовуються швидкі евристичні фільтри, а більш повільні та точні нейромережі використовуються на другому рівні.

Така комплексна система аналізу вразливостей могла б інтегруватися в процес розробки ПЗ відповідно до стандартів безпеки, таких як Microsoft SDL, OWASP та ISO 27034.

Зокрема, вона може застосовуватися на етапах проектування архітектури системи, розробки компонентів, інтеграції та тестування. Автоматизована перевірка допоможе своєчасно виявляти дефекти безпеки та виправляти їх до виходу готового продукту.

Така інтеграція сприятиме розробці безпечнішого ПЗ, підвищуючи його надійність та захищеність. В результаті, поєднання методів аналізу вразливостей у комплексну систему дозволить значно поліпшити якість перевірок безпеки коду.

1.5 Висновки

Таким чином, на підставі проведеного аналізу, що наведено в першому розділі даної дисертаційної роботи, можна зробити висновки, які задають фундамент для подальшого дослідження. В центрі уваги даної роботи знаходиться розробка методів та систем для пошуку та класифікації вразливостей у програмному коді.

Перший розділ встановлює теоретичні основи для розуміння сучасного стану вразливостей у програмному коді та визначає ключові напрямки для подальшого розвитку методів їх детекції. Основна увага приділяється аналізу поточних викликів у сфері кібербезпеки, а також розглядаються існуючі методи та підходи до виявлення вразливостей. Встановлено, що швидкий розвиток технологій та зростання складності програмного забезпечення вимагають нових, більш ефективних методів детекції.

Дана робота виокремлює важливість розвитку систем, які можуть автоматизувати процеси детекції вразливостей, інтегруючи передові технології та методи аналізу. Це включає в себе застосування таких інноваційних підходів, як використання штучного інтелекту та машинного навчання для підвищення точності та ефективності виявлення вразливостей.

У подальших главах планується розглянути практичну реалізацію та оцінку пропонованих методів. Зокрема, другий розділ фокусується на моделюванні об'єкта дослідження та його репрезентації в контексті пошуку вразливостей, що є фундаментальним для розробки ефективної системи

детекції. Третій розділ передбачає розробку самого методу детекції вразливостей, тоді як четвертий розділ зосереджується на експериментах та оцінці результатів роботи розробленої системи.

Висновки першого розділу, таким чином, підкреслюють важливість розробки інтегрованих, автоматизованих систем для детекції вразливостей в програмному коді, що враховують сучасні виклики та технологічні тенденції у сфері кібербезпеки. Це дозволить не лише підвищити рівень безпеки програмних систем, але й забезпечить більш ефективний та своєчасний відгук на потенційні загрози.

2 Моделювання об'єкту дослідження та його репрезентація в контексті пошуку вразливостей

2.1 Моделі представлення програмного коду

На сьогоднішній день існує кілька моделей представлення програмного коду [25] [35] [36]. Ефективність використання цих моделей залежить від задачі яка вирішується. Для реалізації системи пошуку вразливостей розглядаються наступні моделі представлення програмного коду: текстова, AST (Abstract Syntax Tree), CFG (Control Flow Graph) та PDG (Program Dependence Graph). Дані моделі є найбільш уживаними серед наукових робіт, присвячених аналізу програмного коду.

Текстове представлення коду, є найбільш базовим, оскільки воно просто відображає код як послідовність текстових символів або токенів. Такі токени, що утворюються в результаті лексичного аналізу, представляють собою елементи синтаксичних конструкцій мови програмування. Однак, основною проблемою текстового представлення є його низька інформативність та сильна залежність від конкретної мови програмування. Це означає, що однакові за функціональністю фрагменти коду на різних мовах будуть мати різне текстове представлення, що ускладнює процес адаптації системи виявлення вразливостей до різних мов програмування.

Приклад текстового представлення коду, написаного мовою програмування C, зображено на рис. 2.1:

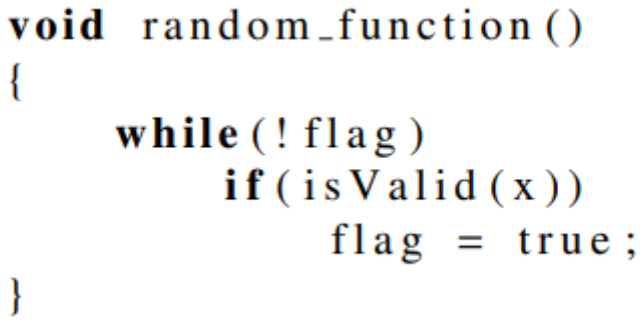

Рис. 2.1 - Текстове представлення коду на мові C [34]

Абстрактне Синтаксичне Дерево (AST) є структурою даних, яка використовується для представлення синтаксичної структури програмного коду у вигляді ієрархічного дерева. Елементами цього дерева є токени вихідного коду, що організовані у специфічному порядку. У такому дереві, вузли, що не є листками, зазвичай представляють синтаксичні конструкції, такі як оператори або декларації, тоді як листкові вузли відображають операнди, змінні або літерали.

Особливістю AST є те, що воно відображає структуру програми, а не її фактичний синтаксис. Це означає, що AST абстрагується від багатьох деталей конкретної мови програмування, таких як розміщення дужок, відступи та інші елементи форматування, зосереджуючись на логічній та структурній організації коду. Ця абстракція дозволяє більш глибоко аналізувати програму, ідентифікувати залежності та взаємодії між різними частинами коду, що є особливо корисним у задачах оптимізації коду, виявлення помилок та вразливостей.

Граф Потоку Керування (Control Flow Graph, CFG) є фундаментальною структурою даних у комп'ютерній науці, використовуваною для представлення логічної послідовності виконання програмного коду. Ця структура є орієнтованим графом, де кожен вузол представляє окремі елементи програми, такі як предикати (умовні вирази) та оператори. Ребра графу, які з'єднують ці вузли, ілюструють потік

керування між ними, тобто порядок, в якому ці оператори та умови будуть виконуватися під час роботи програми.

Особливістю кожного ребра у CFG є наявність міток, які вказують на специфічні умови, за яких потік керування перейде від одного вузла до іншого. Ці мітки є критично важливими для розуміння логіки програми, оскільки вони дозволяють визначати, за яких обставин будуть виконуватися певні частини коду.

CFG широко використовується у задачах оптимізації програмного коду, статичного аналізу, виявлення помилок та вразливостей, оскільки дозволяє зрозуміти процес виконання програми та виявити потенційні проблеми в логіці потоку даних.

Граф Залежностей Програми (Program Dependence Graph, PDG) є складною структурою даних у галузі комп'ютерних наук, призначеною для детального відображення залежностей у програмному коді. У PDG, вузли представляють різні елементи програми, зокрема предикати (умовні вирази) та оператори. Ребра цього орієнтованого графу створюють візуальне представлення залежностей, які існують між цими вузлами, поділяючись на два основних типи: ребра залежності від управління та ребра залежності від даних.

Ребра залежності від управління в PDG ілюструють, як вибір шляху виконання програми, визначений результатом предиката, впливає на виконання інших частин коду. Наприклад, якщо результат умовного виразу визначає, чи буде виконуватися певний блок коду, це ребро відображатиме цю залежність.

З іншого боку, ребра залежності від даних відображають вплив, який одна змінна чи вираз має на інші. Це включає в себе ситуації, коли одна змінна використовується для обчислення або впливу на значення іншої. Таке представлення дозволяє глибше зрозуміти взаємозв'язки між різними частинами коду, особливо у випадках складних програм з багатьма

залежностями. Рис. 2.2 демонструє AST, CFG та PDG для фрагменту коду представленого на рис. 2.1:

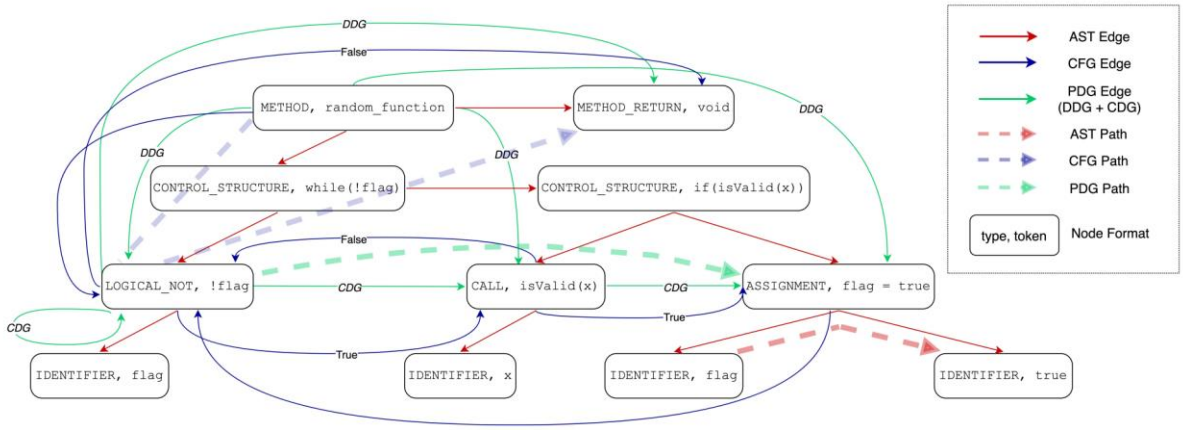

Рис. 2.2 - AST, CFG та PDG для фрагменту коду з рис. 2.1. [34]

У рамках цієї роботи обрано використання Абстрактного Синтаксичного Дерева (AST) як основного способу репрезентації програмного коду під час аналізу. Хоча і Граф Потоку Керування (CFG), і Граф Залежностей Програми (PDG) могли б забезпечити більш глибокий контекст і, відповідно, потенційно покращити роботу аналітичної моделі завдяки їх здатності детально відображати потік виконання програми та взаємозв'язки між різними елементами коду, використання цих методів репрезентації в даному дослідженні не вважається доцільним.

Основною причиною цього є значні витрати часу та обчислювальних ресурсів, необхідних для створення та обробки CFG та PDG. Реалізація цих складних структур даних вимагає значної кількості обчислювальної пам'яті та процесорного часу, що може стати обтяжливим для системи, особливо при аналізі великих обсягів коду або складних програмних систем.

Враховуючи ці обмеження, вибір на користь AST як моделі репрезентації коду є оптимальним рішенням. AST забезпечує достатній рівень деталізації для аналізу синтаксичної структури коду, при цьому не

потребує високого обчислювального навантаження, на відміну від CFG та PDG.

2.2 Аналіз програмного коду з використанням гібридного методу пошуку та класифікації вразливостей

В даній роботі пропонується створити систему аналізу програмного коду з використанням гібридного методу пошуку та класифікації вразливостей. Дана система повинна виконувати 2 основні функції:

1. Пошук вразливих фрагментів вихідного коду. Це передбачає ідентифікацію ділянок коду, які з високою ймовірністю містять вразливості, що можуть бути використані зловмисниками.
2. Класифікація знайдених вразливих фрагментів відповідно до загальноприйнятої таксономії вразливостей програмного забезпечення CWE (Common Weakness Enumeration).

Формально, роботу системи аналізу програмного коду з використанням гібридного методу пошуку та класифікації вразливостей можна визначити наступним чином:

Нехай, C - є вихідним кодом програми. Функція $g: C \rightarrow A$ - перетворює вихідний код у абстрактне синтаксичне дерево. Функція $r: A \rightarrow R$ створює множину проміжних представлень фрагментів коду R на основі абстрактного синтаксичного дерева. Модель пошуку вразливостей $m_s: R \rightarrow Y$ шукає відображення проміжного представлення $r_i \in R$, $i \in \{1,2,\ldots,n\}$, де n - кількість проміжних представлень, $Y \in [0,1]$, де 1 позначає вразливість, а 0 відсутність вразливості у проміжному представленні фрагменту коду r_i. Тоді, якщо $m_s(r_i) = 1$, модель класифікації вразливості $m_c: R \rightarrow V$ шукає відображення проміжного

представлення фрагменту коду r_i, для визначення класу вразливості, де $V\epsilon\{v_1, v_{2,}\dots v_j\}$, де j – кількість класів вразливостей, а V - множина класів вразливості згідно таксономії CWE [33].

Алгоритм роботи системи аналізу коду з використанням гібридного підходу пошуку та класифікації вразливостей можемо записати наступним чином:

$a_c = g(c)$

$R_c = r(a_c)$, де $R_c = \{(s_1, t_1, t'_1, p_1), \dots, (s_n, t_n, t'_n, p_n)\}$

$\forall r_i \in R_c$:

$y_i = m_s(p_i)$

$if\ y_i = 1$:

$v_i = m_c(s_i)$

Принцип роботи запропонованої системи зображено на Рис. 2.3:

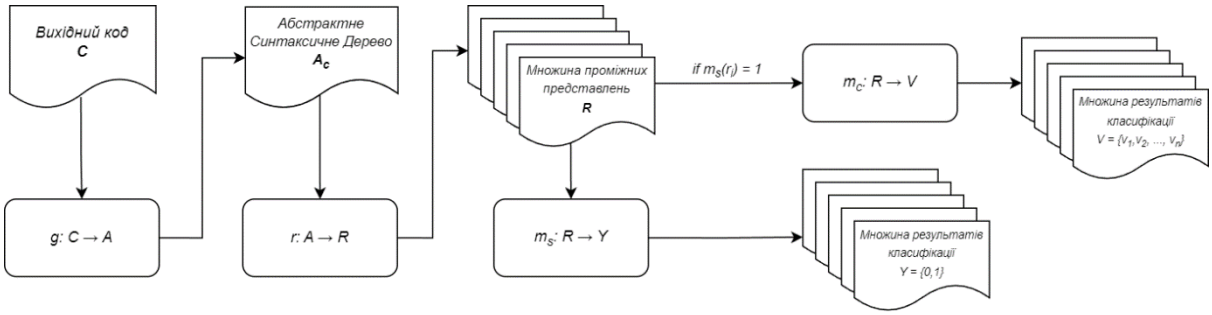

Рис. 2.3 - Принцип роботи системи аналізу програмного коду з використанням гібридного методу пошуку та класифікації вразливостей

Таким чином, вперше було запропоновано гібридний метод аналізу програмного коду, що поєднує методи глибокого навчання та методи виявлення подібності коду для пошуку та класифікації вразливості в коді, який дозволяє ефективно виконувати пошук вразливостей в коді, а також класифікувати з високою точністю знайдені вразливості.

Запропонований поділ задачі класифікації вразливостей на два етапи дозволяє досягти більшої ефективності завдяки спеціалізації кожної з моделей.

Зокрема, модель пошуку вразливостей фокусується виключно на завданні розпізнавання ознак, характерних для вразливих фрагментів коду. Вона навчається визначати узагальнені шаблони та аномалії, типові для різних категорій слабких місць програми.

Натомість модель багатокласової класифікації спеціалізується виключно на розпізнаванні конкретних типів вразливостей. Вона оперує більш вузькими шаблонами та особливостями, характерними для певних категорій вразливостей згідно зі стандартизованими базами, на кшталт CWE.

Такий поділ компетенцій дає можливість кожній моделі глибше спеціалізуватися у рамках своєї предметної області, не розпорошуючи зусилля на суміжні задачі. Це дозволяє досягати більшої точності класифікації за рахунок вузької спеціалізації кожного класифікатора.

Отже, запропонована архітектура із поділом на спеціалізовані моделі пошуку та класифікації вразливостей дозволяє досягти більшого синергетичного ефекту завдяки можливості налаштування кожного компоненту під конкретну роль у загальному процесі аналізу безпеки програмного коду.

2.3 Метод побудови проміжного представлення програмного коду

Для ефективного аналізу вразливостей у програмному коді необхідно побудувати адекватне проміжне представлення, яке б відображало найбільш значущі фрагменти коду з точки зору аналізу на предмет вразливостей. Рис 2.1 демонструє загальний процес аналізу програмного

коду, в рамках якого процес побудови множини проміжних представлень фрагментів коду R з AST A описується функцією $r: A \rightarrow R$.

Функція r представляє собою композицію декількох функцій, які викликаються послідовно. Формально це можна записати наступним чином:

Нехай A – абстрактне синтаксичне дерево програмного коду. Функція $s: A \times K \rightarrow S$ будує множину зрізів S абстрактного синтаксичного дерева A, де зріз $s_i \in S$. Кожен зріз будується навколо ключової точки $k_j \in K$, де $j = \{1,2,\dots,z\}$, де $z -$ загальна кількість ключових точок. Функція $t: S \rightarrow T$ будує відображення зрізу $s_i \in S$ у текстове представлення $t_i \in T$. Функція $t': T \rightarrow T'$ виконує нормалізацію текстового представлення $t_i \in T$. Функція $p: T' \rightarrow R$ будує проміжне представлення у векторному просторі. Тоді функція r визначається як композиція наступних функцій:

$$r = p \circ t' \circ t \circ s \qquad (2.1)$$

Елементи $r_i \in R$ представляють собою впорядкований набір:

$$r_i = (s_i, t_i, t'_i, p_i) \qquad (2.2)$$

Схематично метод побудови проміжного представлення програмного коду зображено на Рис. 2.4:

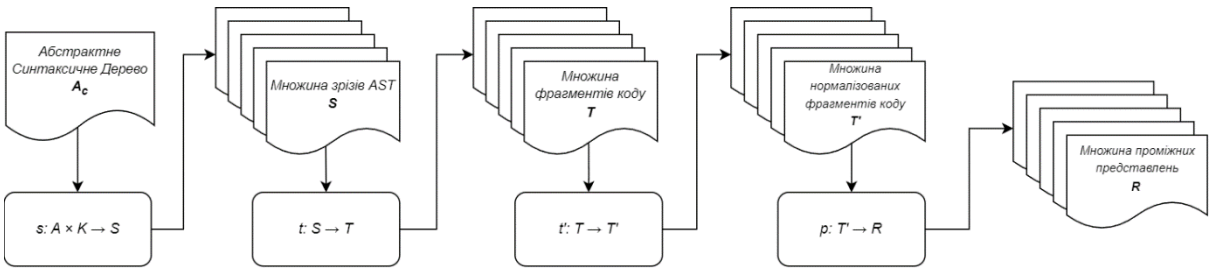

Рис. 2.4 - Метод побудови проміжного представлення програмного коду

Алгоритм побудови проміжного представлення програмного коду можна записати наступним чином:

$\forall k_j \in K, j = \{1, 2, \dots z\}$:

$$s_j = s(a_c, k_j)$$

$$t_j = t(s_j)$$
$$t'_j = t'(t_j)$$
$$p_j = p(t'_j)$$
$$r_j = (s_j, t_j, t'_j, p_j)$$

Таким чином, отримав подальший розвиток метод побудови проміжного представлення програмного коду у вигляді кодового гаджету, який відрізняється від існуючих методів наявністю обмеження по розміру локального контексту відносно ключової точки, що дозволило зменшити результуючий розмір кодових гаджетів та підвищити точність класифікації при подальшому аналізі нейронною мережею.

Запропонований підхід дозволяє виділити навколо кожної ключової точки локальний фрагмент AST разом з контекстом, який трансформується у векторне представлення.

Перевагою запропонованого підходу є акцентування уваги саме на потенційно небезпечних ділянках коду, які пов’язані з ключовими точками, що суттєво підвищує ефективність подальшої роботи моделей класифікації вразливостей.

2.4 Метод пошуку вразливостей в програмному коді з використанням нейронної мережі

Після побудови проміжного представлення програмного коду (рис. 2.2), система аналізу програмного коду виконує аналіз отриманих фрагментів коду на предмет наявності вразливості безпеки. Фактично, метод пошуку вразливостей в програмному коді вирішує задачу бінарної класифікації, де клас «0» позначає відсутність вразливості в програмному коді, а клас «1» позначає наявність вразливості.

Метод пошуку вразливостей в програмному коді складається з двох основних фаз роботи: тренування та виконання. Фаза тренування полягає в тому що модель навчається розпізнавати шаблони вразливостей в коді, в результаті чого формуються вихідні коефіцієнти моделі пошуку вразливостей. Дані вагові коефіцієнти будуть використані у фазі виконання, для здійснення бінарної класифікації фрагментів коду на предмет наявності в них вразливості.

Формально, фазу тренування можна описати наступним чином:
Нехай, C_s - є множиною семплів програм які містять або не містять вразливості. Функція $g: C \rightarrow A$ - перетворює вихідний код у абстрактне синтаксичне дерево. Функція $r: A \rightarrow R$ створює множину проміжних представлень фрагментів коду R на основі абстрактного синтаксичного дерева. Тоді задачею фази тренування буде знайти відображення $l: R \times W \rightarrow Y$. Схематично фаза тренування для методу пошуку вразливостей в коді зображено на рис. 2.5:

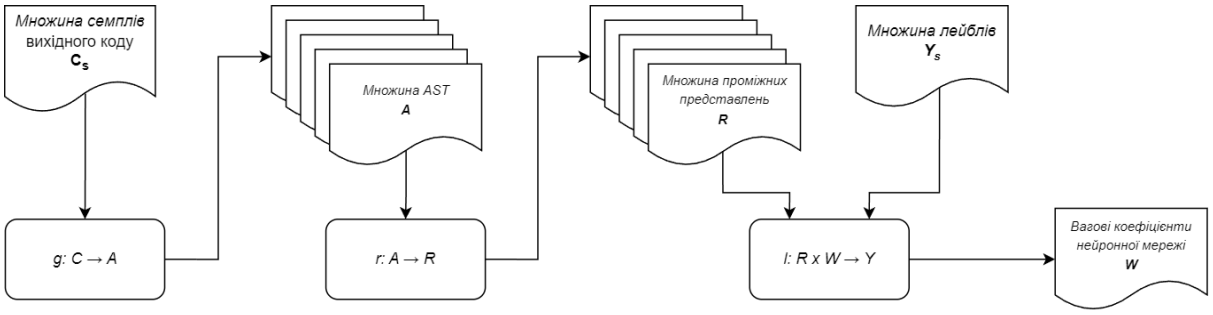

Рис. 2.5 - Фаза тренування методу пошуку вразливостей з використанням нейронної мережі

Алгоритм роботи фази тренування методу пошуку вразливостей з використанням нейронної мережі, зображеного на рис. 2.3, можна записати наступним чином:

$\forall c_i \in C_s$:

$a_i = g(c_i)$

$R_i = r(a_i)$, де $R_i = \{(s_1, t_1, t'_1, p_1), \dots, (s_n, t_n, t'_n, p_n)\}$

$\forall r_j \in R_i$:

$$y_j = l(p_j, W)$$

$$e_j = err(y_j, y_{sj})$$

$$W = opt(W, e_j)$$

Рис. 2.6 схематично позначає фазу виконання методу пошуку вразливостей з використанням нейронних мереж:

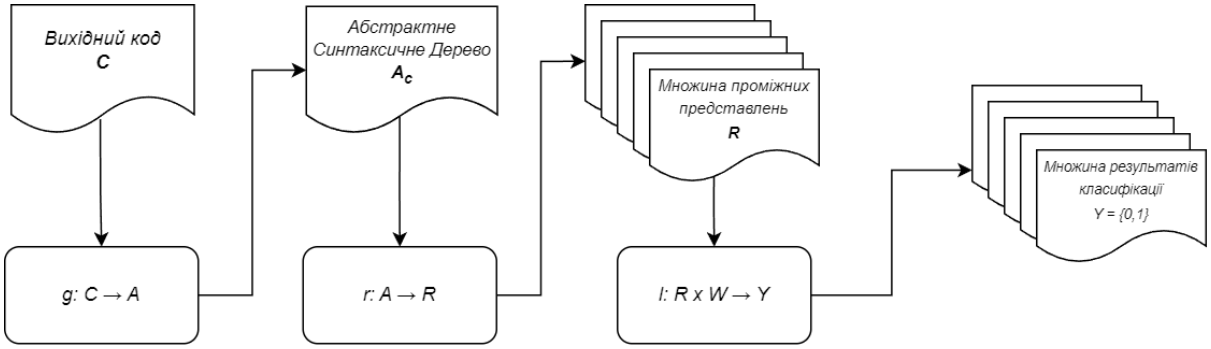

Рис. 2.6 - Фаза виконання методу пошуку вразливостей з використанням нейронної мережі

Як показано на Рис. 2.6, вагові коефіцієнти *W*, що використовуються для побудови відображення проміжного представлення *R* у лейбл *Y*, були отримані в рамках фази тренування (Рис. 2.5).

Алгоритм роботи фази виконання для методу пошуку вразливостей з використанням нейронної мережі виглядає наступним чином:

$a_c = g(C)$

$R_c = r(a_c)$, де $R_c = \{(s_1, t_1, t'_1, p_1), \dots, (s_n, t_n, t'_n, p_n)\}$

$\forall r_j \in R_c$:

$$y_j = l(p_j, W)$$

2.5 Метод класифікації вразливостей в програмному коді з використанням ковзного хешування

Наступним етапом аналізу коду є класифікація знайдених вразливостей згідно таксономії CWE. Класифікація вразливостей здійснюється з використанням ковзного хешування AST. Даний метод працює у дві фази: підготовка та виконання. На відміну від методу пошуку вразливостей, метод класифікації вразливостей не використовує нейронні мережі та не потребує тренування. Натомість, в рамках даного методу виконується попередня підготовка бази знань, яка в подальшому буде використовуватися у фазі виконання.

Фазу підготовки в рамках методу класифікації вразливостей з використанням ковзного хешування AST формально можна визначати наступним чином:

Формально, фазу тренування можна описати наступним чином:
Нехай, C_{cwe} - є множиною семплів програм які містять вразливість $V = \{v_1, v_2, \dots, v_j\}$, де j – кількість класів вразливостей, згідно таксономії CWE, $L_w -$ розмір вікна для ковзного хешування. Функція $g: C \rightarrow A$ - перетворює вихідний код у абстрактне синтаксичне дерево. Функція $r: A \rightarrow R$ створює множину проміжних представлень фрагментів коду R на основі абстрактного синтаксичного дерева. Функція $h: R \rightarrow H$ будує множину хешів H_b для проміжного представлення. Схематично фаза підготовки для методу класифікації вразливостей з використанням ковзного хешування зображено на Рис. 2.7:

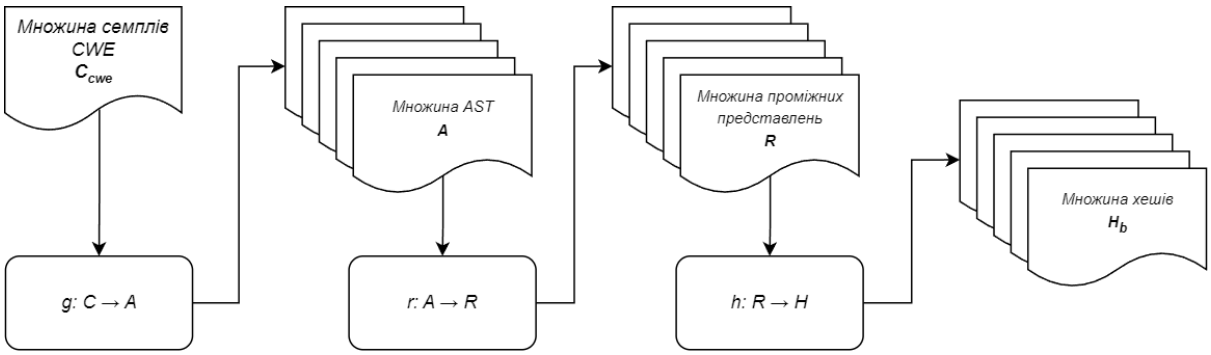

Рис. 2.7 - Фаза підготовки методу класифікації вразливостей з використанням ковзного хешування

Алгоритм роботи фази підготовки методу класифікації вразливостей з використанням ковзного хешування складається з наступних етапів:

$\forall \mathrm{c}_i \in C_{cwe}$:

$a_i = g(\mathrm{c}_i)$

$R_i = r(a_i)$, де $R_i = \{(s_1, t_1, t'_1, p_1), \dots, (s_n, t_n, t'_n, p_n)\}$

$\forall r_j \in R_i$:

$\forall n_k \in s_j$, де $n_k = \{n_{j1}, n_{j2}, \dots, n_{jL_w}\}$,

де k

$= \{1,2, \dots, N - L_w\}$, де $N -$ кількість вузлів у s_j :

$h_{jk} = h(n_k)$

$h_j = \{h_{j1}, h_{j1}, \dots, h_{jN-L_w}\}$

$H_b = \{h_1, h_2, \dots, h_j\}$

Фаза виконання методу класифікації вразливостей з використанням ковзного хешування проходить після фази виконання методу пошуку вразливостей з використанням нейронної мережі. Вхідними даними на цьому етапі виступає проміжне представлення фрагменту коду, який містить вразливість.

В рамках фази виконання, проміжне представлення потенційно вразливого фрагменту коду хешується. Отриману множину хешів порівнюють з вже наявними в базі знань хешами вразливостей. Клас вразливостей з яким було виявлено найбільше співпадінь і буде визначеним класом.

Формально, роботу методу класифікації вразливостей з використанням ковзного хешування можна описати наступним чином:

Нехай $R_c -$ проміжне представлення фрагменту коду C. Функція $h: R \to H$ будує множину хешів H_c для проміжного представлення. Функція $x: H_c \cap H_b \to V$ виконує перетин множини хешів вхідного коду та хешів з бази знань, отриманих на етапі підготовки (Рис. 2.7). Схематично фаза виконання методу класифікації вразливостей з використанням ковзного хешування зображено на Рис. 2.8:

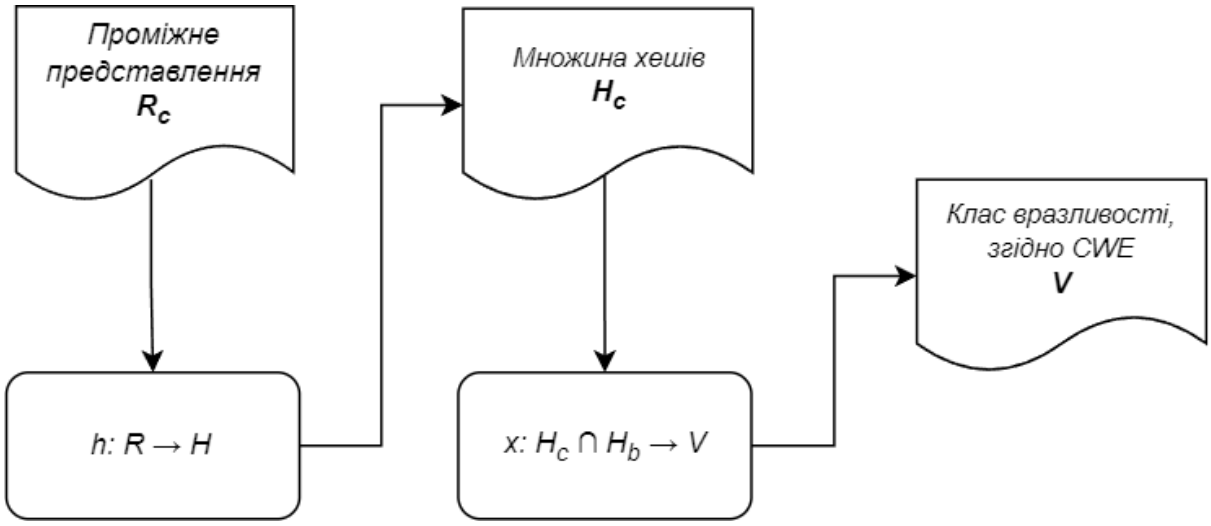

Рис. 2.8 - Фаза виконання методу класифікації вразливостей з використанням ковзного хешування

Алгоритм роботи фази виконання, представленої на Рис. 2.8, складається з наступних кроків:

$\forall r_j \in R_i$:

$$\forall n_k \in s_j, \text{де } n_k = \{n_{j1}, n_{j2}, \dots, n_{jL_w}\},$$

$$\text{де } k = \{1,2, \dots, N - L_w\}, \text{де } N - \text{кількість вузлів у } s_j :$$

$$h_{jk} = h(n_k)$$

$$h_j = \{h_{j1}, h_{j1}, \dots, h_{jN-L_w}\}$$

$$v_j = x(h_j, H_b)$$

$$v_c = \max(v_1, v_2, \dots, v_j)$$

Таким чином, вперше було запропоновано метод класифікації вразливостей в програмному коді з використанням ковзного хешування абстрактного синтаксичного дерева, який відрізняється від існуючих методів тим що використовує метод виявлення подібності коду для ефективної класифікації вразливостей без необхідності використання навчальної вибірки великого об’єму.

2.6 Висновки

У другому розділі роботи було здійснено ґрунтовне формалізоване моделювання задачі автоматизованого пошуку вразливостей у програмному коді із застосуванням математичного апарату. Це дозволило з високою точністю описати вхідні дані, етапи перетворення та бажані результати системи у вигляді послідовності формальних перетворень.

Ключовим результатом розділу є пропозиція нової гібридної архітектури, яка інтегрує потужності глибинних нейронних мереж для контекстно-залежного аналізу вразливостей та метод ковзного хешування абстрактних синтаксичних дерев з метою прискорення пошуку і класифікації вразливостей за відомими шаблонами. Такий симбіоз сприяє взаємо доповненню підходів та дозволяє подолати окремі недоліки кожного

з них, створюючи найсприятливіші умови для високоточного пошуку прихованих вразливостей у програмному коді.

На базі цієї гібридної архітектури розроблено математичні моделі складових компонент системи:

- Модуля аналізу коду за допомогою нейронних мереж.
- Модуля класифікації вразливостей з використанням ковзного хешування AST.
- Загальної гібридної моделі системи, котра інтегрує обидва підходи.

Особлива увага приділялася формалізації роботи складових моделей, визначенню функцій та встановленню залежностей між змінними, які описують перетворення даних в межах системи. За рахунок цього забезпечено прозорість та однозначність розуміння запропонованих математичних моделей, що є необхідним підґрунтям як для їх практичного втілення, так і подальшого аналізу та вдосконалення.

Окремо, були розглянуті різноманітні методи репрезентації програмного коду для цілей його аналізу, включаючи:

- Текстове представлення у вигляді токенів-символів та лексем.
- Абстрактні синтаксичні дерева, які відображують логічну структуру коду.
- Граф потоку керування та граф залежностей програми для аналізу взаємодій фрагментів коду.

Після порівняльного аналізу за різними критеріями такими як складність, ресурсоємність, інформативність, в роботі вибрано абстрактні синтаксичні дерева як оптимальний спосіб подання програмного коду для наступного аналізу на предмет наявності вразливостей. Це обумовлено раціональним балансом між адекватним рівнем деталізації структури та обсягом ресурсів.

Таким чином, у другому розділі роботи сформовано методологічне підґрунтя як для подальшої розробки, так і наукової оцінки результативності запропонованої системи автоматизованої верифікації програмного забезпечення. Визначення точних математичних моделей та формальний опис процедури перетворення вхідних даних забезпечують однозначність розуміння та можливості варіації складових модулів системи, а також їх взаємозамінності. Це створює гнучкий фундамент для подальших удосконалень архітектури системи. В той же час, чітка формалізація вирішуваної задачі разом з обраними методами представлення та перетворення даних задають послідовність етапів розв'язання, що є необхідною умовою для проведення об'єктивної наукової оцінки отриманих результатів.

3 Розробка системи аналізу програмного коду з використанням гібридного методу пошуку вразливостей

3.1 Побудова проміжного представлення програмного коду

3.1.1 Побудова абстрактного синтаксичного дерева

Абстрактне синтаксичне дерево (AST) є одним з ключових інструментів аналізу та представлення структури програмного коду. Воно дозволяє подати вихідний код у вигляді ієрархічного дерева, де кожен вузол відповідає певному фрагменту коду чи синтаксичній конструкції. Завдяки AST можна не лише візуалізувати логічну організацію програми, але й ефективно проводити її аналіз, включаючи пошук потенційних вразливостей.

Однак для коректної побудови AST необхідно дотримуватися певних вимог. Зокрема, якщо програмний код містить посилання чи використовує зовнішні бібліотеки, відповідні залежності мають бути наявними в середовищі побудови AST. Інакше можлива ситуація, коли частини коду, які містять виклики невідомих функцій чи звернення до відсутніх бібліотек, не будуть включені до побудованого AST. Це, в свою чергу, унеможливить подальший повноцінний аналіз цих фрагментів коду на предмет безпеки.

Існує декілька популярних інструментів та бібліотек для аналізу та обробки C/C++ коду, зокрема для генерації абстрактних синтаксичних дерев (AST)

Кожен з вказаних інструментів має свої унікальні особливості і застосування в аналізі та обробці коду C++:

1. Clang [37] - це передова частина компілятора для мов C, C++ та Objective-C, заснована на інфраструктурі LLVM. Clang славиться своєю

міцністю, підтримкою сучасних можливостей мови C++ і використовується в багатьох інструментах для статичного аналізу та рефакторингу.

2. GCC-XML [38] - цей інструмент здатен генерувати XML-представлення AST для коду C++, але він не так активно підтримується, як Clang.

3. Tree-sitter [39] - це універсальна бібліотека парсингу, яка ефективно обробляє великі бази коду і підтримує багато мов програмування, включаючи C++.

4. PyCParser [40] (для Python) - це бібліотека Python для парсингу коду C і C++ і генерації AST, зручна для роботи з C++ в середовищі Python.

5. ROSE Compiler [41] - цей компілятор — відкрита інфраструктура для створення перекладачів із коду в код, інструментів статичного аналізу тощо. Він підтримує C++ і надає API на C++ для роботи з AST.

7. ANTLR [42] (з граматикою C++) — це потужний генератор парсерів, який дозволяє визначати власні правила граматики для різних мов програмування, включаючи C++.

В даній роботі обрано Clang через його широкі можливості, активну підтримку та інтеграцію з екосистемою LLVM, що дозволяє ефективно виконувати аналіз та перетворення C/C++ коду для вирішення завдань детекції вразливостей.

Отже, перед побудовою AST для конкретного програмного файлу необхідно переконатися, що:

1. Встановлено усі бібліотеки та залежності, які використовуються у цьому файлі. Їх відсутність може призвести до часткового чи повного ігнорування окремих фрагментів коду під час генерації AST.
2. Шляхи пошуку бібліотек налаштовані коректно відповідно до середовища, де відбувається побудова AST. Інакше система не зможе знайти потрібні залежності навіть за їх наявності.

3. Використовується актуальна версія інструментарію для аналізу та компіляції. Застарілі компілятори можуть некоректно інтерпретувати сучасний синтаксис мови C/C++, що призведе до помилок під час генерації AST.

Якщо усі ці умови дотримані, можна сподіватися, що побудоване AST адекватно відображатиме структуру та зміст вихідного програмного коду. Інакше результуюче AST може бути неповним або містити структурні дефекти, що унеможливить його подальше використання для аналізу коду чи пошуку вразливостей. Для демонстрації процесу побудови AST розглянемо простий фрагмент коду на мові C++, зображений на Рис. 3.1:

```cpp
#include <iostream>
#include "helper.h"

void processArray(int array[], size_t size) {

  Helper helper;
  helper.initialize();

  for(int i = 0; i < size; ++i) {
    helper.update(array[i]);
  }

  helper.finalize();

  std::cout << "Done!" << std::endl;

}

int main() {

  int data[] = {1, 2, 3, 4};

  processArray(data, 4);

  return 0;

}
```

Рис. 3.1 - Фрагмент C++ коду

Цей код використовує стандартну бібліотеку iostream для введення/виведення, а також посилається на користувацьку бібліотеку helper.h, яка містить оголошення та реалізацію класу Helper.

Перед побудовою AST для цього файлу необхідно переконатися, що:

- Бібліотека iostream доступна. Зазвичай вона входить до стандартної бібліотеки C++ і не потребує додаткових дій.
- Файл helper.h та відповідна бібліотека, що реалізує клас Helper, присутні та доступні для компілятора. Інакше відповідні фрагменти коду можуть бути проігноровані під час генерації AST.

Припустимо, що усі умови дотримано і компілятор може успішно проаналізувати та обробити наведений вище код. Тоді в результаті роботи відповідних інструментів буде побудовано AST, фрагмент якого представлений на Рис. 3.2:


```
{
  "id": "0x81aa68",
  "kind": "TranslationUnitDecl",
  "loc": {},
  "range": {
    "begin": {},
    "end": {}
  },
  "inner": [
    {
      "id": "0x81b340",
      "kind": "TypedefDecl",
      "loc": {},
      "range": {
        "begin": {},
        "end": {}
      },
      "isImplicit": true,
      "name": "__int128_t",
      "type": {
        "qualType": "__int128"
      },
      "inner": [
        {
          "id": "0x81b000",
          "kind": "BuiltinType",
          "type": {
            "qualType": "__int128"
          }
        }
      ]
    },
```


Рис. 3.2 - Фрагмент побудованого AST: корінь дерева

Як бачимо, кореневим елементом AST є вузол TranslationUnitDecl, який репрезентує весь трансляційний блок. Він містить дочірні вузли для кожного окремого елемента цього блоку: директиви include, функціональні прототипи та визначення.

На наступному рівні відображено тіло функцій: для processArray це блок CompoundStmt (Рис. 3.3), а для main - ReturnStmt (Рис. 3.4). Кожен з цих блоків, в свою чергу, складається з дочірніх вузлів, що відповідають конкретним операторам чи конструкціям мови: оголошення змінної, цикл ForStmt (Рис. 3.5), виклик функції тощо.

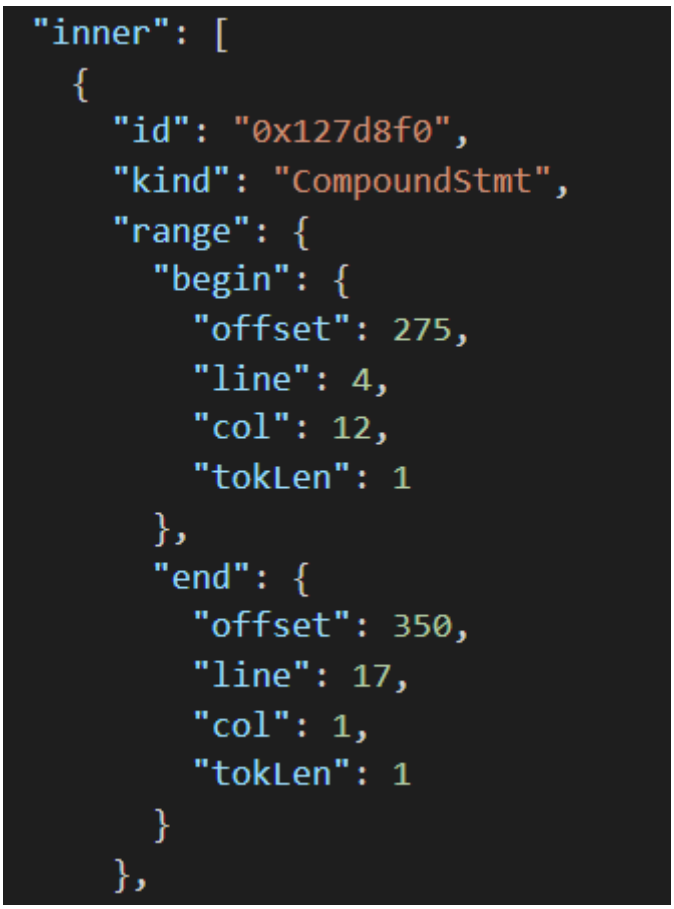

Рис. 3.3 - Батьківський блок функції processArray

```
{
  "id": "0x127d8e0",
  "kind": "ReturnStmt",
  "range": {
    "begin": {
      "offset": 339,
      "line": 25,
      "col": 3,
      "tokLen": 6
    },
    "end": {
      "offset": 346,
      "col": 10,
      "tokLen": 1
    }
  },
```


Рис. 3.4 - Батьківський блок конструкції return


```
{
  "id": "0x1274220",
  "kind": "ForStmt",
  "range": {
    "begin": {
      "offset": 133,
      "line": 9,
      "col": 3,
      "tokLen": 3
    },
    "end": {
      "offset": 198,
      "line": 11,
      "col": 3,
      "tokLen": 1
    }
  },
```


Рис. 3.5 - Батьківський блок циклу for

Завдяки такій ієрархічній структурі, AST дозволяє чітко бачити взаємозв'язки між елементами програми та їх вкладеність. Це спрощує аналіз коду та виявлення потенційно небезпечних фрагментів, адже з

контексту AST можна визначити як певна змінна чи функція використовується в інших частинах програми.

Варто відзначити декілька важливих моментів щодо побудови AST:

Існують різні способи візуального подання AST - UML-подібні діаграми, JSON, XML тощо. Втім сутність залишається незмінною: ієрархічна структура вузлів з відповідними зв'язками.

Глибина деталізації AST може варіюватися залежно від налаштувань аналізатора коду. Інколи доцільно абстрагуватися від надмірних низькорівневих подробиць задля компактності.

За необхідності, AST може бути доповнене додатковою інформацією про атрибути чи властивості окремих вузлів. Це розширює можливості подальшого аналізу.

3.1.2 Побудова кодових гаджетів

Кодовий гаджет являє собою фрагмент вихідного коду, що охоплює семантично пов'язані між собою рядки програми з певною **ключовою точкою** - потенційно вразливим викликом функції чи операцією.

Ключові точки являють собою фрагменти програми, які можуть містити потенційно небезпечні операції. Найчастіше це різноманітні виклики функцій, особливо тих, що належать до стандартних бібліотек - робота з пам’яттю, введенням/виведенням, мережевими операціям та ін.

Прикладами типових вразливих викликів функцій можуть бути:

- *malloc, calloc, realloc* - виділення та звільнення пам'яті
- *memcpy, strcpy, strcat* - робота з буферами та рядками
- *fopen, fread, fwrite* - читання/запис файлів
- *system, exec* - виконання зовнішніх команд
- *connect, bind, listen* - мережеві операції

Тобто ключовими точками виступають виклики функцій, які при некоректному використанні можуть призвести до типових вразливостей - переповнень буферів [47][48], витоків пам'яті [49], подвійного звільнення [50] тощо. Виявлення таких точок дозволяє звузити область пошуку потенційних дефектів безпеки.

На основі кожної ключової точки може бути сформовано відповідний кодовий гаджет - конструкція, що складається з:

1. Самої ключової точки (виклику вразливої функції)
2. Фрагментів коду, які пов’язані з ключовою точкою: аргументи або повернені значення

Такий кодовий гаджет формує семантично завершений блок програми, який може бути проаналізовано на предмет наявності вразливостей більш ізольовано від решти коду.

Основні етапи формування кодового гаджету:

1. **Вибір ключової точки**. На основі заздалегідь визначеного списку ключових точок, виконується пошук по AST ключових точок. На даному етапі важливо вирізняти ключову точку, яка викликається саме у коді, який аналізується, а не в межах виклику в стандартній бібліотеці C++. За допомогою метаданих, присутніх в AST, можливо визначити походження елементу дерева.

2. **Побудова зрізу AST**. Навколо обраної ключової точки виділяється фрагмент AST, який охоплює вузли, пов'язані з цією функцією. Глибина занурення в дерево AST визначається параметром алгоритму. Чи більший рівень вкладеності, тим більшими будуть результуючі кодові гаджети. Великий розмір кодового гаджету може негативно вплинути на подальшу точність класифікації, а також швидкодію методу,, тому емпіричним шляхом було підібрано розмір кодового гаджету 100.

3. **Екстракція коду з AST**. Із сформованого зрізу AST витягуються відповідні рядки вихідного коду, які об’єднуються в єдину конструкцію - кодовий гаджет.

4. **Знеособлення даних**. Ідентифікатори змінних, класів та функцій замінюються на узагальнені імена: VAR1, VAR2, FUNC1, CLASS1 тощо для уникнення перенавчання та покращення узагальнюючої властивості.

Різновиди кодових гаджетів. Залежно від типу ключової точки, виділяють два різновиди кодових гаджетів:

1. **Гаджети прямого проходу** - коли аналізується використання значення, що повертається з потенційно-вразливої функції (Рис. 3.6):

```
int* items = (int*) malloc(100); //Ключова точка
items[0] = 123;
```

Рис. 3.6 - Кодовий гаджет прямого проходу

2. **Гаджети зворотного проходу** - коли аналізується передача параметрів до вразливої функції (Рис. 3.7):

```
char src[SIZE];
char dest[SIZE];

strcpy(dest, src); // Ключова точка
```

Рис. 3.7 - Кодовий гаджет зворотного проходу

Такий розподіл дозволяє точніше проаналізувати вплив ключової точки на логіку програми в залежності від напрямку зв'язку.

Переваги кодових гаджетів для аналізу вразливостей:

- Дають цілісне уявлення про фрагмент коду та його оточення
- Містять важливий контекст навколо потенційних вразливостей
- Легше піддаються автоматизованому аналізу порівняно з повним кодом програми
- Спрощують процес категоризації та класифікації знайдених вразливостей
- Покращують повторне використання даних для машинного навчання

Таким чином, кодові гаджети являють собою зручну одиницю аналізу, яка дозволяє підвищити ефективність виявлення прихованих вразливостей у програмному коді комплексних систем за рахунок відсіювання несуттєвих деталей реалізації. Разом з тим, вони зберігають цінну інформацію про семантичні зв'язки між ключовими точками та оточенням, що є визначальним фактором для формування адекватних висновків щодо безпеки коду.

Схематичне представлення процесу побудови кодового гаджету зображено на Рис. 3.8:

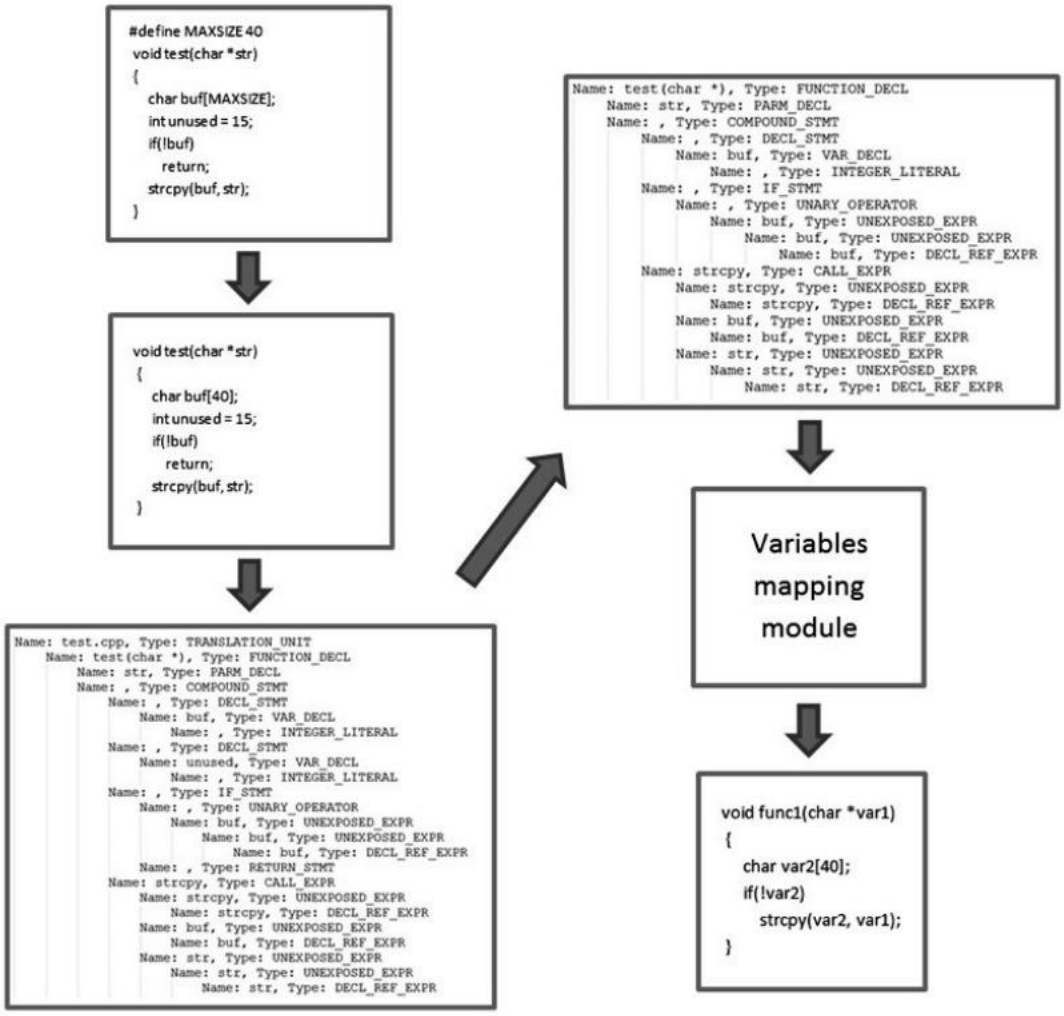

Рис. 3.8 - Процес формування кодового гаджету

3.1.3 Побудова векторного представлення коду

Трансформація текстових даних в семантичні вектори є методом взаємодії з обчислювальними системами для виконання завдань обробки природної мови та вирішення проблем за допомогою математичних підходів. В рамках даного дослідження передбачається використання векторизації для перетворення кодового гаджета у формат, придатний для подальшого використання в нейронних мережах. В даній главі розглянуті можливі алгоритми, які б дозволили це зробити.

Описуючи проблему представлення даних як особливість безпосереднього представлення текстових елементів, варто також відзначити структуру незмінності представлення для подальшої конструкції моделі. Модель векторизації у формі "мішка слів" вирішує цю проблему.

Для формування вхідного вектора для документу представлення елемент за елементом агрегується та обчислюється їх середнє значення. Вхідний вектор x у прикладі класифікації мови містить нормалізовану кількість біграм у документі D. Цей вектор може бути розкладений на середнє значення D векторів, кожен з яких відповідає певній позиції у документі i:

$$x = \frac{1}{|D|}\sum_{i=1}^{|D|} x^{D_{[i]}} \tag{3.1}$$

де $D_{[i]}$ - це біграма в документі на позиції i, а $x^{D_{[i]}}$ - це вектор із однією одиницею у позиції, що відповідає біграмі, і нулями на інших позиціях.

Таким чином, ми отримуємо векторне представлення документа фіксованого розміру словника, яке буде використовуватися пізніше. Це дозволяє знизити навантаження на обчислювальну потужність та використовувати цей метод в багатьох задачах. Серед недоліків цього підходу слід відзначити, що він не враховує порядок елементів тексту.

Проблема використання методу "мішка слів" (BOW) полягає у нерівномірному розподілі текстових елементів у векторах представлення, тобто рідкі елементи або загальновживані слова не відрізняються. Для вирішення цієї проблеми був впроваджений метод векторизації TF-IDF [43], який складається з двох частин: частота термінів (TF) та обернена частота документу (IDF). Обчислення таких значень виглядає наступним чином:

$$TF \sim IDF_{td} = TF_{td} \cdot IDF_{tc} \tag{3.2}$$

де TF (Частота термінів) - обчислення частоти входження текстового елемента в документ. Для вказаного текстового елемента цей показник може бути визначений як відношення кількості разів, коли елемент з'являється в документі, до загальної кількості слів у документі.

$$TF_{td} = \frac{N(t)}{N_d} \tag{3.3}$$

де $N(t)$ - це кількість входжень t в документ d, а N_d - загальна кількість слів у документі d.

Інверсійна частота документів (IDF) - це метрика, яка визначає важливість слова в корпусі багатьох документів. Вона обчислюється як логарифмічне відношення кількості всіх документів до кількості документів із зазначеним словом. Формалізуючи це, маємо наступне визначення формули:

$$IDF_{tc} = \frac{N_c}{N_c(t)} \tag{3.4}$$

де N_c - це кількість документів у корпусі c, а $N_c(t)$ - кількість документів, які містять слово t у корпусі c.

Таким чином, метод TFIDF хоч і визначає важливість слів у документах відносно корпусу текстів, але не враховує порядок слів і семантику, він також може вимагати значних обчислювальних ресурсів. Цей підхід корисний для виділення ключових слів, але не дозволяє враховувати глибокий зміст тексту.

Алгоритм Word2Vec, широко відомий та використовуваний, був розроблений Томашем Міколовим та його науковим колективом в рамках серії наукових статей [44], [45]. Word2Vec ініціюється базовою моделлю нейромережі для обробки природної мови та піддається подальшим модифікаціям з метою забезпечення більш ефективного процесу навчання. Важливо зауважити, що Word2Vec не представляє собою окремий алгоритм, а складається з програмного пакету, що реалізує два різних

способи представлення контексту (CBOW та Skip-Gram) та два різних вдосконалення (Negative-Sampling та Hierarchical Softmax), як показано на Рис. 3.9:

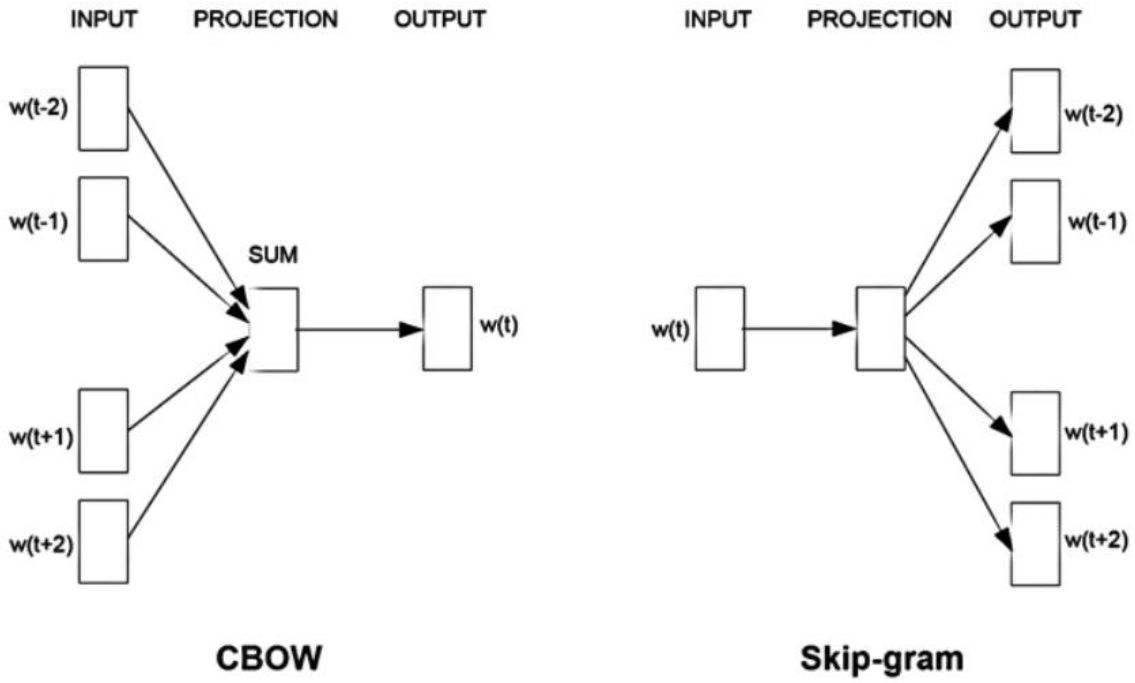

Рис. 3.9 - Архітектура моделі Word2Vec [45]

Word2Vec демонструє здатність захоплювати як синтаксичні, так і семантичні взаємозв'язки між різними словами у текстовому корпусі, що надає багатший контекст для розуміння асоціацій слів. Крім того, він вирізняється компактними та гнучкими векторами вбудовування, у відміну від попередніх алгоритмів, де розмір вектору вбудовування був пропорційним розміру словника, що робить його більш ефективним [46] та пристосованим до різних наборів даних. Таким чином, можна прийти до висновку, що використання моделі word2vec є більш доцільним, ніж BoW та TFIDF.

Векторизація кодових гаджетів відбувається наступним чином:

1. Токенізація кодових гаджетів: перший етап полягає в розбитті кодових гаджетів на окремі токени.
2. Тренування моделі: на наступному етапі токенізовані кодові гаджети використовуються для тренування моделі word2vec.

Отримання векторних представлень: після тренування для кожного токена або гаджету може бути отримано векторне представлення.

Кожен кодовий гаджет потребує перетворення вектором через його символьне представлення. Для цього ми розбиваємо символьне представлення кодового гаджету на послідовність токенів за допомогою лексичного аналізу, включаючи ідентифікатори, ключові слова, оператори і символи. Наприклад, символьне представлення кодового гаджету,

"strcpy(VAR5, VAR2);"

представлене послідовністю з 7 токенів:

"strcpy", "(", "VAR5", ",", "VAR2", ")", і ";"

Це призводить до створення великого корпусу токенів. Для перетворення цих токенів у вектори ми використовуємо інструмент word2vec, який вибрано через його широке застосування у текстовому аналізі [51]. Цей інструмент базується на ідеї розподіленого представлення, яке відображає токен у ціле число, яке потім перетворюється у вектор фіксованої довжини [52].

Оскільки кодові гаджети можуть мати різну кількість токенів, відповідні вектори можуть мати різну довжину. Оскільки BLSTM приймає вектори однакової довжини на вхід, нам потрібно внести корективи. Для цього ми вводимо параметр τ як фіксовану довжину векторів, відповідних кодовим гаджетам.

1. Коли вектор коротший за τ, існують два випадки: якщо кодовий гаджет створений зі зворотного відрізка або створений шляхом комбінування декількох зворотних відрізків, ми доповнюємо

нулями в початку вектора; в іншому випадку, ми доповнюємо нулями в кінці вектора.

2. Коли вектор довший за τ, також є два випадки: якщо кодовий гаджет створений з одного зворотного відрізка або створений шляхом комбінування декількох зворотних відрізків, ми видаляємо початкову частину вектора; в іншому випадку, ми видаляємо кінцеву частину вектора.

Це забезпечує, що останній оператор в кожному кодовому гаджеті, створеному зі зворотного відрізка, є викликом функції бібліотеки/API, і що перший оператор в кожному кодовому гаджеті, створеному зі зворотного відрізка, є викликом функції бібліотеки/API. В результаті кожний кодовий гаджет представлений у вигляді вектора довжиною τ бітів. Довжина векторів пов'язана з кількістю прихованих вузлів на кожному рівні нейронної мережі, який є параметром, що може бути налаштований для покращення точності виявлення вразливостей.

3.2 Метод пошуку вразливостей в програмному коді з використанням глибокого навчання

3.2.1 Вибір архітектури нейронної мережі для вирішення задачі пошуку вразливостей

Для вирішення задачі пошуку вразливостей в програмному коді застосовується широкий спектр архітектур нейронних мереж. Кожна з них має свої переваги та недоліки. Розглянемо основні типи, що використовуються:

Рекурентні нейронні мережі (RNN) [73]. Рекурентні нейронні мережі є одним з найбільш поширених підходів для виявлення вразливостей в програмному коді за допомогою методів машинного навчання [72]. Це

пов'язано з тим, що RNN добре підходять для моделювання послідовностей, а програмний код являє собою послідовність токенів (змінних, ключових слів, операторів тощо) [25][26][60].

Основною перевагою RNN є те, що вони здатні запам'ятовувати контекст у вигляді внутрішнього стану. Це дозволяє аналізувати код не окремими фрагментами, а з урахуванням логіки всієї програми. На відміну від feed-forward нейронних мереж, RNN розуміють порядок слідування команд та їхній взаємозв'язок. Для підвищення ефективності роботи з довгими послідовностями застосовуються спеціальні модифікації RNN - LSTM та GRU [25][26][60].

Згорткові нейронні мережі (CNN) [74]. На відміну від RNN, згорткові нейронні мережі рідше використовуються в задачах аналізу програмного коду, проте все одно мають певні переваги. Основним принципом роботи CNN є використання операції згортки - вона дозволяє мережі автоматично виділяти важливі локальні ознаки з вхідних даних.

Оскільки багато типів вразливостей пов'язані з певними шаблонами кодування або синтаксичними конструкціями, CNN можуть ефективно їх розпізнавати. Наприклад, мережа Idea2Code використовує CNN для класифікації функцій коду та генерації відповідного синтаксису мовою Python [75]. Така здатність до синтаксичного аналізу може застосовуватись і в системах пошуку вразливостей на основі CNN.

Основним недоліком CNN є необхідність великої кількості даних для навчання, що є проблемою через обмежену наявність датасетів вразливого коду. Однак це можна частково вирішити за допомогою попередньо навчених моделей обробки природної мови та технік підсилення даних. Також CNN не враховують контекст всієї програми на відміну від RNN.

Двонаправлена LSTM мережа (BLSTM) [76]. BLSTM є варіантом архітектури LSTM, що була спеціально розроблена для роботи з послідовностями даних. У контексті аналізу вразливостей BLSTM може

виявитися особливо корисною завдяки своїй здатності вивчати контекст з обох кінців послідовності.

BLSTM містить два шари LSTM, які працюють у протилежних напрямках – один у прямому, інший – у зворотньому. Це означає, що кожен часовий крок у послідовності отримує інформацію як з минулого, так і з майбутнього в контексті даної послідовності.

Вхідні дані: Аналогічно звичайній LSTM, BLSTM приймає на вхід послідовність даних. У контексті аналізу вразливостей це може бути послідовність токенів або інструкцій у коді.

Прямий шар LSTM: Обробляє дані, починаючи від першого елемента до останнього.

Зворотній шар LSTM: Обробляє дані, починаючи від останнього елемента до першого.

Об'єднання: Виходи з обох шарів LSTM об'єднуються на кожному часовому кроці, щоб отримати більш повний контекст.

Завдяки своїй бідирекційній структурі, BLSTM може вивчати залежності в коді, які звичайні LSTM можуть проґавити. Наприклад, в контексті пошуку вразливостей, важливо не тільки дивитися на те, який код йде перед потенційною вразливістю, але і те, який код слідує за нею. BLSTM дозволяє це реалізувати, враховуючи контекст з обох боків потенційної вразливості.

3.2.2 Процес пошуку вразливостей в програмному коді

Після побудови проміжного векторного представлення кодових гаджетів, наступним кроком є безпосередньо пошук вразливостей за допомогою розробленої нейромережевої моделі. Цей процес складається з декількох етапів:

Попередня обробка даних. На цьому етапі вхідні вектори кодових гаджетів піддаються низці перетворень для приведення даних до

оптимального формату для подальшого аналізу моделлю. Зокрема, виконуються наступні процедури:

Нормалізація векторів для мінімізації впливу викидів та екстремальних значень. Застосовуються різні схеми нормалізації, включаючи нормалізацію Мін-Макс, Z-нормалізацію, тощо.

Балансування класів для усунення дисбалансу між вразливим та невразливим кодом. Це досягається за рахунок дублювання зразків менших класів або видалення надлишкових представників більших класів.

Розбиття даних на навчальну, валідаційну та тестову вибірки у співвідношенні 60/20/20 або 70/15/15.

Такі перетворення забезпечують створення оптимального вхідного набору даних, придатного як для ефективного навчання моделі, так і для адекватної оцінки її метрик якості.

Навчання моделі. На цьому етапі відбувається безпосереднє тренування розробленої BLSTM-моделі на підготовлених векторних даних з метою налаштування її вагових коефіцієнтів для подальшої класифікації нових зразків коду.

Процес навчання виконується ітераційно шляхом багаторазового проходження вхідних даних через модель та коригування її параметрів за допомогою градієнтних методів оптимізації з метою мінімізації втрат і помилок класифікації. Ключові параметри процесу навчання:

- Функція втрат - бінарна крос-ентропія
- Оптимізатор - Adam
- Швидкість навчання - 0.001
- Кількість епох - 15
- Розмір батчу - 32

Динаміка зміни функції втрат на навчальній та валідаційній вибірках відображає якість навчання моделі. Його завершення відбувається за умови

досягнення плато для цих метрик (коли подальше навчання не веде до значущого поліпшення результатів).

Оцінка моделі. Після завершення навчання виконується оцінка якості роботи BLSTM-моделі на тестовому наборі даних, який не брав участі в тренуванні. Це дозволяє об'єктивно виміряти її узагальнюючу здатність до класифікації раніше невідомих зразків коду.

В якості ключових метрик оцінки використовуються:

- Точність (accuracy) - загальна частка правильних результатів серед усіх випадків
- Відсоток хибно позитивних результатів (FPR) - частка помилково ідентифікованих вразливостей
- AUC-ROC - інтегральний показник якості, заснований на характеристиці операційних параметрів

Бажаними є високі значення точності та AUC при низькому рівні FPR. Ці метрики відображають здатність моделі ефективно відрізняти вразливий та безпечний код.

Аналіз результатів. Отримані метрики якості моделі на тестовій вибірці аналізуються для оцінки загальної ефективності розробленого рішення у задачі детекції вразливостей в коді.

Також вивчаються розподіли кількості семплів по класах (вразливий/невразливий код) у розрізі різних діапазонів вихідних значень ймовірностей, отриманих на виході BLSTM-моделі. Це дозволяє встановити оптимальні пороги прийняття рішень щодо наявності вразливості.

Крім того, аналізуються окремі приклади хибно класифікованих зразків для виявлення можливих шаблонів помилок моделі. Ця інформація може використовуватись для подальшого вдосконалення архітектури чи гіперпараметрів нейромережі.

За результатами пошуку вразливостей формуються відповідні аналітичні звіти, що містять як загальну статистику щодо частки потенційно небезпечних фрагментів коду, так і їх детальний перелік з точною локалізацією.

Ці відомості можуть використовуватись розробниками та архітекторами ПЗ для оперативного реагування на виявлені проблеми безпеки, а також для вдосконалення загальної архітектури програмних систем з метою запобігання потенційним вразливостям в майбутньому.

Таким чином, запропонований процес дозволяє ефективно реалізувати пошук вразливостей в програмному коді за допомогою нейромережевої моделі, що базується на глибокому контекстному аналізі даних з використанням бідирекційних LSTM. За рахунок автоматичного навчання на великих масивах даних, модель може точно ідентифікувати складні шаблони, що вказують на потенційні дефекти безпеки, значно оптимізуючи та підвищуючи результативність процесу верифікації коду.

3.3 Метод класифікації вразливостей в програмному коді з використанням ковзного хешування AST

3.3.1 Підготовка бази знань хешів вразливостей

Підготовка бази знань хешів вразливостей є першим етапом реалізації методу класифікації на основі ковзного хешування AST. Від повноти та якості цієї бази безпосередньо залежатиме точність розпізнавання типів вразливостей системою.

Процес формування бази знань включає наступні кроки:

Формування вибірки вразливого коду. Перш за все, необхідно зібрати репрезентативні приклади вразливого коду, які охоплюють максимально широкий спектр потенційних вразливостей. Для цього доцільно скомбінувати такі джерела:

- Відкриті бази вразливостей (CVE Details, NVD)
- Спеціалізовані набори вразливого коду (SARD)

Побудова AST. Наступним кроком для кожного фрагменту коду потрібно побудувати AST - абстрактне синтаксичне дерево, яке відображає логічну структуру та зв'язки елементів програми. Саме ця структура аналізуватиметься для генерації хешів.

Ковзне хешування AST. Після отримання "деревовидних" представлень, кожне з них піддається ітераційній процедурі ковзного хешування з фіксованим розміром вікна N. Результатом є набори унікальних бітових хеш-рядків, які "згортають" локальні фрагменти AST.

Збереження результатів. Сформовані множини хешів для кожного випадку вразливості зберігаються у базі даних чи іншому сховищі, що дозволяє ефективно індексувати та виконувати пошук даних. Також зазвичай зберігається супутня інформація: клас вразливості, локалізація, CVE-ідентифікатор тощо.

Приклад вразливості з датасету показано на Рис. 3.10:

```
void CWE78_OS_Command_Injection__wchar_t_file_w32_spawnv_12_bad()
{
    wchar_t * data;
    wchar_t dataBuffer[100] = COMMAND_ARG2;
    data = dataBuffer;
    if(globalReturnsTrueOrFalse())
    {
        {
            /* Read input from a file */
            size_t dataLen = wcslen(data);
            FILE * pFile;
            /* if there is room in data, attempt to read the input from a file */
            if (100-dataLen > 1)
            {
                pFile = fopen(FILENAME, "r");
                if (pFile != NULL)
                {
                    /* POTENTIAL FLAW: Read data from a file */
                    if (fgetws(data+dataLen, (int)(100-dataLen), pFile) == NULL)
                    {
                        printLine("fgetws() failed");
                        /* Restore NUL terminator if fgetws fails */
                        data[dataLen] = L'\0';
                    }
                    fclose(pFile);
                }
            }
        }
    }
    else
    {
        /* FIX: Append a fixed string to data (not user / external input) */
        wcscat(data, L"*.*");
    }
    {
        wchar_t *args[] = {COMMAND_INT_PATH, COMMAND_ARG1, COMMAND_ARG3, NULL};
        /* wspawnv - specify the path where the command is located */
        /* POTENTIAL FLAW: Execute command without validating input possibly leading to command injection */
        _wspawnv(_P_WAIT, COMMAND_INT_PATH, args);
    }
}
```

Рис. 3.10 - Приклад вразливості CWE78 з датасету SARD

Схематичне представлення методу побудови бази знань хешів вразливостей представлено на Рис. 3.11:

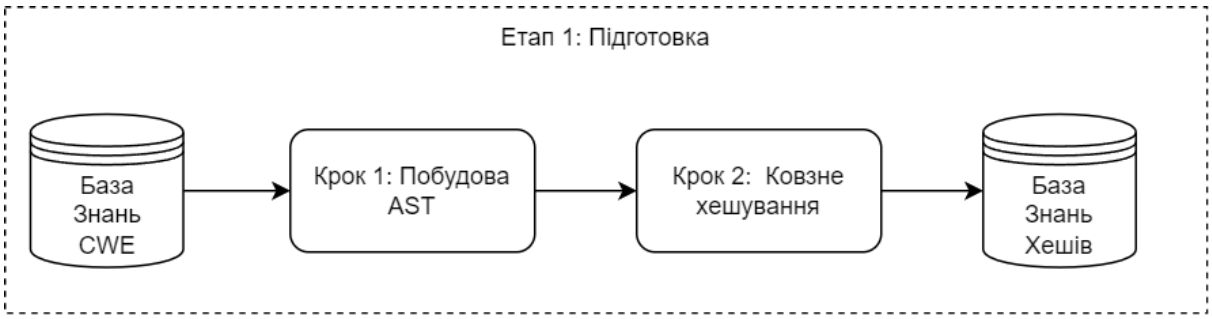

Рис. 3.11 - Метод побудови бази знань хешів вразливостей

Приклад файлу з побудованими хешами вразливості та мета інформацією зображено на Рис. 3.12:

Рис. 3.12 - Обчислені хеші з локацією у файлі для вразливості при розмірі вікна N=3

Створена база знань на основі хеш-відбитків вразливих фрагментів програмного коду є фундаментом методу класифікації вразливостей. Її репрезентативність, повнота та якість безпосередньо визначатимуть результативність системи в реальних умовах застосування для аналізу безпеки ПЗ. Тому формування, розширення та вдосконалення цієї бази знань є одним з ключових факторів успіху запропонованого підходу.

3.3.2 Процес класифікації вразливостей з використанням методу ковзного хешування AST

Процес класифікації вразливостей є наступним логічним кроком роботи системи після ідентифікації потенційно небезпечних фрагментів коду за допомогою нейромережевої моделі.

Метою даного етапу є визначення типу знайдених вразливостей відповідно до загальноприйнятої системи класифікації вразливостей програмного забезпечення CWE.

Класифікація CWE є ієрархічною таксономією, що містить сотні різних типів вразливостей, систематизованих за категоріями в залежності від природи уразливості, механізму реалізації атаки тощо.

Процес класифікації вразливостей включає наступні етапи:

Формування вхідних даних. На цьому кроці фрагменти вихідного коду, які були ідентифіковані на етапі детекції вразливостей як потенційно уразливі з ймовірністю вище встановленого порогу (наприклад, 70%), формуються для подальшої класифікації.

В якості даних використовуються зрізи AST цих фрагментів коду, отримані на етапі побудови кодових гаджетів. Саме AST містить усю необхідну інформацію про структуру та синтаксис коду, що є ключовими ознаками для класифікації типу вразливості.

Ковзне хешування вузлів AST. Кожен AST піддається процедурі ковзного хешування - ітераційного обходу дерева у глибину із обчислення хеш-значень для послідовних фіксованих вікон з N вузлів. Розмір вікна N є важливим параметром, що впливає на якість класифікації.

Застосування криптографічно стійкої хеш-функції (наприклад, MD5) дозволяє отримати унікальні бітові рядки фіксованої довжини, які відображають структуру кожного вікна AST.

Порівняння з базою хешів вразливостей. Набори хешів, згенеровані для кожного фрагменту коду, далі порівнюються із заздалегідь

підготовленою базою знань еталонних хешів вразливостей різних типів згідно з CWE.

Чим вищий рівень збігів хешів конкретного фрагменту коду з якимось еталоном з бази знань, тим вища ймовірність, що цей фрагмент містить вразливість відповідного класу. Таким чином відбувається класифікація.

Варто зазначити, що такий підхід дозволяє не просто визначити клас вразливості, а назвати кілька потенційних класів вразливості. На Рис. 3.13 схематично зображено метод класифікації вразливостей з використанням ковзного хешування AST:

Рис. 3.13 - Робота алгоритму пошуку вразливостей на основі ковзного хешування AST

Переваги запропонованого підходу:

- Можливість класифікувати сотні типів вразливостей за універсальною системою CWE;
- Висока швидкодія та масштабованість методу;
- Простота розгортання та використання рішення;
- Потенціал для інкрементного навчання шляхом поповнення бази знань.

Недоліками є певна чутливість до синтаксичних мутацій коду, які не змінюють його логіки, а також залежність якості класифікації від обсягу бази знань еталонів вразливостей.

Тому актуальним напрямком вдосконалення є дослідження альтернативних алгоритмів хешування AST, які б краще враховували семантичну схожість коду, не зважаючи на неістотні синтаксичні відмінності. Перспективним видається використання методів на зразок SimHash для побудови вразливих хеш-відбитків коду.

Ще одним напрямком є розвиток бази знань хешів за рахунок додавання нових зразків реальних вразливостей, знайдених в процесі функціонування системи. Їх інтеграція в якості еталонів дозволить покращити релевантність бази вразливостей до конкретних прикладних задач та предметних областей аналізу програмного коду.

Таким чином, подальші дослідження дадуть можливість підвищити ефективність та універсальність методу класифікації вразливостей на основі ковзного хешування AST для використання в широкому спектрі практичних задач верифікації програмного забезпечення.

3.3.3 Визначення оптимального розміру вікна для хешування

Розмір вікна N для ковзного хешування AST є важливим параметром, що впливає на результативність методу класифікації вразливостей. Занадто мале значення призведе до втрати цінного контексту та зниження точності, в той час як надмірно велике - збільшить обчислювальну складність та уповільнить швидкодію алгоритму.

Для емпіричного визначення оптимального розміру вікна N пропонується наступна методика:

1. Для кожного з можливих значень N в діапазоні від 1 до K виконати процедуру ковзного хешування AST з бази вразливостей та обчислити відсоток збігів з еталонними хешами для кожного N.

Також для контролю обчислити аналогічний показник для випадкової вибірки невразливих фрагментів коду з різних проектів.

2. Для кожного N побудувати графік залежності рівня збігів від розміру вікна для вразливого та невразливого коду.
3. Вибрати значення N, при якому буде спостерігатися найбільша різниця (контраст) між показником збігів для вразливих та безпечних фрагментів коду.

Таке N буде оптимальним, оскільки дозволяє найкращим чином диференціювати вразливий код від невразливого при класифікації.

Таким чином, результат роботи алгоритму для вибору оптимального значення розміру вікна для хешування виглядає наступним чином (рис. 3.14):

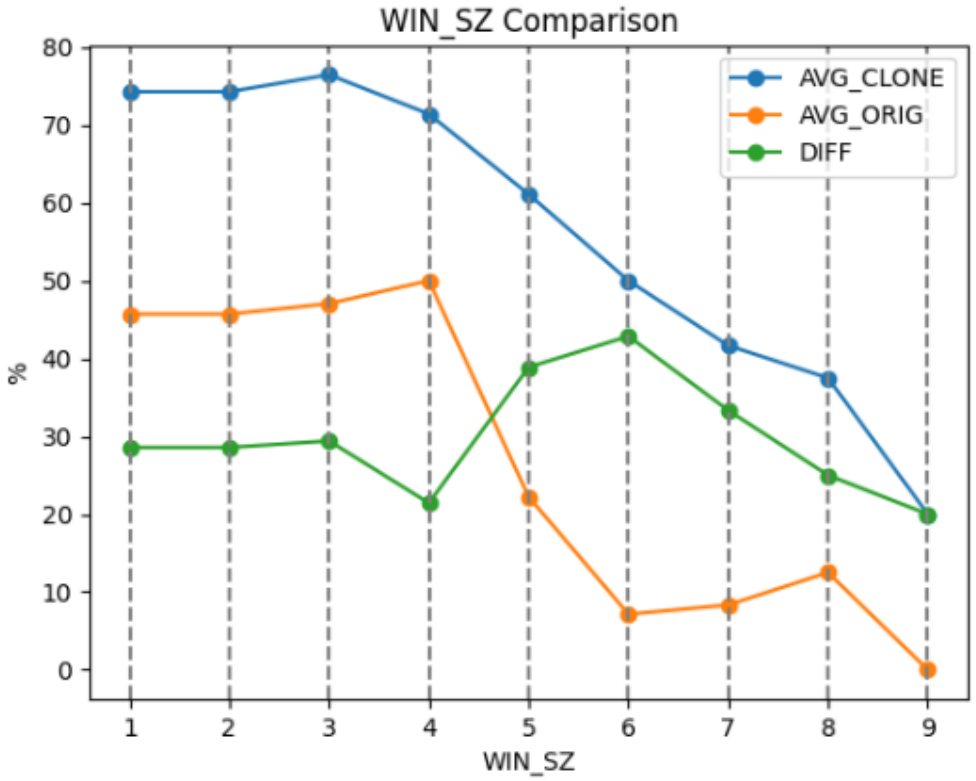

Рис. 3. 14 - Визначення оптимального розміру вікна для хешування

Де, AVG_CLONE - порогове значення, при якому код вважається плагіатом,

AVG_ORIG - порогове значення при якому код вважається оригінальним,

DIFF - модуль різниці AVG_CLONE та AVG_ORIG.

Як видно з рис. 3.14, при N=6 спостерігається найбільший розрив між кривими для вразливого та безпечного коду. Отже, саме 6 є оптимальним розміром вікна в цьому випадку.

Переваги запропонованої евристики:

- Автоматичне емпіричне визначення N без необхідності ручного підбору
- Урахування специфіки конкретних даних, на яких навчається система
- Можливість повторної оптимізації N при зміні вхідних даних
- Простота реалізації методики

Разом з тим, описаний підхід має деякі обмеження:

- Потребує наявності репрезентативних даних як для вразливого, так і безпечного коду
- Вимагає додаткових обчислювальних витрат для перебору варіантів N
- Не гарантує глобального оптимуму для всіх можливих випадків

Тому актуальним є дослідження альтернативних методів визначення оптимального розміру вікна ковзного хешування AST, які могли б враховувати природні кластери даних, апріорні знання щодо типових розмірів вразливих конструкцій тощо.

Можливості для вдосконалення включають застосування еволюційних алгоритмів, методів кластерного аналізу, а також побудову

ансамблів моделей з різними N, з наступним агрегуванням їх висновків щодо класифікації вразливостей.

Загалом, визначення оптимального розміру вікна ковзного хешування AST залишається важливою задачею для підвищення загальної точності методу класифікації вразливостей на основі запропонованого підходу. Комбінація емпіричних та аналітичних стратегій оптимізації цього параметра надає можливості для подальшого вдосконалення системи.

3.4 Висновки

У третьому розділі дисертації здійснено детальний аналіз існуючих методологій та підходів до детекції вразливостей у програмному коді, методів представлення програмного коду, а також методів векторизації програмного коду.

Аналіз також охоплює методи, що ґрунтуються на технологіях глибокого навчання, зокрема на різновидах нейронних мереж. Ці методи продемонстрували значний потенціал у виявленні прихованих закономірностей у великомасштабних наборах даних. Однак, вони можуть стикатися з викликами, пов'язаними з масштабуванням та адаптацією до нових даних.

Перспективним напрямком є також використання абстрактних синтаксичних дерев (AST), що дозволяють здійснювати структурний аналіз коду. Проте традиційні методи, що базуються на AST, можуть бути надмірно ресурсоємними та виявитися неефективними при обробці великих масивів даних.

У відповідь на ці недоліки, у дисертації запропоновано дві інноваційні моделі для детекції вразливостей:

1. Модель на основі двонаправленої рекурентної нейронної мережі довгої короткочасної пам'яті (BLSTM) яка забезпечує контекстний

аналіз, враховуючи як попередні, так і наступні фрагменти коду при ідентифікації вразливостей. Ця модель може бути адаптована та донавчена на специфічних даних проекту, що підвищує її точність та адаптивність.

2. Модель, заснована на ковзному хешуванні вузлів AST і порівнянні з базою відомих вразливостей яка забезпечує швидке виявлення типових вразливостей без потреби у складному контекстному аналізі.

Ці моделі, доповнюючи одна одну, формують комплексну систему для детекції вразливостей, підвищуючи надійність та точність виявлення потенційних проблем у програмному коді.

4 Експерименти та результати роботи системи аналізу програмного коду з використанням гібридного методу пошуку та класифікації вразливостей

4.1 Експериментальні результати роботи системи аналізу програмного коду в задачі пошуку вразливостей

У даній главі проводиться експериментальна оцінка розробленої системи оцінки безпеки програмного коду у контексті задачі пошуку вразливостей. Оскільки запропонована система вирішує дві ключові задачі – безпосередньо пошук вразливостей та їх подальша класифікація, ефективність її роботи оцінюється окремо для кожного з цих компонентів.

Для оцінки моделі пошуку вразливостей було використано датасет SARD (Software Assurance Reference Dataset) [54] з бази даних вразливостей NVD [55]. Датасет містить 38801 зразок програмного коду написаного на мові C++ та 92645 зразків коду на мові C. Кожен зразок складається з двох варіантів однієї програми – один містить певну вразливість, а інший є безпечною версією без даної вразливості. Ці два варіанти розділені директивою препроцесора.

Код було перетворено у векторне представлення з використанням кодових гаджетів та Word2Vec, як описано у попередніх розділах. Розмір вектору складав 100. Таким чином, вхідними даними для моделі були вектори довжиною 100, які відображали фрагменти коду.

Датасет був розділений на навчальну, валідаційну та тестову вибірки, частка яких від загальної кількості семплів складала 80%, 10%, 10% відповідно.

Для класифікації коду на вразливий/невразливий була використана двонаправлена рекурентна нейронна мережа з довгою короткочасною пам’яттю (BLSTM). Мережа складалася з 4 BLSTM шарів по 256

прихованих одиниць кожен, шару дропауту з ймовірністю випадіння 0.3 для регуляризації, та вихідного сигмоїдного шару для отримання ймовірності належності до класу "вразливість".’

Функція втрат визначалася як бінарна крос-ентропія. Мережа тренувалася протягом 15 епох із використанням оптимізатора Adam [61], швидкістю навчання 0.001 та розміром батчу 32.

Для оцінки ефективності моделі пошуку вразливостей було обрано три ключові метрики:

1. Точність - ця метрика показує відсоток правильно класифікованих вразливих та безпечних фрагментів коду відносно загальної кількості зразків. Точність розраховується за формулою:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

де TP - правильно ідентифіковані вразливі фрагменти, TN - ідентифіковані безпечні фрагменти, FP - хибно позитивні, тобто безпечні фрагменти, ідентифіковані як вразливі, FN - хибно негативні, тобто вразливі фрагменти, ідентифіковані як безпечні.

2. Відсоток хибно позитивних результатів (FPR) - ця метрика показує частку хибно позитивних результатів серед усіх насправді негативних (безпечних) випадків. FPR розраховується за формулою:

$$FPR = \frac{FP}{FP + TN}$$

Нижче значення FPR вказує на меншу кількість хибних сигналів, щодо наявності вразливостей там, де їх насправді немає.

3. Площа під кривою ROC (AUC) - цей показник являє собою інтегральну метрику якості бінарної класифікації на всіх можливих порогових значеннях. AUC може приймати значення від 0 до 1, де вище значення вказує на кращу загальну ефективність класифікатора.

Показники роботи моделі для тестової вибірки датасету SARD продемонстровані в Таблиця 4.1:

Таблиця 4.1. Показники роботи моделі пошуку вразливостей з використанням нейронної мережі на тестовій вибірці датасету SARD

Точність (%)	**FPR (%)**	**AUC**
94.1	4.4	0.97

Результати розробленої моделі пошуку вразливостей з використанням нейронної мережі були порівняні з іншими моделями пошуку вразливостей в коді, що базуються на різних підходах: пошук за шаблоном (Flawfinder [16], Checkmarx [12]), глибоке навчання (VulDeePecker [25], SySeVR [26]), пошук подібності (VUDDY [19], VulPecker [20]). Результати наведені у Таблиця 4.2:

Таблиця 4.2. Результати порівняння моделі пошуку вразливостей з використанням нейронної мережі та іншими рішеннями

Модель \ Метрика	**Точність (%)**	**FPR (%)**
Flawfinder	69.8	23.9
Checkmarx	72.9	20.8
Vulpecker	79.4	12.4
VUDDY	82.7	10.2
VulDeePecker	92.2	8.5

SySeVR	97.0	2.3
Запропонована модель пошуку вразливостей	**94.1**	**4.4**

Також була проведена оцінка вищевказаних моделей за критерієм швидкодії (T) та витрат пам’яті (RAM). Результати оцінки представлені в Таблиця 4.3:

Таблиця 4.3. Результати оцінки швидкодії та витрат ресурсів в моделях пошуку вразливостей

Модель \ Метрика	**Середній час обробки програми (с)**	**Мінімальна об’єм оперативної пам’яті (MB)**
Flawfinder	0.9	512
Checkmarx	1.2	512
Vulpecker	8.4	2048
VUDDY	12.5	2048
VulDeePecker	3.2	1024
SySeVR	15.7	4096
Запропонована модель пошуку вразливостей	**2.9**	**1024**

Оскільки розроблена модель пошуку вразливостей з використанням нейронної мережі є ідейним продовженням моделі VulDeePecker [25], доцільно буде провести більш детальну оцінку у порівнянні з VulDeePecker.

Для навчання моделі VulDeePecker в якості навчального набору були використані заздалегідь підготовлені кодові гаджети, які були отримані від авторів оригінальної статті [25]. Початковий набір даних містив у собі 8122

приклади коду, які мали вразливості переповнення буфера, а також 1729 прикладів коду з вразливостями, пов’язаними з некоректним управлінням ресурсами. Цей набір був поділений на навчальний та тестовий набори в співвідношенні 80% та 20%, відповідно.

Таблиця 4.4. Порівняння точності класифікації моделей VulDeePecker та запропонованої моделі на основі нейронної мережі

Вибірка / Точність	**Запропонована модель**	**VulDeePecker**
Навчальна	97.9	96.4
Тестова	94.1	92.2

Для додаткової оцінки ефективності моделей, навчених на даних з різних джерел, було проведено експеримент, в рамках якого здійснювався пошук вразливостей в у вихідному коді протоколу Bitcoin [56]. Результати цього експерименту, представлені в таблиці 4.5. Даний експеримент дозволяє провести порівняння моделей за різними метриками, що вказують на кількість знайдених кодових гаджетів. Такий аналіз допомагає краще зрозуміти, наскільки ефективні обидві моделі в задачі пошуку вразливостей у великих кодових базах. Також даний експеримент допомагає визначити працездатність моделей та застосовність в практичних задачах безпеки програмного забезпечення.

Таблиця 4.5. Порівняння моделі пошуку вразливостей з моделлю VulDeePecker

Метрика	**VulDeePecker**	**Запропонована модель пошуку вразливостей**
Кількість кодових гаджетів	1510	

Кількість кодових гаджетів позначена як “не вразливі”	970	1326
Кількість кодових гаджетів позначена як “вразливі”	540	184
Кількість кодових гаджетів позначена як “не вразливі” обома моделями	802	802
Кількість кодових гаджетів позначена як “вразливі” обома моделями	16	16
Кількість унікальних кодових гаджетів позначена як “не вразливі”	168	524
Кількість унікальних кодових гаджетів позначена як “вразливі”	524	168

Оскільки модель пошуку вразливостей в коді видає результат в діапазоні від 0 до 1, значення близькі до 1 можна інтерпретувати як наявність чіткого шаблону вразливості, тоді як значення близькі до 0 показують відсутність шаблону вразливості в коді. Таблиця 4.6 демонструє розподіл вихідних значень моделі пошуку вразливостей для класифікованих як «вразливий» чи «не вразливий» фрагментів коду, де P – це результат прогнозування вразливості фрагменту коду, $P \in [0, 1]$.

Таблиця 4.6. Розподіл вихідних значень моделі пошуку вразливостей для проекту Bitcoin

P	**Клас**	**Кількість**
$0.9 < P \leq 1.0$	Вразливий	40
$0.8 < P \leq 0.9$	Вразливий	39

$0.7 < P \leq 0.8$	Вразливий	61
$0.6 < P \leq 0.7$	Вразливий	20
$0.5 < P \leq 0.6$	Вразливий	24
$0.4 < P \leq 0.5$	Не вразливий	43
$0.3 < P \leq 0.4$	Не вразливий	77
$0.2 < P \leq 0.3$	Не вразливий	341
$0.1 < P \leq 0.2$	Не вразливий	470
$0.0 \leq P \leq 0.1$	Не вразливий	579

4.2 Результати роботи системи аналізу програмного коду з використанням гібридного методу пошуку та класифікації вразливостей

У цій главі представлені результати експериментів з оцінки здатності розробленої системи класифікувати знайдені вразливості відповідно до загальноприйнятої таксономії CWE (Common Weakness Enumeration) [33]. Метою є визначення точності віднесення виявлених вразливих фрагментів коду до конкретних класів згідно з базою CWE на основі запропонованого підходу з використанням ковзного хешування AST. Оскільки кожен фрагмент коду у використаному датасеті SARD містить відомий клас вразливості, окрім оцінки ефективності пошуку, можна також виконати оцінку точності класифікації знайдених вразливостей. Датасет SARD налічує близько 150 класів вразливостей, згідно таксономії CWE.

Для оцінки ефективності класифікації використовувалася метрика багатокласової точності (multiclass accuracy), яка визначає загальну частку правильно класифікованих вразливостей серед усіх тестових зразків з урахуванням усіх наявних класів CWE. Також застосовувалася інтегральна метрика “локальне розгортання”, яка відображає можливість розгортання

розробленого рішення на обчислювальних системах відносно невеликої потужності, що є важливим для можливості його використання малими та середніми IT-компаніями. Результати порівнювалися з існуючими підходами за цими метриками.

Метрика багатокласової точності обчислюється за формулою:

$$Accuracy_m = \frac{1}{|L|}\sum_{l=1}^{l\in L}\frac{TP_l + TN_l}{TP_l + TN_l + FP_l + FN_l}$$

де L – кількість класів, TP_l – кількість правдиво позитивних результатів відносно класу $l \in L$, TN_l – кількість правдиво негативних результатів відносно класу $l \in L$, FP_l- кількість хибно позитивних результатів відносно класу $l \in L$, FN_l- кількість хибно негативних результатів відносно класу $l \in L$.

Модель класифікації вразливостей з використанням ковзного хешування не вимагає навчального датасету, проте дана модель вимагає підготовки бази знань хешів вразливостей. В залежності від кількості наявних класів вразливостей в базі знань – стільки різних класів модель і зможе класифікувати.

Для першого експерименту використовується 4 різні бази знань з хешами і відповідно вибірка з датасету SARD, яка містить лише наявні у базі знань вразливості:

1. База знань, яка містить по 1 екземпляру кожного класу CWE, який присутній в датасеті SARD. Дана база містить 164 записи.
2. База знань, що містить 100 найбільш частих CWE, що присутні в датасеті SARD.
3. База знань, що містить 50 найбільш частих CWE, що присутні в датасеті SARD.
4. База знань, що містить 10 найбільш частих CWE, що присутні в датасеті SARD.

Результати класифікації семплів з датасету SARD представлені в Таблиця 4.7:

Таблиця 4.7. Результати багатокласової класифікації для різної кількості семплів в базі знань

Кількість класів / **Метрика**	$L = 164$	$L = 100$	$L = 50$	$L = 10$
Багатокласова точність, %	27.9	34.4	47.5	89.1

Другий експеримент в задачі класифікації вразливостей полягає в порівнянні запропонованої моделі з використанням ковзного хешування AST та інших рішень. Оскільки класифікації вразливостей в інших рішеннях відбувалася з-поміж 40 класів, для даного експерименту було сформовано базу знань з 40 класами, що використовувались при оцінці точності інших рішень. До рішень, з якими порівнюється дана модель належать як рішення, які використовують глибоке навчання (µVulDeePecker [62]), так і великі мовні моделі (CodeBERT [63], GraphCodeBert [64], AIBugHunter [65]). Результати порівняння цих моделей з запропонованим рішенням представлено в Таблиця 4.8:

Таблиця 4.8. Порівняння запропонованої моделі класифікації вразливостей з існуючими рішеннями за критерієм багатокласової точності

Модель	**Багатокласова точність (%),** $L = 40$
µVulDeePecker	37.8
Запропонована модель класифікації вразливостей	51.1

CodeBERT	59.0
GraphCodeBert	62.0
AIBugHunter	63.0

Оскільки великі мовні моделі як [63] [64] [65] потребують великої обчислювальної потужності, в даній роботі вводиться метрика можливості локального розгортання моделі. Поява даної метрики пов’язана з тим, що розгортання моделей для аналізу коду на потужностях третьої сторони (напр. хмарна інфраструктура) суперечать практикам безпеки програмного продукту [66], оскільки відправка коду третій стороні це потенційних вектор вразливості. Таким чином, результати порівняння вищезазначених моделей за критерієм можливості локального розгортання представлено в Таблиця 4.9:

Таблиця 4.9. Порівняння запропонованої моделі класифікації вразливостей з існуючими рішеннями за критерієм можливості локального розгортання

Модель	**Локальне розгортання**
µVulDeePecker	Так
Запропонована модель класифікації вразливостей	Так
CodeBERT	Ні
GraphCodeBert	Ні
AIBugHunter	Ні

Наступний експеримент демонструє результати класифікації знайдених вразливостей у проекті Bitcoin [56] за допомогою моделі класифікації вразливостей. Для цього експерименту база знань містила 10

найбільш поширених, згідно датасету SARD, CWE. Розподіл класів знайдених вразливостей представлений на Рис. 4.1:

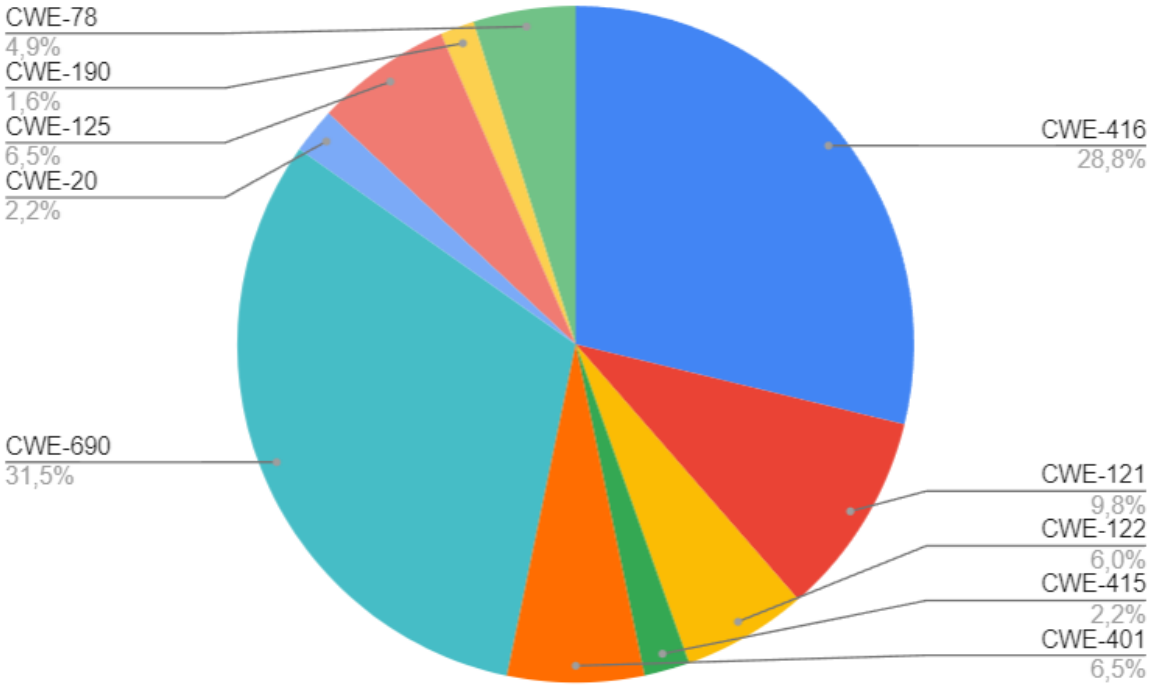

Рис. 4.1 - Розподіл класів вразливостей в проекті Bitcoin

4.3 Обговорення результатів дослідження

Результати експериментальної перевірки запропонованої системи оцінки безпеки програмного коду підтвердили її високу ефективність у задачі пошуку та класифікації вразливостей.

Зокрема, модель пошуку вразливостей на основі нейронних мереж продемонструвала 94.1% точності детекції вразливих фрагментів коду на тестовій вибірці датасету SARD. Це на 1.9% вище, ніж у моделі VulDeePecker, яка є одним з найкращих існуючих аналогів на основі глибинного навчання. Розроблена модель також показала нижчу частку хибно позитивних результатів (4.4% проти 8.5% у VulDeePecker), що свідчить про більш точне розпізнавання відсутності вразливостей у фрагментах коду.

Перевага запропонованої моделі пояснюється застосуванням удосконаленої побудови проміжного представлення програмного коду. Обмеження розміру кодового гаджету дозволило скоротити час роботи алгоритму, та залишити найбільш релевантний контекст для кожної залежності.

Втім, порівняно з найкращими представниками, система SySeVR показала дещо вищу точність на рівні 97%. Це пояснюється більш складною архітектурою SySeVR, яка побудована на основі графових нейронних мереж.

З іншого боку, складність побудови проміжного представлення SySeVR негативно впливає на швидкодію та ресурсоємність моделі. Наприклад, час аналізу однієї програми в SySeVR складає 15.7 секунд, тоді як запропоноване рішення займає лише 2.9 секунд у середньому. Обсяг оперативної пам'яті також скоротився майже у 4 рази (до 1 Гб). Це робить систему значно більш масштабованою та придатною для інтеграції у робочі процеси розробки ПЗ, де швидкодія має вирішальне значення.

Модель класифікації вразливостей з використанням ковзного хешування AST, показала перевагу над популярними аналогами. Зокрема, точність класифікації для 40 класів склала 51.1%, що на 13.3% вище за μVulDeePecker - одну з кращих RNN-моделей для даної задачі.

Перевага пояснюється спроможністю алгоритму ковзного хешування AST ефективно виявляти структурні відповідності між вразливим кодом та базою знань, навіть за наявності незначних відмінностей. Моделі глибокого навчання не здатні повною мірою узагальнювати такі особливості при наявності великої кількості класів.

Разом з тим, найкращу точність 63% продемонструвала спеціалізована модель AIBugHunter. Ця перевага зумовлена застосуванням у AIBugHunter великої мовної моделі, попередньо навченої на величезних

обсягах коду різних мов програмування. Такі моделі здатні вловлювати глибокі латентні закономірності та взаємозв'язки в коді. Однак вони є надзвичайно ресурс ємними, що унеможливлює їх локальне розгортання в рамках невеликих IT-проектів чи компаній.

Натомість запропонований підхід демонструє прийнятну точність для 40 класів 51.1% без потреби у великих обчислювальних ресурсах. Це робить його доступним для широкого кола користувачів. До того ж, його точність може бути покращена за рахунок розширення бази знань новими зразками вразливого коду.

Як показано в таблиці 4.7, при використанні повного обсягу бази знань із 164 класами вразливостей точність класифікації склала 27.9%. Це пояснюється складністю завдання розпізнавання такої великої кількості класів.

Поступове зменшення кількості класів у базі знань покращило результати моделі. Зокрема, для 100, 50 та 10 класів точність склала 34.4%, 47.5% та 89.1% відповідно.

Таку тенденцію можна пояснити тим, що при меншій кількості класів для розпізнавання, модель має більше даних на кожен окремий клас. Це дозволяє їй краще навчитися розрізняти вужчий спектр вразливостей.

Отже, хоча запропонований метод і демонструє гірші результати за умов великої кількості класів, зі зменшенням їх числа його ефективність швидко зростає. Тому для практичного застосування доцільно обмежуватися 10-50 найбільш поширеними та небезпечними типами вразливостей, що забезпечить задовільну якість класифікації.

Експерименти з реальними проектами також підтвердили практичну цінність розробленого рішення. Зокрема, згідно даних, наведених на рис. 4.1, найбільш поширеним типом вразливостей у коді Bitcoin виявилися вразливості розіменування нульового вказівника (CWE-690) - 58 випадки.

Ці вразливості пов'язані з відсутністю перевірок вказівників, перш ніж використовувати їх.

Друге місце по частоті займають вразливості використання після звільнення (CWE-416) - 53 випадки. Ці дефекти обумовлені помилками в управлінні пам'яттю та можуть призводити до серйозних наслідків.

Третіми за поширеністю є переповнення буферу (CWE-121, CWE-122) з 29 випадками (18 базовані на стеку, 11 на купі).

Такий розподіл вказує на те, що основні проблеми в коді Bitcoin пов'язані з недоліками реалізації базових механізмів безпеки: перевірок логіки, захисту пам'яті та обмеження доступу.

Отже, проведені експерименти дозволяють зробити висновок про доцільність застосування гібридного підходу, поєднуючого потужність глибинних нейронних мереж та ефективність алгоритмів пошуку подібності коду. Такий симбіоз дає можливість компенсувати недоліки кожного з методів, створюючи сприятливі умови для високоточної детекції вразливостей.

Завдяки врахуванню контекстних зв'язків у коді, запропонована система здатна ідентифікувати складні, неочевидні вразливості, які складно виявити за допомогою традиційних підходів. Це суттєво розширює можливості автоматизованої оцінки безпеки програмного забезпечення та відкриває шлях до подальшого вдосконалення методів машинного аналізу коду.

Одним з перспективних напрямків подальшого розвитку запропонованої моделі пошуку вразливостей на основі нейронної мережі є інтеграція механізму уваги (attention). Ці методи дозволяють мережі фокусуватися на найбільш інформативних фрагментах вхідних даних, ігноруючи менш значущі.

Як показано в роботах [67] [68], застосування уваги може підвищити якість вилучення ознак з послідовностей та покращити загальну точність

моделей. Замість того, щоб розглядати кодовий гаджет цілком, мережа здатна виділяти лише ті токени чи фрагменти, які несуть найбільше семантичне навантаження з точки зору класифікації вразливості. Це дозволяє позбутися "шуму" та сконцентруватися на найсуттєвіших ознаках. Крім підвищення точності, такий підхід може оптимізувати швидкодію моделі та зменшити витрати обчислювальних ресурсів при навчанні моделі.

Альтернативним варіантом є застосування графових нейронних мереж (GNNs) [69]. В цьому випадку вхідні дані подаються не у вигляді лінійної послідовності токенів, а у вигляді графа залежностей між ними. Таке подання дозволяє точніше моделювати складні зв'язки в коді та краще вловлювати ознаки, що вказують на приховані вразливості.

На відміну від RNN чи CNN, GNN можуть явно враховувати взаємодію між віддаленими фрагментами коду завдяки графовій топології. Це сприяє покращенню контекстного аналізу та здатності узагальнювати знання на нових проектах і мовах програмування [70].

Для моделі класифікації вразливостей на основі ковзного хешування AST перспективним є використання алгоритмів подібного хешування, таких як SimHash [71]. Вони дозволяють генерувати хеші, відстань між якими відображає ступінь подібності вхідних даних. За рахунок цього можна не лише знаходити точні збіги хешів як запропоновано в дисертації, а й вимірювати "відстань" між хешами AST тестованого та еталонного коду. Чим менша вона, тим ймовірніше дані фрагменти є подібними і містять аналогічні вразливості.

Така модифікація дозволить підвищити стійкість алгоритму до можливих мутацій вразливого коду, коли зловмисники навмисно вносять незначні зміни для уникнення детекції. Навіть за умови таких модифікацій, метод на основі SimHash з високою ймовірністю зможе визначити подібність з відомими вразливостями та класифікувати загрозу.

Отже, інтеграція механізмів уваги, графові нейромережі та алгоритми подібності хешів надають значний потенціал для вдосконалення запропонованої в дисертації системи детекції вразливостей. Комбінація цих методів дозволить підвищити якість аналізу та стійкість до нових типів загроз в умовах бурхливого розвитку технологій та складності програмних систем.

4.4 Практичне застосування розробленого рішення

4.4.1 Інтеграція системи аналізу програмного коду в процеси безперервної інтеграції

Ефективна інтеграція розробленої системи аналізу програмного коду в процеси безперервної інтеграції вимагає ретельного планування та налаштування. Нижче наведено детальний покроковий алгоритм такої інтеграції:

1. Вибір системи безперервної інтеграції. Найбільш популярними на сьогодні є GitHub Actions, Jenkins, GitLab CI, CircleCI, Travis CI, TeamCity тощо. Вибір конкретної системи залежить від платформи розробки, мови програмування, інфраструктури проекту.
2. Налаштування оточення CI. Необхідно здійснити налаштування усіх залежностей, баз даних та інших сервісів, потрібних для коректного запуску системи аналізу коду. Для цього можуть використовуватися контейнери або віртуальні машини.
3. Інтеграція системи аналізу коду в конвеєр CI. Додається новий крок пайплайну CI, що запускатиме розроблену систему аналізу програмного коду.

4. Перевірка працездатності. Необхідно виконати тестові запуски пайплайна CI та переконатися, що система аналізу коректно виконується без помилок для різних версій коду.
5. Налаштування звітності. Результати роботи системи необхідно представити у звітах CI у зручному для перегляду форматі. Зазвичай це спеціальні розділи звіту, таблиці чи діаграми.
6. Налаштування повідомлень про помилки. Якщо система аналізу виявляє вразливості у коді, CI має генерувати відповідні помилки чи попередження або навіть призупиняти збірку. Реалізація залежить від системи CI.
7. Інтеграція з системою відстеження помилок. Знайдені вразливості можуть автоматично відкривати задачі у системах класу Jira чи Redmine з їх подальшим відстеженням та моніторингом статусу виправлення.
8. Налаштування автоматичного тестування. Бажано автоматизувати запуск тестів безпеки після внесення змін, пов'язаних із виправленням вразливостей, для перевірки результативності коду.
9. Додавання аналізу змін. Доцільно інтегрувати можливість аналізу коду лише для фрагментів, які зазнали змін, а не для усього кодової бази повністю. Це оптимізує час роботи.
10. Додавання кодогляду. CI може автоматично ініціювати перевірку змін розробниками з правами кодогляду у разі внесення змін у вразливі фрагменти.

В деяких випадках доцільно реалізувати два окремих етапи аналізу: попередній швидкий скан та повний глибинний аналіз. Швидкий скан може запускатися для кожної збірки, тоді як повний аналіз – лише один раз на добу чи за запитом.

Переваги інтеграції у CI:

1. Виявлення вразливостей на ранніх стадіях розробки ще до потрапляння коду у репозиторій
2. Автоматизація та неперервність аналізу безпеки
3. Швидкий зворотний зв'язок для розробників
4. Можливість донавчання системи аналізу та покращення точності
5. Убезпечення від людського фактору: забудькуватості чи неуважності запустити перевірку вручну

Разом з тим потрібно врахувати наступні аспекти:

1. Підвищене навантаження на інфраструктуру CI/CD через додаткові перевірки
2. Можливе уповільнення роботи пайплайнів
3. Необхідність налаштування гнучких тригерів, які б не спричиняли надмірних запитів та перебудов проектів
4. Додаткові витрати на збільшення потужностей у випадку хмарних CI/CD рішень

Отже, існує низка аспектів, які потрібно врахувати при інтеграції системи аналізу безпеки програмного коду в платформи безперервної інтеграції. Водночас це дозволяє автоматизувати виявлення вразливостей на ранніх етапах розробки та покращити загальний стан безпеки проектів.

Схема вбудови розробленої системи в процеси безперервної інтеграції зображена на Рис. 4.2:

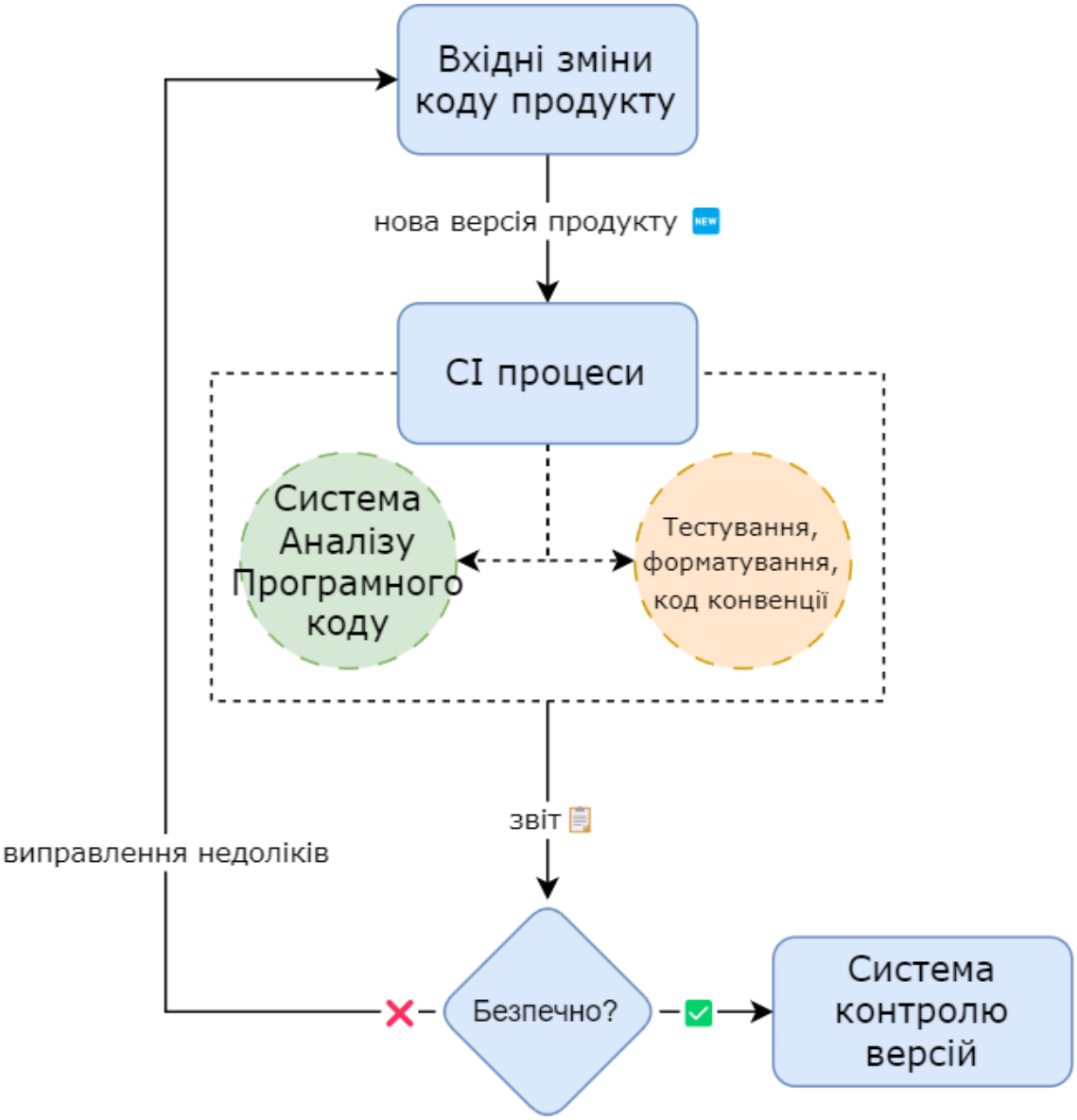

Рис. 4.2 - Схема вбудови системи аналізу програмного коду (САПК) в процеси безперервної інтеграції

4.4.2 Інтеграція системи аналізу програмного коду в процес передрелізної верифікації коду

Ефективна інтеграція розробленої системи аналізу програмного коду в процес передрелізної верифікації коду в рамках методології Microsoft SDL вимагає детального планування та врахування низки аспектів.

По-перше, необхідно чітко визначити ролі та зони відповідальності фахівців з інформаційної безпеки та розробників програмного забезпечення в контексті передрелізної верифікації. Рекомендовано створити окрему групу з інформаційної безпеки, до складу якої можуть входити як технічні фахівці (аналітики безпеки, пентестери), так і фахівці з ризик-менеджменту. Ця група має бути відповідальною за проведення оцінки безпеки, тестування на проникнення, а також аналіз результатів, генерованих розробленою системою аналізу коду.

З іншого боку, команди розробки ПЗ мають фокусуватися виключно на розробці та виправленні вразливостей відповідно до знайдених проблем. Підтримання чіткого розмежування обов'язків допомагає уникнути конфліктів інтересів, а також сприяє більш плідній комунікації та співпраці між групами.

Крім того, важливо забезпечити високий рівень автоматизації процесів шляхом інтеграції розробленої системи з наявними платформами та інструментами перевірки безпеки. Це включає:

1. Інтеграцію з системою контролю версій (Git, SVN тощо) для отримання останніх версій коду програмних продуктів.
2. Інтеграцію з системою управління вразливостями та задачами (Jira, Redmine), де буде автоматично створюватися та відстежуватися статус знайдених вразливостей.
3. Забезпечення можливості автоматичного формування звітів за результатами роботи системи у зручній для аналізу формі (HTML, PDF).
4. Інтеграцію з хмарними платформами безпеки (AWS Security Hub, Azure Sentinel) для кореляції даних, поглибленого аналізу та формування загальної картини ризиків програмного забезпечення.

Також важливо забезпечити процес навчання та підвищення кваліфікації фахівців з інформаційної безпеки, задіяних у верифікації коду перед релізом. Це необхідно для вдосконалення їх навичок аналізу програмного коду, ознайомлення з архітектурою та бізнес-логікою програмних систем компанії, а також формування розуміння можливих атак та загроз. Рекомендовано проводити як внутрішні тренінги та навчання на робочому місці, так і зовнішні спеціалізовані курси з питань безпеки ПЗ.

В контексті аналізу результатів роботи розробленої системи під час передрелізної верифікації також важливо враховувати можливі хибно позитивні спрацювання, коли фрагмент коду помилково ідентифікується як вразливий.

Для мінімізації таких випадків фахівці мають проводити ретельний аналіз кожної потенційної вразливості з урахуванням особливостей архітектури та бізнес-логіки програмного продукту. Це включає перевірку:

- чи може користувач реально вплинути на вразливий фрагмент коду, наприклад, через інтерфейс програми;
- чи не містить код компенсуючих механізмів безпеки, які запобігають реальним атакам, не дивлячись на деякі недоліки;
- чи не є функціональність, що містить вразливість, вимкненою за замовчуванням або доступною лише для адміністраторів.

У разі наявності сумніву щодо можливості експлуатації певного фрагменту коду, його не варто класифікувати як вразливість, поки не буде одержано достатніх доказів реальної небезпеки.

Крім аналізу окремих вразливостей, результати роботи системи мають вивчатися в комплексі для оцінки загального рівня захищеності програмного продукту. Метою є не лише реагування на конкретні проблеми, а комплексне поліпшення архітектури та дизайну системи з точки зору безпеки в довгостроковій перспективі. Для цього доцільно

проводити регулярні наради між фахівцями з безпеки, архітекторами та розробниками з метою спільного аналізу тенденцій, визначення найкритичніших компонентів системи та планування заходів посилення їх захисту.

Аналіз результатів роботи системи має також відображатися у регулярній звітності перед вищим керівництвом щодо загального стану безпеки програмних продуктів компанії. Це дозволить приймати зважені рішення щодо необхідності додаткових інвестицій у підвищення якості та захист коду.

Нарешті, важливою є можливість постійного навчання та вдосконалення самої системи аналізу коду. Оскільки корпус даних про нові вразливості постійно поповнюється, а програмні продукти компанії зазнають змін, потрібно забезпечити безперервне доналаштування та оновлення моделей системи на основі нових даних. Це сприятиме підвищенню точності детекції з часом за рахунок самонавчання та адаптації моделей нейронних мереж до внутрішніх особливостей ПЗ компанії.

Отже, успішна інтеграція системи аналізу програмного коду в процеси передрелізної верифікації має враховувати низку організаційних, методологічних та технологічних аспектів. Це дозволить максимізувати ефект від застосування такого рішення шляхом вбудови його в існуючі процедури та інструменти перевірки безпеки програмних продуктів. Як результат, забезпечується більш точна оцінка захищеності коду, адаптована до особливостей бізнес-логіки компанії, а також відбувається більш швидке реагування на виявлені вразливості та їх ефективне виправлення.

Процес інтеграції розробленої системи в фазу передрелізної верифікації безпеки графічно зображено на Рис. 4.3:

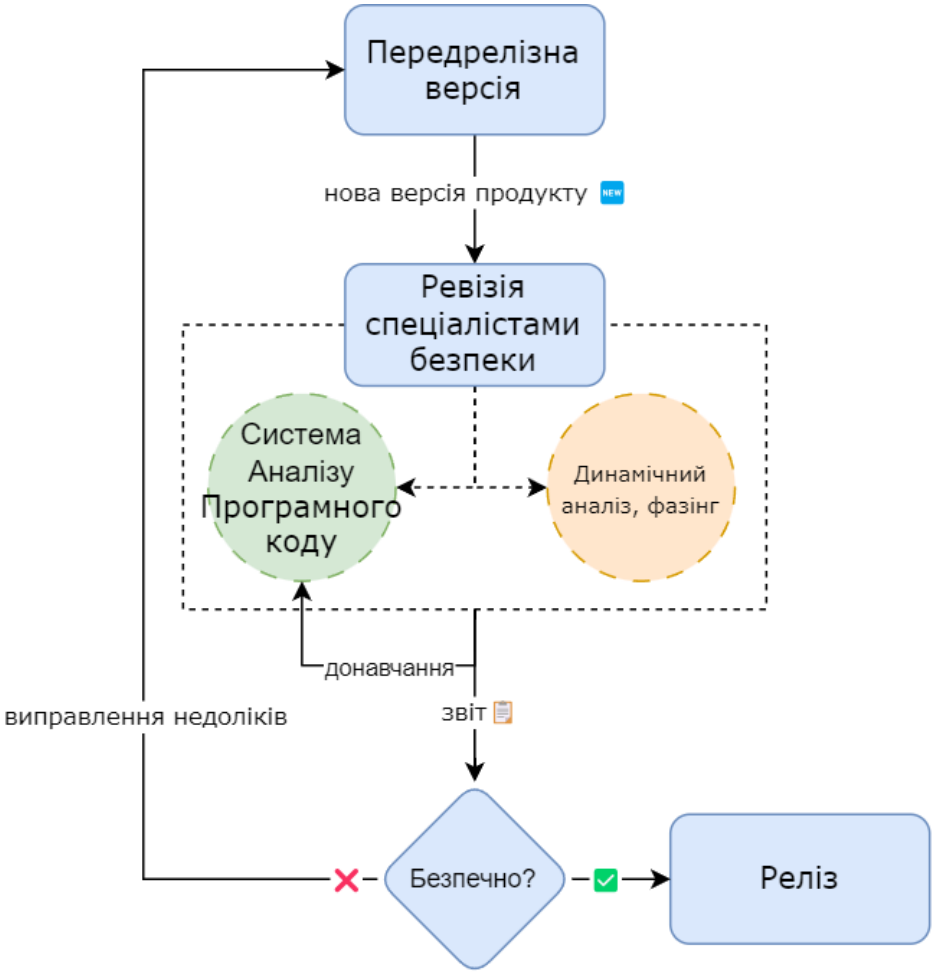

Рис. 4.3 - Схема інтеграції системи аналізу програмного коду (САПК) в процеси передрелізної верифікації програмного продукту

4.5 Апробація результатів дослідження

Результати досліджень, представлених у даній дисертаційній роботі, пройшли апробацію та були оприлюднені у низці наукових публікацій.

Зокрема, основні положення щодо розробки та застосування системи аналізу програмного коду з використанням гібридного методу пошуку та класифікації вразливостей було представлено в 4 публікаціях у фахових наукових виданнях.

Перша стаття [57] присвячена поглибленому аналізу існуючих методів детекції вразливостей у програмному коді на основі глибокого навчання. Розглядаються переваги та обмеження підходів з використанням

абстрактних синтаксичних дерев, графів залежностей та інших моделей представлення коду.

Друга публікація [58] представляє розроблений автором інноваційний алгоритм виявлення дублікатів коду на основі AST та демонструє його ефективність для задачі пошуку прихованих вразливостей.

У статті [59] подано аналіз методів машинного навчання, зокрема на основі дерев рішень, для детекції подібних фрагментів коду та оцінено їх можливості і недоліки у контексті безпеки ПЗ.

Нарешті, робота [60] присвячена вивченню застосування глибинних нейронних мереж у системах класу VulDetect для автоматизованого пошуку вразливостей за допомогою навчання на прикладах вразливого коду.

Разом ці публікації сформували наукове підґрунтя та підтвердили результативність розробки комплексної системи автоматизованої верифікації безпеки програмного забезпечення на основі поєднання методів штучного інтелекту, зокрема глибинних нейронних мереж, та алгоритмів структурного аналізу коду з використанням абстрактних синтаксичних дерев.

У дослідженні [57] проведено глибинний аналіз сучасних методів пошуку вразливостей в програмному коді, акцентуючи увагу на потужності методів глибокого навчання. Розглядаються різні підходи, включаючи використання абстрактних синтаксичних дерев (AST) та графів залежностей програми. Одним з ключових відкриттів є те, що моделі, які використовують AST, показують високу точність у виявленні вразливостей, проте вони можуть бути обмежені при аналізі великих обсягів даних через складність структури дерева.

Щодо методів, що базуються на графах залежностей, вони відкривають нові можливості для виявлення складних вразливостей, зокрема проблем з неправильним іменуванням змінних та їх неправильним

використанням. Проте, великий розмір графів та об'єм взаємозв'язків можуть негативно впливати на точність моделей.

Огляд вказує на критичну потребу в подальшому вдосконаленні існуючих методів, зокрема через комбінування AST та аналізу графів для створення більш точних та масштабованих рішень. Враховуючи це, дослідники та розробники мають значні перспективи для розробки нових, більш ефективних систем виявлення вразливостей, які можуть значно підвищити рівень безпеки програмного забезпечення.

У статті [58] представлено інноваційний підхід до виявлення дублікатів у програмному коді на основі абстрактного синтаксичного дерева (AST). Автори зосереджують увагу на важливості виявлення клонованих фрагментів коду, оскільки вони можуть маскувати потенційні вразливості та призводити до помилок у програмному забезпеченні. Використовуючи AST, дослідження демонструє, як можна ефективно ідентифікувати такі дублікати, що відкриває шлях для більш глибокого аналізу безпеки коду. Хоча методика особливо ефективна для виявлення SQL-ін'єкцій, вона також успішно справляється з іншими вразливостями, такими як витоки пам'яті, розіменування нульових вказівників та переповнення буферу. Проте, варто зазначити, що, не дивлячись на високу точність виявлення дублікатів, метод може виявляти помилкові позитивні сигнали, особливо при роботі з SQL-ін'єкціями. Незважаючи на це, розроблений підхід має великий потенціал для покращення процесів розробки програмного забезпечення, спрощуючи виявлення потенційних вразливостей та підвищуючи якість кінцевого продукту.

У статті [59] представлено аналіз алгоритмів машинного навчання для виявлення дублікатів в програмному коді. Автори зосереджуються на методі, що використовує дерево рішень, яке показує результати у класифікації та виявленні дублікатів коду. Проте, відсутність глибокого аналізу в контексті виявлення вразливостей в програмному забезпеченні

робить цей метод менш ефективним для виявлення потенційних ризиків безпеки. Особливо це стосується складних вразливостей, які вимагають детального розуміння контексту виконання коду та не можуть бути виявлені простим порівнянням фрагментів коду. Крім того, хоча метод дерева рішень демонструє хороші показники точності, він може бути неефективним у сценаріях з великими обсягами даних або коду, що постійно змінюється, через обмежену здатність адаптації до нових шаблонів вразливостей. Таким чином, необхідні подальші дослідження для оптимізації цього підходу з метою підвищення його ефективності у виявленні вразливостей.

У статті [60] автори презентують систему VulDetect для виявлення вразливостей у вихідному коді, яка використовує методи глибокого навчання для створення правил, які визначають, чи є фрагмент коду вразливим. Цей підхід є удосконаленням методу, запропонованого в VulDeePecker, і використовує AST (абстрактне синтаксичне дерево) для представлення вихідного коду. Автори порівнюють результати виявлення вразливостей обох систем на проекті Bitcoin, демонструючи ефективність своєї методики. Система VulDetect підкреслює значущість використання глибокого навчання в області безпеки програмного забезпечення, особливо з огляду на здатність цих методів адаптуватися до нових видів загроз шляхом навчання з даних. Однак, хоча результати є обнадійливими, автори також натякають на необхідність подальших досліджень для оптимізації системи в контексті реального світу, де вразливості можуть бути більш складними та різноманітними. Ця робота закликає до подальшого вдосконалення автоматизованих систем виявлення вразливостей, підкреслюючи необхідність інтеграції сучасних методів машинного навчання та глибокого аналізу для ефективного і своєчасного виявлення потенційних загроз.

4.6 Висновки

У четвертому розділі дисертаційної роботи було проведено експериментальне дослідження розробленої системи аналізу програмного коду, що дозволило оцінити практичну цінність запропонованого підходу.

Основним завдання експериментальної фази було встановити ефективність окремих компонентів системи, які вирішують дві ключові задачі: 1) безпосередньо пошук вразливостей у коді; 2) класифікація виявлених вразливостей згідно таксономії CWE.

Для оцінки моделі пошуку вразливостей використовувався спеціалізований набір даних SARD з бази вразливостей NVD, який містить як вразливі, так і безпечні фрагменти коду. Розроблена модель на основі двонаправлених рекурентних нейронних мереж продемонструвала 94.1% точність детекції вразливостей, що є близьким до точності роботи існуючих моделей. Запропонована система перевершує їх за швидкодією та можливістю масштабування, що дозволяє ефективно інтегрувати її у процеси розробки ПЗ в рамках безперервної інтеграції чи DevSecOps.

Компонент класифікації вразливостей на основі алгоритму ковзного хешування абстрактних синтаксичних дерев також показав задовільну якість. Зокрема, при класифікації серед 40 типів вразливостей точність склала 51.1%. Це на 13.3% вище за інші мережі глибокого навчання, проте нижче ніж у спеціалізованих великих мовних моделей, як AIBugHunter (63%). Втім, на відміну від великих мовних моделей, запропонований метод є масштабованим та може впроваджуватися локально в рамках будь-яких IT проектів за прийнятну ціну.

Таким чином, проведені експерименти підтвердили гіпотезу щодо ефективності поєднання методів машинного навчання та алгоритмів пошуку подібності коду в рамках єдиної інтегрованої системи аналізу програмного коду. Кожен з компонентів дозволяє подолати недоліки

іншого, формуючи найсприятливіші умови для ефективного виявлення вразливостей різної природи та складності.

Також була продемонстрована можливість практичної інтеграції системи в процеси безперервної інтеграції та передрелізної верифікації коду. Зокрема, застосування розробленого рішення у процесі CI дозволяє автоматизовано аналізувати зміни у коді в міру його розробки, оперативно виявляти вразливості та сприяти їх своєчасному виправленню ще до потрапляння у репозиторії чи системи контролю версій.

З іншого боку, інтеграція з фахівцями безпеки на передрелізних етапах, згідно з моделлю Microsoft SDL, дозволяє проводити ретельнішу верифікацію та класифікацію загроз на основі висновків експертів та доопрацьовувати систему для підвищення точності аналізу конкретних програмних проектів.

Сформульовані сценарії вбудови системи в процеси CI/CD та робочі процедури фахівців кібербезпеки демонструють реальні можливості апробації та вдосконалення запропонованого у дисертації рішення викликах комплексних ІТ проектів. Завдяки адаптивності та гнучкості архітектури, система має великий потенціал для подальшого розвитку та індивідуального підлаштування під потреби конкретних організацій.

Отже, на основі експериментальних досліджень підтверджена гіпотеза щодо підвищення ефективності пошуку та класифікації вразливостей за рахунок поєднання глибокого навчання з алгоритмами, заснованими на структурному аналізі коду. Проведена оцінка у порівнянні з альтернативними підходами та виявлено переваги та недоліки запропонованої системи. Також продемонстровано практичне застосування розробленого рішення в рамках стандартних процесів розробки та верифікації ПЗ, що підтверджує його цінність для реальних ІТ-компаній та проектів.

ВИСНОВКИ

У даній дисертаційній роботі представлено комплексне дослідження, присвячене розробці системи аналізу програмного коду з використанням гібридного методу пошуку та класифікації вразливостей. Робота містить в собі як теоретичне підґрунтя та формалізоване моделювання задачі, так і практичну реалізацію та апробацію розроблених моделей. Основними результатами даної роботи є:

1. вперше запропоновано гібридний метод аналізу програмного коду, що поєднує методи глибокого навчання та методи виявлення подібності коду для пошуку та класифікації вразливості в коді, який дозволяє ефективно виконувати пошук вразливостей в коді, а також класифікувати з високою точністю знайдені вразливості;

2. отримав подальший розвиток метод побудови проміжного представлення програмного коду у вигляді кодового гаджету, який відрізняється від існуючих методів наявністю обмеження по розміру локального контексту відносно ключової точки, що дозволило зменшити результуючий розмір кодових гаджетів та підвищити точність класифікації при подальшому аналізі нейронною мережею;

3. вперше запропоновано метод класифікації вразливостей в програмному коді з використанням ковзного хешування абстрактного синтаксичного дерева, який відрізняється від існуючих методів тим що використовує метод виявлення подібності коду для ефективної класифікації вразливостей без необхідності використання навчальної вибірки великого об’єму;

У першому розділі здійснено ґрунтовний аналіз сучасного стану кібербезпеки, проаналізовано динаміку зростання кількості вразливостей у програмному забезпеченні, що підкреслює важливість та необхідність розробки нових підходів до автоматизації їх детекції. Визначено основні

принципи, стандарти та найкращі практики безпечної розробки ПЗ. Окремий акцент зроблено на аналізі можливостей машинного навчання для вирішення задачі ідентифікації неочевидних вразливостей.

У другому розділі здійснено математичне моделювання системи пошуку вразливостей, визначено її вхідні дані, етапи перетворення, структурні одиниці та бажані результати у вигляді формальних відношень та залежностей. Запропоновано гібридний метод аналізу програмного коду, що поєднує метод з використанням глибокого навчання для пошуку вразливостей та метод виявлення подібності коду для класифікації вразливих фрагментів коду.

У третьому розділі запропоновано подальший розвиток методу побудови проміжного представлення програмного коду, що дозволило покращити роботу існуючого методу пошуку вразливостей на основі нейронної мережі. Завдяки обмеженню розміру кодового гаджету, вдалося пришвидшити процес побудови та розмір проміжного представлення коду, без втрати точності класифікації моделі нейронної мережі.

Також було розроблено метод класифікації вразливостей на основі алгоритму ковзного хешування вузлів абстрактного синтаксичного дерева з використанням бази знань відомих вразливих конструкцій. Це дозволяє ефективно та швидко визначати тип знайденої вразливості шляхом порівняння її структури з еталонами.

У четвертому розділі проведено експериментальне дослідження розробленої системи, що підтвердило її високу ефективність у задачах пошуку та класифікації вразливостей. Зокрема, модель пошуку вразливостей на основі нейронних мереж продемонструвала 94.1% точності детекції. Модель класифікації показала 51.1% точності для 40 класів вразливостей. Також була продемонстрована можливість інтеграції системи в процеси CI/CD та передрелізної верифікації коду.

Розроблена в дисертації система аналізу програмного коду має значне практичне застосування. Зокрема, запропонований метод дозволяє знаходити вразливості в коді мовами C/C++, а також класифікувати тип знайденої вразливості відповідно до загальноприйнятої таксономії CWE.

Це надає можливість розробникам швидше реагувати на критичні вразливості та раціонально розподіляти ресурси на їх виправлення. Як свідчить практика, застосування системи дозволило скоротити загальний час усунення дефектів безпеки в коді майже вдвічі.

Крім того, створене програмне рішення було інтегроване в технологічний процес розробки IT-компанії у вигляді сервісу автоматизованої перевірки безпеки коду. Це надало можливість регулярно контролювати рівень захищеності програмних продуктів на різних стадіях розробки та своєчасно реагувати на потенційні дефекти.

Ефективність системи також була підтверджена тестуванням на реальних проектах з відкритим вихідним кодом. Зокрема, за допомогою розробленого рішення вдалося виявити та підтвердити вразливість в популярному додатку Microsoft Terminal.

Незважаючи на продемонстровану ефективність, запропоноване рішення має певні обмеження. Зі збільшенням кількості класів у базі знань хешів система показала значне зниження показника точності класифікації. Тому, потенційним напрямом для поліпшення є застосування алгоритму подібності хешів, як SimHash, що забезпечить більшу стійкість до можливих мутацій вразливого коду в задачі класифікації вразливості.

Ще одним напрямком удосконалення є розширення підтримки мов програмування. Наразі система орієнтована переважно на аналіз коду C/C++, тоді як підтримка інших популярних мов, таких як Java, JavaScript чи Python, залишається обмеженою.

Оскільки технології штучного інтелекту неупинно розвиваються, з’являються нові архітектури нейронних мереж, що можуть бути

використані в задачі пошуку вразливостей. Інтеграція механізмів уваги та графових нейронних мереж відкриває можливості для підвищення точності та оптимізації швидкодії моделей машинного навчання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The Latest Cyber Crime Statistics (updated December 2023) | AAG IT Support. AAG IT Services. URL: https://aag-it.com/the-latest-cyber-crime-statistics/ (дата звернення: 11.12.2023).
2. 25+ Cyber Security Vulnerability Statistics and Facts of 2023. Comparitech. URL: https://www.comparitech.com/blog/information-security/cybersecurity-vulnerability-statistics/ (дата звернення: 11.12.2023).
3. Beyond Memory Corruption Vulnerabilities – A Security Extinction and Future of Exploitation. Trellix | Revolutionary Threat Detection and Response. URL: https://www.trellix.com/about/newsroom/stories/research/beyond-memory-corruption-vulnerabilities/ (дата звернення: 11.12.2023).
4. Number of common vulnerabilities and exposures 2023 | Statista. *Statista*. URL: https://www.statista.com/statistics/500755/worldwide-common-vulnerabilities-and-exposures/ (дата звернення: 11.12.2023).
5. NVD - cve-2021-44228. NVD - Home. URL: https://nvd.nist.gov/vuln/detail/cve-2021-44228 (дата звернення: 11.12.2023).
6. NVD - CVE-1999-0517. NVD - Home. URL: https://nvd.nist.gov/vuln/detail/CVE-1999-0517 (дата звернення: 11.12.2023).
7. Microsoft Security Development Lifecycle. URL: https://www.microsoft.com/en-us/securityengineering/sdl/practices (дата звернення: 11.12.2023).
8. OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation. OWASP Foundation, the Open Source

Foundation for Application Security | OWASP Foundation. URL: https://owasp.org/ (дата звернення: 11.12.2023).

9. ISO/IEC 27034-1:2011. *ISO*. URL: https://www.iso.org/standard/44378.html (дата звернення: 11.12.2023).
10. Agrafiotis, I., Nurse, J. R., Goldsmith, M., Creese, S., & Upton, D. (2018). A taxonomy of cyber-harms: Defining the impacts of cyber-attacks and understanding how they propagate. Journal of Cybersecurity, 4(1), tyy006.
11. Al-Fedaghi, S., & Alkandari, A. (2011). On security development lifecycle: Conceptual description of vulnerabilities, risks, and threats. International Journal of Digital Content Technology and its Applications, 5(05), 296-306.
12. Application Security Testing Company | Software Security Testing Solutions | Checkmarx. Checkmarx.com. URL: https://www.checkmarx.com/ (дата звернення: 11.12.2023).
13. Google Code. Google Code. URL: https://code.google.com/ (дата звернення: 11.12.2023).
14. Coverity Scan - Static Analysis. *Coverity Scan - Static Analysis*. URL: https://scan.coverity.com/ (дата звернення: 11.12.2023).
15. Static Code Analyzer | Static Code Analysis Security | CyberRes. Micro Focus is now OpenText. URL: https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer (дата звернення: 11.12.2023).
16. Flawfinder Home Page. *David A. Wheeler's Personal Home Page*. URL: http://www.dwheeler.com/flawfinder (дата звернення: 11.12.2023).
17. Cui, L., Cui, J., Hao, Z., Li, L., Ding, Z., & Liu, Y. (2022). An empirical study of vulnerability discovery methods over the past ten years. Computers & Security, 120, 102817.

18. Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., & Chen, Z. (2021). Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing, 19(4), 2244-2258.
19. Kim, S., Woo, S., Lee, H., & Oh, H. (2017, May). Vuddy: A scalable approach for vulnerable code clone discovery. In 2017 IEEE Symposium on Security and Privacy (SP) (pp. 595-614). IEEE.
20. Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., & Hu, J. (2016, December). Vulpecker: an automated vulnerability detection system based on code similarity analysis. In Proceedings of the 32nd annual conference on computer security applications (pp. 201-213).
21. Guo, J., Cheng, J., & Cleland-Huang, J. (2017, May). Semantically enhanced software traceability using deep learning techniques. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE) (pp. 3-14). IEEE.
22. White, M., Tufano, M., Vendome, C., & Poshyvanyk, D. (2016, August). Deep learning code fragments for code clone detection. In Proceedings of the 31st IEEE/ACM international conference on automated software engineering (pp. 87-98).
23. Shin, E. C. R., Song, D., & Moazzezi, R. (2015). Recognizing functions in binaries with neural networks. In 24th USENIX security symposium (USENIX Security 15) (pp. 611-626).
24. Lin, G., Zhang, J., Luo, W., Pan, L., & Xiang, Y. (2017, October). POSTER: Vulnerability discovery with function representation learning from unlabeled projects. In Proceedings of the 2017 ACM SIGSAC conference on computer and communications security (pp. 2539-2541).
25. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., ... & Zhong, Y. (2018). Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.

26. Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., & Chen, Z. (2021). Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing, 19(4), 2244-2258.
27. Li, J., He, P., Zhu, J., & Lyu, M. R. (2017, July). Software defect prediction via convolutional neural network. In 2017 IEEE international conference on software quality, reliability and security (QRS) (pp. 318-328). IEEE.
28. Geng, Q., Zhou, Z., & Cao, X. (2018). Survey of recent progress in semantic image segmentation with CNNs. Science China Information Sciences, 61, 1-18.
29. Wang, J., Sun, J., Lin, H., Dong, H., & Zhang, S. (2017). Convolutional neural networks for expert recommendation in community question answering. Science China Information Sciences, 60, 1-9.
30. Wang, S., Liu, T., & Tan, L. (2016, May). Automatically learning semantic features for defect prediction. In Proceedings of the 38th International Conference on Software Engineering (pp. 297-308).
31. Yang, X., Lo, D., Xia, X., Zhang, Y., & Sun, J. (2015, August). Deep learning for just-in-time defect prediction. In 2015 IEEE International Conference on Software Quality, Reliability and Security (pp. 17-26). IEEE.
32. Ghaffarian, S. M., & Shahriari, H. R. (2017). Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Computing Surveys (CSUR), 50(4), 1-36.
33. CWE - Common Weakness Enumeration. CWE - Common Weakness Enumeration. URL: https://cwe.mitre.org/ (дата звернення: 11.12.2023).
34. Swarna, K. C., Saji Mathews, N., Vagavolu, D., & Chimalakonda, S. (2021). On the Impact of Multiple Source Code Representations on Software Engineering Tasks--An Empirical Study. arXiv e-prints, arXiv-2106.

35. Yamaguchi, F., Golde, N., Arp, D., & Rieck, K. (2014, May). Modeling and discovering vulnerabilities with code property graphs. In 2014 IEEE Symposium on Security and Privacy (pp. 590-604). IEEE.
36. Ferrante, J., Ottenstein, K. J., & Warren, J. D. (1987). The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS), 9(3), 319-349.
37. Clang C Language Family Frontend for LLVM. *Clang C Language Family Frontend for LLVM*. URL: https://clang.llvm.org/ (дата звернення: 11.12.2023).
38. GCC-XML. URL: https://www.gccxml.org/HTML/Index.html (дата звернення: 11.12.2023).
39. Tree-sitter | Introduction. Tree-sitter. URL: https://tree-sitter.github.io/tree-sitter/ (дата звернення: 11.12.2023).
40. pycparser. PyPI. URL: https://pypi.org/project/pycparser/ (дата звернення: 11.12.2023).
41. ROSE: Main Page. Redirect Page. URL: http://rosecompiler.org/ROSE_HTML_Reference/index.html (дата звернення: 11.12.2023).
42. ANTLR. *ANTLR*. URL: https://www.antlr.org/ (дата звернення: 11.12.2023).
43. Sparck Jones, K. (1972). A statistical interpretation of term specificity and its application in retrieval. Journal of documentation, 28(1), 11-21.
44. Mikolov, T. (2012). Statistical language models based on neural networks. Presentation at Google, Mountain View, 2nd April, 80(26).
45. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.
46. Singh, A. K., & Shashi, M. (2019). Vectorization of text documents for identifying unifiable news articles. International Journal of Advanced Computer Science and Applications, 10(7).

47. CWE - CWE-121: Stack-based Buffer Overflow (4.13). *CWE - Common Weakness Enumeration*. URL: https://cwe.mitre.org/data/definitions/121.html (дата звернення: 11.12.2023).
48. CWE - CWE-122: Heap-based Buffer Overflow (4.13). CWE - Common Weakness Enumeration. URL: https://cwe.mitre.org/data/definitions/122.html (дата звернення: 11.12.2023).
49. CWE - CWE-401: Missing Release of Memory after Effective Lifetime (4.13). *CWE - Common Weakness Enumeration*. URL: https://cwe.mitre.org/data/definitions/401.html (дата звернення: 11.12.2023).
50. CWE - CWE-415: Double Free (4.13). CWE - Common Weakness Enumeration. URL: https://cwe.mitre.org/data/definitions/415.html (дата звернення: 11.12.2023).
51. Wolf, L., Hanani, Y., Bar, K., & Dershowitz, N. (2014). Joint word2vec Networks for Bilingual Semantic Representations. Int. J. Comput. Linguistics Appl., 5(1), 27-42.
52. Hinton, G. E. (1986). Distributed Representations//Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume I. Foundations/Ed. by DE Rumelhart & JL McClelland–Cambridge.
53. TensorFlow. TensorFlow. URL: https://www.tensorflow.org/ (дата звернення: 11.12.2023).
54. NIST Software Assurance Reference Dataset. *NIST Software Assurance Reference Dataset*. URL: https://samate.nist.gov/SARD/ (дата звернення: 11.12.2023).
55. NIST Software Assurance Reference Dataset. NIST Software Assurance Reference Dataset. URL: https://samate.nist.gov/SARD/ (дата звернення: 11.12.2023).

56. GitHub - bitcoin/bitcoin: Bitcoin Core integration/staging tree. *GitHub*. URL: https://github.com/bitcoin/bitcoin (дата звернення: 11.12.2023).
57. Kubiuk, Y., & Kyselov, G. (2021). Comparative analysis of approaches to source code vulnerability detection based on deep learning methods. Technology audit and production reserves, 3(2), 59.
58. Kubiuk, Y., & Kyselov, G. (2023). Development of an algorithm for code clone detection in source code based on abstract syntax tree. Technology audit and production reserves, 4(2 (72)), 33-36.
59. Kaliuzhna, T., & Kubiuk, Y. (2022). Analysis of machine learning methods in the task of searching duplicates in the software code. Technology audit and production reserves, 4(2 (66)), 6-13.
60. Chernousov, A., Savchenko, A., Osadchyi, S., Kubiuk, Y., Kostenko, Y., & Likhomanov, D. (2019). Deep learning based automatic software defects detection framework.
61. Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochASTic optimization." arXiv preprint arXiv:1412.6980 (2014).
62. Zou, D., Wang, S., Xu, S., Li, Z., & Jin, H. (2019). μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. IEEE Transactions on Dependable and Secure Computing, 18(5), 2224-2236.
63. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.
64. Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., ... & Zhou, M. (2020). Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.
65. Fu, M., Tantithamthavorn, C., Le, T., Kume, Y., Nguyen, V., Phung, D., & Grundy, J. (2024). AIBugHunter: A Practical tool for predicting,

classifying and repairing software vulnerabilities. Empirical Software Engineering, 29(1), 4.

66. Gupta, S., & Gupta, B. B. (2017). Detection, avoidance, and attack pattern mechanisms in modern web application vulnerabilities: present and future challenges. International Journal of Cloud Applications and Computing (IJCAC), 7(3), 1-43.
67. Luong, M. T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. arXiv preprint arXiv:1508.04025.
68. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.
69. Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., ... & Sun, M. (2020). Graph neural networks: A review of methods and applications. AI open, 1, 57-81.
70. Allamanis, M., Brockschmidt, M., & Khademi, M. (2017). Learning to represent programs with graphs. arXiv preprint arXiv:1711.00740.
71. Sadowski, C., & Levin, G. (2007). Simhash: Hash-based similarity detection.
72. Steenhoek, B., Rahman, M. M., Jiles, R., & Le, W. (2023, May). An empirical study of deep learning models for vulnerability detection. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (pp. 2237-2248). IEEE.
73. Medsker, L. R., & Jain, L. C. (2001). Recurrent neural networks. Design and Applications, 5(64-67), 2.
74. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

75. Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. (2018). Mapping language to code in programmatic context. arXiv preprint arXiv:1808.09588.
76. Zhang, S., Zheng, D., Hu, X., & Yang, M. (2015, October). Bidirectional long short-term memory networks for relation classification. In Proceedings of the 29th Pacific Asia conference on language, information and computation (pp. 73-78).
77. MD5 [Електронний ресурс] – Режим доступу до ресурсу: https://www.ietf.org/rfc/rfc1321.txt. (дата звернення: 24.01.2024).
78. Wang, J., Huang, Z., Liu, H., Yang, N., & Xiao, Y. (2023). Defecthunter: A novel llm-driven boosted-conformer-based code vulnerability detection mechanism. arXiv preprint arXiv:2309.15324.
79. Zhou, X., Han, D., & Lo, D. (2021, September). Assessing generalizability of codebert. In 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 425-436). IEEE.
80. Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2008). The graph neural network model. IEEE transactions on neural networks, 20(1), 61-80.
81. Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., ... & Sun, M. (2020). Graph neural networks: A review of methods and applications. AI open, 1, 57-81.

ДОДАТОК А. СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА ЗА ТЕМОЮ ДИСЕРТАЦІЇ

1. Kubiuk, Y., Chernousov, A., Savchenko, A., Osadchyi, S., Kostenko, Y., & Likhomanov, D. (2019). Deep learning based automatic software defects detection framework. (Здобувачем запропоновано архітектуру системи, розроблено нейромережеву модель пошуку вразливостей, проведено експерименти)

2. Kubiuk, Y., & Kyselov, G. (2021). Comparative analysis of approaches to source code vulnerability detection based on deep learning methods. Technology audit and production reserves, 3(2), 59. (Здобувачем проведено аналіз існуючих методів детекції вразливостей, сформульовано вимоги до удосконалення підходів, запропоновано концепцію поєднання методів)

3. Kaliuzhna, T., & Kubiuk, Y. (2022). Analysis of machine learning methods in the task of searching duplicates in the software code. Technology audit and production reserves, 4(2 (66)), 6-13. (Здобувач приймав участь у проведенні аналізу ефективності методів машинного навчання для пошуку дублікатів коду, а також визначав експерименти, що мають бути проведені при вирішення задач пошуку подібності коду та пошуку вразливостей)

4. Kubiuk, Y., & Kyselov, G. (2023). Development of an algorithm for code clone detection in source code based on abstract syntax tree. Technology audit and production reserves, 4(2 (72)), 33-36. (Здобувачем запропоновано метод детекції клонів коду на основі ковзного хешування AST та алгоритм його роботи, проведено експерименти та проаналізовано результати)

Статті у неперіодичний збірниках наукових праць:

5. Черноусов, А. В., Савченко, А. Ю., & Куб'юк, Є. Ю. Методи виявлення помилок безпеки в програмному забезпеченні на основі глибинного навчання, XVII Всеукраїнська науково-практична конференція студентів, аспірантів та молодих вчених «Теоретичні і прикладні проблеми фізики,

математики та інформатики» (Україна, м. Київ, 25-26 квітня 2019 р.) : матеріали конференції. – Київ : КПІ ім. Ігоря Сікорського, 2019. (Здобувачем досліджено ефективність методів глибинного навчання для детекції дефектів ПЗ, проаналізовано архітектури нейромереж, розроблено рекомендації щодо застосування)

ДОДАТОК Б. ДОВІДКА ПРО ВПРОВАДЖЕННЯ В НАВЧАЛЬНИЙ ПРОЦЕС

ЗАТВЕРДЖУЮ

Заступник директора НН ІПСА

НТУУ "КПІ ім. Ігоря Сікорського"

з науково-педагогічної роботи, д.т.н., професор

_____ Віктор РОМАНЕНКО

ДОВІДКА
про впровадження результатів дисертаційної роботи
Куб'юка Євгенія Юрійовича
у навчальний процес НН ІПСА НТУУ «КПІ ім. Ігоря Сікорського»

У процесі виконання дисертаційної роботи Куб'юком Євгенієм Юрійовичем отримано ряд нових наукових результатів стосовно розробки методів та засобів автоматизованого аналізу програмного коду на предмет вразливостей. Результати дисертаційного дослідження використані в проекті СП 2023-2 «Забезпечення захищеності і цифрової доступності веб-застосунків інтелектуальних розподілених середовищ» (номер держреєстрації 0123U101334), що виконується згідно Тематичного плану виконання ініціативних кафедральних науково-дослідних робіт Навчально-наукового інституту прикладного системного аналізу КПІ ім. Ігоря Сікорського, та впроваджені в навчальному процесі кафедри системного проєктування.

Гібридний метод аналізу програмного коду, що поєднує глибоке навчання для пошуку вразливостей та ковзне хешування AST для класифікації типу вразливості, використано в методичному забезпеченні лабораторного практикуму з дисципліни «Сучасні технології проектування програмних систем». Впровадження результатів дисертаційної роботи Куб'юка Є.Ю. в навчальний процес дозволило підвищити якість підготовки студентів другого рівня вищої освіти (наукових магістрів) спеціальності 122 – комп'ютерні науки.

Завідувач кафедри системного проектування, д.т.н., професор — Вадим МУХІН

Вчений секретар кафедри системного проектування, к.т.н. — Олександр БЕЗНОСИК

ДОДАТОК В. АКТ ВПРОВАДЖЕННЯ НА ПІДПРИЄМСТВІ

«ЗАТВЕРДЖУЮ»

директор з інформаційної безпеки, Самсунг РнД Інститут Україна, український центр досліджень та розробок Samsung, к.ф.-м.н.
Мохонько Олексій Анатолійович

« 05 » березня 2024 року

АКТ

впровадження результатів досліджень дисертаційної роботи
Куб'юка Євгенія Юрійовича на тему «Аналіз програмного коду з використанням гібридного методу пошуку та класифікації вразливостей» на здобуття наукового ступеня кандидата технічних наук за спеціальністю 122 – комп'ютерні науки

Підрозділ із досліджень та розробки у галузі кібербезпеки Самсунг РнД Інститут Україна інформує, що, у контексті вирішення завдань автоматизації аналізу програмного коду, було використано результати опублікованих Куб'юком Є.Ю. матеріалів наукових робіт за темою «Аналіз програмного коду з використанням гібридного методу пошуку та класифікації вразливостей». Зокрема, методи представлення вихідного коду у вигляді сукупності code gadgets, що є результатом трансформації абстрактного синтаксичного дерева коду, а також методи пошуку клонів у коді. Особливістю даного підходу є забезпечення високого рівня точності пошуку та класифікації вразливостей в C/C++ коді, при збереженні малої тривалості процедури аналізу, а також низький рівень хибно-позитивних спрацювань.

Застосування запропонованого Куб'юком Є.Ю. методу спростило процес ручного аналізу програмного коду на предмет вразливосей спеціалістами з кібербезпеки та дозволило підвищити якість продукту. Розглянуті та запропоновані у дисертаційній роботі методи використано і при розробці пропозицій щодо майбутніх проектів.

к.т.н., старший інженер
Самсунг РнД Інститут Україна,
керівник лабораторії "Platform Security Lab"

Сінельнікова Ольга Ігорівна

ТОВ «САМСУНГ РнД ІНСТИТУТ УКРАЇНА» 44648320

04 березня 2024 року