ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
МІНІСТЕРСТВА ОСВІТИ І НАУКИ УКРАЇНИ

Кваліфікаційна наукова праця
на правах рукопису

СЄРГЄЄВ ЄВГЕНІЙ ВІТАЛІЙОВИЧ

УДК 004.03:004.49:004.45

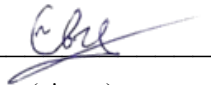
ДИСЕРТАЦІЯ

МЕТОДИ ТА ЗАСОБИ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ В ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННІ КОМП'ЮТЕРНИХ СИСТЕМ

123 Комп'ютерна інженерія
(шифр і назва спеціальності)

12 Інформаційні технології
(галузь знань)

Подається на здобуття наукового ступеня доктора філософії
Дисертація містить результати власних досліджень. Подані до захисту наукові положення є власним напрацюванням. Всі використані ідеї, наукові результати, цитати супроводжуються належними посиланнями на їх авторів та джерела опублікування. Всі частини тексту дисертації, під час написання яких використовувались технології штучного інтелекту, перевірені та відредаговані особисто.



_____ Є. В. Сєргєєв
(підпис)

Наукові керівники:
Савенко Олег Станіславович, доктор технічних наук, професор
Кльоц Юрій Павлович, кандидат технічних наук, доцент

Хмельницький – 2026

АНОТАЦІЯ

Сєргєєв Є. В. Методи та засоби виявлення вразливостей в програмному забезпеченні комп'ютерних систем. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії з галузі знань 12 Інформаційні технології за спеціальністю 123 Комп'ютерна інженерія. – Хмельницький національний університет, Хмельницький. 2026.

Вирішення задачі підвищення точності виявлення вразливостей у програмному забезпеченні комп'ютерних систем є однією з актуальних наукових проблем інформаційної безпеки. Серед найнебезпечніших помилок мов C/C++ залишаються переповнення буфера, які призводять до неконтрольованого перезапису пам'яті та виконання довільного коду. Існуючі статичні та динамічні інструменти аналізу коду мають обмеження щодо точності та масштабованості, що вимагає розроблення нових підходів, здатних формалізувати природу вразливостей та забезпечити їх автоматизоване виявлення у промислових проєктах та конвеєрах автоматизованого збирання та розгортання(CI/CD).

У дисертації здійснено аналіз методів та засобів виявлення вразливостей типу переповнення буферу у програмному забезпеченні мов C/C++, підходів статичного аналізу коду, методів машинного навчання та гібридних моделей, а також сучасних рішень їх інтеграції в конвеєр автоматизованого збирання та розгортання. У роботі розроблено нову орієнтовану графову модель переповнення буфера, що формалізує потоки даних та керування у програмах C/C++ і забезпечує основу для кількісної оцінки ризику експлуатації вразливостей, розроблено новий метод автоматизованого виявлення вразливостей типу переповнення буфера, який враховує просторові й контекстні залежності між елементами програмного коду на основі графових моделей та нейромережевої архітектури типу YOLO/Transformer, розроблено новий метод підготовки та обробки даних для тренування нейронних детекторів на основі сегментації орієнтованих графів і перетворення підграфів у багатоканальні зображення з класами стек, купа та помилка на одиницю, розроблено новий метод композитної оцінки ризику експлуатації виявлених

вразливостей з інтеграцією в конвеєри інтеграції та розгортання для автоматизованого визначення пріоритетів виправлень і блокування небезпечних збірок, а також здійснено постановку експериментів і проведено експериментальні дослідження із запропонованими моделями та методами.

Об’єктом дослідження є процес виявлення та аналізу вразливостей у програмному забезпеченні комп’ютерних систем.

Предметом дослідження є методи представлення переповнення буфера, методи автоматизованого виявлення вразливостей типу переповнення буферу у коді C/C++, методи оцінювання ризику та алгоритми інтеграції результатів у конвеєри автоматизованого збирання та розгортання.

Метою дисертаційного дослідження є підвищення точності виявлення вразливостей у програмному забезпеченні комп’ютерних систем шляхом формалізації переповнення буфера, створення нейромережевих детекторів та впровадження механізмів композитної оцінки ризику для автоматизованого прийняття рішень.

Наукова новизна отриманих результатів полягає в наступному:

1) удосконалено модель процесу виявлення вразливостей, в якій на відміну від відомих передбачено інтеграцію графової моделі, нейромережевого детектора та модуля композитної оцінки ризику в конвеєри автоматизованого збирання та розгортання, що дає змогу забезпечити підтримку повного циклу аналізу, тобто від початкового коду до блокування небезпечних збірок;

2) розроблено новий метод автоматизованого виявлення вразливостей «переповнення буфера», який на відміну від відомих враховує просторові й контекстні залежності між елементами програмного коду на основі графових моделей та нейромережевої архітектури YOLO/Transformer, що дало змогу підвищити точність і повноту виявлення переповнень буфера у системному програмному забезпеченні;

3) розроблено новий метод підготовки та обробки даних для тренування нейронних детекторів, який на відміну від відомих характеризується побудовою та сегментацією орієнтованих графів і перетворенням інформативних підграфів у

багатоканальні зображення з класами стек, купа та помилки на одиницю, що дало змогу формувати відтворювані навчальні вибірки, узгоджені з кореневими причинами вразливостей, та підвищити ефективність навчання нейромережевих архітектур для обробки зображень;

4) розроблено новий метод композитної оцінки ризику експлуатації виявлених вразливостей, який на відміну від відомих характеризується інтеграцією у конвеєри автоматизованого збирання та розгортання та узгодженням показників ризику з результатами нейромережевого детектування, що дає змогу автоматизувати визначення пріоритету виправлень, ранжування вразливостей за рівнем ризику і блокування небезпечних збірок.

Практичне значення отриманих результатів. За результатами виконаних досліджень розроблено комплекс моделей та методів виявлення вразливостей типу переповнення буферу у програмному забезпеченні комп'ютерних систем та програмні засоби для їх застосування. Використання запропонованих методів дозволяє підвищити точність і швидкодію аналізу коду та інтегрувати автоматизоване виявлення вразливостей у конвеєри автоматизованого збирання та розгортання. Ефективність розроблених рішень підтверджено експериментальними дослідженнями. Нейромережевий детектор на основі графових моделей показав покращення точності та повноти виявлення, а метод композитної оцінки ризику забезпечує автоматизований пріоритет вразливостей.

Теоретичні та практичні результати дослідження впроваджені в ТОВ «Nolt technologies» (м.Хмельницький, Акт від 16.02.2026), ТОВ «ІТТ» (м. Хмельницький, Акт від 16.02.2026), а також, в освітньому процесі Хмельницького національного університету (Акт від 25.02.2026) при викладанні дисциплін на кафедрі комп'ютерної інженерії та інформаційних систем для здобувачів спеціальності F7 Комп'ютерна інженерія, зокрема в курсах «Безпека та захист комп'ютерних систем», «Моделювання та методи оптимізації в наукових та експериментальних дослідженнях», «Методології забезпечення якості, надійності, гарантоздатності та безпеки комп'ютерних систем та мереж».

У вступі представлено обґрунтування актуальності наукової задачі забезпечення підвищення точності виявлення вразливостей у програмному забезпеченні, сформульовано мету, завдання, об’єкт і предмет дослідження, показано зв’язок тематики дисертації з науковими програмами і проєктами, наведено відомості про наукову новизну, практичне значення, апробацію результатів та публікації автора.

У першому розділі здійснено аналіз предметної області дослідження, класифікацію вразливостей програмного забезпечення та розглянуто відомі методи їх виявлення, зокрема статичний і динамічний аналіз, підходи машинного навчання та глибинних нейронних мереж. Проаналізовано існуючі інструменти виявлення переповнення буфера та їх інтеграцію в процеси розробки ПЗ, охарактеризовано вимоги DevSecOps-підходів та виявлено недоліки наявних рішень.

У другому розділі розроблено формальну модель вразливостей переповнення буфера на основі уніфікованого графа програми з анотаціями буферів, операцій роботи з пам’яттю та потоків даних. Визначено локальні та шляхові показники ризику, побудовано математичну модель композитної оцінки ризику експлуатації вразливостей, сформульовано критерії віднесення буферів до класів Stack/Heap/Off-by-one та вимоги до відтворюваності побудови графів.

У третьому розділі розроблено нейромережеві методи виявлення вразливостей переповнення буфера на основі графових представлень коду: метод YOLO-типу, що працює з растрованими підграфами; метод на основі трансформерної архітектури. Описано метод підготовки даних на основі вибору інформативних підграфів, їх нормалізації та перетворення у багатоканальні зображення, наведено схему навчання і валідації моделей, а також показано інтеграцію результатів аналізу в метод композитної оцінки ризику.

У четвертому розділі представлено програмну реалізацію запропонованих моделей і методів, описано архітектуру програмного комплексу для виявлення переповнення буфера та обчислення композитної оцінки ризику, інтеграцію детектора в конвеєри автоматизованого збирання та розгортання, а також наведено результати експериментальних досліджень, порівняння з відомими інструментами

статичного аналізу та нейромережевими підходами, виконано аналіз точності, повноти та продуктивності запропонованих рішень.

У висновках представлено отримані наукові та практичні результати дослідження.

У додатках представлено наукові публікації, в яких відображено основні наукові результати роботи, акти впровадження результатів роботи.

Ключові слова: комп’ютерні системи, зловмисні програми, бази даних вразливостей, переповнення буфера, OWASP, нейронні мережі, YOLO, трансформер, розпізнавання зображень, паралельне обчислення, CNN, комп'ютерний зір, вразливості, безпека.

ANNOTATION

Sierhieiev Ye. V. Methods and means of detecting vulnerabilities in computer system software. – Qualification scientific work as a manuscript.

Dissertation for the degree of Doctor of Philosophy in the specialty 123 – Computer Engineering. – Khmelnytskyi National University, Khmelnytskyi, 2026.

Increasing the efficiency of vulnerability detection in computer system software is a current scientific problem in information security. Among the most dangerous errors in the C/C++ languages, buffer overflows remain, which lead to uncontrolled memory overwriting and arbitrary code execution. Existing static and dynamic code analysis tools have limitations in terms of accuracy and scalability, which require the development of new approaches that can formalise the nature of vulnerabilities and ensure their automated detection in industrial projects and CI/CD processes.

The dissertation analyzes methods and tools for detecting buffer overflow vulnerabilities in C/C++ software, including static code analysis approaches, machine learning techniques, hybrid models, and modern solutions for integrating them into CI/CD pipelines.

The object of the study is the process of identifying and analyzing vulnerabilities in computer system software.

The subject of the research is methods for representing buffer overflows, methods for automated detection of buffer overflow vulnerabilities in C/C++ code, risk assessment methods, and algorithms for integrating results into CI/CD pipelines.

The dissertation research aims to enhance the effectiveness of vulnerability detection techniques in software by formalising buffer overflows, developing neural network detectors, and implementing composite risk assessment mechanisms for automated decision-making.

The scientific novelty of the results obtained is as follows:

1) the model of the vulnerability detection process has been improved, in which, unlike the known ones, integration of a graph model, a neural network detector and a composite risk assessment module into CI/CD pipelines is provided, which makes it possible to support the full cycle of analysis - from the source code to the blocking of dangerous assemblies;

2) a new method of automated detection of "buffer overflow" vulnerabilities was developed, which, unlike the known ones, takes into account spatial and contextual dependencies between program code elements based on graph models and the YOLO/Transformer neural network architecture, which makes it possible to increase the accuracy and completeness of detecting buffer overflows in system software;

3) a new method of data preparation and processing for training neural detectors was developed, which, unlike the known ones, is characterized by the construction and segmentation of directed graphs and the transformation of informative subgraphs into multi-channel images with classes of stack, heap and off-by-one errors, which made it possible to form reproducible training samples, consistent with the root causes of vulnerabilities, and increase the effectiveness of learning neural network architectures for image processing;

4) a new method of composite risk assessment of the exploitation of discovered vulnerabilities has been developed, which, unlike the known ones, is characterized by integration into CI/CD pipelines and matching of risk indicators with the results of neural network detection, which makes it possible to automate the determination of the priority

of fixes, the ranking of vulnerabilities by risk level, and the blocking of dangerous assemblies.

Practical significance of the results obtained. Based on the research findings, a set of models and methods for detecting buffer overflow vulnerabilities in software, along with software tools for their implementation, was developed. The use of these proposed methods enhances the accuracy and speed of code analysis and allows for the integration of automated vulnerability detection into CI/CD processes. The effectiveness of the developed solutions was validated through experimental studies: a neural network detector based on graph models demonstrated improved accuracy and completeness in detection, while the composite risk assessment method facilitates automated prioritisation of vulnerabilities.

The theoretical and practical results of the research are implemented in "Nolt technologies" LLC (Khmelnytskyi) (Act of 16.02.2026), ITT LLC (Khmelnytskyi) (Act of 16.02.2026), as well as in the educational process of the Khmelnytskyi National University(Act of 25.02.2026) in the teaching of disciplines at the Department of Computer Engineering and Information Systems for students of the F7 Computer Engineering specialty, in particular in the courses "Security and protection of computer systems", "Modeling and optimization methods in scientific and experimental research", "Methodologies for ensuring the quality, reliability, warranty and security of computer systems and networks".

The introduction presents the justification of the relevance of the scientific task of ensuring the effectiveness of methods and tools for detecting vulnerabilities in software, outlines the goal, objectives, object, and subject of the research, demonstrates the connection between the dissertation topic and scientific programmes and projects, provides information about the scientific novelty, practical significance, validation of the results, and the author's publications.

The first section examines the subject area of the study, classifies software vulnerabilities, and reviews known detection methods, including static and dynamic analysis, machine learning techniques, and deep neural networks. It analyses existing buffer overflow detection tools and their integration into software development

processes. The requirements of CI/CD and DevSecOps approaches are described, and the limitations of current solutions are highlighted.

In the second section, a formal model of buffer overflow vulnerabilities is developed based on a unified programme graph with buffer annotations, memory operations, and data flows. Local and path risk indicators are determined; a mathematical model for composite vulnerability exploitation risk assessment is constructed; criteria for assigning buffers to Stack/Heap/Off-by-one classes and requirements for graph construction reproducibility are formulated.

In the third section, neural network methods for detecting buffer overflow vulnerabilities based on graph representations of code are developed. These are a YOLO-type method working with rasterised subgraphs and a method based on a transformer architecture. A data preparation method based on the selection of informative subgraphs, their normalisation and transformation into multichannel images is described. The model training and validation scheme is presented, and the integration of the analysis results into the composite risk assessment method is also shown.

The fourth section presents the software implementation of the proposed models and methods, describes the architecture of the software complex for detecting buffer overflows and calculating the composite risk score, the integration of the detector into CI/CD pipelines, as well as the results of experimental studies. A comparison with well-known static analysis tools and neural network approaches is presented. An analysis of the accuracy, completeness, and performance of the proposed solutions is performed.

The conclusions present the scientific and practical results obtained from the research.

The Appendices include scientific publications that reflect the main scientific findings of the work, as well as acts of implementing its results.

Keywords: computer systems, malicious program, vulnerability database, buffer overflow, OWASP, neural networks, YOLO, transformer, image recognition, parallel computing, CNN, computer vision, vulnerabilities, security.

Список публікацій здобувача за темою дисертації

Наукові праці, в яких опубліковані основні наукові результати дисертації:

1. Сєргєєв Є., Каштальян А., Ковальчук В., Савенко О., Іванченко О. Ефективність і вдосконалення SAST у контексті SQL Injection вразливостей. *Information Technology: Computer Science, Software Engineering and Cyber Security*. 2024. № 3. С. 149-158. DOI: https://doi.org/10.32782/IT/2024-3-16

2. Сєргєєв Є.В., Савенко О.С. Виявлення вразливостей переповнення буфера в системному програмному забезпеченні на основі графа та моделі трансформатора. *Вчені записки ТНУ імені В.І. Вернадського. Серія: Технічні науки.* 2025. № 6. С. 318 – 327. DOI: https://doi.org/10.32782/2663-5941/2025.6.2/43

3. Сєргєєв Є.В. Підготовка даних на основі графіків для виявлення вразливостей переповнення буфера в коді в рамках CI/CD-процесів. *Herald Of Khmelnytskyi National University. Technical Sciences*. 2026. № 361(1), Pp. 316–322. DOI: https://doi.org/10.31891/2307-5732-2026-361-45

4. Сєргєєв, Є.В., Кльоц Ю.П. Композитна оцінка ризику переповнення буфера і її трансляція в дії CI/CD. *MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES*. 2025. № 84(4), Pp. 89–94. DOI: https://doi.org/10.31891/2219-9365-2025-84-10

Праці, які засвідчують апробацію матеріалів дисертації:

5. Sierhieiev Y., Paiuk V., Sachenko A., Nicheporuk A., Kwiecien A. A graph-based vulnerability detection method. (2024) *CEUR Workshop Proceedings*, 3675, pp. 388-401. *The 5th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS-2024)* : CEUR-Workshop Proceedings. Vol. 3675. (Khmelnytskyi, March 2024). Khmelnytskyi, 2024. Pp. 343-355. URL: https://ceur-ws.org/Vol-3675/paper25.pdf

6. Sierhieiev Y., Paiuk V., Nicheporuk A., Kwiecien A., Huralnyk O. Detection and prediction of the vulnerabilities in software systems based on behavioral analysis with machine learning. (2024) *CEUR Workshop Proceedings*, 3736, pp. 239-254. *The 1th* Proceedings of the 1st International Workshop on Intelligent & CyberPhysical

Systems (ICyberPhyS 2024) Khmelnytskyi, Ukraine, June 28, 2024 : CEUR-Workshop Proceedings. Vol. 3736. (Khmelnytskyi, Ukraine, June 28, 2024). Khmelnytskyi, 2024. Pp. 239-254. URL: https://ceur-ws.org/Vol-3736/paper18.pdf

7. Savenko O., Lips S., Gaj P., Sierhieiev Y. Graph-based data preparation for detecting buffer overflow vulnerabilities in code within CI/CD pipelines. (2025) CEUR Workshop Proceedings, 4163, pp. 1–10. The 2nd International Workshop on Advanced Applied Information Technologies: AI & DSS (AdvAIT-2025), December 05, 2025, Khmelnytskyi, Ukraine: CEUR-Workshop Proceedings. Vol. 4163. Khmelnytskyi, 2025. Pp. 1–10. URL: https://ceur-ws.org/Vol-4163/paper19.pdf

8. Savenko O., Sierhieiev Y., Gaj P., Balej J. Using artificial intelligence in the context of buffer overflow vulnerabilities. *The 2nd International Workshop on Intelligent & CyberPhysical Systems (ICyberPhyS 2025), CEUR Workshop Proceedings*. Vol. 4013. Khmelnytskyi, Ukraine, 4 July 2025. Pp. 211–220. URL: https://ceur-ws.org/Vol-4013/paper17.pdf

9. Savenko O., Gaj P., Sierhieiev Ye. Detection of buffer overflow vulnerabilities in system software based on a graph and transformer model. *AISSLE-2025: The International Workshop on Applied Intelligent Security Systems in Law Enforcement*, October, 30–31, 2025, Vinnytsia, Ukraine. Pp. 292-305. URL: https://ceur-ws.org/Vol-4126/paper17.pdf

10. Sierhieiev Y., Paiuk V., Savenko O., Drozd A. Improvement of effectiveness for Static Application Security Testing for detection of SQL Injection vulnerabilities. *IEEE 14th International Conference on Dependable Systems, Services and Technologies (DESSERT-2024)* : Proceedings. Athens, Greece, October 11–13, 2024. Pp. 1–6. DOI: https://doi.org/10.1109/DESSERT65323.2024.11122171

Публікації, які додатково відображають наукові результати дисертації:

11. Сєргєєв Є. В., Савенко О. С. Авторське свідоцтво №143407. Україна. Комп'ютерна програма «OverflowGuard: система аналізу переповнення буфера та оцінки ризику в CI/CD». Дата реєстрації: 23 лютого 2026 р.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ…… 14
ВСТУП……... 15
РОЗДІЛ 1 АНАЛІЗ МЕТОДІВ ТА ЗАСОБІВ РОЗПІЗНАВАННЯ ВРАЗЛИВОСТЕЙ В ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННЮ КОМП'ЮТЕРНИХ СИСТЕМ…….. 25
1.1 Вразливості, класифікація вразливостей, етапи прояву вразливостей… 25
1.2 Методи виявлення вразливостей……. 35
1.3 Засоби виявлення вразливостей програмного забезпечення комп'ютерних систем…….. 41
1.4 Перспективні стратегії до виявлення вразливостей програмного забезпечення комп'ютерних систем……... 47
1.5 Постановка задачі дослідження……... 49
1.6 Висновки до першого розділу…….. 50
РОЗДІЛ 2 МОДЕЛІ ВРАЗЛИВОСТЕЙ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНИХ СИСТЕМ ТА ПРОЦЕСУ ЇХ ВИЯВЛЕННЯ…….. 52
2.1 Визначення області дослідження ……... 53
2.2 Моделі класів вразливостей……. 65
2.3 Формування трьох класів вразливостей на основі баз вразливостей…… 75
2.4 Модель процесу виявлення вразливостей в ПЗКС на основі нейромережі YOLO…….. 80
2.5 Узагальнена модель процесу виявлення вразливостей у ПЗКС та оцінювання ризику……... 88
2.6 Висновки до другого розділу……... 92
РОЗДІЛ 3 МЕТОДИ АНАЛІЗУ ТА ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ ТИПУ ПЕРЕПОВНЕННЯ БУФЕРУ НА ОСНОВІ РОЗРОБЛЕНИХ МОДЕЛЕЙ…… 94
3.1 Підготовки даних для виявлення переповнень пам'яті…… 94
3.2 Метод автоматичного виявлення вразливостей на основі допрацьованої нейронної мережі YOLO…… 107
3.3 Метод захисту програмного коду комп'ютерних систем……. 121

3.4 Висновки до третього розділу……. 135
РОЗДІЛ 4 ВИЗНАЧЕННЯ ЕФЕКТИВНОСТІ МОДЕЛЕЙ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНИХ СИСТЕМ…… 137
4.1 Опис системи та місця кожного методу……. 137
4.2 Методика отримання результатів та відтворюваність ……. 148
4.3 Експерименти і аналіз результатів…….. 154
4.4 Висновки до четвертого розділу……. 173
ВИСНОВКИ……... 175
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ…….. 177
ДОДАТОК А Список публікацій здобувача…… 196
ДОДАТОК Б Акти впровадження…… 199
ДОДАТОК В Керівництво користувача…….. 205
ДОДАТОК Г Лістинг програмного коду (фрагмент)……. 211

ПЕРЕЛІК СКОРОЧЕНЬ

ПЗКС – Програмне забезпечення комп’ютерних систем

Метод 1 – Метод підготовки даних для виявлення переповнень пам’яті

Метод 2 – Метод автоматичного виявлення вразливостей на основі допрацьованої нейронної мережі YOLO

Метод 3 – Метод захисту програмного коду комп’ютерних систем.

ШІ – Artificial intelligence (штучний інтелект)

AdaBoost– Adaptive Boosting (мета-алгоритм статистичної класифікації)

CNN – Convolutional Neural Network (згорткова нейронна мережа)

GT – Ground truth (валідаційна вибірка образів для тестування детектора)

KLT – Kanade–Lucas–Tomasi (алгоритм Канаде–Лукаса–Томазі)

LF – Loss Function (функція втрат)

MH – Machine Learning (машинне навчання)

OD – Object Detection (пошук об’єкта)

PR – Pattern recognition (розпізнавання образів)

ReLU – Rectified Linear Unit (випрямлений лінійний вузол)

RTK – Real Time Kinematic (позиціонування в режимі реального часу)

SE – Squeeze and Networks (Стискання та мережі)

SSD – Single Shot Detector (однопрохідна нейронно-мережева архітектура)

YOLO – You Only Look Once (однопрохідна нейронно-мережева архітектура)

WC – Weak Classifiers (слабкі класифікатори)

ВСТУП

Обґрунтування вибору теми дослідження. Сучасний комп’ютерний світ є сукупністю різноманітних і надскладних обчислювальних пристроїв, систем обробки інформації, телекомунікаційних технологій, програмного забезпечення та засобів його проектування. Взаємодіючи між собою, ці компоненти формують критичну інфраструктуру, яка забезпечує функціонування державних, військових, промислових, фінансових та побутових інформаційних систем. Зі зростанням складності програмних комплексів, масштабів розподілених обчислень та обсягів програмного коду стрімко зростає й кількість потенційних вразливостей у програмному забезпеченні комп’ютерних систем.

Під програмним забезпеченням комп'ютерних систем (ПЗКС) приймемо системні компоненти, які безпосередньо взаємодіють із ресурсами обчислювальної платформи, такими як пам'ять, процесор і пристроями введення або виведення та системними інтерфейсами операційних систем. До цих складових відносяться операційні системи та їх підсистеми, драйвери, системні служби, прошивки чи вбудоване програмне забезпечення, а також системні бібліотеки. Вони функціонують на рівні управління ресурсами та виконання базових операцій.

Чим відповідальнішою є сфера застосування комп’ютерних інформаційних технологій, тим більш критичними стають вимоги до надійності та безпеки програмного забезпечення, яке реалізує функції збору, накопичення, обробки, передачі та зберігання даних. Зловмисний вплив на програмні компоненти інформаційних систем здійснюється через експлуатацію вразливостей у коді, що призводить до порушення конфіденційності, цілісності та доступності інформації, компрометації цільових систем, а в окремих випадках – до повного виведення з ладу критичних об’єктів інфраструктури. Особливо небезпечними є вразливості, пов’язані з некоректною роботою з пам’яттю, зокрема переповненням буферів, які традиційно залишаються одним із найпоширеніших та найважчих для виявлення класів вразливостей у мовах системного програмування C/C++.

У межах даного дослідження розглядаються вразливості, характерні для системного програмного забезпечення комп’ютерних систем, зокрема операційних систем, драйверів пристроїв, вбудованих та низькорівневих програмних компонентів. Основна увага приділяється програмному коду, написаному на мовах C та C-подібних, оскільки ці мови характеризуються прямим управлінням пам'яттю і високою ймовірністю виникнення помилок при доступі до неї. У цій роботі розглянуто вразливості, пов’язані з порушенням цілісності пам’яті. Зокрема, увагу приділено переповненню буфера в стеку (Stack Overflow), переповненню буфера в купі (Heap Overflow) та помилкам індексації типу off-by-one.

Значний внесок у розробку теоретичних основ і практичних методів виявлення вразливостей у програмному забезпеченні комп’ютерних систем зробили українські та іноземні вчені: С. Бучик [150], L. Allodi [5], R. Amankwah [7], A. Anwar [9], Ö. Aslan [10; 11], M. Humayun [57], D. Li [76], Z. Li [77], G. Lin [79], Z. Shen [124], X. Zhou [149]. Використанню методів статичного та динамічного аналізу для виявлення вразливостей у програмному забезпеченні КС присвячені роботи R. Chandrashekhar [24], L. Do [36], N. Kasmawi [64], A. Mohaidat [94], T. Oyetoyan [104], S. Panjwani [106], S. Shah [122], J. Softić [130], C. Vassallo [141], Y. Wang [142]. Запропоновано низку підходів, що базуються на формальних методах, евристичному аналізі, машинному навчанні й глибинних нейронних мережах, проте їх практична точність істотно залежить від представлення початкового коду, повноти врахування контексту виконання та здатності масштабуватися на великі програмні системи.

Незважаючи на значний обсяг проведених наукових досліджень і отриманих результатів, на сьогодні надзвичайно актуальною залишається задача підвищення точності методів і засобів виявлення вразливостей у програмному забезпеченні комп’ютерних систем, зокрема в умовах складної архітектури, розгалужених шляхів виконання та обмежень реального часу. Актуальною є побудова таких моделей і методів аналізу коду, які поєднують формальні представлення програм (графові моделі потоків керування і даних, моделі пам’яті) з інструментами інтелектуального аналізу (глибинні нейронні мережі, гібридні архітектури), що

дозволить підвищити повноту та точність виявлення небезпечних конструкцій, насамперед переповнення буфера, і забезпечити їх інтеграцію у промислові процеси розробки системного програмного забезпечення комп'ютерних систем.

Отже, актуальною науковою задачею є розробка моделей, методів та засобів виявлення вразливостей у системного програмному забезпеченні комп'ютерних систем, орієнтованих на формалізоване представлення програмних об'єктів, кількісну оцінку ризику експлуатації та інтеграцію розроблених рішень у сучасні технології розробки й супроводу програмного забезпечення, для підвищення точності виявлення.

Зазначена науково-прикладна задача відповідає предметній області Стандарту вищої освіти України зі спеціальності 123 – Комп'ютерна інженерія для третього (освітньо-наукового) рівня вищої освіти, зокрема, такому об'єкту вивчення та діяльності, як « - … цифрові комп'ютери та комп'ютерні системи, …, методи та способи подання, отримання, зберігання, передавання, опрацювання та захисту в них інформації, …; - інформаційні процеси, технології, методи, способи, інструментальні засоби та системи для дослідження, … й експлуатації комп'ютерів та комп'ютерних систем і мереж, …, IT-інфраструктур, розроблення, верифікації та розгортання програмного забезпечення та систем у хмарних та інших середовищах, а також процедури та засоби підтримки та керування життєвим циклом, забезпечення якості, надійності та безпеки.».

Зв'язок роботи з науковими програмами, планами, темами.

Дисертаційне дослідження виконувалось у рамках науково-дослідної тематики Хмельницького національного університету: держбюджетної науково-дослідної теми №2Б-2024 «Система виявлення ЗПЗ та комп'ютерних атак в корпоративних мережах з використанням хибних об'єктів атак та пасток» (номер держреєстрації 0124U000980); держбюджетної науково-дослідної теми №1Б-2026 «Система забезпечення стійкості до витоку конфіденційної інформації в корпоративних мережах в умовах впливів комп'ютерних атак» (номер держреєстрації 0126U002082), в яких автор дисертації був виконавцем.

Мета і завдання дослідження.

Об’єктом дослідження є процес виявлення та аналізу вразливостей у програмному забезпеченні комп’ютерних систем.

Предметом дослідження є методи представлення переповнення буфера, методи автоматизованого виявлення вразливостей типу переповнення буферу у коді C/C++, методи оцінювання ризику та алгоритми інтеграції результатів у конвеєри автоматизованого збирання та розгортання.

Метою дисертаційного дослідження є підвищення точності виявлення вразливостей у програмному забезпеченні комп’ютерних систем шляхом формалізації переповнення буфера, створення нейромережевих детекторів та впровадження механізмів композитної оцінки ризику для автоматизованого прийняття рішень.

Задачі дослідження формулюються в роботі наступним чином:

1. Провести системний аналіз існуючих підходів до виявлення вразливостей у програмному забезпеченні, визначити їх переваги й недоліки та сформувати вимоги до автоматизованих засобів детектування переповнення буфера.

2. Створити формальні моделі класів вразливості типу переповнення буфера у вигляді орієнтованого графа з атрибутами вузлів і ребер, що відображають залежності між даними й керуванням.

3. Удосконалити модель процесу виявлення вразливості типу переповнення буфера, в якій здійснити інтеграцію графової моделі, нейромережевого детектора та модуля композитної оцінки ризику в конвеєри автоматизованого збирання та розгортання, для забезпечення підтримки повного циклу аналізу коду та підвищення точності виявлення вразливості типу переповнення буфера.

4. Розробити метод машинного виявлення переповнення буфера з використанням нейромережевої архітектури YOLO/Transformer, визначити правила сегментації графів та формування навчальних вибірок.

5. Розробити метод підготовки та обробки даних на основі розмітки початкового коду, побудові та сегментуванні орієнтованих графів, перетворенні підграфів у багатоканальні зображення з класами Stack/Heap/Off-by-one.

6. Розробити метод композитної оцінки ризику та алгоритм інтеграції в конвеєрах автоматизованого збирання та розгортання для кількісного оцінювання критичності виявлених вразливостей та автоматизованого управління процесом розгортання.

7. Реалізувати прототипи програмних засобів та провести експериментальні дослідження, інтегрувати їх у середовища розробки, оцінити точність і швидкодію порівняно з існуючими сканерами та сформулювати практичні рекомендації.

Методи дослідження. У роботі застосовувалися: теорія графів та елементи абстрактної алгебри для побудови формальних моделей; методи машинного навчання – конволюційні та трансформерні нейронні мережі; теоретичні основи інформаційних технологій і захисту інформації; статистичні методи оцінки якості; практики DevSecOps для інтеграції рішень у конвеєри автоматизованого збирання та розгортання.

Наукова новизна отриманих результатів полягає в наступному:

1) удосконалено модель процесу виявлення вразливостей, в якій на відміну від відомих передбачено інтеграцію графової моделі, нейромережевого детектора та модуля композитної оцінки ризику в конвеєри автоматизованого збирання та розгортання, що дає змогу забезпечити підтримку повного циклу аналізу, тобто від початкового коду до блокування небезпечних збірок;

2) розроблено новий метод автоматизованого виявлення вразливостей «переповнення буфера», який на відміну від відомих враховує просторові й контекстні залежності між елементами програмного коду на основі графових моделей та нейромережевої архітектури YOLO/Transformer, що дало змогу підвищити точність і повноту виявлення переповнень буфера у системному програмному забезпеченні;

3) розроблено новий метод підготовки та обробки даних для тренування нейронних детекторів, який на відміну від відомих характеризується побудовою та сегментацією орієнтованих графів і перетворенням інформативних підграфів у багатоканальні зображення з класами стек, купа та off-by-one помилки, що дало змогу формувати відтворювані навчальні вибірки, узгоджені з кореневими

причинами вразливостей, та підвищити ефективність навчання нейромережевих архітектур для обробки зображень;

4) розроблено новий метод композитної оцінки ризику експлуатації виявлених вразливостей, який на відміну від відомих характеризується інтеграцією у конвеєри автоматизованого збирання та розгортання та узгодженням показників ризику з результатами нейромережевого детектування, що дає змогу автоматизувати визначення пріоритету виправлень, ранжування вразливостей за рівнем ризику і блокування небезпечних збірок.

Обґрунтованість і достовірність наукових положень, висновків і рекомендацій. Достовірність результатів підтверджується коректним застосуванням математичного апарату, використанням різнорідних навчальних та тестових вибірок (open-source проєкти, синтетичні приклади), багатократними експериментами та порівнянням із відомими статичними й динамічними сканерами. Реалізовані прототипи забезпечують можливість практичної перевірки методів.

Практичне значення отриманих результатів. За результатами виконаних досліджень розроблено комплекс моделей та методів виявлення вразливостей типу переповнення буферу у програмному забезпеченні комп’ютерних систем та програмні засоби для їх застосування. Використання запропонованих методів дозволяє підвищити точність і швидкодію аналізу коду та інтегрувати автоматизоване виявлення вразливостей у конвеєри автоматизованого збирання та розгортання. Ефективність розроблених рішень підтверджено експериментальними дослідженнями. Нейромережевий детектор на основі графових моделей показав покращення точності та повноти виявлення, а метод композитної оцінки ризику забезпечує автоматизований пріоритет вразливостей.

У результаті проведених експериментальних досліджень було доведено точність роботи розробленої моделі та методів: використання запропонованого нейромережевого детектора на основі графових представлень коду забезпечує підвищення точності та повноти виявлення переповнення буфера і зменшує час аналізу порівняно з базовими статичними інструментами. Запропонований метод композитної оцінки ризику дає можливість автоматизувати пріоритезацію

виявлених вразливостей і прийняття рішень щодо блокування або дозволу збірок у конвеєрах автоматизованого збирання та розгортання.

Теоретичні та практичні результати дослідження впроваджені в ТОВ «Nolt technologies» (м. Хмельницький, Акт від 16.02.2026), ТОВ «ІТТ» (м. Хмельницький, Акт від 16.02.2026), а також, в освітньому процесі Хмельницького національного університету (Акт від 25.02.2026) при викладанні дисциплін на кафедрі комп'ютерної інженерії та інформаційних систем для здобувачів спеціальності F7 Комп'ютерна інженерія, зокрема в курсах «Безпека та захист комп'ютерних систем», «Моделювання та методи оптимізації в наукових та експериментальних дослідженнях», «Методології забезпечення якості, надійності, гарантоздатності та безпеки комп'ютерних систем та мереж».

Особистий внесок здобувача. Всі основні результати дисертаційного дослідження, які представлені до захисту, отримані автором особисто. Постановка наукових задач, розроблення моделей, методів, програмних засобів та проведення експериментальних досліджень виконані у межах єдиної наукової концепції

У роботах, опублікованих одноосібно автором, отримано наступні результати: [154] – розроблено підхід до побудови графових моделей програмного коду та підготовки даних на їх основі для задачі виявлення вразливостей переповнення буфера у конвеєрах автоматизованого збирання та розгортання; [153] – запропоновано композитну оцінку ризику переповнення буфера та схему її інтеграції у конвеєрах автоматизованого збирання та розгортання для підтримки прийняття рішень щодо якості та безпеки програмного забезпечення.

У роботах, які опубліковані у співавторстві, автору належать основні ідеї, теоретична та практична розробка положень, відображених у характеристиці наукової новизни отриманих результатів, а саме: [128; 155] – здійснено формування вимог до сучасних статичних методів аналізу безпеки, розроблення підходів для підвищення ефективності SAST-засобів у контексті SQL-ін'єкцій, вдосконалення методу статичного тестування безпеки застосунків у контексті вразливостей SQL Injection; [119, 156] – здійснено побудову графової моделі представлення програм та розроблення нейромережевого методу детектування на

основі трансформерної архітектури; [120, 127] – розроблення підходу до побудови графових моделей коду для задачі виявлення вразливостей, визначення схеми перетворення таких моделей у вхідні дані для нейронних мереж і обґрунтування переваг графового представлення порівняно з лінійним; [121] – розроблення формальних моделей вразливостей переповнення буфера та нейромережевого методу їх виявлення на основі архітектури YOLO з використанням графових представлень коду; [126] – розроблення підходу до аналізу поведінкових характеристик програмних систем з використанням методів машинного навчання для виявлення та прогнозування вразливостей, а також адаптація отриманих положень для формування показників ризику; [152] – розроблення програмного засобу «OverflowGuard: система аналізу переповнення буфера та оцінки ризику в CI/CD».

У роботах, які опубліковані у співавторстві, співавторам належать такі результати: [128] – визначено стратегію тестування безпеки застосунків в корпоративних мережах (Дрозд А.), визначено концепцію здійснення тестування застосунків в корпоративних системах (Савенко О.), здійснено постановку експерименту щодо вразливостей застосунків (Паюк В.); [155] – здійснено предметний аналіз, узагальнено існуючі підходів SAST та постановка завдання дослідження (Савенко О.), здійснено постановку, підготовку та проведення частини експериментальних процедур (Каштальян А., Ковальчук В.), проведено аналіз типових показників для досягнення ефективності виявлення вразливостей (Іванченко О.); [119] – здійснено постановку завдання дослідження щодо виявлення переповнення буфера, а також підготовку експериментальних даних/вибірок (Savenko O.), налаштування обчислювальних експериментів (Gaj P.); [156] – визначено область дослідження та проведено постановку завдання дослідження, постановку експерименту та підготовку експериментальних даних/вибірок, аналіз результатів (Савенко О.); [120] – здійснено постановку завдання дослідження та участь у порівняльному аналізі альтернативних подань коду (Savenko O.), підготовка експериментальних сценаріїв та узгодження результатів (Gaj P.); [127] – здійснено постановку завдання дослідження та участь

у аналізі результатів (Sachenko A.), здійснення порівняльного аналізу альтернативних подань програмного коду та формування експериментальних сценаріїв (Paiuk V.), інтерпретація результатів експериментів (Kwiecien A., Nicheporuk A.); [121] – здійснено постановку завдання дослідження щодо формальних моделей вразливостей переповнення буфера, розроблення підходів до подання програмного коду графовими моделями (Savenko O. S.), участь у виборі та налаштуванні компонентів нейромережевої архітектури для експериментів (Gaj P., Balej J.); [126] – здійснено вибір методів машинного навчання до задач аналізу програмного коду та систематизація характеристик програмних систем (Paiuk V.), участь у зборі та систематизації даних для завдання дослідження (Nicheporuk A., Huralnyk O.), верифікація результатів дослідження (Kwiecien A.); [152] – здійснено розроблення та постановку вимог до програмного засобу «OverflowGuard: система аналізу переповнення буфера та оцінки ризику в CI/CD» (Савенко О. С.).

Апробація результатів дисертації. Апробацію основних положень, ідей, висновків дисертаційної роботи проведено на науковому семінарі кафедри комп’ютерної інженерії та інформаційних систем у Хмельницькому національному університеті. Наукові результати дисертації доповідалися також на таких наукових заходах: 5th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS-2024), Khmelnytskyi, Ukraine, March 2024; 1st International Workshop on Intelligent & CyberPhysical Systems (ICyberPhyS-2024), Khmelnytskyi, Ukraine, June 28, 2024; 2nd International Workshop on Intelligent & CyberPhysical Systems (ICyberPhyS-2025), Khmelnytskyi, Ukraine, July 4, 2025; The 2nd International Workshop on Advanced Applied Information Technologies: AI & DSS (AdvAIT-2025), Khmelnytskyi, Ukraine, December 5, 2025; 2024 IEEE 14th International Conference on Dependable Systems, Services and Technologies (DeSSerT-2024), Athens, Greece, October 11–13, 2024; International Workshop on Applied Intelligent Security Systems in Law Enforcement (AISSLE-2025), Vinnytsia, Ukraine, October, 30–31, 2025; наукових семінарах кафедри комп’ютерної інженерії та інформаційних систем Хмельницького національного університету.

Публікації. За результатами проведених досліджень основні наукові результати опубліковано у 4 наукових статтях у фахових наукових журналах України [153-156]. Апробація засвідчена публікаціями 6 праць в матеріалах міжнародних та всеукраїнських конференцій [119-121; 126-128], з яких три праці індексовані у наукометричній базі Scopus [126-128] і отримано одне авторське свідоцтво на твір [152].

Структура та обсяг дисертації. Дисертаційна робота складається з анотації, змісту, переліку умовних скорочень, вступу, чотирьох розділів, висновку, списку використаних джерел та чотирьох додатків. Повний обсяг роботи містить 233 сторінки друкованого тексту, з них анотація – на 11 стор., зміст – на 2 стор., перелік умовних скорочень – на 1 стор., основний текст – на 150 стор., список із 157 використаних джерел – на 19 стор., додатки – на 32 стор. Дисертація містить 13 рисунків та 24 таблиці.

РОЗДІЛ 1
АНАЛІЗ МЕТОДІВ ТА ЗАСОБІВ РОЗПІЗНАВАННЯ ВРАЗЛИВОСТЕЙ В ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННІ КОМП'ЮТЕРНИХ СИСТЕМ

Багато наукових задач, які спрямовані на забезпечення безпеки програмних систем в різних галузях промисловості, потребують застосування методів виявлення вразливостей. Використання баз даних вразливостей є важливим компонентом комплексного підходу до забезпечення безпеки програмних систем, який включає різні методи та інструменти для виявлення, аналізу та усунення вразливостей. Наразі актуальним завданням є розробка нових ефективних методів виявлення неприпустимих подій у роботі комп'ютерних систем (КС), які можуть бути наслідками як технічних збоїв, так і несанкціонованих впливів. Одним з ключових напрямків розвитку цієї сфери є створення систем, здатних виявляти та реагувати на різноманітні типи порушень, включаючи ті, що виникають у результаті складних та раніше невідомих атак. При цьому важливо забезпечити можливість виявлення загроз за непрямими ознаками порушення роботи мережі, що дозволить виявляти навіть ті зловмисні впливи, які пролонговані у часі та мають прихований характер.

1.1 Вразливості, класифікація вразливостей, етапи прояву вразливостей

Сучасний комп'ютерний світ є сукупністю різноманітних і дуже складних обчислювальних пристроїв, систем обробки інформації, телекомунікаційних технологій, програмного забезпечення та високоефективних засобів його проектування. Вся ця багатогранна та взаємопов'язана система вирішує велике коло проблем у різних галузях людської діяльності, від простого вирішення шкільних завдань на домашньому персональному комп'ютері до управління складними технологічними процесами.

Чим складніше завдання автоматизації і чим відповідальніша область, у якій використовуються КС, тим більше критичними стають такі властивості, як

надійність і безпека інформаційних ресурсів, що задіяні у процесі збору, накопичення, обробки, передачі та зберігання комп'ютерних даних. Зловмисний вплив на інформацію в процесі функціонування КС різного призначення здійснюється з порушенням її конфіденційності, цілісності та доступності. Вирішення завдань, пов'язаних із запобіганням впливу безпосередньо на інформацію, здійснюється в рамках комплексної проблеми забезпечення безпеки інформації та має досить розвинену науково-методичну базу. При цьому, розглядаючи інформацію як активний експлуатований ресурс, встановлено, що процес забезпечення безпеки інформації включає і забезпечення безпеки програмного забезпечення КС. Цей аспект забезпечення безпеки інформації та засобів її обробки називається експлуатаційною безпекою, оскільки відповідає етапу застосування КС. Останнім часом з'явилися нові проблеми забезпечення безпеки, пов'язані з інформаційними технологіями, які, на думку низки зарубіжних та вітчизняних експертів у галузі їх створення та застосування, значною мірою визначають ефективність комп'ютерних систем [14; 23].

Проблема безпеки програм є однією з першорядних у сфері інформаційної безпеки (ІБ), оскільки наявність вразливостей у програмних ресурсах КС зумовлює можливість реалізації промислових комп'ютерних атак та вірусних епідемій, а також причину різного роду ненавмисних відмов та втрати ресурсів [150]. З урахуванням динамічності та складності програм та розвитку технологій інформаційного протистояння вирішення зазначеної проблеми вимагає безперервного вдосконалення методів контролю, тестування та випробувань, а саме: статичного та динамічного аналізу, функціонального тестування, експертиз бюлетенів безпеки, сканування вразливостей, антивірусного контролю та ін. Синтез та комплексування зазначених підходів мають на увазі систематизацію базових понять безпеки програмного забезпечення (ПЗ) [112].

Проте, незважаючи на те, що подібні роботи регулярно проводяться, у практиці ІБ залишається різне тлумачення поняття вразливості ПЗ, зокрема, найчастіше змішують визначення помилок безпечного програмування та відомих вразливостей мережевих сервісів, недекларованих можливостей та програмних

закладок, прихованих каналів та прихованих криптографічних каналів тощо. Тому виникає не тільки плутанина із засобами виявлення вразливостей та базами вразливостей, а й з цілями та результатами оцінки відповідності.

Система ІБ, яка базується на міжнародних стандартах надзвичайно важлива у сучасному цифровому світі. Їх розробкою займаються Міжнародна організація з стандартизації (ISO) спільно з Міжнародною електротехнічною комісією (IEC). На цей час фахівцями цих організацій розроблена серія стандартів ISO/IEC 27000, що постійно доповнюється новими документами. В Україні діють стандарти серії ДСТУ ISO/IEC 27000. До них відносяться наступні:

1. ДСТУ ISO/IEC 27000:2019 Інформаційні технології. Методи захисту.

2. Системи управління інформаційною безпекою. Огляд і словник термінів (ISO/IEC 27000:2018, IDT) дає змогу переглянути системи управління інформаційною безпекою (СУІБ).

3. ДСТУ ISO/IEC 27001:2015 Інформаційні технології. Методи захисту.

4. Системи управління інформаційною безпекою. Цей стандарт визначає вимоги до проектування, впровадження, підтримки та постійного вдосконалення системи управління інформаційною безпекою з урахуванням обставин організації. Він також містить вимоги для оцінювання та оброблення ризиків інформаційної безпеки, пов’язаних з потребами організації.

В Україні проводяться дослідження щодо розвитку стандартів у галузі вразливостей інформаційних систем взагалі. Безсумнівно це є важливим кроком у розвитку вітчизняної нормативної бази у сфері технічного захисту інформації.

Розглянемо питання класифікації вразливостей програмних систем та пов'язаних із ними понять. Історія виявлення вразливостей у програмному забезпеченні еволюціонувала разом із розвитком КС, підкреслюючи важливість захисту інформації з самого початку. Спочатку методики виявлення вразливостей базувалися переважно на ручному аналізі коду та аудитах безпеки. Такі підходи вимагали великих зусиль та глибокого розуміння внутрішньої логіки програм, але з часом стало зрозуміло, що для ефективного захисту від широкого спектру загроз потрібні більш автоматизовані методи. Сучасні дослідники, такі як Valdés

Rodríguez та інші, розглядають інтеграцію практик безпеки в agile розробку програмного забезпечення, підкреслюючи необхідність адаптації до змінних вимог безпеки та розробки методології, яка дозволяє навіть недосвідченим розробникам створювати більш безпечні програми[140]. Цей підхід вказує на зміну фокусу від традиційних методів до більш гнучких та адаптивних стратегій в рамках швидкого розвитку технологій та програмного забезпечення, що є важливим кроком у забезпеченні надійності та безпеки в сучасному цифровому світі.

У сфері кібербезпеки, виявлення вразливостей в програмному забезпеченні є ключовим аспектом захисту інформаційних систем від зростаючих кіберзагроз. Актуальні дослідження в цій області відкривають нові можливості для вдосконалення методів аналізу та ідентифікації потенційних слабких місць[5]. Вразливість – це слабке місце в захисту системи, отримане помилками чи недосконалістю процедур, реалізації, проєктів, яке загроза може подолати. Практика показує, що вразливості є головною причиною виникнення атак. Загрозою вважають потенційну можливість причинити збиток якимось наперед відомим способом. Загрози ІБ можуть бути здійснені шляхом використання вразливостей системи. Слабкості захисту можуть використовуватися однією або декількома загрозами, що є причиною небажаних інцидентів, які можуть призвести до нестабільного функціонування компонентів інформаційної системи [107; 148].

Більшість сучасних класифікацій загроз і вразливостей допускають їх поділ за етапом життєвого циклу: проектування (архітектура), кодування (реалізація), експлуатація (адміністрування). З погляду практики використання двох основних підходів до контролю безпеки ПЗ – аналізу коду та сканування ресурсів – доцільно відокремити від визначення вразливості поняття дефекту безпеки.

Під дефектом безпеки (weakness, bug) будемо розуміти недолік створення ПЗ, що потенційно впливає на ступінь безпеки інформації. У такому випадку дефект безпеки, що експлуатується, є вразливість (vulnerability), реалізація якої становить загрозу ІБ. Для реалізації вразливості необхідна наявність суб'єкта, що впливає на інформаційну систему та здатного експлуатувати вразливість [110].

У комп'ютерній безпеці, вразливість (*англ. system vulnerability*) – це нездатність системи протистояти реалізації певної загрози або сукупності загроз [157]. Тобто, це певні недоліки в КС, завдяки яким можна навмисно порушити її цілісність і викликати неправильну роботу. Вразливість може виникати в результаті допущених помилок програмування, недоліків, допущених при проектуванні системи, ненадійних паролів, вірусів та інших зловмисних програм, скриптових і SQL-ін'єкцій. Деякі вразливості відомі тільки теоретично, інші ж активно використовуються і мають відомі експлойти [24].

Використовувати вразливість клієнта трохи складніше. Потрібно переконати користувача в необхідності підключення до підробленого сервера, тобто переходу за посиланням у випадку, якщо вразливий клієнт є браузером [69].

Інформацію, що була отримана в результаті виявлення вразливості, можна використати як для написання експлойту, так і для усунення вразливості. Тому в ній однаково зацікавлені обидві сторони – і зловмисник, і виробник програмного забезпечення, яке піддають злому. Характер поширення цієї інформації визначає час, який потрібно розробнику до випуску програмної “латки”. Після закриття вразливості виробником шанс успішного застосування експлойту зменшується. Тому особливу популярність серед зловмисників мають “експлойти нульового дня”, які використовують вразливості, що недавно з'явилися, і які ще не стали відомі громадськості [69].

Наразі можна зустріти ряд як простих, так і складних ієрархічних класифікацій у сфері ІБ, які можна поділити на такі:

- класифікації загроз та атак;
- класифікації зловмисних програм;
- класифікації та реєстри вразливостей;
- класифікації дефектів.

Розглянемо їх детальніше. Класифікації загроз та атак є методично опрацьованими і систематизують різні види штучних і природних, випадкових і зловмисних, внутрішніх і зовнішніх загроз за безліччю різних параметрів [79]. Як правило, ці класифікації виділяють клас загроз, пов'язаний із можливістю

реалізації порушником програмних вразливостей, проте класи вразливостей описуються лише у загальному плані. При цьому ці класифікації є основою для побудови моделей загроз безпеці інформації.

Щодо класифікації зловмисних програм, то розробники засобів антивірусного захисту дотримуються класифікацій зловмисного ПЗ (malware) за моделлю поширення, за способом активації та за дією та іншими параметрами, що дозволяє розробляти ефективні тести та бази сигнатур антивірусів. Ці класифікації корисні при описі підкласу вразливостей експлуатаційного типу, безпосередньо не торкаючись вразливостей етапу проектування та кодування[51].

Історично реєстри вразливостей були зумовлені потребою в регулярному розповсюдженні бюлетенів та зведень про знайдені вразливості для кожного типу та версії програмних продуктів та середовищ. Такі реєстри підтримуються як великими розробниками ПЗ (наприклад, Adobe, Microsoft, RedHat), і різними асоціаціями (US-CERT, Secunia, Open Security Foundation). Останні створили ряд реєстрів, що входять до єдиної системи ідентифікаторів (наприклад, CVE-ID) вразливості ПЗ різних розробників.

Проаналізуємо вид класифікацій дефектів. Він систематизує дефекти безпеки ПЗ при дослідженні початкового коду ПЗ. На відміну від описаних вразливостей, дефекти – це внутрішня властивість кожної реалізації ПЗ або системи.

Більшість дефектів виникає у процесі створення ПЗ. Це можуть бути помилки проектування, помилки кодування програмістів, помилки, допущені при складанні дистрибутива та інтеграції різних версій компонентів ПЗ. [95].

Деякі класифікації включають поняття дефектів інформаційної системи, що пов'язані зі зміною системи та викликані або помилками адміністраторів (наприклад, неправильними налаштуваннями схеми автентифікації, несвоєчасним встановленням оновлень операційної системи або мережевих сервісів), або помилками операторів інформаційних систем (наприклад, слабкими паролями в обліковому записі, некоректним вимкненням комп'ютера).

У табл. 1. представлені поширені класифікації у сфері безпеки ПЗ.

Таблиця 1.1 – Класифікації у сфері безпеки ПЗ

Класифікації ЗПЗ	
1	2
Приклади	Особливості
Mitre MAEC (Malware Attribute Enumeration and Characterization) – перелік та характеристики ознак зловмисного ПЗ	Мова для опису зловмисного ПЗ, що враховує ознаки поведінки, тип атаки тощо.
Symantec Classification – класифікація фірми Symantec	Класифікація виявленого зловмисного ПЗ
Реєстри та класифікації вразливостей програмних систем	
Приклади	Особливості
MITRE CVE (Common Vulnerabilities and Exposures) – загальні вразливості та “незахищеності”	База даних відомих вразливостей
NVD (National Vulnerability Database) – національна база вразливостей США	База вразливостей, що використовує ідентифікатори CVE
OSVDB (Open Security Vulnerability Database) – база вразливостей відкритого доступу	База даних відомих вразливостей
US-CERT Vulnerability Notes Database – база вразливостей	Опис знайдених вразливостей та способів їх виявлення
Бюлетені розробників: 1. Microsoft Bulletin ID. 2. Secunia ID. 3. VUPEN ID.	Зведення знайдених вразливостей
Таксономія Бішопа та Бейлі	Застаріла класифікація вразливостей Unix-систем
Класифікації загроз безпеки та комп'ютерних атак на ресурси системи	
OWASP Top Ten – 10 найпоширеніших загроз для веб-додатків	Десять найактуальніших класів загроз, пов'язаних з уразливістю web-додатків за останній рік
MITRE CAPEC (Common Attack Pattern Enumeration and Classification) – перелік та класифікація поширених типів атак	Всебічна класифікація типів атак
Microsoft STRIDE Threat Model – модель загроз Microsoft	Опис п'яти основних категорій вразливостей
WASC Threat Classification 2.0 – класифікація загроз Консорціуму безпеки web-застосунків	Класифікація вад, загроз web-безпеки, націлена на практичне застосування

Кінець таблиці 1.1

1	2
Класифікації дефектів, внесених у процесі розробки	
MITRE CWE (Common Weaknesses Enumeration) – загальна класифікація дефектів ПЗ	Система класифікації “вад” ПЗ
Fortify Seven Pernicious Kingdoms – 7 руйнівних "царств" компанії HP Fortify	Класифікація дефектів ПЗ на 8 основних видів
CWE/SANS Top 25 Most Dangerous Software Errors – 25 найбільш небезпечних помилок у розробці ПЗ	25 найбільш поширених та небезпечних помилок, які можуть стати причиною вразливості
OWASP CLASP (OWASP Comprehensive, Lightweight, Application Security Process) – опис процесу безпечної розробки додатків	Принципи безпеки організації процесу розробки застосунків
DoD Software Fault Patterns – зразки програмних помилок Міноборони США	Система типів дефектів ПЗ, асоційована з CWE та розроблена з метою автоматизації їх виявлення
Застарілі класифікації: 1. Переліки RISOS/PA. 2. Таксономія Ландвера. 3. Таксономія Асламу. 4. Таксономія Макгоу. 5. Таксономія Вебера 6. Список PLOVER	Перші проекти з часткової каталогізації відомих дефектів безпеки та їх класифікації
MITRE Common Configuration Enumeration (CCE) – загальний реєстр конфігурацій	Ідентифікація проблемних конфігурацій системи, що встановлює відповідність між різними джерелами
Класифікації дефектів, внесених у процесі впровадження та експлуатації	
DPE (Security-Database Default Password Enumeration) - реєстр паролів за промовчанням	База даних паролів за промовчанням для мережевих пристроїв, ПЗ та ОС, призначена для тестування з метою виявлення слабких конфігурацій

З табл. 1.1 подано класифікації в сфері ІБ, але в основному вони орієнтовані на конкретні завдання, будь-то мережеві атаки, вразливість операційних систем або некоректність програмування. Серед розглянутих класифікацій найкращим, з точки зору розробника, є реєстр CWE (за показниками повноти, всебічності класифікації, наявності докладних описів з прикладами коду), а з погляду

адміністратора найефективнішою є класифікація CVE (за показниками обсягу записів, оперативності поновлення). З іншої сторони, класифікація CWE є неоднозначною і досить складною і до того ж повною мірою вирішує питання безпеки конфігурацій.

Також варто зазначити про фундаментальний ресурс для забезпечення безпеки ПЗКС, а саме : бази даних вразливостей. Вони систематизують інформацію про відомі вразливості, умови їх виникнення та можливі методи експлуатації. Поширені бази: CVE; NVD; OWASP. Кожна з них має свої особливості, що робить їх невід'ємною частиною сучасної екосистеми кібербезпеки [76].

CVE є глобальним стандартом для ідентифікації вразливостей і загроз, що забезпечує структурований підхід до опису й аналізу загроз ІБ. Кожна вразливість у базі отримує унікальний ідентифікатор, наприклад, CVE-2023-12345, який гарантує її однозначне розпізнавання серед фахівців і автоматизованих систем. Цей стандарт містить базову інформацію про вразливість, включаючи її тип, можливий вплив, сценарії виникнення та основні аспекти експлуатації. Інтеграція CVE з іншими базами, такими як NVD, дозволяє створювати взаємопов'язану інфраструктуру аналізу вразливостей, що значно спрощує обмін інформацією між різними інструментами та службами безпеки. Окрім цього, CVE використовується для забезпечення відповідності міжнародним стандартам, таким як ISO/IEC 29147 та ISO/IEC 30111, що підвищує його значимість у глобальній системі безпеки[16].

База NVD розширює функціональність CVE, додаючи більш детальну та структуровану інформацію. Зокрема, NVD пропонує рейтинги серйозності (CVSS), які дозволяють оцінити критичність кожної вразливості, враховуючи її технічні характеристики та потенційний вплив. У цій базі також зберігається інформація про підтримувані операційні системи, платформи та фреймворки, що надає повне уявлення про середовище виконання. NVD забезпечує доступ до даних у форматах JSON та XML, що сприяє інтеграції цих даних в автоматизовані системи аналізу, дозволяючи значно пришвидшити процеси ідентифікації та оцінки вразливостей. Завдяки рекомендаціям і прикладам коду, які надаються в базі, фахівці з безпеки можуть ефективно усувати вразливості, використовуючи найкращі практики.

Окрім цього, NVD активно співпрацює з великими виробниками програмного забезпечення КС, забезпечуючи інтеграцію своїх даних у процеси розробки та тестування [101].

OWASP фокусується на забезпеченні безпеки веб-застосунків, надаючи розробникам і адміністраторам широкий спектр інструментів та рекомендацій. Одним із ключових елементів є "OWASP Top Ten" – перелік найпоширеніших вразливостей, таких як SQL-ін’єкції, XSS та переповнення буфера. Цей перелік постійно оновлюється на основі аналізу глобальних загроз, що робить його актуальним і практичним інструментом для розробників. OWASP також надає інструменти для автоматизації аналізу безпеки, серед яких особливо виділяється ZAP (Zed Attack Proxy). Цей інструмент дозволяє імітувати атаки на веб-застосунки, виявляючи потенційні вразливості ще до їх експлуатації. Окрім ZAP, OWASP пропонує інші фреймворки, наприклад, ASVS (Application Security Verification Standard), що надає структурований підхід до оцінки безпеки веб-застосунків. Ресурси OWASP включають також інструкції з безпечного кодування, довідники з управління ризиками та інструменти для впровадження політик безпеки, що сприяє створенню надійного програмного забезпечення комп'ютерних систем [64].

Щодо етапів виявлення вразливостей, то у великих організаціях, які працюють із цінною інформацією, часто виникає необхідність контролю безпеки. Робота сканера вразливості полягає у перевірці застосунків, виявленні слабких місць, якими можуть скористатися сторонні для доступу до секретної інформації. Програма дозволяє виявити зловмисний код, перевірити ПЗКС та ресурси системи, створити звіт [95]. Отже, можна виокремити такі етапи сканування, як:

1) збір інформації, тобто ідентифікацію активних пристроїв, виявлення запущених сервісів, портів, програм, версій, ip-адрес;

2) виявлення потенційних вразливостей, тобто сканер перевіряє базу даних на наявність відомих варіантів з урахуванням ступеня ризику;

3) аналіз сканування, підтвердження факту фактора ризику;

4) формування звіту, що містить рекомендації щодо забезпечення інформаційної безпеки.

Таким чином, вразливості ПЗКС становлять актуальну проблему при експлуатації КС. Їх класифікації є різними та підтверджують потребу в постійному розвитку процесів з їх виявлення. Етапи прояву вразливостей є різними і це ускладнює можливості до створення універсальних підходів до їх виявлення.

1.2 Методи виявлення вразливостей

Сучасне ПЗКС для виявлення вразливостей класифікується за групами відповідно до реалізованих у ньому конкретних методів пошуку вразливостей. Дослідженню виявлення вразливостей в КС для виявлення зловмисного програмного забезпечення (ЗПЗ) і КА приділяють увагу багато дослідників. Науковці розробили багато різних методів виявлення [24; 63; 36]. Проте, важливим є не лише реалізація одного чи групи методів в спеціалізованих інструментах, а й розробка нових методів, які дозволять ефективніше знаходити та усувати вразливості. У цьому контексті значну увагу слід приділити дослідженню існуючих аналітичних підходів та технологій, що використовуються для виявлення вразливостей, а також розробці інноваційних методів для забезпечення надійності та безпеки програмних систем.

Результати статистики кількості вразливостей зображені на рис. 1.1 демонструють стрімку динаміку збільшення їх кількості щорічно. Це суттєво актуалізує проблему створення спеціалізованих ПЗ з врахуванням їх можливих функціонувань в умовах впливу ЗПЗ.

Отже, проаналізуємо відомі методи виявлення вразливостей у ПЗ.

Метод статичного аналізу безпеки (SAST) базується на детальному перегляді початкового коду, бінарних файлів та бібліотек без їх виконання для виявлення потенційних вразливостей. Цей метод дозволяє ідентифікувати такі проблеми, як SQL-ін'єкції, XSS та інші типи вразливостей ще на етапі розробки [65]. SAST використовує кореляційний аналіз, метрики вразливостей та моделювання загроз

для визначення потенційних шляхів атак. Це підвищує ефективність виявлення вразливостей шляхом створення карт вразливостей та дерев атак на основі інформації з CWE (Common Weakness Enumeration) та CVE (Common Vulnerabilities and Exposures).

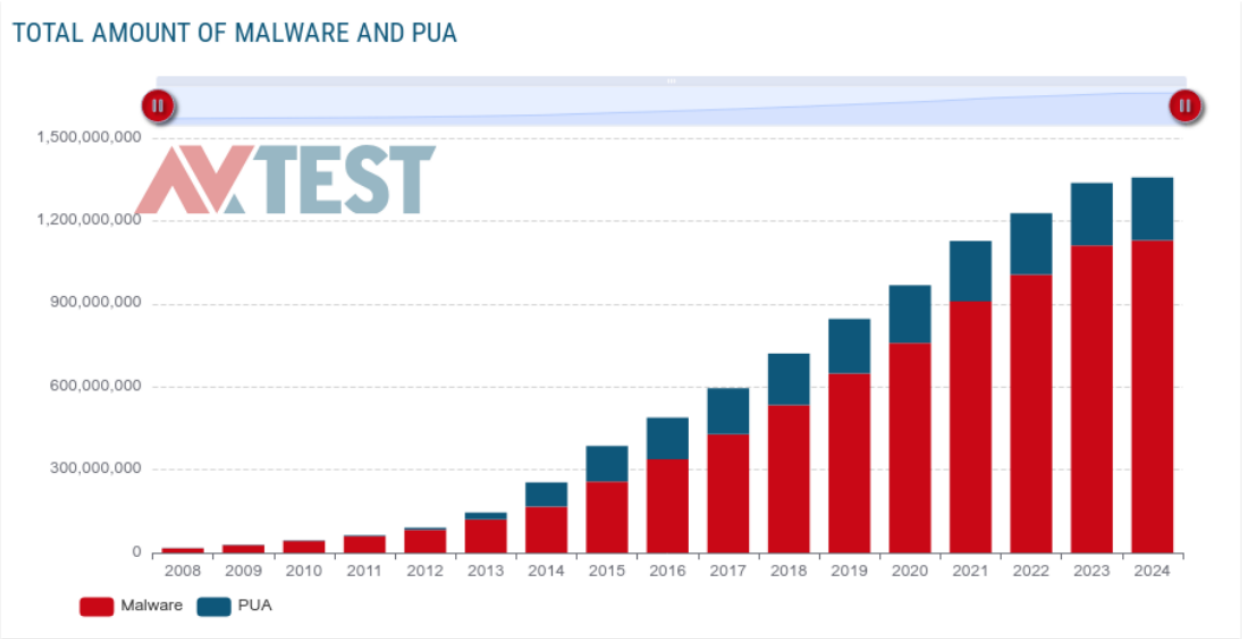


Рисунок 1.1 – Результати статистики кількості вразливостей з ресурсу https://www.av-test.org/en/statistics/malware/

Однак, метод SAST має свої недоліки. Основні проблеми включають високу залежність від технічних ресурсів та потребу в постійному оновленні інструментів та методик, щоб відповідати новітнім викликам у сфері кібербезпеки. Крім того, SAST не може виявити вразливості, які проявляються лише під час виконання програми, що обмежує його ефективність у виявленні деяких типів атак. [147].

Відомо, що невиявлені вразливості можуть призвести до значних витрат для IT компаній. Дослідження Myriam Dunn Cavelty показують, що швидке виправлення є важливим для уникнення втрат, пов'язаних з вразливістю та її публічним розголошенням [22]. Вона вказує на значні витрати, пов'язані з недостатньою інфраструктурою тестування програмного забезпечення; тому необхідний інструмент для виявлення вразливостей.

Інструменти статичного аналізу (SAST) є практичним вибором для активного виявлення вразливостей у програмному забезпеченні під час розробки.

Національна база даних вразливостей (NVD) збирає загальні вразливості та експлойти (CVE), класифіковані через загальні слабкі місця (CWE). NVD, також, класифікує ступінь небезпеки вразливостей, пов'язаних з конкретними CVE. SAST-інструменти розроблені для виявлення вразливостей, пов'язаних з CWE. Хоча багато SAST-інструментів можуть виявляти однакові вразливості, лише деякі з них можуть виявляти більш специфічні та складні CWE. NIST надає тестові набори SARD, які орієнтовані на найпоширеніші мови програмування для оцінки SAST-інструментів.

Серед досліджень, що здійснюються, можна виділити дві підгрупи: дослідження, що фокусуються на зручності використання; дослідження, що фокусуються на адаптації та інтеграції інструментів у процеси організацій (підприємств, установ тощо).

Метод статичного тестування безпеки застосунків (SAST) був представлений на початку 2000-х років. Останніми роками його популярність значно зросла, оскільки багато комерційних (Checkmarx, 2021; Microfocus, 2021; Veracode, 2021; SonarSource, 2021; Github, 2021; Snyk, 2021) і з відкритим кодом (of Maryland U, 2021; Facebook, 2021; Krüger et al., 2018) інструментів стали доступними. Таким чином, дослідження цього методу має велике промислове значення. Зокрема, дослідники вивчали можливість використання методу статичного аналізу. Nguyen Quang Do та інші провели опитування серед розробників з німецької компанії Software AG і проаналізували дані про використання інструментів для двох проектів компанії [99]. Вони виявили потреби та мотивації розробників при використанні інструментів статичного аналізу. На основі цього вони надали рекомендації щодо нових функцій та дослідницьких ідей для майбутнього розгляду. Подібне дослідження було проведено раніше в Microsoft Christakis та Bird. Вони провели опитування та інтерв'ю з розробниками, а також проаналізували дані про інциденти на живих сайтах, що є високо критичними помилками, які обробляють чергові інженери. Вони виявили проблеми зручності використання та функціональні можливості, які повинні надавати інструменти програмного аналізу для кращої зручності використання [99].

Vassallo та інші досліджували різні контексти, в яких розробники використовують статичні інструменти, такі як середовище розробки, огляд та безперервна інтеграція. Вони також досліджували налаштування для цих контекстів і з'ясували, що більшість розробників використовують однакові налаштування у всіх середовищах (IDE, безперервна інтеграція чи огляд). У порівнянні з цим, наше дослідження спрямоване лише на IDE. Вони провели опитування серед 42 учасників. Для підтвердження своїх висновків вони провели інтерв'ю з одинадцятьма розробниками з шести компаній [104: 141; 21].

Розглянемо дослідження, які зосереджуються на виявленні вразливостей безпеки за допомогою інструментів статичного аналізу. Smith та інші запросили 10 розробників з одного проекту виконати чотири завдання з використанням розширеної версії FindBugs. Учасників попросили усно пояснити свої думки, які автори використали для формулювання запитань. Потім вони використовували метод сортування карток для отримання відповідних висновків. Порівняно з цими дослідженнями, наша робота є єдиною, яка фокусується на конкретних параметрах налаштування [98].

Smith та інші оцінювали зручність користування інтерфейсами чотирьох SAST-інструментів. Вони використовували евристичні методи та експеримент, після якого проводили інтерв'ю з дванадцятьма учасниками. У дослідженні з декількома етапами, Patnaik та інші проаналізували 2491 запитання на Stack Overflow, щоб виявити проблеми зручності, з якими стикаються розробники при використанні криптографічних бібліотек. Порівняно з усіма цими дослідженнями, встановлено, що робота не орієнтована на аспект зручності використання інструменту [97; 98].

Для забезпечення ефективного виявлення вразливостей використання методу SAST повинно бути доповнене іншими методами, такими як динамічний аналіз безпеки (DAST), аналіз складу програмного забезпечення (SCA), тестування проникнення та захист у реальному часі (RASP). Взаємодія цих методів дозволяє не тільки технічно аналізувати вразливості, але й управляти ними на кількох рівнях

архітектури застосунків, що значно підвищує загальну безпеку програмного забезпечення.

Для ПЗКС важливим є аналіз, який не потребує виконання коду (SAST і ML-моделі поверх коду/графів). Це пов’язано з тим, що динамічні тестування часто підпадають під вплив апаратних компонентів, конфігурацій ОС чи RTOS та збільшують ризики дестабілізації середовища. Тому в цій роботі основну увагу приділено виявленню фрагментів коду, критичних для пам'яті, а також формальним умовам переповнення.

Розглянемо метод Fuzzing, що заснований на створенні великої кількості тестових випадків для виявлення вразливостей у ПЗ шляхом аналізу результатів його роботи після введення цих тестів. На відміну від традиційних методів контролю цілісності або сигнатурного аналізу, метод Fuzzing дозволяє виявляти невідомі раніше вразливості, для яких ще не існують специфічні сигнатури [137]. Дослідження науковців Huang Y., Wang Z., Ou H., Chi Y. продемонструвало високу ефективність фаззингу (Fuzzing) в контексті мобільного офісного ПЗ на платформі Android, використовуючи багатоаспектні підходи для генерації тестових випадків та аналізу результатів [56]. Fuzzing – є методом виявлення безпеки, який вводить недійсні або випадкові дані в програму та виводить поведінку, яка не очікується, тим самим ідентифікуючи помилки в програмі та потенційні вразливості. Основна ідея fuzzing полягає у генерації даних, де проводяться випробування для збою джерельної програми, а також у виборі правильних інструментів для моніторингу процесу. Fuzzing може генерувати велику кількість хибнопозитивних результатів, оскільки не завжди можливо точно визначити, чи є виявлена поведінка програми дійсною вразливістю. Цей метод вимагає значних ресурсів для генерації та обробки тестових випадків, особливо при використанні масштабних тестів з великою кількістю вхідних даних, а також може не виявити вразливості, які активуються лише під дуже специфічними умовами або потребують складної взаємодії з користувачем [7]. Методи фаззингу продовжують еволюціонувати, включаючи розробку інтелектуальних систем, здатних адаптувати стратегії генерації даних для підвищення ефективності виявлення. Запропонована система пошуку вразливостей

на основі фаззингу демонструє свою життєздатність і може надати підтримку розробникам для покращення безпеки програмного забезпечення.

В процесі Fuzzing використовуються різноманітні техніки генерації тестових випадків, включаючи методи на основі мутацій, генерації та Char-RNN, що забезпечує широкий спектр покриття тестів та виявлення вразливостей з різних сторін [56].

Основні переваги методу Fuzzing включають здатність виявляти нові та непередбачені вразливості, особливо в складних програмних системах, таких як мобільні офісні застосунки на платформі Android. Метод дозволяє автоматизовано створювати тестові випадки, виконувати їх та аналізувати результати, що включають як звичайний вихід програми, так і створені винятки. Це значно підвищує ефективність виявлення вразливостей та дозволяє розробникам швидко реагувати на знайдені проблеми. [56].

Недоліки методу Fuzzing полягають у тому, що він потребує значних обчислювальних ресурсів та часу для створення і аналізу великої кількості тестових випадків. Крім того, ефективність Fuzzing може бути обмежена типом застосунку, оскільки він найкраще працює для виявлення вразливостей пам'яті та подібних помилок у програмному забезпеченні. Наприклад, Fuzzing може не виявити логічні помилки або вразливості, які не призводять до негайних винятків чи помилок [56].

Таким чином, метод Fuzzing є потужним інструментом для виявлення вразливостей у ПЗ, особливо в умовах, коли традиційні методи можуть бути неефективними. Його застосування дозволяє підвищити загальну безпеку програмних систем та забезпечити їх стійкість до нових типів загроз.

Дослідники вважають, що одним з найбільш ефективних методів виявлення вразливостей є використання баз даних вразливостей. Національна база даних про вразливості (NVD) є загальнодоступним джерелом даних, яке зберігає стандартизовану інформацію про зареєстровані вразливості програмного забезпечення. З моменту заснування у 1997 році NVD опублікувала інформацію про понад 43 000 вразливостей програмного забезпечення, які впливають на понад

17 000 програмних застосунків. Ця інформація є потенційно цінною для розуміння тенденцій і закономірностей у вразливостях ПЗ, що дозволяє краще керувати безпекою КС [76].

Використання NVD як важливого джерела інформації про вразливості ПЗ дозволяє розробникам та аналітикам безпеки ідентифікувати відомі вразливості та передбачати потенційні загрози. Зокрема, інформація з NVD допомагає в аналізі тенденцій, що дозволяє створювати прогнозні моделі для оцінки ймовірності появи нових вразливостей у конкретних програмних застосунках. Це може включати використання методів інтелектуального аналізу даних та алгоритмів машинного навчання для побудови моделей, які передбачають час до появи наступної вразливості для певного ПЗ.

Однак, існують певні обмеження у використанні даних NVD для прогнозування. Незважаючи на великий обсяг даних, вони мають обмежену здатність до точного прогнозування нових вразливостей, за винятком кількох постачальників та програмних застосунків. Проведені дослідження показали, що дані NVD загалом мають низьку прогностичну силу, що може бути пов'язано з обмеженнями у якості та повноті наявної інформації. Попри це, використання баз даних вразливостей залишається важливим інструментом для ідентифікації та оцінки ризиків, пов'язаних з безпекою ПЗ, а також для розробки стратегій захисту від відомих загроз.

Таким чином, використання баз даних вразливостей є важливим компонентом комплексного підходу до забезпечення безпеки ПЗКС, який включає різні методи та інструменти для виявлення, аналізу та усунення вразливостей.

1.3 Засоби виявлення вразливостей програмного забезпечення комп’ютерних систем

Збільшення використання електронних пристроїв призвело до збільшення вразливостей в операційних системах і програмах. Оскільки все більше і більше пристроїв підключаються до Інтернету, то вони стають потенційними цілями для

кіберзлочинців, які прагнуть використати вразливості в цих системах. Ці вразливості можуть мати різні форми, наприклад невиправлене ПЗКС, слабкі паролі або погано налаштовані параметри [113] .

За словами Фернандеса, ПЗ, яке використовується для промисловості, здебільшого містить помилки, недоліки та вразливості, якими, природно, можуть скористатися зловмисники [43]. Усунення вразливостей і недоліків не є простим і повним вирішенням проблеми. Кількість вразливостей зростає, а захист КС небезпечним ПЗ не є ефективним способом боротьби зі зловмисниками [145]. Дослідники Шах і Мехтре пояснили різні методи тестування на проникнення та оцінки вразливості [122].

Використання актуальних версій ПЗ захищає його від ряду атак. Панджвані та інші досліджували кореляцію між скануванням портів і атаками [106]. Вони зазначають, що лише сканування портів не є показником майбутньої атаки, однак комбінації вразливості та сканування портів вважаються передвісниками майбутньої атаки. На нашу думку, щоб гарантувати безпеку КС, слід регулярно проводити тестування, що показує слабкі місця за допомогою деяких існуючих засобів виявлення вразливостей. Крім того, регулярне проведення тестів на проникнення підвищує обізнаність і знижує ризик атак.

Сканування вразливостей – це структурований і цілеспрямований процес, метою якого є пошук потенційних слабких місць у середовищі однієї організації, зануреного в безпеку Інтернету Yuji Rose. Цей процес передбачає використання спеціальних засобів і методів для виявлення вразливостей, що можуть використовувати кіберзагрози. Ця процедура є необхідною частиною ширшого плану управління вразливістю. Такі програми призначені для запобігання впливу кібервторгнень на організацію та зниження ризиків від витоку даних.

Виявлено, що сканування вразливостей працює як перевірка стану безпеки цифрових систем. Воно систематично оцінює та аналізує інфраструктуру безпеки, щоб розпізнати будь-які вразливості, які можуть виявити організації до потенційних ризиків. Виявивши ці слабкі сторони, організації можуть вжити заходів і своєчасно їх усунути та виправити, зрештою покращуючи загальний стан

кібербезпеки та здатність протистояти потенційним кіберзагрозам. Важливою метою є посилення захисту організації, зниження ймовірності успішних кібератак і мінімізація впливу будь-яких потенційних загроз безпеки.

Таким чином, сканер вразливостей – це інструмент, розроблений для автоматичного пошуку відкритих вразливостей мережі та повідомлення про них. Сканер вразливостей автоматично перевіряє мережу або систему, щоб визначити потенційні слабкі місця безпеки, що дозволяє організації отримати повну картину стану своєї безпеки. Ці засоби спрощують процеси аналізу та звітування, дозволяючи командам безпеки вживати необхідних заходів для ефективного виправлення та підвищення безпеки програмного забезпечення [94].

Дослідник Кох та інші запропонували техніку доступу до віддалених хостів за доменним іменем, якщо зловмисники не зможуть отримати доступ до серверів за IP-адресою автоматизованих мережевих сканерів [70]. Проте, запропонована методика не вирішує проблему повністю, а лише вимагає більше часу для сканування жертви атак. Засоби сканування мережі корисні для людей, які проводять тести на проникнення, однак ці інструменти також використовуються зловмисниками для виявлення вразливих хостів у мережі. Знайшовши вразливі хости та способи використання вразливостей, потрібно небагато часу, щоб почати руйнівні кібератаки.

Розглянемо засіб, що використовується для виявлення та оцінки вразливостей у системі чи мережі - VAPT (Vulnerability Assessment and Penetration Testing). VAPT – це засіб пошуку, що використовується для виявлення та усунення потенційних ризиків безпеці, перш ніж ними зможуть скористатися зловмисники. Оцінка вразливості включає використання спеціалізованих інструментів і методів для сканування системи або мережі на відомі вразливості, такі як невиправлене ПЗКС, слабкі паролі та неправильно налаштовані параметри. Після виявлення цих вразливостей проводиться тест на проникнення, щоб імітувати атаку на систему чи мережу та оцінити ефективність існуючих засобів контролю безпеки. Тест на проникнення – це імітована кібератака на КС, мережу або веб-програму для оцінки їх безпеки. Існує кілька різних категорій тестування на проникнення, які

розрізняються залежно від обсягу, цілей і методів тестування. Методи, які використовуються в пентесті, ідентичні тим, які використовуються під час реальної хакерської атаки [15]. Отже, основна відмінність між оцінкою вразливості та тестуванням на проникнення полягає в фокусі кожного підходу. Оцінка вразливості зосереджена на пошуку слабких місць у системі чи мережі, тоді як тестування на проникнення зосереджено на перевірці міцності засобів захисту, які існують для захисту від цих слабких місць.

Ще однією ключовою відмінністю між оцінкою вразливості та тестуванням на проникнення є рівень деталізації та комплексності кожного підходу. Оцінка вразливості зазвичай включає автоматичне сканування, яке здатне визначити широкий спектр потенційних вразливостей, але воно може виявити не всі можливі слабкі місця в системі чи мережі. Тестування на проникнення, з іншого боку, як правило, є більш глибоким і ретельним процесом, який передбачає імітацію різних типів атак і оцінку того, наскільки система або мережа здатні протистояти цим атакам [130].

Дослідник Khraisat A. акцентує на складнощах точного виявлення вторгнень у контексті постійного розвитку методів кібератак [140]. Він підкреслює необхідність адаптації систем виявлення вторгнень до змінюваних загроз, що вимагає інтеграції передових технологій, зокрема машинного навчання, для покращення точності та швидкості реакції на кібератаки. Машинне навчання включає в себе процес видобування знань з великих масивів даних. Він використовує набір правил, методів або складних "перетворювальних функцій", які можуть застосовуватися для виявлення шаблонів даних або для розпізнавання або прогнозування поведінки. Такі техніки, як кластеризація, нейронні мережі, асоціативні правила, дерева рішень, генетичні алгоритми та методи найближчих сусідів, застосовувались для аналізу даних про вторгнення. Методи машинного навчання можуть створювати високий рівень хибних спрацьовувань або мати низьку точність через проблеми з оновленням інформації про нові атаки. Це стає особливо критичним при спробі виявлення атак "нульового дня", які не мають попередньо відомих сигнатур або шаблонів поведінки. Деякі підходи, особливо

ансамблеві методи, можуть вимагати значних обчислювальних ресурсів та часу для навчання на великих наборах даних, що ускладнює їх використання в реальному часі [7]. Машинне навчання дозволяє аналізувати великі обсяги даних та виявляти складні залежності між характеристиками програм, що може сприяти більш ефективному виявленню вразливостей [89].

Однак, як показують дослідження, машинне навчання не вирішує всі завдання і вимагає подальших удосконалень, зокрема у плані адаптації до нових видів загроз та зменшення кількості помилкових спрацьовувань [124]. Незважаючи на значний потенціал, застосування машинного навчання у кібербезпеці супроводжується певними викликами. Одним з основних є забезпечення якості та актуальності даних для навчання моделей. Некоректно підібрані або застарілі дані можуть призвести до помилок у виявленні, підвищуючи ризик пропуску реальних загроз або генерації хибних сповіщень. Додатково, складність та "чорна скринька" деяких алгоритмів машинного навчання ускладнюють інтерпретацію результатів та їх інтеграцію в існуючі системи безпеки [7; 9].

Застосування нейронних мереж дає змогу автоматизувати процеси аналізу, адаптуватися до нових викликів та ідентифікувати невідомі раніше патерни, що розширює можливості традиційних методів, таких як статичний і динамічний аналіз, які часто обмежені застарілими правилами та шаблонами. Однією з головних переваг нейронних мереж є їх здатність працювати з великими даними, автоматично аналізуючи складні взаємозв’язки у коді.

Крім того, нейронні мережі дозволяють інтегрувати різні джерела інформації, такі як структура коду, середовище виконання, метадані та версії бібліотек. Це сприяє комплексному аналізу ризиків і дає можливість оцінювати не лише технічні аспекти вразливостей, але й потенційний вплив на бізнес[132].

Розглянемо детальніше типи нейронних мереж. Для аналізу ПЗКС та виявлення вразливостей можна використовувати різноманітні типи нейронних мереж, кожен із яких має унікальні особливості й переваги. Вони дозволяють вирішувати специфічні задачі, пов’язані з аналізом коду, враховуючи його структуру, контекст виконання та взаємодію між компонентами.

Рекурентні нейронні мережі (RNN) особливо ефективні для роботи з послідовними даними, такими як початковий код, що має ієрархічну структуру або залежності між функціями. Вони забезпечують збереження контексту попередніх ітерацій, що критично важливо для аналізу довгих послідовностей і складних програмних залежностей. Наприклад, RNN дозволяють аналізувати, як зміна значення змінної у певній функції може вплинути на поведінку всієї програми. Хоча RNN мають обмеження, пов’язані з проблемою затухання градієнтів, сучасні їх варіації, такі як LSTM (Long Short-Term Memory) або GRU (Gated Recurrent Unit), частково вирішують цю проблему, що дозволяє працювати з довгостроковими залежностями [77].

Згорткові нейронні мережі (CNN) здебільшого використовуються для аналізу даних із просторовою структурою, таких як зображення, але їх також можна адаптувати для роботи з системним програмним кодом операційної системи. У цьому випадку код представляється у вигляді графів або матриць залежності, що дозволяє CNN виявляти локальні патерни, наприклад специфічні комбінації викликів функцій чи аномальні послідовності дій. CNN мають здатність автоматично виділяти ознаки, що робить їх особливо корисними для виявлення складних шаблонів. Наприклад, при аналізі динамічно виділеної пам’яті вони можуть ідентифікувати підозрілі виклики функцій, які можуть призвести до переповнення буфера чи використання звільненої пам’яті [19].

Зауважимо, що трансформери, такі як моделі BERT або GPT, представляють найсучасніший підхід до аналізу текстових даних. Використання механізму уваги (attention) дозволяє трансформерам ефективно аналізувати взаємозв’язки у коді, враховуючи залежності між його частинами навіть на великій відстані. Це робить трансформери ідеальними для виявлення вразливостей, які залежать від взаємодії між різними компонентами програмного забезпечення комп'ютерних систем. Наприклад, трансформер може враховувати, як зміна змінної у функції впливає на виклики, які знаходяться у зовсім іншій частині програми операційної системи. Отже, завдяки своїй здатності працювати з довгими контекстами трансформери ідеально підходять для аналізу великих кодових баз із мільйонами рядків коду [50].

Автоматизація процесу виявлення вразливостей отримала розвиток з появою інструментів статичного аналізу коду, які дозволяли ідентифікувати потенційні проблеми без необхідності виконання програм. Ці інструменти аналізують код на предмет відомих шаблонів помилок та небезпечних практик програмування, пропонуючи розробникам рекомендації щодо їх усунення [9].

Статичний аналіз використовується для виявлення вразливостей у програмах без необхідності їх виконання. Цей метод дозволяє ідентифікувати слабкі місця у програмному коді до його фактичного використання у середовищі користувача. Статичний аналіз може не виявити всі вразливості, особливо ті, які вимагають виконання коду для активації, такі як деякі типи вразливостей часу виконання. В них велика кількість хибнопозитивних результатів може стати проблемою, оскільки аналізатори можуть інтерпретувати коректний код як потенційно вразливий, а також статичні аналізатори вимагають глибокого розуміння початкового коду програми та її залежності, що може ускладнити їх використання неспеціалістами [7]. Паралельно розвивається динамічний аналіз, який, на відміну від статичного, вимагає виконання програми в контрольованому середовищі для виявлення помилок, що проявляються лише під час її роботи [5].

Отже, серед основних причин, через які КС може бути зламана, є наявність вразливостей у системі, використання незахищених мереж і нездатність відстежувати та виявляти підозрілу активність у системі. Для запобігання таким причинам злому КС треба вчасно використовувати засоби виявлення вразливостей.

1.4 Перспективні стратегії до виявлення вразливостей програмного забезпечення комп’ютерних систем

Комплексний захист комп'ютерних систем сучасного рівня потребує використання різних засобів безпеки, таких як системи виявлення мережевих атак, системи захисту від спаму, антивіруси, міжмережеві екрани (firewall), сканери безпеки тощо. При цьому зростання кількості апаратних та програмних засобів захисту систем значно збільшує обсяг аналізованої інформації, необхідний

контролю безпеки. Як наслідок, адміністратори мережі повинні приділяти значний час аналізу рутинної інформації, що знижує продуктивність роботи та, відповідно, впливає на оперативне прийняття рішень щодо підтримки функціонування комп'ютерної мережі. Таким чином, виникає протиріччя між збільшенням обсягу інформації, яку доцільно аналізувати для запобігання загрозам, і оперативністю управління мережею.

Під загрозою з точки зору безпеки слід розуміти сукупність умов і факторів, що потенційно призводять до порушення функціонування комп'ютерної мережі в цілому, у тому числі контрольованих мереж активів (даних), а також окремих користувачів [27].

Загрози зазвичай поділяються на навмисні (усвідомлене заподіяння шкоди) та природні. До перших належать несанкціоновані підключення, витік інформації, порушення функціонування мережі тощо, а до других – форс-мажорні обставини та нещасні випадки, а також помилки внаслідок збоїв апаратури [57]. При цьому значна частина загроз безпеці є наслідком людського фактору, тобто відсутності у користувачів необхідної компетентності, а також ігнорування службових інструкцій, наприклад, недбале поводження з паролями. Аналіз безпеки комп'ютерної мережі вимагає враховувати всі види загроз, однак якщо природні загрози досить легко формалізуються в плані ризиків і захист від них досить лінійний, то загрози, що мають причиною людський фактор, вимагають особливої уваги через непередбачуваність дій навіть за відсутності наміру заподіяння шкоди. З іншої сторони, людський фактор має значення в тих випадках, коли дисфункція системи якимось чином загрожує людині, що особливо важливо з погляду часу, коли робототехніка стане звичною у побуті і відноситься до найближчого майбутнього [72].

У статті [30] розглянуто важливість безперервної оцінки ризиків у контексті Secure DevOps. Автор підкреслює, що безпека часто розглядається як другорядний аспект у розробці ПЗ, що може призвести до серйозних вразливостей і додаткових витрат у майбутньому. Він звертає увагу на те, що багато організацій

недооцінюють значення кібербезпеки, часто приділяючи їй увагу лише після виникнення негативних наслідків від атак [30].

В контексті дослідження, важливим є розгляд методів, що дозволяють здійснювати безперервну оцінку ризиків та інтегрувати безпеку в процес розробки. Це включає навчання команд, впровадження автоматизованих інструментів для сканування вразливостей, а також розвиток культури, орієнтованої на безпеку. Застосування таких підходів може значно зменшити ризики компрометації даних та підвищити загальний рівень захищеності програмних систем.

Таким чином, сучасні стратегії забезпечення кібербезпеки в КС повинні включати комплексні заходи, які охоплюють як технологічні аспекти, так і організаційні зміни. Інтеграція безпеки в DevOps процеси є актуальним завданням, яке вимагає уваги дослідників та практиків для створення більш стійких та надійних інформаційних систем.

1.5 Постановка задачі дослідження

Для розв’язання задачі підвищення точності виявлення вразливостей у ПЗКС шляхом формалізації переповнення буфера, створення нейромережевих детекторів та впровадження механізмів композитної оцінки ризику для автоматизованого прийняття рішень потрібно вирішити такі завдання:

1. Провести системний аналіз існуючих підходів до виявлення вразливостей у програмному забезпеченні, визначити їх переваги й недоліки та сформувати вимоги до автоматизованих засобів детектування переповнення буфера.

2. Створити формальні моделі класів вразливості типу переповнення буфера у вигляді орієнтованого графа з атрибутами вузлів і ребер, що відображають залежності між даними й керуванням.

3. Удосконалити модель процесу виявлення вразливості типу переповнення буфера, в якій здійснити інтеграцію графової моделі, нейромережевого детектора та модуля композитної оцінки ризику в конвеєри автоматизованого збирання та

розгортання, для забезпечення підтримки повного циклу аналізу коду та підвищення точності виявлення вразливості типу переповнення буфера.

4. Розробити метод машинного виявлення переповнення буфера з використанням нейромережевої архітектури YOLO/Transformer, визначити правила сегментації графів та формування навчальних вибірок.

5. Розробити метод підготовки та обробки даних на основі розмітки початкового коду, побудові та сегментуванні орієнтованих графів, перетворенні підграфів у багатоканальні зображення з класами Stack/Heap/Off-by-one.

6. Розробити метод композитної оцінки ризику та алгоритм інтеграції в конвеєрах автоматизованого збирання та розгортання для кількісного оцінювання критичності виявлених вразливостей та автоматизованого управління процесом розгортання.

7. Реалізувати прототипи програмних засобів та провести експериментальні дослідження, інтегрувати їх у середовища розробки, оцінити точність і швидкодію порівняно з існуючими сканерами та сформулювати практичні рекомендації.

1.6 Висновки до першого розділу

Таким чином, проведено аналіз сучасного стану та актуальних напрямів розвитку методів і засобів виявлення вразливостей у ПЗКС. Було визначено, що проблема забезпечення ІБ залишається надзвичайно актуальною через постійне зростання кількості та складності вразливостей, які виявляються у сучасних програмних продуктах.

Досліджено найбільш поширені типи вразливостей. Особливої увагу було приділено аналізу вразливостей типу переповнення буферу, SQL-ін’єкцій, XSS-атак, логічних помилок, а також інших критичних загроз, які часто призводять до серйозних наслідків. Зокрема, на основі аналізу відкритих баз даних CVE, NVD та OWASP вдалося систематизувати та класифікувати найбільш типові патерни виникнення цих загроз. Це дозволило сформулювати чіткі вимоги до розробки

нових, більш ефективних методів автоматизованого аналізу програмного коду, які здатні оперативно та надійно ідентифікувати подібні вразливості.

Проведено порівняльний аналіз існуючих методів статичного та динамічного аналізу програмного коду. Виявлено, що хоча класичні засоби аналізу (наприклад, SonarQube, Checkmarx, Fortify) залишаються популярними та ефективними в окремих випадках, їхня загальна точність і глибина аналізу складних структурних залежностей у коді залишається недостатньою. Ці інструменти не завжди здатні ефективно виявляти складні та контекстуальні загрози, що підкреслює необхідність подальшого розвитку технологій машинного навчання і штучного інтелекту у цій сфері.

Виявлено сучасні тенденції у сфері автоматизації аналізу безпеки, зокрема перспективи інтеграції нейронних мереж та інших методів штучного інтелекту у процеси Continuous Integration та Continuous Deployment (CI/CD). Встановлено, що такі інтегровані рішення дозволяють суттєво підвищити рівень безпеки на ранніх етапах розробки, що зменшує витрати на усунення вразливостей після випуску програмних продуктів. Цей висновок був підкріплений аналізом практичного досвіду використання існуючих рішень, який підтвердив високу ефективність поєднання класичних методів статичного аналізу з новітніми підходами на основі нейронних мереж.

Загалом проведений аналіз підтвердив необхідність і перспективність розробки нових, більш досконалих моделей та алгоритмів аналізу коду. Отримані результати стали основою для подальшого формулювання задач, а також дозволили обґрунтувати доцільність використання адаптованих нейромережевих моделей, зокрема YOLO, для автоматизованого виявлення критичних вразливостей типу переповнення буферу.

Основні результати розділу опубліковані у працях [120; 121; 126 – 128; 155; 156].

РОЗДІЛ 2
МОДЕЛІ ВРАЗЛИВОСТЕЙ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНИХ СИСТЕМ ТА ПРОЦЕСУ ЇХ ВИЯВЛЕННЯ

У сучасних умовах забезпечення інформаційної безпеки КС є надзвичайно складним завданням через постійне зростання кількості та різноманітності вразливостей, що виникають у процесі розробки та експлуатації ПЗКС. Для ефективного вирішення цієї проблеми потрібно мати чітко визначені та формалізовані механізми, які дозволяють описати і прогнозувати можливі ризики ще на етапі проектування та розробки ПЗКС.

Саме тому використано формальний підхід, який базується на створенні моделей виникнення вразливостей у системних програмах та операційних системах. Акцент зроблено на шести найбільш поширених і критичних класах вразливостей – Buffer Overflow (переповнення буфера), Stack Overflow (переповнення стеку), Heap Overflow (переповнення купи), Off-by-One Error (помилка індексації масивів), Integer Overflow (цілочисельне переповнення) та Race Conditions (умови перегонів). Вибір саме цих класів вразливостей обґрунтовується їх широким розповсюдженням, високим потенціалом загроз для безпеки інформаційних систем, а також доступністю великої кількості реальних даних для детального аналізу та формалізації умов їх виникнення.

Для побудови моделей враховано специфіку програмного коду, умови виникнення ризиків та їх потенційний вплив на безпеку систем. Використання інформації з відкритих баз даних, таких як CVE, NVD та OWASP, дозволило уточнити і деталізувати критерії, які лягли в основу розроблених математичних моделей. На цій основі були створені графові представлення, що суттєво спрощують автоматизований аналіз програмного коду та локалізацію критичних зон, що потенційно містять вразливості.

Таким чином, створення таких формальних моделей є необхідною основою для подальшої розробки ефективних інструментів аналізу коду, які можуть бути інтегровані з технологіями штучного інтелекту та іншими сучасними методами, що

забезпечують оперативне виявлення й усунення загроз на ранніх етапах життєвого циклу програмних систем у комп'ютерних системах.

2.1 Визначення області дослідження

З розвитком технологій, які залучають штучний інтелект, великі дані та хмарні обчислення, комп’ютерні системи стають дедалі складнішими. Більшість операційних систем містять мільйони рядків програмного коду, що обумовлює істотне підвищення ймовірності появи вразливостей. Зростання складності, інтеграція численних модулів, бібліотек і зовнішніх API, а також розгортання програм на різноманітних платформах ускладнюють тестування та аудит безпеки, сприяючи виникненню значної кількості недоліків у структурі програмного коду. Ці фактори створюють передумови для появи критичних вразливостей, зокрема пов’язаних із переповненням буфера програм комп’ютерних систем.

Проблема дослідження полягає у недостатній автоматизації процесів виявлення вразливостей у ПЗКС, зокрема у системному програмному забезпеченні, яке застосовується в операційних системах реального часу (RTOS) та низькорівневих драйверах пристроїв. Необхідність оперативної та точної ідентифікації вразливостей обумовлена високими ризиками експлуатації цих систем у критичних галузях, таких як “розумний будинок”, IoT-пристрої та промислова автоматизація. У цій області під аналізованими фрагментами ПЗКС будуть розглядатись функції та основні елементи, які виконують операції доступу і зміни даних у пам'яті. Такі фрагменти створюють групи важливих елементів при формуванні графів і подальшій оцінці ризиків, адже вони безпосередньо впливають на дотримання умов переповнення.

Здійснюватимемо дослідження системного та вбудованого ПЗКС (ядро й модулі ОС, драйвери, системні служби, низькорівневі бібліотеки), де використовуються C/C++ мови. Цим видам ПЗ притаманне ручне управління пам'яттю, а також активна робота з вказівниками і буферами. У цьому контексті переповнення буфера стає особливо важливим, оскільки може призвести не лише

до відмови в обслуговуванні, але й до збою в управлінні потоком і навіть ескалації привілеїв. Тому область дослідження обмежена виявленням переповнень стеку/кучі та помилок off-by-one у коді ПЗКС. Водночас, вразливості, які здебільшого стосуються веб-застосунків і інформаційних систем (SQL-ін’єкції, XSS, помилки автентифікації/авторизації, логічні дефекти бізнес-процесів), не розглядаються в цій роботі.

Сценарій застосування можна описати згідно такої моделі: розробник або інженер отримує для перевірки спеціалізоване ПЗ, наприклад, RTOS або драйвер для платформи на базі мікроконтролерів (наприклад, Arduino). Це ПЗ потрібно перевірити на наявність типових вразливостей, таких як переповнення буфера, перш ніж воно буде впроваджене в реальну КС, яка керує пристроями “розумного будинку”. Тому дослідження сфокусовано винятково на вразливостях системного програмного забезпечення, зокрема на операційних системах реального часу (таких як FreeRTOS, ZephyrOS), драйверах та низькорівневих утилітах. Особлива увага приділяється аналізу механізмів виникнення переповнення буфера, оскільки ці вразливості найбільш поширені в системних програмах і мають серйозні наслідки при їх експлуатації.

Переповнення буфера належить до найбільш розповсюджених і небезпечних вразливостей. За даними CVE (Common Vulnerabilities and Exposures), воно входить до переліку найчастіше експлуатованих вразливостей, які здатні істотно загрожувати інформаційній безпеці. Експлуатація буферних переповнень дозволяє зловмисникам досягати виконання довільного коду, здійснювати ескалацію привілеїв, порушувати цілісність та конфіденційність даних. Економічні наслідки таких атак можуть бути надзвичайно значними, оскільки включають не лише прямі фінансові втрати, але й витрати на відновлення працездатності систем, судові витрати та втрату репутації організацій.

Аналіз проблеми переповнення буфера доцільний як з точки зору розвитку теоретичних методів, так і практичних інструментів виявлення та запобігання вразливостям. Чітко виражені умови виникнення та механізми експлуатації, що супроводжують буферні переповнення, уможливлюють їх формалізацію,

розроблення математичних моделей та алгоритмічних підходів. Це, своєю чергою, сприяє ефективнішому розумінню сутності проблеми, дає змогу уніфікувати та систематизувати наявні знання, а також створювати дієві методи аналізу та автоматизовані засоби виявлення вразливостей на рівні коду. Тому, необхідним є розроблення модедей переповнення буфера комп'ютерних систем як класу вразливостей, що поширені у різноманітних мовах програмування та середовищах виконання.

Особливе місце при вирішенні цієї проблеми займають бази даних вразливостей, такі як CVE, NVD та OWASP, які містять опис і приклади відомих типів вразливостей, їх умови виникнення та можливі наслідки. Ці бази стають ключовим джерелом для формування шаблонів аналізу коду та навчання систем, заснованих на машинному навчанні. Використання нейронних мереж дозволяє виявляти не лише відомі вразливості, а й раніше невідомі патерни, аналізуючи складні взаємозв'язки між компонентами коду. Завдяки здатності моделювати поведінку програмних структур нейронні мережі забезпечують адаптивний підхід до виявлення аномалій.

Також завдяки аналізу баз даних вразливостей CVE та NVD визначено типові класи вразливостей, які найбільш поширені у системних програмах та операційних системах реального часу. Розглядатимемо наступні типові класи вразливостей.

1. Buffer Overflow (переповнення буфера).
2. Off-by-One Error (помилка індексації масивів).
3. Heap Overflow (переповнення купи).
4. Integer Overflow (цілочисельне переповнення).
5. Race Conditions (умови перегонів).
6. Stack Overflow (переповнення стеку).

Ці класи вразливостей є характерними саме для ПЗКС, зокрема систем реального часу. Також вони особливо актуальні для платформи Arduino та популярних операційних систем реального часу, таких як FreeRTOS, ZephyrOS і NuttX RTOS, що використовуються у пристроях IoT і системах автоматизації "розумних будинків".

Зважаючи на виклики, пов’язані з переповненням буфера, потрібно здійснити розроблення моделей, які дозволять не лише формалізувати наявні знання, але й запропонувати нові методи аналізу та ідентифікації вразливостей. Використання таких моделей у поєднанні з машинним навчанням створить основу для розробки інструментів, які підвищать рівень безпеки ПЗКС та знизять ризики, пов’язані з експлуатацією переповнення буферу.

Переповнення буфера є одним із найбільш поширених і небезпечних типів вразливостей в операційних системах та системному програмному забезпеченні, таких як драйвери пристроїв. Здатність зловмисників експлуатувати такі вразливості залежить від специфіки їхнього прояву, структури програмного коду та особливостей середовища виконання. Переповнення буфера може спричинити аварійне завершення або зависання програми комп'ютерної або операційної системи, що веде до відмови обслуговування (denial of service, DoS). Тому для систематизації підходів до аналізу цієї проблеми необхідно розглянути основні типи переповнень буфера, кожен із яких має унікальні механізми виникнення та методи експлуатації. Аналіз здійснюватимемо для ПЗ, написаного на системних мовах програмування C, C++ (стандарти C++11, C++14, C++17, C++20) та спеціалізованій мові Arduino (C з розширеннями Arduino), оскільки саме ці мови широко використовуються для написання операційних систем реального часу та драйверів пристроїв. Вибір цих мов зумовлений їх домінуючою роллю в розробці ПЗ для мікроконтролерів та систем керування. Насамперед, розглянемо такі види переповнень, як переповнення в стековому кадрі та переповнення в купі.

Переповнення стеку, є класичною вразливістю, яка виникає через перевищення обсягу вхідних даних над виділеним розміром буфера у стеку. Це призводить до неконтрольованого перезапису критичних областей пам’яті, включаючи адресу повернення. Зловмисник може цим скористатися для спрямування виконання програми на зловмисний код, який він розмістив у пам’яті. До того ж, переповнення стеку може стати основою для атак, спрямованих на компрометацію системи, і воно є одним із найпоширеніших методів експлуатації вразливостей у сучасному програмному забезпеченні комп'ютерних систем.

Модель вразливості переповнення стеку базується на зберіганні локальних змінних, параметрів функцій і адреси повернення у стеку під час виконання програми:

$$M = \langle L, P, R \rangle, \tag{2.1}$$

де M – стек; L – множина локальних змін; P – множина параметрів функцій; R – множина адрес повернення.

Переповнення виникає, коли введені дані перевищують розмір виділеного буфера, що призводить до перезапису сусідніх областей пам’яті. У таких випадках перезапис може вплинути на адресу повернення, дозволяючи зловмиснику спрямувати виконання програми на інший сегмент пам’яті. Це створює можливість впровадження зловмисного коду або маніпуляції контрольним потоком програми. Зокрема, відсутність перевірки меж даних під час використання небезпечних функцій, що маніпулюють пам’яттю, таких як gets, strcpy, або scanf, значно спрощує експлуатацію вразливості у драйверах та системних програмах. Зловмисник може сформувати спеціальний вхідний рядок, який змінить адресу повернення функції таким чином, щоб програма почала виконувати код, розташований у буфері або іншій області пам’яті. У деяких випадках цей код може містити "ROP-гаджети" – короткі послідовності інструкцій, що дозволяють виконати складні дії, комбінуючи фрагменти існуючого коду програми.

Наслідки стекового переповнення можуть бути значними і часто залишаються непоміченими до моменту експлуатації. Серед них – аварійні збої у роботі ПЗКС, порушення цілісності даних та компрометація конфіденційності. Це актуалізує необхідність впровадження ефективних механізмів раннього виявлення, а для цього потрібно розглянути безпосередньо методи атак.

У сучасних умовах, навіть із впровадженням захисних механізмів, таких як ASLR (Address Space Layout Randomization) і DEP (Data Execution Prevention), переповнення стеку залишається ключовою проблемою через еволюцію технічних атак. Зокрема, Return-Oriented Programming (ROP) є методикою, яка дозволяє зловмисникам використовувати існуючий код у пам’яті для створення ланцюга

виконання зловмисних дій без необхідності впровадження власного коду. Зловмисники використовують додаткові техніки, такі як створення “трамплінів” – коротких інструкцій, які передають управління на адреси в пам’яті, що містять інші корисні фрагменти коду.

$$T = \{(a_i, j_i) \mid j_i : a_i \to a_j, a_i \epsilon G_u\}, \quad (2.2)$$

де T – множина трамплінів; a_i – адреса, за якою розташована коротка інструкція (наприклад, jmp, ret, call); j_i – інструкція, яка передає управління на іншу адресу; a_j – адреса корисного коду; G_u – множина “корисних” (використовуваних у атаці) інструкцій у пам’яті.

Це підтверджує гнучкість атак, які можуть бути здійснені на основі переповнення стеку, навіть у системах із сучасними засобами захисту. Наприклад, у сценарії з ROP-гаджетами зловмисник може створити вхідний рядок, який містить адресу гаджету для виклику системної команди, такої як `execve("/bin/sh")`, що забезпечує доступ до оболонки. Це дозволяє атакуючому виконувати команди із привілеями поточного користувача, створюючи значну загрозу для цілісності системи.

Розглянемо метод атак, що полягає в перезапису адреси повернення функції, який задамо множиною адрес повернення після атаки:

$$R' = R \backslash \{r_v\} \cup \{r_m\}, \quad (2.3)$$

де R' – змінена множина адрес повернення після атаки; R – початкова множина допустимих адрес повернення; r_v – справжня адреса повернення, яка була записана у стек; r_m – підставлена зловмисником адреса.

У цьому випадку атакуючий використовує переповнення для заміни адреси повернення на вказівник, що вказує на зловмисний код. Такий підхід дозволяє зловмиснику виконувати довільні дії у системі, включаючи зміну конфігурацій, завантаження додаткових компонентів або ініціювання подальших атак.

Таким чином, наслідками цієї вразливості може бути:

1. Виконання довільного коду в контексті користувача.
2. Компрометація системи, включаючи доступ до конфіденційної інформації.

3. Поширення зловмисних компонентів через мережеві канали або уразливі файли.

Розглянемо модель атаки, що базується на переповненні купи (heap), яка є критично важливою категорією буферних переповнень комп'ютерних систем і суттєво відрізняється своїм механізмом виникнення та експлуатації від переповнення стеку. Основна відмінність полягає в тому, що пам'ять у купі виділяється динамічно під час виконання програми у операційній системі, і це створює унікальні можливості для маніпуляцій із пам'яттю. Відомо, що переповнення купи відбувається у випадках, коли обсяг даних, записаних у виділений блок пам'яті, перевищує його розмір. Це може призводити до перезапису сусідніх блоків, метаданих менеджера пам'яті, або навіть інших критичних структур, що відкриває шлях для компрометації систем:

$$\exists B_i \in H: S(D_i) > S(B_i) \Longrightarrow C(B_{i+1} \cup M_{i+1}), \quad (2.4)$$

де $H = \{B_1, B_2, \dots, B_n\}$ – множина блоків пам'яті в купі; B_i – виділений блок пам'яті; D_i – дані, які записуються у B_i; M_{i+1} – метадані менеджера пам'яті після B_i; M_{i+1}– критичні структури або службові покажчики; $S(...)$ – функція, яка означає розмір даних; $C(...)$ – функція, яка означає пошкодження або модифікацію вказаних структур.

До того ж, на відміну від переповнення стеку, яке часто орієнтоване на зміну адреси повернення функції, переповнення купи є особливо небезпечним через динамічний характер виділення пам'яті, що робить його важким для виявлення стандартними інструментами аналізу. Механізм переповнення купи базується на порушенні структури пам'яті КС, яка використовується для динамічних виділень. Коли обсяг даних, записаних у блок пам'яті, перевищує розмір цього блоку, починається перезапис сусідніх областей пам'яті. Цей перезапис може зачіпати як інші блоки пам'яті, так і службові дані, які зберігають інформацію про стан купи. Наприклад, зловмисник може змінити значення таких параметрів, як prev_size або size, які є частиною метаданих блоків пам'яті, щоб створити фальшиві зв'язки між

блоками. Це може призвести до некоректного звільнення пам’яті, або її перенаправлення на інші області, контрольовані зловмисником.

Часто зловмисники використовують переповнення купи для обходу захисних механізмів КС, таких як ASLR або DEP, маніпулюючи метаданими або розташуванням об’єктів у пам’яті. Зокрема, вони можуть експлуатувати вразливості для перезапису важливих параметрів у суміжних структурах, отримуючи контроль над критичною функціональністю.

Також варто зауважити, що особливу вразливість мають системи, які використовують зв’язані списки для керування вільними блоками пам’яті. Наприклад, сучасні алгоритми керування пам’яттю, такі як glibc malloc, покладаються на метадані для визначення розташування і розміру блоків пам’яті. Зміна цих метаданих через переповнення створює умови для експлуатації, включаючи "use-after-free" та "double free" атаки. Такі маніпуляції можуть викликати помилки у роботі менеджера пам’яті, що, у свою чергу, дозволяє зловмиснику змінювати довільні області пам’яті.

Отже, перейдемо до розгляду одного із найнебезпечніших методів атак на основі переповнення купи, а саме: Use-After-Free, який базується на використанні пам’яті після її звільнення. Зловмисник отримує можливість використовувати пам’ять, що вже виділена для іншого процесу, змінюючи її вміст на свій розсуд. Наприклад, уразливість такого типу дозволяє підміняти об’єкти пам’яті, які застосовуються для автентифікації, відкриваючи шлях до виконання зловмисних сценаріїв. Use-After-Free часто використовується в поєднанні з іншими техніками, такими як переповнення буфера, для створення багатоступінчастих атак.

Розглянемо ще одним поширений метод: Heap Spraying, який полягає у заповненні великих областей пам’яті спеціально сформованими даними. Ця техніка створює умови, за яких зловмисний код розташовується у передбачуваній частині пам’яті, що дозволяє зловмисникам обійти механізми захисту, такі як ASLR. Heap Spraying часто використовується для підготовки пам’яті до наступного етапу атаки, наприклад, для реалізації ROP (Return-Oriented Programming). Зазначимо, що поєднання Heap Spraying із JIT Spraying дозволяє атакуючим

впроваджувати зловмисний код навіть у системах із сучасними механізмами безпеки.

Варто зауважити, що комбіновані техніки, такі як маніпуляція метаданими пам’яті, або зловживання механізмами обробки помилок, дозволяють зловмисникам отримати високий рівень контролю над пам’яттю. Вони можуть виконувати довільний код, порушувати цілісність даних або спричиняти збої у роботі критичних компонентів системи. Завдяки цим методам, переповнення купи системних програм стало одним із найбільш поширених інструментів у арсеналі кіберзлочинців.

Отже, проаналізуємо типовий приклад переповнення купи представлений у вразливості CVE- 2021-33034, яка була виявлена в NuttX RTOS. Ця вразливість виникла через переповнення буфера в драйвері ядра srv.sys, спричинене обробкою спеціально сформованих SMB-запитів без належної перевірки розміру вхідних даних. У результаті, зловмисник може перезаписати критичні області пам’яті ядра, що дає змогу підвищити привілеї та виконати довільний код у системі.

Як бачимо, ця атака демонструє, наскільки критичним є управління динамічною пам’яттю у програмах комп'ютерних систем. Відсутність перевірки меж введених даних дозволяє зловмисникам впливати на поведінку програми, змінювати критичні параметри та отримувати доступ до конфіденційної інформації. Багато подібних атак націлені не лише на безпосереднє виконання коду, але й на зміну конфігурації системи для подальших компрометацій.

Перейдемо до розгляду і аналізу помилки на одиницю або помилки неврахованої одиниці (англ. *off-by-one error*). Off-by-One помилки є поширеним класом дефектів програмного забезпечення операційних систем, які виникають через неточності у формулюванні граничних умов у роботі з масивами. Такі помилки зазвичай виникають у випадках, коли індексація або обчислення меж масиву виконуються некоректно. У багатьох мовах програмування, зокрема C та C++, індексація масивів починається з нуля, що часто призводить до помилок при роботі з граничними елементами. Наприклад, цикл, який проходить елементи

масиву, може виконувати одну зайву або одну меншу ітерацію, ніж необхідно, що призводить до виходу за межі масиву.

Помилки типу Off-by-One мають низку особливостей. Вони можуть бути важкими для виявлення під час статичного або динамічного тестування, оскільки їхній вплив може проявлятися лише в певних умовах або для конкретних наборів даних. У контексті системної безпеки такі помилки є особливо небезпечними, адже вони створюють можливості для зловмисників експлуатувати пам’ять за межами допустимих меж. Це може призводити до перезапису критичних даних, таких як адреси повернення функцій, таблиці вказівників або контрольні структури, що безпосередньо впливає на поведінку програми комп'ютерних систем.

Виявлено, що часто помилки цього типу виникають у програмах операційних систем, що працюють із динамічними структурами даних, такими як списки або хеш-таблиці. У таких структурах помилки в індексації можуть призводити до руйнування внутрішньої логіки, наприклад, некоректної зміни зв’язків між елементами або порушення цілісності даних. У багатовимірних масивах Off-by-One помилки можуть проявлятися при неправильному розрахунку індексів для доступу до елементів, що створює складніші умови для їх виявлення і виправлення.

Варто зазначити, що наслідки Off-by-One помилок можуть бути багатогранними. Вони варіюються від мінімальних порушень функціональності програми до критичних компрометацій безпеки. Вихід за межі масиву здатен призвести до пошкодження даних у сусідніх областях пам’яті. Це може впливати на цілісність програмного забезпечення комп'ютерних систем, стабільність роботи системи, а також відкривати можливості для виконання довільного коду. У контексті безпеки такі помилки особливо небезпечні, оскільки вони можуть бути використані для отримання несанкціонованого доступу до конфіденційних даних або змінювати поведінку програми операційної системи на користь зловмисників.

Вразливість CVE- 2021-3622 є прикладом Off-by-One помилки, яка була виявлена у Linux Kernel USB. Помилка виникла в реалізації функціоналу обробки URL у бібліотеці libcurl, зокрема при роботі з FILE URL. Невірна перевірка та обробка шляху до файлу дозволяє зловмиснику обійти обмеження доступу та

отримати доступ до довільних локальних файлів. Ця вразливість відкриває шлях до витоку конфіденційної інформації та може бути використана для подальших атак на систему.

Отже, для мінімізації ризиків, пов’язаних із Off-by-One помилками, необхідно впроваджувати кілька рівнів захисту, а саме:

1) використання сучасних інструментів статичного аналізу коду, які здатні ідентифікувати можливі порушення граничних умов у роботі з масивами;

2) створення і використання перевірених бібліотек для обробки масивів, які мають вбудовані механізми захисту від виходу за межі;

3) ретельне тестування програм комп'ютерних систем із застосуванням методів fuzzing, що дозволяють перевірити поведінку програми в умовах неочікуваних або аномальних даних;

4) формалізація алгоритмів доступу до масивів через створення математичних моделей, які описують сценарії роботи з даними у багатовимірних або динамічних структурах.

Також, варто зазначити, що значна частина вразливостей виникає через некоректну перевірку введених даних, зокрема їх обсягу, формату або структури. У ситуаціях, коли розмір буфера не враховується належним чином, введення надмірного обсягу даних може спричинити перезапис критичних областей пам’яті. Це стосується як стекового переповнення, коли дані порушують коректність виконання програми операційної системи через зміну адрес повернення, так і переповнення купи, де вплив поширюється на метадані пам’яті, створюючи можливості для атак. Off-by-One помилки, у свою чергу, виникають унаслідок невірного визначення меж масивів, що також призводить до порушення цілісності пам’яті. Для прикладу, функції, що обробляють рядкові дані, нерідко нехтують перевіркою довжини введення, дозволяючи зловмисникам використати цю прогалину для маніпуляцій із внутрішньою логікою програми комп'ютерної системи. Важливо враховувати, що навіть незначні відхилення у контролі меж можуть мати серйозні наслідки, якщо програма працює з динамічними структурами даних або багатовимірними масивами.

Проаналізувавши вразливості програмного забезпечення комп'ютерних систем, ми дійшли висновку, що багато відомих вразливостей пов’язано з використанням функцій, які не забезпечують належного контролю за пам’яттю. Наприклад, функції strcpy, sprintf або gets дозволяють введення надмірного обсягу даних без перевірки їх довжини, що створює умови для переповнення. У випадку стекового переповнення це може призвести до перезапису адреси повернення, дозволяючи зловмисникам перенаправити виконання програми операційної системи на зловмисний код. Переповнення купи змінює структуру метаданих динамічної пам’яті, що відкриває шлях для зловживань, включаючи некоректне звільнення пам’яті або створення фальшивих блоків.

Також, важливу роль у зменшенні ризиків відіграє впровадження безпечних альтернатив, таких як strncpy або snprintf. Однак, навіть ці функції вимагають ретельного тестування, оскільки їх ефективність залежить від коректності реалізації. Крім того, автоматизація перевірки за допомогою інструментів статичного аналізу сприяє своєчасному виявленню небезпечних фрагментів коду.

Досліджено, що однією з ключових загроз, спільною для всіх розглянутих типів вразливостей, є можливість порушення цілісності пам’яті. Наприклад, переповнення стеку часто зачіпає адреси повернення або таблиці вказівників, дозволяючи маніпулювати логікою програми комп'ютерної системи. Переповнення купи створює умови для змін у метаданих динамічної пам’яті, що уможливлює реалізацію складних атак, таких як "use-after-free". Off-by-One помилки, хоча й менш очевидні, також здатні змінювати критичні дані, розташовані поруч із межами масиву, що відкриває можливості для впровадження зловмисних інструкцій. Зловмисники часто комбінують ці вразливості для побудови складних атак, які можуть включати отримання привілейованого доступу, викрадення даних або компрометацію системи загалом. Зокрема, у багатопотокових середовищах такі вразливості мають ще серйозніші наслідки, оскільки помилки одного потоку можуть впливати на стабільність всієї програми комп'ютерної системи. Тому, систематизація даних про ці типи вразливостей і їх формалізація сприяють розробці ефективних механізмів захисту. Важливою

складовою є інтеграція практик безпечного кодування, включаючи обов’язкову перевірку введення, використання інструментів динамічного та статичного аналізу, а також впровадження fuzzing-тестів для виявлення аномальних сценаріїв роботи. Навчання розробників і створення внутрішніх стандартів програмування комп'ютерних систем також сприяють підвищенню загального рівня безпеки. Лише через комплексний підхід, який поєднує сучасні технології та науковий аналіз, можна мінімізувати ризики, пов’язані з цими критичними вразливостями.

Таким чином, кожен із розглянутих типів вразливостей характеризується окремими умовами виникнення, що залежать від типу пам’яті, у якій відбувається переповнення, або логічних помилок у програмному коді. Це дозволяє зосередити увагу на їх структурних особливостях та наслідках, які вони спричиняють у контексті безпеки програмного забезпечення комп'ютерних систем.

Отже, різні класи вразливостей, включаючи стекове переповнення, переповнення купи та Off-by-One помилки, демонструють спільні риси, що визначають їх критичний вплив на безпеку ПЗКС. Незважаючи на специфічність кожного типу, загальні механізми та причини їхнього виникнення підкреслюють нагальну потребу в удосконаленні методів програмування, орієнтованих на запобігання помилкам у роботі з пам’яттю.

2.2 Моделі класів вразливостей

З баз вразливостей CVE та NVD було визначено шість класів вразливостей, які є типових для системних програм та операційних систем реального часу: переповнення буфера; помилка індексації масивів (Off-by-One Error); переповнення купи(Heap Overflow); цілочисельне переповнення (Integer Overflow); умови перегонів (Race Conditions) і переповнення стеку (Stack Overflow). Поділ на шість класів подано в табл. 2.1, але для зручності подальшого використання моделей у нейромережевих алгоритмах виявлення вразливостей, інтегруємо ці класи у три узагальнені класи, виходячи із спільних механізмів їхнього виникнення та експлуатації.

Таким чином, виділено в табл. 2.1 наступні класи вразливостей ПЗКС:

1) Stack Overflow (переповнення стеку);
2) Heap Overflow (переповнення купи);
3) Off-by-One Error (помилка індексації масивів).

Таблиця 2.1 – Інтеграція типових класів вразливостей у три узагальнені категорії

Початкові класи вразливостей	Узагальнені класи вразливостей	Причина інтеграції початкових класів вразливостей
Buffer Overflow (переповнення буфера)	Stack Overflow (клас 1)	Механізм переповнення буфера є загальним, але залежить від типу пам’яті (стек або купа).
Stack Overflow (переповнення стеку)	Stack Overflow (клас 1)	Явно визначений тип переповнення, що відбувається у стековій пам’яті, характерний перезаписом адрес повернення.
Heap Overflow (переповнення купи)	Heap Overflow (клас 2)	Явно визначений тип переповнення, що відбувається у динамічно виділеній пам’яті (купа), з ризиком пошкодження метаданих.
Integer Overflow (цілочисельне переповнення)	Stack Overflow (клас 1)	Цілочисельне переповнення часто є передумовою переповнення буфера через некоректний розрахунок розмірів.
Race Conditions (умови перегонів)	Heap Overflow (клас 2)	Виникає при одночасному неконтрольованому доступі до спільної пам’яті, часто спричиняючи порушення меж динамічних буферів у купі.
Off-by-One Error (помилка індексації масивів)	Off-by-One Error (клас 3)	Відмінний механізм виникнення, пов’язаний із логічними помилками у граничних умовах індексації масивів.

Інші класи, такі як Integer Overflow та Race Conditions, інтегруються у зазначені категорії через їхню близькість за механізмом прояву та наслідками. Клас Integer Overflow часто є передумовою виникнення Stack Overflow або Heap Overflow, оскільки саме некоректне визначення розміру буфера є характерною

ознакою цих вразливостей. Аналогічно, Race Conditions, хоча й мають специфічний механізм, можуть створювати умови для виникнення переповнень буферів через неконтрольований доступ до пам’яті у багатопотоковому середовищі. Тому, зведення кількості класів до трьох класів дозволяє оптимізувати і реалізувати автоматизований аналіз з подальшою класифікацію вразливостей.

Формалізація умов виникнення вразливостей типу переповнення буферу є ключовим етапом у створенні математичних моделей для їх аналізу та запобігання. Основна умова, за якої відбувається переповнення буфера, виглядає наступним чином:

$$|d_i| > |b_s|, \tag{2.5}$$

де d_i – обсяг введених даних; b_s – розмір виділеного буфера.

Ця базова нерівність визначає критичний момент, коли вхідні дані перевищують ємність буфера, що може призводити до порушення цілісності пам’яті. Такі порушення є передумовою для неконтрольованого виконання програми, зокрема через перезапис критичних областей пам’яті, таких як адреси повернення функцій або таблиці вказівників.

У контексті динамічного управління пам’яттю, де розмір буфера може змінюватися під час виконання програми, модель уточняється наступним виразом:

$$|d_i| > max(b_s(t)), \tag{2.6}$$

де t – момент часу виконання операції.

Цей підхід враховує динамічну природу змін у пам’яті та забезпечує більшу гнучкість моделі під час аналізу реальних програмних середовищ комп'ютерних систем.

Для подальшого ускладнення моделі можна враховувати специфіку типу буфера, його ініціалізацію та методи роботи з пам’яттю. Наприклад, для динамічно виділеної пам’яті у функціях malloc або calloc необхідно враховувати вплив метаданих пам’яті на фактичний розмір доступної області.

Також, для аналізу взаємодій між компонентами коду, що можуть стати джерелами вразливостей, розглянемо представлення програми комп'ютерної

системи у вигляді графової моделей. Графові моделі дозволяють досліджувати потік даних, взаємодію функцій і їхню залежність від вхідних даних.

Графова модель представляється як орієнтований граф:

$$G = (V, E), \tag{2.7}$$

де V – множина вершин, кожна з яких відповідає функції, змінній або буферу у програмі комп'ютерної системи; E – множина орієнтованих ребер, які відображають потік даних між компонентами.

Кожне ребро $e_{ij} \in E$ характеризується вагою w_{ij}, що відображає обсяг переданих даних або ступінь критичності операції. Якщо вага ребра перевищує ємність буфера у вершині прийому, це сигналізує про потенційну вразливість:

$$w_{ij} > |b_j|, \tag{2.8}$$

де b_j – розмір буфера у вершині V_j.

Крім того, графова модель дозволяє не лише виявляти точки ризику, але й аналізувати поведінку програми комп'ютерної системи у складних сценаріях, таких як рекурсивні виклики функцій або обробка великих масивів даних у циклах. У таких випадках ймовірність виникнення переповнення буфера значно зростає, особливо при роботі з великими обсягами динамічно виділеної пам’яті.

Варто зазначити, що для глибшого аналізу граф може бути доповнений атрибутами, які відображають специфіку використання пам’яті. Наприклад:

1. Тип даних, що передаються між компонентами.

2. Метод ініціалізації буфера (статичний або динамічний).

3. Контекст виконання (локальна чи глобальна пам’ять).

Це дозволяє моделі враховувати особливості середовища виконання та створювати більш точні інструменти для автоматизованого аналізу.

Графова модель також є зручною для візуалізації потоків даних, що значно полегшує процес виявлення складних взаємозв’язків між компонентами програми операційної системи. Наприклад, у великих програмах комп'ютерних систем, де сотні функцій взаємодіють між собою, візуалізація дозволяє швидко

ідентифікувати критичні точки, які потребують додаткової перевірки або оптимізації.

Перейдемо до формалізації окремих моделей. Почнемо розгляд із взаємодії між буфером і адресою повернення стеку. Як було зазначено вище, стекове переповнення є однією з найпоширеніших вразливостей, яка виникає через перевищення обсягу введених даних над розміром буфера, виділеного у стеку. Цей тип вразливості має критичні наслідки, оскільки дозволяє зловмисникам маніпулювати пам’яттю програми комп'ютерної системи, зокрема адресами повернення функцій.

Умову виникнення переповнення стека задамо наступним чином:

$$\exists i \in \mathrm{I} : D_i > B_i, \tag{2.9}$$

де D_i – обсяг вхідних даних i; B_i – розмір буфера, виділеного у стеку.
Формула фіксує локальну умову переповнення стекового буфера та визначає здатність перезапису змінити адресу повернення. Наслідком цієї умови є випадок, коли переписано кілька елементів, що можна задати такою формулою:

$$\mathrm{R}' = (\mathrm{R} \setminus R_v) \cup R_m, \tag{2.10}$$

де $R_v \subseteq R$ – підмножина валідних адрес повернення, які були перезаписані внаслідок атаки, R_m – підмножина адрес, підставлених зловмисником, R' – множина адрес повернення після атаки.

Отже, ця формалізація підкреслює, що основна умова виникнення вразливості – це використання функцій, які не забезпечують перевірки меж. Коли обсяг введених даних перевищує розмір буфера, відбувається перезапис сусідніх областей пам’яті, включаючи критичні структури, такі як адреси повернення функцій. Переписана адреса повернення може бути використана для спрямування виконання програми у операційній системі на зловмисний код, інтегрований у пам’ять.

Для складніших сценаріїв, зокрема у рекурсивних функціях або багаторівневих буферах, модель може бути розширена. Наприклад, якщо взаємодія

між кількома буферами створює залежності, то можна ввести додаткову множину $\boldsymbol{U}$, яка відображає такі взаємозв'язки:

$$U = \{(b_i, b_j) | b_i \text{ і } b_j \text{ взаємодіють через небезпечну функцію}\}, \quad (2.11)$$

де U – множина взаємозв'язків між буферами (виклики, спільні дані, каскадні копіювання); b_i, b_j – буфери (ідентифікатори/області пам'яті), між якими є взаємодія;

У цьому випадку оцінка ризиків переповнення буфера комп'ютерної системи враховує не лише окремі буфери, але й їх взаємодії у програмі комп'ютерної системи.

Перейдемо до розгляду моделі захисту на основі Stack Canaries, що є ефективним механізмом запобігання стековому переповненню, який базується на використанні контрольного значення canary. Це значення додається між локальними змінними функції та адресою повернення і служить індикатором спроби маніпуляції пам'яттю. Перед виконанням інструкції повернення програма перевіряє, чи залишилося це значення незмінним.

Отже, процес захисного механізму (Stack Canaries) можна формалізувати наступним чином:

1) ініціалізація контрольного значення, тобто при виклику функції програма КС генерує випадкове значення canary так:

$$\text{контрольне\_значення} = \text{випадкове\_значення}();$$

2) перевірка перед поверненням, тобто перед виконанням інструкції повернення здійснюється перевірка цілісності canary так:

$$\text{якщо } (\text{поточне\_контрольне\_значення} \neq \text{початкове\_контрольне\_значення}), \text{ то завершити виконання}()$$

3) умову безпеки задамо так:

$$\forall (i, b) \in W : \text{якщо } S_i(i) > S_b(b), \text{то } (\text{поточне\_контрольне\_значення} = \text{початкове\_контрольне\_значення}), \quad (2.12)$$

де W– множина пар (дані, буфер); $S_i(i)$ – розмір вхідних даних; $S_b(b)$ – розмір буфера, виділеного у стеку.

Таким чином, цей механізм дозволяє ефективно виявляти спроби переповнення буфера та запобігати перезапису адреси повернення, що є ключовим елементом більшості атак на основі стекового переповнення.

Крім того, для обробки складніших сценаріїв, таких як використання ROP-гаджетів (Return-Oriented Programming), модель може бути доповнена параметрами, які враховують динаміку виконання ПЗКС. Наприклад, оцінка ризиків переповнення може враховувати тип введених даних та їх відповідність типу буфера:

$$F(d_i, b) = \begin{cases} 1, & \text{якщо } S_i(i) > S_b(b) \\ 0, & \text{інакше} \end{cases}, \tag{2.13}$$

де $F(d_i, b)$– індикатор виконання критерію; d_i – i-та порція введення/даних; b – відповідний буфер; $S_i(i)$ – розмір вхідних даних; $S_b(b)$ – розмір буфера, виділеного у стеку.

Ця функція дозволяє автоматизувати процес аналізу ризиків та виявлення вразливостей у складних програмних системах. Математична формалізація процесів захисту та оцінки умов переповнення є основою для створення інструментів автоматизованого аналізу безпеки програмного забезпечення.

Розглянемо формалізації моделей на основі умов переповнення купи. Як уже зазначалося, переповнення купи виникає внаслідок перевищення обсягу введених даних над розміром динамічно виділеного блоку пам’яті. Важливим аспектом цієї вразливості є зміна метаданих, які використовуються менеджером пам’яті для контролю над виділенням і звільненням блоків пам’яті. Некоректна модифікація метаданих спричиняє порушення цілісності купи, що може призвести до неконтрольованого виконання системної програми операційної системи або компрометації системи, тоді умову переповнення купи задамо так:

$$\exists j \in J : D_j > H_j \ \wedge \ M' = M - \{m_o\} \cup \{m_e\}, \tag{2.14}$$

де D_j – введені дані; H_j – розмір блоку пам’яті; M' – множина змінених метаданих купи; m_o – оригінальні метадані; m_e – змінені зловмисником метадані;

Це означає, що існує блок H, для якого розмір введених даних перевищує виділений розмір буфера, а метадані M' було змінено. Така зміна порушує логіку роботи менеджера пам’яті, що може викликати серйозні наслідки, а саме:

1) некоректне звільнення пам’яті, тобто зміна параметрів prev і next, які пов’язують блоки у двозв’язному списку вільної пам’яті, може призвести до помилок типу Use-After-Free або Double Free;

2) порушення вказівників, тобто модифікація вказівників дозволяє перенаправляти операції алокації чи звільнення на довільні ділянки пам’яті, відкриваючи шлях до виконання зловмисного коду.

3) аварійне завершення програми операційної системи, тобто зміна службової інформації часто призводить до збою роботи менеджера пам’яті, що унеможливлює подальше виконання програми.

Також, варто звернути увагу на формалізацію атак типу Heap Spraying, що є однією з поширених технік атак, які використовуються у поєднанні з переповненням купи. Основна ідея полягає у багаторазовому виділенні блоків пам’яті з одночасним записом у них однакових або схожих даних, які включають зловмисний код. Це дозволяє зловмиснику збільшити ймовірність потрапляння вказівника виконання у ділянку пам’яті, що містить зловмисні дані. Розглянемо основні компоненти моделі Heap Spraying.

1. Множина блоків B:

$$B = \{b_1, b_2, \dots, b_n\} \subseteq H, \tag{2.15}$$

де b_i – блок пам’яті, виділений під час атаки; H – блок пам’яті для якого розмір введених даних перевищує виділений розмір буфера;

Шаблон даних $\boldsymbol{\delta}$: Кожен блок заповнюється шаблоном $\boldsymbol{\delta}$, який містить зловмисний код:

$$\forall b_i \in B : M(b_i, \delta), \tag{2.16}$$

де M – функція memset.

2. Ймовірність компрометації, тобто якщо через переповнення або некоректну роботу менеджера пам’яті вказівник виконання (IP) потрапляє в один із блоків $\boldsymbol{b_i}$, то система може бути скомпрометована:

$$P_r = \sum_{i=1}^{n} p(IP \to b_i), \tag{2.17}$$

де $p(IP \to b_i)$ – ймовірність того, що IP перенаправлено до блоку b_i.

Тепер перейдемо формалізації третього класу вразливостей – *Off-by-One* помилки. Off-by-One помилки належать до класу логічних вразливостей, що виникають через некоректне визначення граничних умов під час роботи з масивами. Зазвичай вони пов’язані з неправильним використанням операторів порівняння (наприклад, < замість <=) або неточною індексацією, коли розробник плутає значення верхньої та нижньої меж ітерації. Формально можна визначити множину всіх потенційно вразливих доступів до масиву, якщо існують індекси $\boldsymbol{i}$, які виходять за припустимий діапазон:

$$i \notin [0, n-1], \tag{2.18}$$

де n – розмір масиву.

У випадку, коли програміст помилково змінює межі ітерацій, набір індексів $\{\ \boldsymbol{i}\ \}$ може містити щонайменше один елемент, що порушує цю умову. Особливо небезпечними є ті сценарії, де масиви обробляються у вкладених або циклічних структурах, адже навіть незначна помилка в обчисленні індексів може призводити до виходу за межі допустимої області.

Згідно з формалізмом, нехай i – індекс доступу до масиву, а n – розмір масиву. Тоді умову Off-by-One помилки можна описати так:

$$\exists i : i \notin [0, n-1], \tag{2.19}$$

де $\boldsymbol{i}$ – індекс доступу до масиву; n – розмір масиву.

Формула фіксує факт виходу індексації за коректний діапазон, що створює умови для доступу/запису поза межами структури. Якщо така ситуація можлива, у програмі комп'ютерної системи присутній ризик логічної помилки, що веде до

виходу за межі масиву. У більш складних випадках, коли індекс i динамічно обчислюється залежно від вхідних даних або станів інших змінних, необхідно враховувати додаткові обмеження та вирази, що описують залежності між змінними та розміром масиву.

Отже, наслідки Off-by-One помилок багатогранні, але одним із найважливіших є неконтрольований вплив на сусідні змінні та структури даних у пам’яті. Коли індекс i виходить за межі масиву $[0, n-1]$, програма може зчитувати або записувати інформацію до комірок пам’яті, які належать іншим змінним, об’єктам або службовим структурам. Наступним прикладом наслідків Off-by-One помилок у системному програмному забезпеченні операційних систем може бути пошкодження локальних змінних. У мові C/C++ локальні змінні функції, як правило, розташовуються безпосередньо поруч із масивом у стеку. Вихід за межі масиву на 1 елемент або більше (наприклад, запис в елемент з індексом $\boldsymbol{n}-1$) може змінити значення сусідніх змінних, що призводить до непередбачуваної поведінки. Формально можна вважати, що існує відображення $\boldsymbol{\pi}$ зі змінних локальної області видимості на адреси пам’яті. Якщо комірка, до якої відбувся запис, належить іншій змінній $\boldsymbol{x}$, то:

$$\exists i : A[i] \rightarrow x, i \notin [0, n-1], \tag{2.20}$$

де $A[i] \rightarrow x$ перезапис значення змінної x через помилковий доступ.

Така ситуація призводить до неконтрольованої зміни значень сусідніх змінних у пам'яті, що може порушити коректну роботу комп’ютерної системи.

Також, окрім локальних змінних, Off-by-One помилки можуть впливати на службові структури, наприклад, таблиці вказівників або метадані динамічної пам’яті. Це може призвести до перезапису службових структур. При роботі з динамічно виділеними масивами (в купі), Off-by-One помилки можуть пошкодити метадані менеджера пам'яті ком’ютерної системи. Це створює передумови для виникнення Heap Overflow або інших складних атак. У таком випадку зловмисник може використати некоректно змінені метадані для подальших маніпуляцій,

зокрема ініціації переповнення купи (Heap Overflow) або використання уразливостей класу Use-After-Free.

Хоча Off-by-One помилки найчастіше призводять до локальних пошкоджень пам’яті або аварійного завершення, у певних сценаріях вони можуть бути експлуатовані аналогічно до класичного буферного переповнення. Якщо масив зберігає дані, безпосередньо пов’язані з виконуваним кодом (наприклад, динамічно формовані адреси або інші покажчики), некоректний запис може дозволити зловмиснику модифікувати ці вказівники та спрямувати виконання програми на зловмисну ділянку пам’яті комп'ютерної системи. Формалізація таких ситуацій дає змогу ефективно ідентифікувати їх за допомогою нейромережевих алгоритмів автоматизованого аналізу.

Таким чином, зазначені моделі забезпечують чіткі формальні критерії для класифікації та ідентифікації типових вразливостей у ПЗКС. Це дозволяє використовувати їх як основу для автоматизованих систем аналізу на базі нейронних мереж. Дані моделі забезпечують можливість створення навчальних вибірок для нейромереж, які будуть ефективно ідентифікувати вразливості за характерними математичними ознаками. Ці формули потребують інтеграції із базами даних CVE та NVD для створення комплексних систем автоматичного аналізу безпеки.

2.3 Формування трьох класів вразливостей на основі баз вразливостей

Бази вразливостей акумулюють широкий спектр даних, які необхідні для ідентифікації, аналізу та запобігання експлуатації загроз. Опис вразливостей включає їхню класифікацію, механізми виникнення та потенційні наслідки. Наприклад, опис SQL-ін’єкції може містити детальний сценарій експлуатації, що демонструє, як зловмисник може отримати доступ до конфіденційних даних через некоректну обробку запитів. Умови виникнення зазвичай представлені через приклади коду або конфігурацій, які демонструють, як вразливість може бути викликана, наприклад, через використання функцій strcpy або gets у C/C++.

Середовище виконання охоплює дані про операційні системи, платформи та специфічні версії програмного забезпечення комп'ютерних систем, які є вразливими. Ця інформація дозволяє розробникам і адміністраторам оперативно ідентифікувати, чи стосується вразливість їхніх систем. Рекомендації для запобігання проблемам включають практичні інструменти та методики, такі як заміна небезпечних функцій на безпечні аналоги або впровадження багатофакторної автентифікації для зменшення ризиків експлуатації.

Первинними джерелами є CVE/NVD (ідентифікатори CVE з нормалізованими полями в NVD) та таксономія OWASP/CWE. З цих баз відбираємо записи, релевантні ПЗКС: фільтрація за CPE на ядра ОС; RTOS (FreeRTOS, Zephyr, NuttX тощо); драйвери пристроїв; системні бібліотеки; фільтрація за мовою/стеком технологій (C/C++/Arduino);за ознаками сімейства CWE-119/787 (Out-of-Bounds Write), CWE-121 (Stack-based BOF), CWE-122 (Heap-based BOF), CWE-193 (Off-by-one). Додатково пріоритет мають записи з технічними деталями (вказані функції/структури, умови виникнення), наявним PoC/експлуатацією або CVSS v3 $\geq$ 7.0, а також важливо виключити веб-класи вразливостей (SQLi, XSS, auth), якщо вони не пов’язані з роботою пам’яті в системному коді. Такий фільтр фокусується на типових для ПЗКС сценаріях: обмежені ресурси; інтенсивні операції з пам’яттю; прямий доступ до апаратури.

Відібрані з баз приклади відносяться до класів за ознакою розташування помилки та типом порушення меж доступу з опертям на класифікатор CWE. До класу “переповнення стеку (Stack Overflow)” відносять записи з CWE-121, а також загальні записи з CWE-119/787, якщо з опису прямо випливає модифікація стекових структур (адреса повернення, локальні змінні, кадр виклику). До класу “переповнення купи (Heap Overflow)” відносять записи з CWE-122 та відповідні CWE-119/787 у випадках, коли задіяні динамічні виділення/звільнення пам’яті або метадані купи (зокрема структури керування на кшталт tcache, fastbin), а також ситуації use-after-free чи double-free, що призводять до запису поза межі. До класу “помилки індексації на одну позицію (Off-by-One)” належать записи з CWE-193 та

інші граничні помилки індексації (±1), коли це однозначно випливає з офіційного опису у базі.

Суміжні категорії, зокрема CWE-190 (переповнення цілих) і стани змагання, включаються у вибірку лише за умови, що з офіційного опису та/або супровідних матеріалів (advisory/PoC) очевидно, що наслідком є саме переповнення буфера. У такому разі віднесення виконується за місцем реалізації наслідку (стек або купа). Така процедура забезпечує узгодженість вибірки з предметною областю програмних засобів комп'ютерних систем, для яких характерні використання таких мов програмування, як C/C++/Arduino та інтенсивні операції з пам'яттю за обмежених ресурсів.

Інформація з баз вразливостей активно використовується у статичному та динамічному аналізі коду. Інструменти статичного аналізу, такі як SonarQube чи Checkmarx, застосовують шаблони небезпечного коду з баз CVE/NVD для створення правил перевірки. Наприклад, якщо у початковому коді використовується функція strcpy, що потенційно може викликати переповнення буфера, система аналізу сигналізує про ризик. Окрім виявлення вразливостей, ці інструменти надають рекомендації щодо їх виправлення, що дозволяє інтегрувати безпеку в процес розробки на ранніх етапах.

У динамічному аналізі, який здійснюють такі інструменти, як OWASP ZAP, використовуються сценарії атак для перевірки поведінки системи у реальних умовах. Наприклад, ZAP може автоматично генерувати запити, які імітують дії зловмисника, що дозволяє виявляти вразливості до SQL-ін'єкцій або XSS. Крім того, використання CVSS-рейтингів дозволяє оцінити вплив кожної вразливості, враховуючи потенційні ризики для бізнесу та технічні наслідки.

Щодо ролі баз даних в узагальненій моделі, то вони є основним джерелом інформації для побудови математичних моделей виявлення та запобігання загрозам. Вони забезпечують стандартизацію даних, спрощують інтеграцію між різними інструментами та сприяють розробці ефективних методів захисту. Аналіз даних із CVE, NVD та OWASP дозволяє формувати моделі, які враховують широкий спектр умов виникнення вразливостей та надають точні рекомендації для

їх усунення. Наприклад, інтеграція даних із CVE у моделі машинного навчання дозволяє автоматизувати процеси ідентифікації вразливостей, використовуючи великий обсяг структурованих даних для навчання нейронних мереж. Ці бази не лише слугують інструментом аналізу, а й створюють основу для впровадження комплексних систем захисту інформаційних систем. Отже, завдяки їх використанню можна забезпечити глибше розуміння вразливостей, що дозволяє формувати більш стійкі та адаптивні системи безпеки в умовах динамічних кіберзагроз.

Сформуємо шаблони для пошуку вразливостей. Дані з баз вразливостей, таких як CVE, NVD та OWASP, є основою для створення шаблонів, які допомагають ідентифікувати вразливості у ПЗКС. Ці шаблони формуються на основі типових помилок у коді, механізмів експлуатації та сценаріїв виникнення загроз, а їх використання дозволяє автоматизувати аналіз великих обсягів коду, знижуючи ризик пропуску потенційних загроз. В основі автоматичного аналізу лежить створення чітких формальних шаблонів для кожного з визначених класів: Stack Overflow, Heap Overflow та Off-by-One Error.

Ознакове поле вразливостей визначається як множина формальних ознак, за якими вразливості однозначно класифікуються у відповідні класи. Для аналізу коду кожній вразливості відповідає набір ознак O:

$$O = \{F, C, D, M\}, \tag{2.21}$$

де F – використання конкретних функцій (наприклад, strcpy, memcpy); C – умови перевірки меж (наявність або відсутність таких перевірок); D – тип та обсяг вхідних даних; M – особливості управління пам’яттю (динамічна чи статична пам'ять, наявність метаданих).

Розглянемо типові шаблони вразливостей, згруповані за трьома класами на основі аналізу даних із баз CVE та NVD. Визначення умови вразливості переповнення стеку(Stack Overflow), що є Шаблон №1 (CVE-2023-21604):

$$\exists f \in F: (f = memcpy \vee f = strcpy) \wedge (D > B) \wedge (\neg C), \tag{2.22}$$

де f – використана небезпечна функція; D – розмір вхідних даних; B – розмір буфера у стеку; C – перевірка меж.

Наприклад, використання функції *memcpy* без перевірки довжини введення призводить до переповнення буфера і перезапису адреси повернення.

Наступним шаблоном є приклад вразливості Return-to-libc(CVE-2021-4034), тобто формалізація атаки з маніпуляцією адресою повернення через змінні середовища:

$$\exists e \in E: |e| > B_e, \tag{2.23}$$

де e – змінна середовища, B_e – розмір буфера для зберігання змінних середовища.

Перейдемо до класу переповнення купи (Heap Overflow). Прикладом може бути переповнення буфера при обробці спеціально сформованих SMB-запитів без перевірки розміру вхідних даних. Як наприклад у першому шаблоні (CVE-2021-33034, NuttX RTOS):

$$\exists H \in \mathbb{H}: (|D_H| > |B_H|) \wedge (\Delta M \neq \emptyset) \tag{2.24}$$

де H – блок пам’яті в купі; D_H – обсяг даних, які записуються у цей блок; B_H – розмір виділеного блоку; ΔM – множина змінених метаданих купи.

Також розглянемо другий шаблон Heap Spraying, який можна формалізувати наступним чином:

$$\forall B_i \in B: M(B_i, \delta), \qquad P(IP \in B_i) \to 1 \tag{2.25}$$

де B_i – блок пам’яті; $M(B_i, \delta)$ – заповнення блоку шаблоном δ; IP – вказівник виконання.

Перейдемо до класу помилки індексації масивів (Off-by-One Error). Прикладом може бути використання індексації масиву без належної перевірки меж, коли доступ здійснюється за межами допустимих індексів. Як наприклад у шаблоні (CVE-2021-3622, Linux Kernel USB):

$$\exists A: \exists i \in \mathbb{Z} : (i = |A| \vee i = -1) \wedge f(A, i) = 1 \tag{2.26}$$

де A – масив даних; i – індекс доступу до елемента; $|A|$ – розмір масиву; $f(A,i)$ – операція доступу до елемента масиву за індексом i.

Таким чином, на основі даних із баз CVE та NVD, шаблони формалізуються у вигляді множини S:

$$S = \{S_s, S_h, S_o\}, \tag{2.27}$$

де S – множина шаблонів; S_s – шаблони стекових переповнень; S_h – шаблони переповнень у купі; S_o – шаблони помилок індексації (off-by-one).

Це “словник” формальних умов, за яким автоматичний аналіз співвідносить фрагменти коду з відповідними типами вразливостей. Кожна множина містить типові формальні умови, які дозволяють однозначно класифікувати вразливість за ознаками з баз:

1) S_s – вразливості, що характеризуються переповненням буфера в стеку та перезаписом адреси повернення;

2) S_h – переповнення буферів у купі з потенційною зміною метаданих пам’яті;

3) S_o – логічні помилки, що виникають через неправильне визначення меж масивів.

Кожен шаблон визначає специфічну множину формальних ознак O, завдяки якій здійснюється автоматизоване віднесення вразливостей до одного з цих трьох класів. Ці формальні шаблони використовуються для подальшого навчання моделей на основі нейромережевих алгоритмів. Такий підхід є основою автоматичного виявлення і класифікації вразливостей ПЗКС. Навчання моделей із використанням баз вразливостей забезпечує аналіз коду та дозволяє швидко адаптуватися до нових загроз. Такі моделі можуть інтегруватися в інструменти статичного та динамічного аналізу.

2.4 Модель процесу виявлення вразливостей в ПЗКС на основі нейромережі YOLO

Розглянемо процес автоматизованого виявлення вразливостей у програмному забезпеченні комп’ютерних систем на основі нейромережевої моделі

YOLO. Буде використано сформовані три узагальнені класи вразливостей: Stack Overflow; Heap Overflow; Off-by-One Error. Розглянемо технологічний процес, який міститиме кроки від підготовки даних до класифікації трьох узагальнених класів вразливостей. Початковий програмний код проходить статичний аналіз (побудова CFG/DFG), формуються ознакові вектори або візуальні подання (зображення від 416×416) згідно з шаблонами, інтегрованими з даними CVE/NVD. Далі, на цих поданнях виконується детектор YOLO із трьома вихідними класами (Stack Overflow, Heap Overflow, Off-by-One): “спина” (backbone) екстрагує ознаки; “шийка” (FPN/PAN) агрегує багатомасштабний контекст; “голова” (head) одночасно оцінює рамки, наявність об’єкта та належність до одного з трьох класів. Результат проходить фільтрацію (NMS) та формується у вигляді звіту з посиланням на фрагменти коду і відповідні шаблони.

У даному підході графові моделі коду (CFG/DFG) не лише вивчаються як графи, а й перетворюються у наочне представлення: графове представлення G перетворюється оператором R у багатоканальний тензор-кадр X стандартного розміру (416×416 або 608×608), де кожен канал відповідає певній групі атрибутів чи показників. На створеному "зображенні графа" використовується детектор комп'ютерного зору на основі YOLO, який забезпечує виявлення підозрілих ділянок та їх класифікацію за трьома основними категоріями: Stack, Heap і Off-by-One.

Застосування YOLO в цьому контексті зумовлене не “перетворенням аналізу коду до аналізу зображень”, а зміною підходу до задачі на спрямованість практичного використання в конвеєрах конвеєрах автоматизованого збирання та розгортання, бо важливо не лише виявити, що має місце переповнення, але й отримати точну локалізацію фрагмента, який можна активно зіставити з рядками коду та використати у звіті. Саме тому завдання сформульовано як задачу виявлення об'єктів на структурованому представленні ознак CFG/DFG. Модель видає клас та координати або інтервали кандидатів, а NMS відповідає за узгодження перекриваючих спрацьовувань. Такий підхід є доцільним, оскільки дозволяє здійснити одноетапне виявлення з можливістю роботи на кількох

масштабах та гарантує стабільний формат результатів для автоматизованого тлумачення.

Перейдемо до впровадження описаного конвеєра. На практиці процес аналізу починається зі статичного створення графових моделей CFG/DFG. Після цього формується ознакове представлення згідно з формальними шаблонами, які погоджуються з даними CVE/NVD. Далі проводиться детекція за допомогою YOLO, в якій визначено три класи, та застосовується постобробка NMS. Результати аналізуються і представляються у вигляді набору локалізованих частин коду з оцінкою впевненості та посиланнями на відповідні шаблони. Для кожного виявленого випадку створюється підсумкова оцінка ризику, що застосовується для визначення пріоритетів у перевірці та усуненні вразливих частин під час розробки.

На рис. 2.1 наведено узагальнену схему процесу аналізу та виявлення вразливостей у програмному коді з поділом на три узагальнені класи: переповнення стеку, переповнення купи; помилки індексації на одну позицію. Класифікація на три класи реалізується на рівні маркування $y \in \{ 0, 1, 2 \}$ за формальними шаблонами, у нейромережевій “голові” детектора (вектор класів розмірності три) та у метриках оцінювання, де для кожного класу окремо обчислюються точність (P), повнота (R) і F_1 окремо за кожним класом та macro-F1 як узагальнена ціль для подальшої оптимізації.

Запропонована архітектура ґрунтується на одноетапному детекторі типу YOLO, адаптованому до вхідних подань коду (вектори ознак або зображення розміром 416×416, сформовані на основі графа керування (CFG) та графа потоку даних (DFG)). Архітектура включає компоненту Backbone (послідовність згорток Conv–BN–SiLU зі стрибками 2, що формує піраміду ознак 52×52, 26×26, 13×13), компоненту Neck (PAN-FPN для злиття локальних і глобальних ознак) та Detection Head, у якій для кожної комірки сіті й кожного “якоря” оцінюються координати рамки, об’єктність та вектор класів розмірності три (Stack/Heap/Off-by-One). Процес враховує специфіку кожного класу: для переповнення стеку – вплив на адресу повернення та перевірку захисних маркерів у кадрі виклику; для переповнення купи – модифікацію метаданих менеджера пам’яті та підготовчі

техніки на кшталт “spraying”; для помилок індексації – граничні умови доступу й наслідковий перезапис сусідніх змінних. Далі етапи спільні: формування ознак, нейромережева детекція, зіставлення з формальними шаблонами та обчислення інтегральних показників ризику для пріоритезації виправлень.

Для навчання використовується функція втрат виду:

$$L = \lambda_1 L_b + \lambda_2 L_o + \lambda_3 L_c, \quad (2.28)$$

де L_b – IoU / CIoU-втрата для рамок; L_o – бінарна крос-ентропія об’єктності; L_c – крос-ентропія за трьома класами.

Маркування класів: $y \in \{0{:}\text{Stack}, 1{:}\text{Heap}, 2{:}\text{Off}\}$. Для дисбалансу використано ваги $\lambda_1, \lambda_2, \lambda_3$ за частотою введених трьох класів.

Ініціалізація k-means на статистиці “розмірів уразливих субграфів/вікон коду” (мін. набір 9 якорів на трьох масштабах). Для “зображень” - нормалізація до 416×416, накладання службових шарів (канали з масками вузлів/ребер та типами операцій). Для “векторів” - укладання у псевдо-зображення (канал-wise).

Ймовірності трьох класів для кожного детектованого “фрагмента” коду з координатами (на зображенні) або інтервалами (у тексті) та посиланнями на шаблони вразливостей.

На рис. 2.2 зображено архітектуру уточнюючого класифікатора CNN–LSTM–Attention, яка застосовується для підвищення точності визначення класу вразливості, особливо коли немає достатньої кількості локальних ознак. Класифікатор отримує на вхід ознаки у вигляді векторів, які створені за допомогою графових моделей CFG/DFG або їх перетворення у псевдозображення. На виході цей процес генерує ймовірнісний розподіл, що визначає належність фрагмента коду до класів Stack Overflow, Heap Overflow та Off-by-one. Етап локалізації кандидатів, також відомий як детекція, виконується окремим модулем “YOLO”, який був описаний раніше в цьому розділі. Поточна схема же демонструє саме стадію уточнення та класифікації.

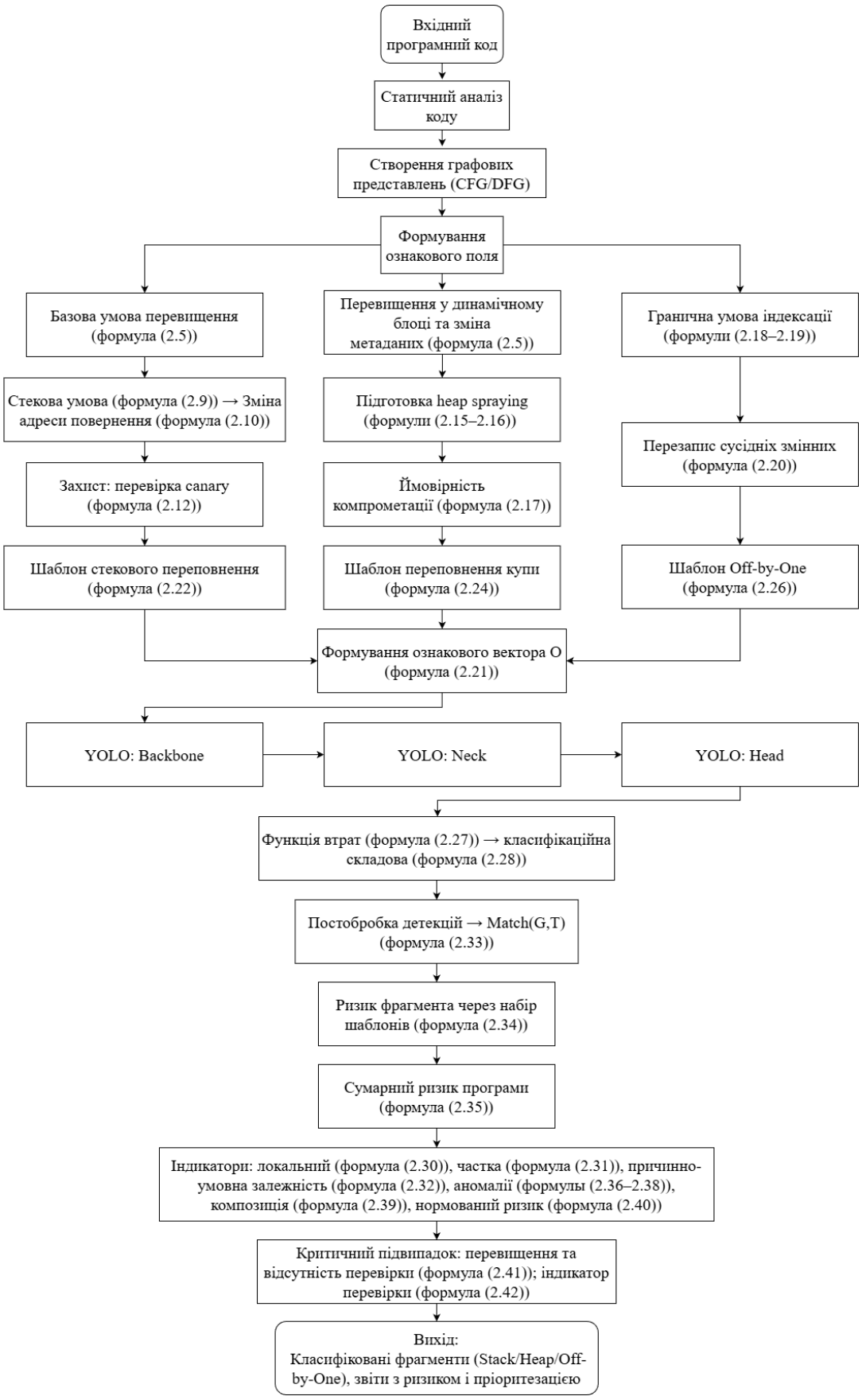


Рисунок 2.1 – Узагальнена схема процесу моделювання та виявлення вразливостей у програмному коді з поділом на три класи та з урахуванням їх специфіки

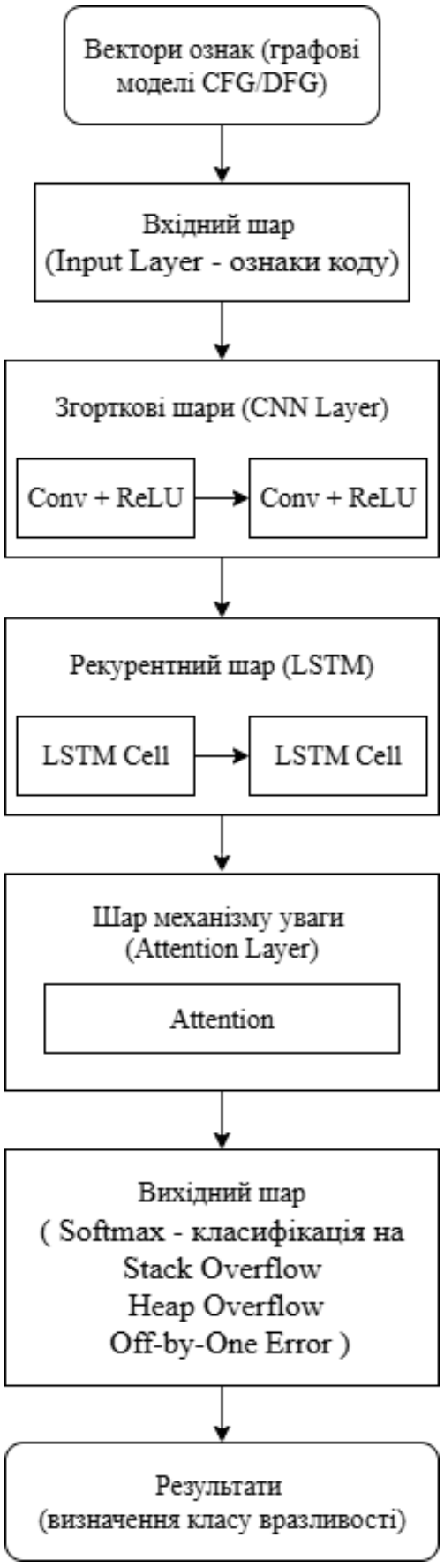


Рисунок 2.2 – Архітектура уточнювального класифікатора CNN – LSTM - Attention для трьох класів вразливостей

На схемі згорткові шари (CNN) відповідають за видобуток локальних шаблонів з ознак, а потім рекурентний шар (LSTM) обробляє цю інформацію, враховуючи послідовний контекст. Далі механізм уваги посилює вплив найінформативніших компонентів представлених даних, а вихідний шар Softmax формує остаточну багатокласову оцінку для трьох різних типів вразливостей.

З огляду на розглянуті вище бази вразливостей, виявлено, що навчання нейронних мереж для аналізу вразливостей базується на великих обсягах структурованих даних, які отримуються з баз вразливостей, таких як CVE, NVD та OWASP. Ці дані забезпечують інформацію про патерни вразливостей, умови їх виникнення та рекомендації щодо усунення. Створення навчальних наборів включає формування позитивних і негативних прикладів коду.

Позитивні приклади містять задокументовані вразливості, такі як використання небезпечних функцій (strcpy) без перевірки розміру введення, тоді як негативні приклади відображають належні практики безпечного кодування, наприклад використання strncpy із перевірками.

Тому, кожен фрагмент коду конвертується у вектор ознак, який враховує ключові характеристики, такі як наявність небезпечних функцій, глибина вкладеності циклів, типи змінних і контекст їх використання. Такі вектори ознак забезпечують стандартизоване представлення коду, яке підходить для машинного навчання. Наприклад, фрагмент коду з потенційно небезпечною функцією може отримати високий ваговий коефіцієнт, що сигналізує про ризик вразливості. Навчальні дані поділяються на тренувальні, валідаційні та тестові підмножини. Тренувальні дані використовуються для оптимізації параметрів моделі, таких як ваги і зміщення. Валідаційні дані дозволяють перевіряти ефективність моделі під час навчання, запобігаючи перенавчанню, тоді як тестові набори слугують для остаточного оцінювання точності моделі на нових прикладах. Навчання моделі здійснюється за допомогою алгоритмів оптимізації, таких як градієнтний спуск, що мінімізує функцію втрат, яка відображає різницю між передбаченими результатами й фактичними мітками вразливостей. Після початкового навчання моделі проводиться ітеративний аналіз помилок для вдосконалення її продуктивності.

Наприклад, якщо модель демонструє низьку точність для певного типу вразливостей, додаються додаткові приклади, які допомагають покращити її здатність до класифікації.

Крім того, тестування моделі на реальних кодових базах дозволяє оцінити її продуктивність у реальних умовах використання, де код може містити нестандартні конструкції або нетипові залежності.

Крім того, навчання нейронних мереж із використанням баз вразливостей забезпечує високий рівень автоматизації аналізу ПЗКС. Це дозволяє швидко ідентифікувати потенційні загрози, навіть у великих і складних проєктах. Адаптивність такого підходу гарантує, що моделі залишатимуться актуальними навіть у динамічних умовах розвитку кіберзагроз, дозволяючи швидко реагувати на появу нових вразливостей.

Дані для тестування включають задокументовані випадки вразливостей, такі як переповнення буфера чи використання небезпечних функцій. Для кожного випадку модель аналізує код та надає оцінку ризику. Наприклад, якщо у фрагменті коду використовується функція strcpy без перевірки довжини введення, модель повинна визначити цей випадок як високоризиковий. Точність передбачень оцінюється на основі співставлення результатів із мітками з баз даних.

Нехай $X = \{x_1, x_2, \dots, x_n\}$ – множина фрагментів коду, що аналізуються, а y_i – реальна мітка для фрагмента x_i (1 – вразливий, 0 – безпечний). Нейронна мережа генерує ймовірність $\hat{y}_i$, що x_i є вразливим. Функція втрат, яка відображає різницю між передбаченими й реальними мітками, обчислюється як:

$$\mathcal{L}(X) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)], \qquad (2.29)$$

де N – кількість фрагментів у множині X; y_i – істинна мітка для фрагмента x_i; $\hat{y}_i$ – передбачена ймовірність вразливості для x_i;

Отже, ця функція втрат дозволяє оптимізувати параметри моделі таким чином, щоб звести до мінімуму помилки класифікації.

Тестування на реальних даних також дозволяє виявляти слабкі місця моделі. Наприклад, якщо модель часто хибно ідентифікує код як небезпечний, це може

свідчити про недостатню кількість негативних прикладів у навчальному наборі. На основі цих спостережень проводиться доопрацювання моделі, включаючи корекцію навчального набору чи зміну архітектури.

Таким чином, розроблено модель процесу виявлення вразливостей у ПЗКС та конкретизовано архітектуру YOLO-детектора з трикласовим виходом (Stack/Heap/Off-by-One), узгоджену з визначеними шаблонами і підготовкою даних з CFG/DFG. Очікуваним ефектом цієї архітектури є зменшення пропусків та помилкових спрацювань за рахунок одноетапної локалізації-класифікації і багатомасштабної агрегації ознак.

2.5 Узагальнена модель процесу виявлення вразливостей у ПЗКС та оцінювання ризику

Модель процесу виявлення вразливостей у ПЗКС базується на поєднанні формалізованих умов виникнення дефектів роботи з пам’яттю та алгоритмічного конвеєра автоматизованого аналізу коду. Вихідним матеріалом слугують фрагменти програм на C/C++ та суміжних діалектах, для яких будується представлення у вигляді графів керування та даних, формується ознакове подання (векторне або візуальне), після чого здійснюється класифікація на підставі заздалегідь визначених шаблонів вразливостей і обчислення показників ризику. Оцінювання виконується на кількох рівнях: локальному (окремий блок коду); кластерному (група фрагментів); програмному (вся сукупність розглянутих блоків). Початковою є умова перевищення обсягу введення над виділеною ємністю буфера:

$$S_i > S_b, \tag{2.30}$$

де S_i – обсяг вхідних даних, а S_b – виділений розмір буфера.

Семантика така, що щойно обсяг введення перевищує місткість буфера, відкривається можливість порушення цілісності пам’яті. Для стекового класу це зазвичай корелює з ризиком перезапису службових слів кадру (зокрема адреси

повернення), для купи – з ушкодженням метаданих менеджера пам’яті, для off-by-one – з виходом індексу рівно на межовий елемент або на один крок за ним. Тим самим (формула (2.29)) є необхідною умовою, яка далі уточнюється контекстом розміщення буфера (стек/купа) та характером індексації (off-by-one).

Локальна індикаторна оцінка для фрагмента x задається як:

$$R_x = \begin{cases} 1,\ S_i > S_b\ \wedge\ C_x = 0; \\ 0, \text{інакше} \end{cases}, \tag{2.31}$$

де R_x – індикатор ризику для блока x; $C_x \in \{ 0,1 \}$ наявність перевірки меж (0 – відсутня, 1 – наявна).

Індикатор зменшує вагу тих випадків, де програміст явно контролює розмір введення (перевірка довжини, безпечні варіанти копіювання, межові assert-и), і навпаки – підсилює значущість фрагментів без таких перевірок. Практично це вирівнює вплив гілок, де перевищення могло бути формально можливим, але фактично блокується захистом, і дозволяє технічно відсіяти “шумові” спрацювання в місцях із коректною валідацією.

Далі узагальнюємо локальні оцінки до програмного рівня. Потрібна часткова метрика, що відбиває питому вагу ризикових фрагментів у всій сукупності проаналізованих блоків – саме її й використовуємо як інтегральний показник для порівняння модулів, версій або різних проєктів. Частку ризикових блоків на рівні програми подамо як:

$$P_A = \frac{n_r}{n}, \tag{2.32}$$

де n_r – кількість фрагментів з умовою (формула (2.29)); n – загальна кількість розглянутих блоків.

У випадках, коли ризик формується не ізольовано, а поширюється вздовж ланцюгів викликів або потоків даних, потрібна причинно-умовна корекція. Вводимо умовні ймовірності для моделювання передачі ризику між пов’язаними компонентами. Це дозволяє підсвітити критичні шляхи в CFG/DFG і пріоритезувати їх під час перевірок так:

$$P(B) = P(B \mid A) \cdot P(A), \tag{2.33}$$

де $P\ (A\ )$ – імовірність ризику в компоненті A, а $P\ (\ B \mid A\ )$ – імовірність виникнення ризику в B за наявності ризику в A.

Під час зіставлення з репертуаром відомих дефектів використовується бінарна функція відповідності “фрагмент ↔ шаблон”:

$$M(G,T) = \begin{cases} 1, G \text{ відповідає шаблону } T; \\ 0, \text{інакше} \end{cases}, \tag{2.34}$$

де G – фрагмент графа або коду; T – еталонний шаблон вразливості.

Тоді, ризик фрагмента через набір шаблонів обчислюємо так:

$$R_x = \sum_{t \in T} M(x,t) \cdot P_t, \tag{2.35}$$

де T – множина шаблонів вразливостей; M(x,t) ∈{ 0,1 } функція відповідності блоку x шаблону t; P_t – ймовірність виявлення ризику для шаблона t.

Сумарний (ненормований) ризик для програми:

$$R = \sum_{x \in X} R_x\ , \tag{2.36}$$

де X – множина всіх елементів блоків коду, які аналізуються.

Ймовірність аномалії для блока x визначимо як відношення спрацьованих умов ризику до усіх перевірених:

$$P(A_x) = \frac{m_x}{k_x}\ , \tag{2.37}$$

де m_x – кількість умов, що сигналізують про ризик у блоці x; k_x – загальна кількість перевірених умов у блоці x.

Середню ймовірність аномалії по програмі визначимо так:

$$\overline{P(A)} = \frac{1}{|X|} \sum_{x \in X} P(A_x), \tag{2.38}$$

де X - множина блоків; $P(A_x)$ – ймовірність аномалії для блока x, що визначається формулою (2.37) як відношення кількості спрацьованих умов ризику до загальної

кількості перевірених умов у блоці x; $|X|$ - потужність множини X (кількість блоків коду в аналізі); .

Для кластерів (груп споріднених фрагментів) використовуємо:

$$P_C(A) = \frac{1}{|C|}\sum_{x \in C} P(A_x), \tag{2.39}$$

де C – кластер блоків;

Композиційна оцінка локального ризику, що поєднує співвідношення “вхід/буфер”, наявність перевірки та імовірність аномалії.

Ризик для блоку x залежить від умов переповнення буфера, перевірки введення даних та ймовірності аномалії:

$$R_x = \left(\frac{S_i}{S_b}\right) \cdot (1 - C_x) \cdot P(A_x), \tag{2.40}$$

де S_i/S_b – співвідношення розміру введених даних до буфера (чим більше, тим вищий ризик); де C_x – булевий індикатор наявності перевірки меж та валідації введення у блоці x; $P(A_x)$ – ймовірність аномалії у блоці x .

Нормований загальний ризик програми:

$$\bar{R} = \frac{1}{|X|}\sum_{x \in X} R_x, \tag{2.41}$$

де $\bar{R}$ – середній ризик; X – множина блоків.

Як окремий випадок є умова перевищення обсягу введення над виділеною ємністю буфера у поєднанні з відсутністю перевірки записується так:

$$S_i > S_b \ \wedge \ C_x \ = \ 0, \tag{2.42}$$

де $\overline{S_i}$ – обсяг введення для розглянутого випадку; S_b – ємність/розмір цільового буфера; C_x – булевий індикатор наявності перевірки меж (1 – перевірка є, 0 – немає).

Формула виділяє найбільш критичну комбінацію – перевищення введення за відсутності перевірки, яку слід виправляти першочергово, тому індикатор перевірки використовується у вигляді:

$$C_x = 0, \tag{2.43}$$

де C_x – булева ознака наявності перевірки меж (0 – відсутня, 1 – наявна).

Процедурно процес реалізується як послідовність етапів: побудова проміжних подань коду (графи керування та даних), формування ознакового простору відповідно до набору шаблонів, статистична оцінка відповідностей та агрегація результатів у показники рівня блока, кластера й програми. Для детектора ознак застосовується згорткова нейромережева архітектура, у якій нижчі рівні виділяють локальні закономірності, проміжні рівні акумулюють багатомасштабний контекст, а вихідні рівні одночасно регресують параметри виділених областей і їх належність до одного з розглянутих класів. Після прогнозування застосовується відсікання надлишкових спрацьовувань, а результати пов’язуються з початковими фрагментами коду та відповідними шаблонами. Така організація дає змогу поєднати строгі формальні критерії з ефективною автоматизованою класифікацією та уніфіковано оцінювати ризики на різних рівнях представлення коду.

2.6 Висновки до другого розділу

Розроблено формальні моделі вразливостей ПЗКС з фокусом на переповнення буферу, а також створено інструментарій для аналізу кодової бази. Виявлено ключові умови, за яких виникають вразливості, сформульовано моделі ризику, ідентифіковано шаблони на основі даних із баз CVE, NVD, OWASP. Створені графічні представлення ризиків дозволяють візуалізувати взаємозв’язки між компонентами програми операційної системи та локалізувати ризикові зони.

Запропоновані моделі дозволяють не лише автоматизувати процес виявлення вразливостей, але й підвищити ефективність аналізу кодової бази. Формалізація умов ризику та інтеграція даних із баз вразливостей надають змогу систематизувати підхід до оцінки безпеки ПЗКС. Використання ймовірнісних

методів для класифікації аномалій та побудова графів залежності допомагають зосередити увагу на найбільш критичних зонах коду.

Встановлено, що розроблені моделі можуть бути інтегровані у конвеєрах автоматизованого збирання та розгортання для автоматизованого аналізу безпеки на ранніх етапах розробки. Використання цих моделей у статичному та динамічному аналізі дозволить виявляти вразливості до їхнього впровадження у виробниче середовище. Крім того, інтеграція з існуючими інструментами аналізу коду, такими як SonarQube або Checkmarx, забезпечить їхню ефективність у великих масштабах. Такий підхід сприятиме зниженню ризиків, економії ресурсів та підвищенню загальної стійкості ПЗКС до кіберзагроз.

Основні результати розділу опубліковані у працях [121; 126; 127; 153; 156].

РОЗДІЛ 3
МЕТОДИ АНАЛІЗУ ТА ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ ТИПУ ПЕРЕПОВНЕННЯ БУФЕРУ НА ОСНОВІ РОЗРОБЛЕНИХ МОДЕЛЕЙ

3.1 Підготовка даних для виявлення переповнень пам’яті

Завданнями першого етапу є побудова відтворюваного представлення програмного коду, що включає інвентар функцій, змінних, буферів і операцій над пам’яттю, а також подальше формування графових подань та багатоканальних кадрів для локалізації й класифікації ризикових фрагментів. У межах розробленого методу фіксуються локальні та графові індикатори і правила маркування класів.

Етап 1. Знімок коду та парсер (формалізація сутностей).

На вхід методу подається фіксований знімок початкового коду проєкту S, що однозначно ідентифікується хешем коміту/тегом і параметрами збірки (версія компілятора, ключі препроцесора, цільова платформа). Виконується нормалізація структури репозиторію (відсіювання нерелевантних каталогів і мов, вилучення двійкових артефактів), уніфікація стилю коду та стабілізація конфігурацій препроцесора для відтворюваності. Парсер P здійснює лексико–синтаксичний розбір і формує інвентар сутностей – множини функцій F, змінних V, буферів B та операцій над пам’яттю O:

$$P = (F, V, B, O), \tag{3.1}$$

де S – фіксований знімок початкового коду; P – парсер; F – множина функцій; V – множина змінних; B – множина буферів; O – множина операцій над пам’яттю. Діаграма процесу відображена на рис. 3.1, де показано перехід від знімка коду S через парсер P до інвентарю $\{F, V, B, O\}$.

Отже, на даному етапі об'єкт дослідження зосереджується на пам'яттєво-критичних операціях ПЗКС. Це гарантує узгодженість подальшої нейромережевої локалізації та калібрування із застосуванням графової інформації, яке стосується конкретних фрагментів, де можливе порушення умов переповнення.

Етап 2. Графові подання (CFG/DFG) та об’єднаний граф.

На підставі означення орієнтованого графа програми (формула (2.7)) та інтерпретації ваг ребер як обсягів передавання даних (формула (2.8)) формується дві взаємодоповнювальні структури: граф викликів функцій (CFG) і граф потоку даних (DFG).

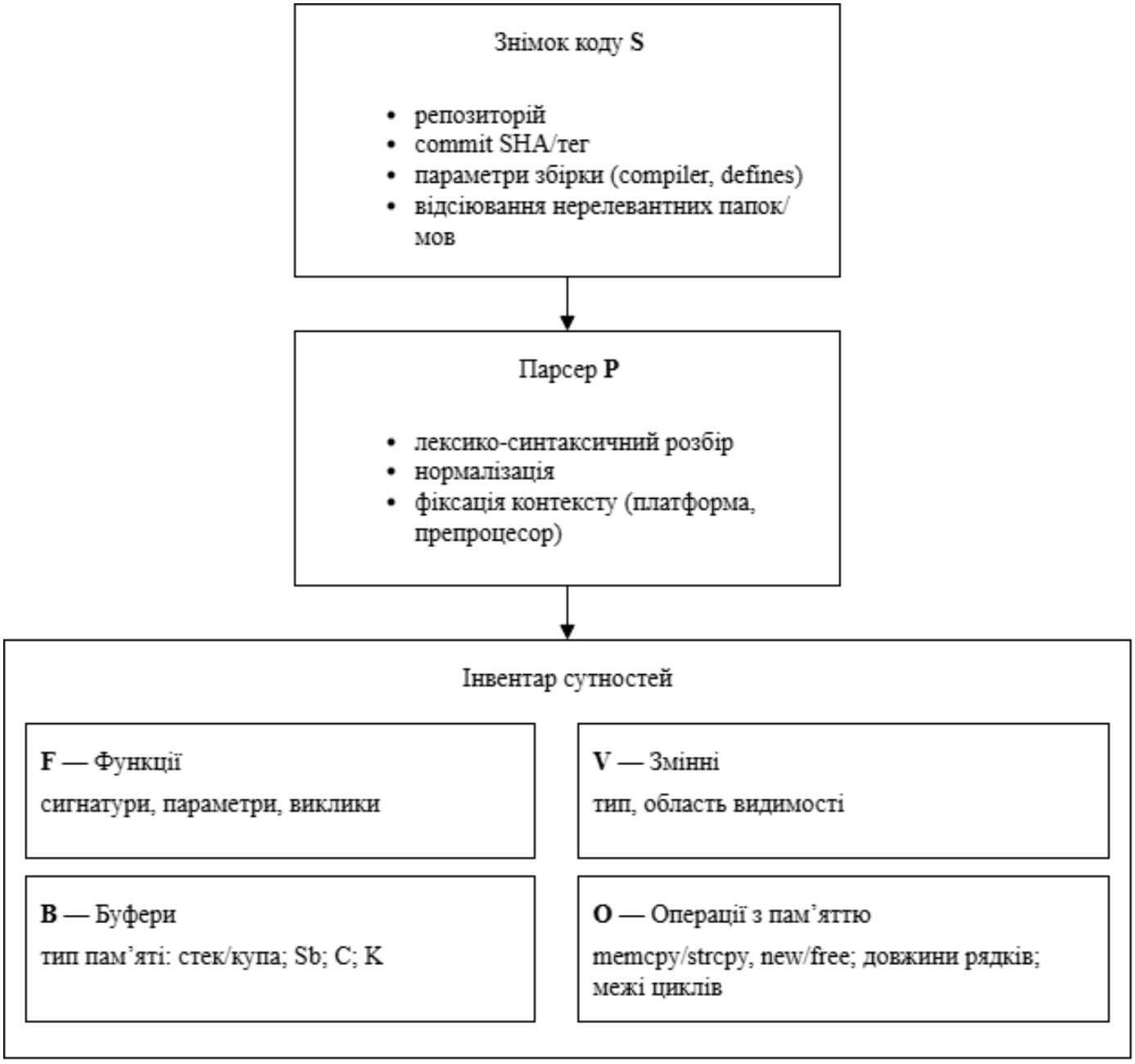


Рисунок 3.1 – Конвеєр від знімка коду до інвентарю {F, V, B, O}

У складі CFG вузлами є функції з орієнтованими ребрами типу *call/ret*, які відображають порядок і напрямок викликів. У складі DFG вузлами виступають буфери та змінні, а ребрами – операції читання/запису з вагами, що наближують

кількість переданих байтів. Для подальшої обробки виконується уніфікація ідентифікаторів вершин та ребер і синтезується єдине подання.

$$G = (V, E), V = V_C \cup V_D, E = E_C \cup E_D, \quad (3.2)$$

де V_C, E_C – множини вершин і ребер CFG (виклики/повернення call/ret); V_D, E_D – множини вершин і ребер DFG (передавання даних *read/write* з вагами $w(e)$); $w(e)$ – оцінка обсягу передавання даних на ребрі згідно з підходом з формули (2.8).

Для збереження відтворюваності фіксується (рис. 3.2) тип кожного ребра в E (одне з: *call, ret, read, write*) та напрямок, а також зберігається вага $w(e)$ для *read/write*.

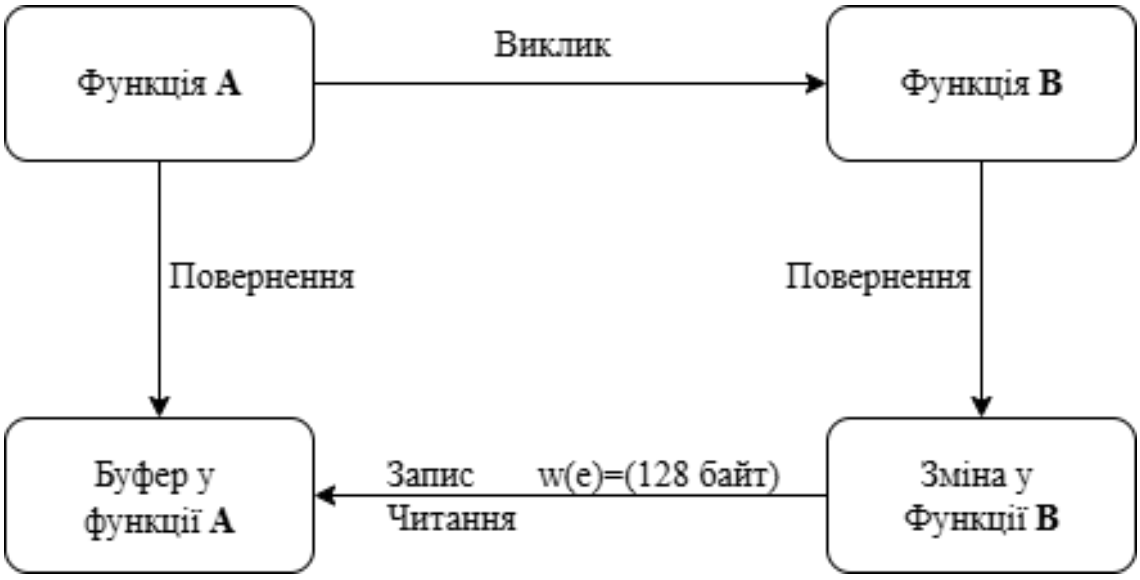


Рисунок 3.2 – Синтез графів викликів (CFG) та потоку даних (DFG) в єдиний граф G (приклад).

Етап 3. Атрибути вузлів/буферів і місткість.

У побудованому графовому поданні кожна вершина $v \in V$, що відповідає буферу або змінній, характеризується набором атрибутів. Такі атрибути дають змогу однозначно інтерпретувати властивості пам’яті, її розташування та механізми захисту. До ключових атрибутів (табл. 3.1) належать:

1) тип даних (рядковий, числовий, структурований);

2) область пам’яті (стек або купа);

3) місткість буфера $S_b(v)$;

4) гварди межових умов (assert, перевірки довжини введення);

5) stack canary як контрольний маркер від переповнення.

Формально вектор атрибутів вершини визначимо так:

$$a(v) = \{\text{тип}, \text{пам}'\text{ять}, S_b(v), \text{гварди}, canary\}, \quad (3.3)$$

де $v \in V$ – вершина графа; $S_b(v)$ - місткість буфера.

Місткість буфера для вершини подамо окремо:

$$S_b(v) \in \mathbb{N}, \quad (3.4)$$

де $S_b(v)$ – виділений розмір пам’яті.

Ці формальні ознаки безпосередньо пов’язані з умовою переповнення, поданою у попередньому розділі: базова нерівність введення > місткість (формули (2.5)–(2.6)) визначає критичний сенс.

Таблиця 3.1 – Атрибути вузла та їх джерела/використання

Атрибут	Джерело даних	Використання у моделі
Тип даних	Аналіз коду, декларації	Визначення семантики операцій
Область пам’яті	Контекст алокації (stack/heap)	Класифікація за класом вразливості
$S_b(v)$ – місткість буфера	Виділення пам’яті (malloc, оголошення масиву)	Перевірка умови “ввід > місткість”
Гварди (assert, if)	Інструкції контролю меж	Зменшення ризику переповнення
Canary	Сервісні дані у стеку	Виявлення спроби перепису адреси повернення

Етап 4. Вага потоку та локальне порушення меж.

Для кожного орієнтованого ребра $e = (u \to v) \in E$ оцінюється очікуваний обсяг передавання даних (у байтах). Ця оцінка використовується для локальної перевірки перевищення місткості буфера–приймача. Відповідно до критерію (формула (2.8)), сигнал ризику з’являється, коли обсяг передавання перевищує місткість цільового буфера:

$$w(e) = S_i(e)\,, \tag{3.5}$$

де $w(e)$ – вага ребра (оцінений обсяг передавання даних, байти); $S_i(e)$ – оцінка кількості байтів, що фактично передаються вздовж е.

Джерела оцінювання подані у табл. 3.2.

Таблиця 3.2 – Джерела для оцінки $S_i(e)$

Джерело / контекст	Як оцінюємо $S_i(e)$	Заувага
Літерали/константи (масиви, рядки)	Довжина літералу (у байтах)	Враховувати термінувальний нуль для `char[]`
Параметри API (`memcpy`, `strncpy`)	За аргументом “length” (якщо статично відомий)	Якщо параметр – змінна, переходити до гілки “вираз/цикли”
Форматні рядки (`sprintf`, тощо)	Оцінка за форматом і максимальними довжинами аргументів	За відсутності меж – верхня оцінка за типом/стандартом
Вирази та індекси у циклах	Оцінка через межі циклів/умов: minf_0(upper bound)\min(\text{upper bound})	Якщо межі не статичні – брати безпечну верхню межу (тип/розмір контейнера)
Результати `strlen`/`sizeof`	Значення функції або розмір типу	Для `sizeof` – статичне значення; для `strlen` – за відомими інваріантами/перевірками
Невідомі/дані ззовні	Консервативна верхня межа за типом призначення	Напр., для `char*` – максимально допустима довжина буфера–приймача $S_b(dst)$

Локальне порушення меж у буфері–приймачі dst(e) фіксується індикаторною функцією:

$$I_{loc}(e) = \mathbb{I}[w(e) > S_b(dst(e))], \tag{3.6}$$

де $S_b(\cdot)$ – місткість відповідного буфера, $\mathbb{I}[\cdot]$ – індикатор (дорівнює 1, якщо умова істинна, і 0 – інакше), dst(e) – вузол–буфер, у який виконується запис/копіювання. Якщо $I_{loc}(e) = 1$, маємо локальний сигнал ризику згідно з формули (2.8).

Етап 5. Граничні індекси (off–by–one) та ознака захищеності.

У межах цього етапу фіксуються звернення за межу буфера на ±1 позицію та наявність механізмів захисту (перевірки меж (табл. 3.3) і stack canary). Формалізація off–by–one доступів узгоджується з формул (2.18)–(2.20); фіксація canary – з формули (2.12).

Для вершини v, що відповідає буферу довжини n з допустимим індексним діапазоном [0, n − 1], вводимо індикатор off–by–one:

$$I_{o1}(v) = \mathbb{I}[\exists k \in \{-1, n\} \text{ та зверення до } v[k]], \tag{3.7}$$

де $I_{o1}(v) \in \{0\ , 1\}$ – індикатор доступу на одну позицію поза межами буфера, n – місткість буфера (елементів), $\mathbb{I}[\cdot]$ – індикатор істинності умови.

Ознаку захищеності подаємо двома індикаторами – наявність коректної перевірки меж (гвард) та активованої "канарейки" у стеку:

$$C(v) = \mathbb{I}[\text{існує коректна перевірка довжиниабодіапазону для } v] \\ K(v) = \mathbb{I}[\text{активний } stackcanary \text{ для фрейму з } v] \tag{3.8}$$

де $C(v) \in \{0,1\}$ – факт наявності перевірки ("умова до операції доступу/копіювання"), що узгоджується з логікою з формул (2.18)–(2.20), $K(v) \in \{0,1\}$ – факт наявності stack canary (за правилом формули (2.12): аналіз параметрів збірки й характерних конструкцій ініціалізації/перевірки у прологах/епілогах функцій).

Автоматичне виявлення складається з таких кроків:

1) $I_{o1}(v)$ - шукаються місця доступу v[k], де статично або за умовами циклів/гвардів можлива підстановка k = −1 або k = n (типові fencepost–помилки: "I ≤ n" замість "I < n", інкремент лічильника після останнього коректного індексу, недокориговані межі після *malloc(n)/new[n]*);

2) $C(v)$ - наявність перед доступом/копіюванням гвардів виду if (k >= 0 && k < n), перевірок довжини перед *memcpy/strncpy/sprintf*, використання "безпечних" API (із явним лімітом), перевірок len < sizeof(v) тощо;

Таблиця 3.3 – Патерни перевірок меж і автоматичне розпізнавання

Патерн у коді / ситуація	Що виділяємо	Як розпізнається автоматично	Примітка
Цикл з умовою `i <= n` (замість `i < n`)	Потенційний доступ до індексу `n`	Аналіз умов циклів: порівняння з верхньою межею `n`	Встановлює $I_{o1}(v)$=1 для буфера довжини n
Доступ `v[i-1]` без гварда `i>0`	Потенційний доступ `-1`	Пошук "мінус один" в індексі без перевірки нижньої межі	$I_{o1}(v) = 1$ за відсутності C(v)
Копіювання рівно `sizeof(v)` байтів у `v`	Вихід за межу на 1 байт для рядків	Матчинг `memcpy(v,*, sizeof(v))`, рядкові API без місця під `\0`	Для `char[]` потрібен запас під термінатор
`strncat/strncpy` без гарантії термінації	Можливий off–by–one при формуванні рядка	Сигнатури викликів + відсутність коректного ліміту/обнулення	Потрібні додаткові гварди або явний `v[n-1]=0`
`malloc(n)` + цикл `i=0..n`	Fencepost у динамічній купі	Зв'язка алокації з подальшими межами циклу	Узгоджується з (2.18)–(2.20)
Відсутність `if (len < n)` перед `memcpy(v,src,len)`	Відсутній гвард меж	Патерн "копіювання без обмеження розміру"	Ставить $C(v) = 0$
Наявність `if (0 <= i && i < n)` або `if (len < sizeof v)`	Коректна перевірка меж	Синтаксичний аналіз умов перед доступом	Ставить $C(v) = 1$
Увімкнено/виявлено stack canary у функції з буфером `v`	Захист від перепису адреси повернення	Перевірка параметрів збірки + характерні прологи/епілоги	Ставить $K(v) = 1$

3) $K(v)$ - фіксація опцій компілятора для canary (наприклад, захищені збірки), шаблонів вставки перевірки в епілозі функції та службових полів у локальному фреймі.

Етап 6. Локальний ризик, поширення графом та агрегація.

На цьому етапі локальні індикатори (перевищення обсягу введення над місткістю, відсутність перевірок меж, ознаки off–by–one тощо) перетворюються на числові оцінки ризику для окремих фрагментів, далі моделюється поширення ризику вздовж шляхів графа залежностей, після чого обчислюються агреговані показники для всього об'єкта аналізу. Узгодження з формальним апаратом розд. 2: умовна передача ризику за ланцюгом – (2.33); відповідність фрагментів шаблонам/патернам та внесок у ризик – (2.34)–(2.35); агрегування, нормування, частка ризикових блоків та середні показники – (2.32), (2.36)–(2.41).

Локальна оцінка ризику. Для елемента x (вузол–буфер або ребро запису/копіювання) вводимо:

$$R_x = \frac{S_i(x)}{S_b(x)} \cdot (1 - C(x)) \cdot P_A(x) \tag{3.9}$$

де $S_i(x)$ – оцінений обсяг передавання, $S_b(x)$ – місткість відповідного буфера, $C(x) \in \{0,1\}$ – індикатор наявності коректної перевірки меж, $P_A(x) \in [0,1]$ – оцінка "аномальності/патерну" згідно з правилами відповідності до формул (2.34)–(2.35).

Для стекових буферів до $(1 - C(x))$ додають поправку на наявність canary за формулою (2.12), наприклад множник $(1 - kK(x))$ з $k\epsilon(0,1]$, де $K(x) \in \{0,1\}$ – індикатор активного canary.

Поширення ризику вздовж шляху. Нехай $\pi : v_0 \to v_1 \to \ldots \to v_m$ – шлях у графі G. За формулою (2.33) ланцюгову оцінку передавання ризику задаємо так:

$$P_{CR}(\pi) = \prod_{i=1}^{m} P(v_i|v_{i-1}), \tag{3.10}$$

де $P(v_i|v_{i-1}) \in [0,1]$– умовна ймовірність того, що ризик у вузлі v_{i-1} індукує ризик у вузлі v_i (оцінюється з урахуванням типу залежності, ваги ребра, наявності перевірок/фільтрів). Для множини альтернативних шляхів між вузлами розглядають максимум (або іншу узгоджену норму) добутків $ChainRisk(\pi)$.

Агрегування і нормування. Для множини релевантних елементів $X \subseteq V \cup E$ визначимо сумарний і нормований ризики:

$$R(G) = \sum_{x \in X} R_x\,,\ \bar{R}(G) = \frac{R(G)}{|X|}, \tag{3.11}$$

де R_x – локальні оцінки (3.9); |X| – кількість елементів; $R(G.$ – сумарний ризик графа G за множиною релевантних елементів X; $\bar{R}(G$ – нормований ризик графа G за множиною X.

Додатково, відповідно до формули (2.32), використовують частку ризикових блоків P_A (на рівні програми/кластера), а за формулами (2.36)–(2.41) – інтегральні та нормовані показники для ранжування критичних зон.

Інтерпретація значення R_x відображає локальну “силу” ризику (перевищення навантаження над місткістю, відсутність гвардів, відповідність патернам), $P_{CR}(\pi)$ – наскрізну вразливість уздовж шляху залежностей R(G) та $\bar{R}(G)$ – загальний та середній рівні небезпеки для аналізованого фрагмента коду/проєкту. Ці показники (табл. 3.4) використовують для пріоритезації знахідок і формування переліку критичних ділянок.

Таблиця 3.4 – Компоненти $P_A(x)$: правила/шаблони та джерела (приклад структури)

Правило/шаблон (узгоджено з формулами (2.34)–(2.35))	Джерело / таксономія (CWE/CVE)	Ознака у коді (сигнатура)	Внесок у PA(x)P_{A}(x)
Небезпечне копіювання без ліміту	CWE–120/121/122	`strcpy`, `sprintf`, `memcpy` без перевірки довжини	підвищує до 0.7–0.9
Запис у стековий буфер без canary	CWE–121	локальний масив + відсутній гвард/`K=0`	+0.1…+0.2 до базового
Off–by–one доступ	CWE–193	`i<=n` у циклі; індекс `-1`/`n`	підвищує до 0.6–0.8
Купове переповнення через довжину	CWE–122/787	`malloc(n)` + цикл `0..n`; `realloc` без перевірок	підвищує до 0.6–0.85

Етап 7. Відбір кандидатів у три класи (Stack / Heap / Off–by–One).

На цьому етапі формується первинна мітка класу для кожного кандидата x (вузол–буфер або ребро запису/копіювання) до початку навчання моделі. Розмітка базується на індикаторах локального перевищення I_{loc} та off–by–one I_{o1}, а також

на контекстних ознаках, що відображають “словник” шаблонів і ознак для трьох класів (формули (2.21)–(2.27); зведення за класами – у табл. 2.1:

$$arg \max_{k \in K}(\lambda_1 I_{loc}(x) + \lambda_2 I_{o1}(x) + \lambda_3 ctx_k(x)) = k, \quad (3.12)$$

де K={Stack, Heap, Off–by–one} – множина класів, $\lambda_{1,2,3} > 0$ – вагові коефіцієнти, I_{loc}(x)∈{0,1} – індикатор локального перевищення (3.6), $I_{o1}(x) \in \{0,1\}$ – індикатор доступу за межу на одну позицію (3.7), $ctx_k(x) \in [0,1]$ – нормована контекстна оцінка відповідності класу k за шаблонами з формул (2.21)–(2.27) (тип пам’яті, характер алокації/деалокації, сигнатури небезпечних викликів, структура циклів тощо).

Налаштування та правила:

1. Якщо $I_{o1}(x) = 1$, пріоритет надається класу Off–by–One (за умови відсутності суперечливих контекстів).

2. Для стекових переповнень у $ctx_{stack}(x)$ враховуються локальні масиви, відсутність/наявність canary за формулою (2.12), шаблони копіювань у локальні буфери.

3. Для купових – зв’язки “алокація→використання→деалокація”, розмірні параметри, цикли “0..n” після *malloc*(n).

4. За рівності аргументів у формулі (3.12) застосовується лексикографічне правило пріоритетів, наприклад: Off1 ≻ Stack ≻ Heap (узгоджується з метою мінімізувати змішування off–by–one з іншими типами).

Після присвоєння міток виконують перевірку балансу класів і, за потреби, корекцію через oversampling/undersampling. Контекстні ознаки $ctx_k(x)$ подано в табл. 3.5.

Етап 8. Відбір кандидатів у три класи (Stack / Heap / Off–by–One).

Для подальньої локалізації та класифікації фрагментів графове подання G перетворюється оператором R на тензор фіксованого розміру (кадр), у якому кожен канал відповідає окремій групі атрибутів/індикаторів. Розміри кадру обираються стандартними для детекторів типу YOLO (416×416 або 608×608), порядок каналів фіксується для відтворюваності.

Таблиця 3.5 – Контекстні ознаки $ctx_k(x)$

Клас k	Ознака (приклад)	Як визначається автоматично	Внесок у $ctx_k(x)$
Stack	Локальний буфер у фреймі функції	Аналіз оголошень/символів, стековий контекст	0.2…0.4
Stack	Небезпечний виклик у локальний буфер (`strcpy` …)	Сигнатури викликів + відсутність гварда	0.3…0.5
Stack	Відсутній canary (за (2.12))	Параметри збірки, пролог/епілог	+0.1…0.2
Heap	Пара “malloc/realloc” + цикл до межі `n`	Зв’язування алокації з подальшим доступом	0.3…0.5
Heap	Копіювання у виділений буфер без перевірки	`memcpy(dst,src,len)` без перевірки на SbS_b	0.2…0.4
Off–by–One	Умова циклу `i <= n` (замість `i < n`)	Аналіз предикатів циклів	0.4…0.6
Off–by–One	Доступ `v[i-1]` без гварда `i>0`	Патерн індексації	0.3…0.5
Спільні (усі k)	Відсутність перевірки довжини перед копіюванням	`if (len < S_b)`/safe–API – ні	+0.1…0.3 до релевантного класу

Побудова каналів спирається на подання графа та ваги ребер (формули (2.7), (2.8)), а також на трикласову постановку завдання, так:

$$X = R(G), \qquad X(:,:,c) = X_c(G), \qquad c = 1,\dots,C\,, \tag{3.13}$$

де $X \in \mathbb{R}^{H\times W\times C}$ – багатоканальний кадр; $H \times W \in \{416 \times 416, 608 \times 608\}$ – просторовий розмір; C – кількість каналів; $X_c(G)$ – канальна функція, що відображає граф G у двовимірну карту ознак (маску/теплокарту) за певним правилом; R(·) – оператор растрування (викладання вершин/ребер на площину з подальшим кодуванням у канали).

Принципи побудови каналів.

1. Кожну вершину та ребро відображають у координатах площини за детермінованою процедурою укладки. Укладку виконують силовим (force-directed)

алгоритмом типу Fruchterman-Reingold, який мінімізує енергію системи "пружин". Між суміжними вершинами діють сили притягання, а між усіма вершинами діють сили відштовхування. Для відтворюваності фіксують seed генератора початкових координат і параметри укладки, зокрема кількість ітерацій та коефіцієнти притягання і відштовхування; у всіх експериментах використовують однакові значення цих параметрів. Отримані координати лінійно нормують до області кадру розміром H x W з відступом від меж і квантують до піксельної сітки. Ребра відображають відрізками, що з'єднують відповідні координати вершин.

2. Значення пікселів – нормовані у [0;1] індикатори або безрозмірні величини (наприклад, ваги w(e), відношення S_i/S_b).

3. Накладання кількох сутностей в одну й ту саму область кадру усувають шляхом агрегації значень пікселів. Для бінарних масок, що кодують наявність вузла або ребра, використовують операцію максимуму. Для індикаторних та інтенсивнісних каналів, які відображають ваги або ризикові індикатори, застосовують сумування з подальшою нормалізацією та насиченням до інтервалу [0; 1]. За такого підходу перекриття не призводить до втрати ознаки, а відображає інтенсивність або щільність ознак у відповідній ділянці кадру. Ідентифікація кожної окремої вершини не є ціллю відображення R(G), оскільки детектор виконує локалізацію регіонів з характерним патерном, а не реконструкцію графа.

Хоча згорткові нейронні мережі переважно виявляють локальні просторові патерни, у представленні R(G) топологічна структура частково зберігається завдяки явному відображенню ребер (зв'язків) у відповідних каналах. Ланцюги залежностей у графі подаються на зображенні як зв'язані траєкторії, а не як сукупність ізольованих точок. Додатково детектор використовує багатомасштабні карти ознак і глибші шари з більшим receptive field, що дає змогу враховувати контекст на різних масштабах. Після локалізації підозрілої області результат прив'язують до відповідного підграфа, тобто до множини вузлів і ребер, які потрапили в цю область. Подальший аналіз виконують уже на графі, де довгі залежності, зокрема поширення впливу (taint), обробляються без обмежень, характерних для локальних згорток.

Етап 9. Відбір кандидатів у три класи (Stack / Heap / Off–by–One).

Для кожного кадру X формується множина анотацій – прямокутників локалізації потенційно ризикових фрагментів коду (за результатами етапів 1–8) із призначеним класом із трьох узагальнених категорій: Stack, Heap, Off–by–one (узгоджено зі “словником” шаблонів та ознак з формул (2.21)–(2.27)).

Структуру множини анотацій подамо як:

$$Y = \{(b_i, k_i)\}, \tag{3.14}$$

де b_i – вектор параметрів прямокутника у вибраному форматі координат; k_i – клас (Stack/Heap/Off–by–one), обґрунтований за відповідними шаблонами розд. 2 (2.21–2.27).

Етап 10. Негативи, баланс і розбиття на підмножини.

На завершальному етапі формується підмножина “жорстких” негативів, виконується вирівнювання класів і проводиться розбиття даних на навчальну, валідаційну та тестову частини з урахуванням запобігання витоку між підмножинами (спліт на рівні проєктів). Критерії негативів узгоджуються з логікою про відсутність одночасно і факту перевищення, і off–by–one, наявність коректної перевірки меж (або активного canary для стеку), а також низька “аномальність” фрагмента за правилами відповідності з формулами (2.42), (2.43), (2.34), (2.35).

$$N = \{x: I_{loc}(x) = 0, \quad I_{o1}(x) = 0, \\ (C(x) = 1 \text{ або } K(x) = 1, \quad P_A(x) < t_A\}, \tag{3.15}$$

де $I_{loc}(x)$ – індикатор локального перевищення (3.6), $I_{o1}(x)$ – індикатор доступу на одну позицію поза межами (3.7), C(x) – наявність коректного гварда меж (3.8), K(x) – наявність stack canary (3.8), $P_A(x)$ – оцінка відповідності патернам/аномаліям (узгоджено з (2.34)–(2.35)), $t_A \in (0,1)$ – фіксований поріг “низької аномальності”.

Після формування позитивів і “жорстких” негативів виконується балансування часток класів (Stack, Heap, Off–by–one) на рівні кадрів і проєктів. Застосовуються:

1) undersampling для перенасичених класів;

2) oversampling (із контрольованими трансформаціями без спотворення топології графа) для дефіцитних класів;

3) за потреби – вагові коефіцієнти класів у функції втрат під час навчання.

Для контролю використовують частку ризикових блоків P_A з формули (2.32) як індикатор “складності” вибірки до/після балансування.

Розподіл виконується на рівні проєктів/репозиторіїв (або їх незалежних модулів), щоб унеможливити перетік артефактів між train/val/test:

1) навчальна: 70 % проєктів;

2) валідаційна: 20 % проєктів;

3) тестова: 10 % проєктів.

Стратифікація за класами зберігається в кожній підмножині. Фіксуються: seed генератора, список проєктів у кожній підмножині та версії комітів, що гарантує відтворюваність.

У межах підготовки даних для виявлення переповнень пам’яті побудовано відтворюваний конвеєр підготовки даних: від знімка коду до графового подання G, розрахунків локальних індикаторів і ризику $\{I_{loc}, I_{o1}, C, K, R_x\}$, моделювання поширення ризику P_{CR} і агрегатів $R(G), \bar{R}(G)$, а також формування багатоканальних кадрів X=R(G) з анотаціями $Y = \{(b_i, k_i)\}$.

Сформовано “жорсткі” негативи за формулою (3.15), виконано балансування класів і спліт 70/20/10 на рівні проєктів. Отримані артефакти X і Y є узгодженим входом для методу автоматичного виявлення вразливостей на основі допрацьованої нейронної мережі YOLO, де повинні бути здійснені локалізація та класифікація переповнень пам’яті.

3.2 Метод автоматичного виявлення вразливостей на основі допрацьованої нейронної мережі YOLO

Завдання методу полягає у побудові процедури локалізації та класифікації фрагментів коду з ризиком переповнення пам’яті на основі підготовлених багатоканальних кадрів та анотацій, з подальшим узгодженням результатів із

формальними показниками ризику. Використовується адаптована архітектура типу YOLO, яка приймає фіксований набір каналів і повертає локалізовані області з класами { Stack, Heap, Off–by–one }.

Особливістю підходу є подальша “ризик–обізнана” пост–обробка: довіра модельних прогнозів калібрується індикаторами з методу підготовки даних для виявлення переповнень пам’яті – локальними оцінками, ланцюговим показником та агрегатами, що дозволяє підвищити пріоритезацію справді небезпечних знахідок і зменшити кількість хибнопозитивних спрацювань у захищених зонах.

Навчання виконується на збалансованих підмножинах зі сплітом 70/20/10 на рівні проєктів, із фіксованими гіперпараметрами та відтворюваними налаштуваннями. Оцінювання проводиться за стандартними метриками детекції (mAP, mAR, F1 у розрізі класів) і, за потреби, за “ризик–зваженими” показниками, що краще відображають вплив кожної знахідки на загальний рівень небезпеки. У підсумку метод формує впорядкований перелік локалізованих фрагментів із класом та скоригованою довірою, сумісний із подальшою інтеграцією у процеси контролю якості й безпеки.

Етап 1. Архітектура та вхідні дані.

Адаптується одноетапний детектор сімейства YOLO до багатоканального представлення програмного коду. Архітектуру YOLO (backbone/neck/head) застосовуватимемо як базову одноетапну схему локалізації. Її наведено для фіксації позначень і забезпечення відтворюваності. Адаптація зводиться до того, що стандартний RGB-вхід замінено на багатоканальний тензор X(H×W×C) з формалізованими семантичними індикаторами, сформований оператором R(G), а трикласову розмітку (формула (3.14)) узгоджено з виходами голови детектора. Додатково параметри anchors і stride підібрано з урахуванням характерних розмірів "об'єктів" на кадрах X. Після виконання NMS застосовано граф-обізнане калібрування довіри, тобто risk-aware калібрування балу, у якому враховуються формальні показники ризику.

Вхідним даним є тензор $X \in \mathbb{R}^{H \times W}$ з фіксованими просторовими розмірами H×W∈{416,608} і впорядкованими каналами X_c. Кожен канал несе формалізовану

семантику з методу підготовки даних для виявлення переповнень пам’яті, тобто мережа від самого входу працює з ознаками, які безпосередньо відображають формальні критерії ризику. Зокрема, топологія графа узгоджена з означенням орієнтованого графа (формула (2.7)), ваги передач w_{ij} – з інтерпретацією (формула (2.8)), а трикласова постановка (Stack/Heap/Off–by–One) відповідає класифікаційним ознакам за формулами (2.21)–(2.27) і словнику шаблонів.

Обчислювальний конвеєр зберігає типову для YOLO трирівневу структуру: “спина” (backbone) φ(·) екстрагує багатомасштабні ознаки на різних stride (наприклад, 8/16/32), “шийка” (FPN/PAN) зливає локальний і глобальний контексти, а “голова” (head) повертає для кожної комірки сітки та “якоря” (anchor) параметри рамки $\hat{b}$, оцінку об’єктності $\hat{p}^{obj}$ і вектор класових ймовірностей $\hat{p}^{cls} \in \Delta^2$ для { Stack, Heap, Off–by–One }. Щоб локалізація була фізично коректною щодо масштабу ризикових фрагментів на кадрах X, прив’язку якорів та stride підбирають під характерні розміри “об’єктів” (типові ширини/висоти боксів, що виникають при растуванні кодових фрагментів/вузлів/ребер). Канальний порядок X_c фіксується ідентичним до підготовки даних для виявлення переповнень пам’яті, щоб забезпечити відтворюваність та однозначність відповідності між компонентами X і семантичними індикаторами (це важливо для подальшої “ризик–обізнаної” пост–обробки автоматичного виявлення вразливостей на основі допрацьованої нейронної мережі YOLO).

Перед обчисленням виконують стандартизацію вхідних каналів: індикаторні канали {0, 1} залишаються бінарними; безрозмірні величини нормуються у [0;1] за єдиними правилами з Методу 1; просторове масштабування до H×W проводять без порушення відносної геометрії шарів–каналів (щоб патерни, пов’язані з формулами (2.5)–(2.6) “ввід > місткість”, не спотворювалися інтерполяцією). Класи в голові моделі відповідають розмітці з (3.14), тому узгодження “формат анотацій ↔ виходи голови” не потребує додаткових перетворень.

$$X = \mathbb{R}^{H \times W}, Z = \varphi(X)\,, \tag{3.16}$$

де X – багатоканальний кадр; H×W∈{416,608} – фіксований розмір вхідного зображення; C – кількість семантичних каналів кадру X позначено як C і вона визначається конфігурацією оператора R(G) (етап 8 методу 1) та в проведених експериментах дорівнює 18; Z – тензор ознак “спини” (backbone), що далі надходить до “шийки” й “голови”.

$$f(Z) \to \{(\hat{b}_j, \hat{p}_j^{obj}, \hat{p}_j^{cls})\}_{j=1}^{N}, \hat{p}_j^{cls} \in \Delta^2 , \tag{3.17}$$

де $\hat{b}_j$ – оцінені параметри рамки (центр, ширина, висота) для j–го прогнозу; $\hat{p}_j^{obj}$ – оцінка наявності об’єкта у рамці; $\hat{p}_j^{cls}$ – тривимірний вектор імовірностей класів { Stack, Heap, Off–by–One }; N – кількість прогнозів на всіх масштабах/якорях.

Формули (3.16) – (3.17) наведено для фіксації позначень і узгодження відображення "вхід X → виходи голови" з форматом анотацій Y (формула (3.14)). Архітектурні блоки детектора при цьому відповідають базовій схемі сімейства YOLO.

Етап 2. Функція втрат і навчання.

Для забезпечення стабільної збіжності модель ініціалізують попередньо навченими вагами базової конфігурації YOLO з подальшим fine-tuning на сформованих кадрах X. Перший згортковий шар, який залежить від кількості каналів C, модифікують під вхід H×W×C. Ваги для додаткових каналів ініціалізують стандартним способом, наприклад за схемою Kaiming або Xavier, або шляхом копіювання чи усереднення базових фільтрів, після чого всю мережу донавчають спільно.

Процедура навчання будується як багатокритеріальна оптимізація, що поєднує три складові: втрату для рамок локалізації (box), для “об’єктності” (objectness) та для класів (class). Сукупна функція втрат зважується коефіцієнтами $\lambda_{box}, \lambda_{obj}, \lambda_{cls}$ з урахуванням важливості кожного компонента та дисбалансу між класами (зокрема дефіцитного Off–by–One). Анотації (формула (3.14)) узгоджені зі “шапкою” моделі, тому відповідність форматів забезпечує пряму підстановку b та класових міток у втрати. Для призначення позитивів/негативів застосовується стандартне правило за порогом IoU (наприклад, позитив при IoU ≥ 0,5, ігнор–зона

0,4–0,5), що вирівнюється зі схемою якорів/stride, обраних на Етапі 1. Щоб зменшити перевагу “легких” негативів, доцільно обмежити їх частку на міні–батч та/або використати фокусувальну модифікацію втрати.

Схема обчислень YOLO з багатоканальним входом і трьохкласовим виходом зображена на рис. 3.3.

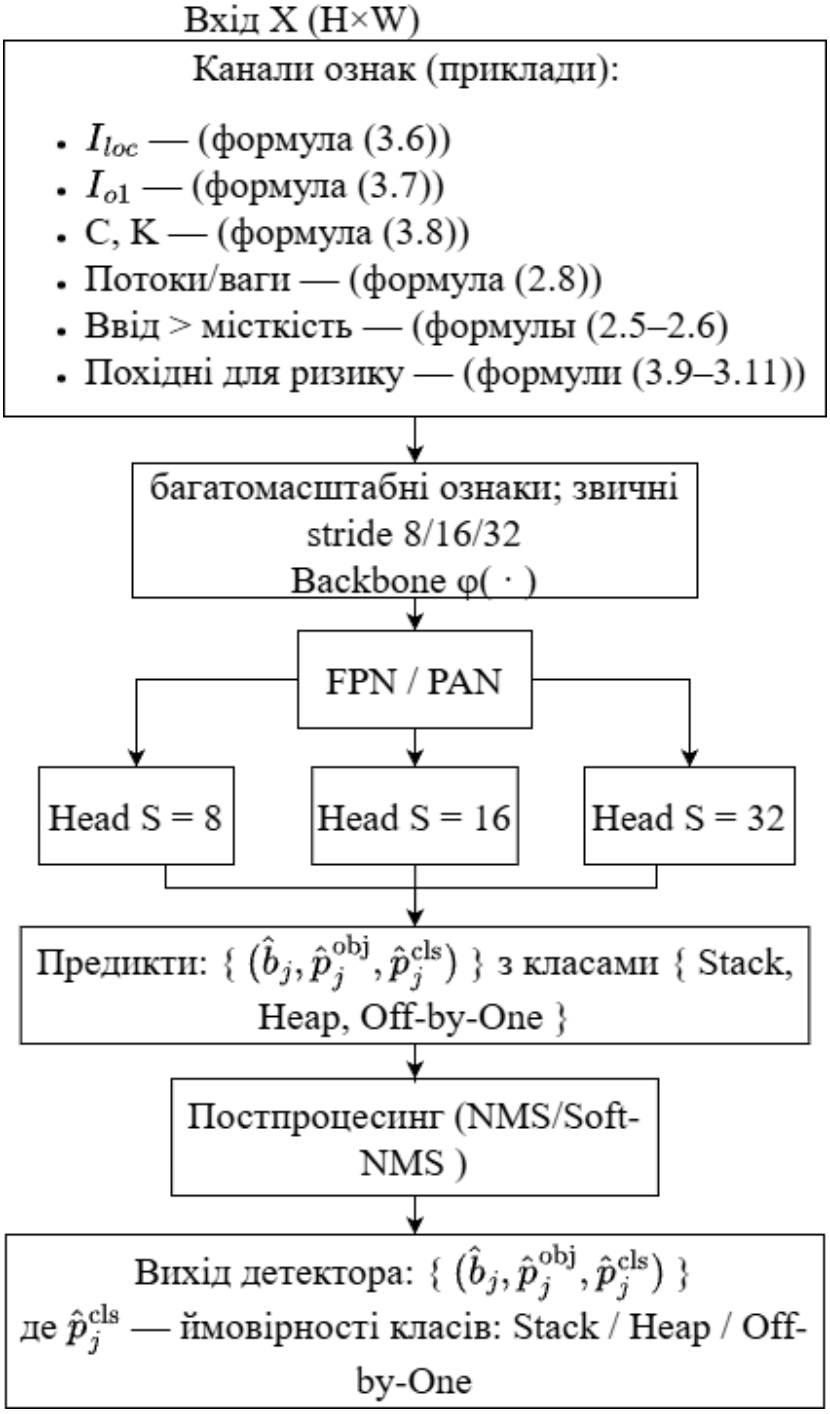


Рисунок 3.3 – Схема обчислень YOLO з багатоканальним входом і трьохкласовим виходом

Складова для локалізації L_{box} моделює відстань між еталонною рамкою b та передбаченням $\hat{b}$; як базову метрику використовуємо GIoU (або CIoU – зафіксувати в тексті конкретний вибір), що підвищує стабільність збіжності при часткових перекриттях. Для "об'єктності" L_{obj} застосовується ALL або Focal Loss (рекомендовано Focal при суттєвому дисбалансі позитивів/негативів у кадрах). Класова складова L_{cls} – крос–ентропія з вагами класів w (надаємо більшу вагу Off–by–One) та помірним label smoothing (наприклад, ε ∈ [0.05;0.1]) для зниження переобучення на "жорстких" патернах. Оптимізація виконується Adam або SGD (із моментом); швидкість навчання n_t керується графіком (cosine/step), рання зупинка – за валідальною метрикою (наприклад, mAP@0.5) з терпимістю p епох (типово 10–20). Для відтворюваності фіксуються seed'и, версії даних та конфігурацій, а також використовується змішана точність (AMP), кліпування градієнта та L2–регуляризація (weight decay).

Аугментації застосовуються лише ті, що не спотворюють топологічний зміст кадру X: масштабування, зсув, помірний поворот, випадковий кроп/паддинг у межах, сумісних з укладкою; колірні перетворення не використовуються (канали – семантичні карти, а не RGB). Будь–які просторові трансформації виконуються синхронно для всіх каналів кадру. Баланс класів у батчі підтримується через вибірку "за класом" і ваги w, а також, за потреби, через oversampling Off–by–One на рівні кадрів/проєктів (узгоджено з Етапом 10 з першого методу). Для стабілізації навчання доречно застосувати "прогрів" (warmup) швидкості навчання, а також план зменшення n_t до кінця тренування.

У подальшому використовується узагальнена функція втрат (формула (2.28)), у якій сума доданків відповідає помилкам локалізації рамок, об'єктності та класової належності. У межах цього методу фіксуємо конкретні вибори для кожного компонента та режиму навчання. Для рамок застосовується метрика типу GIoU (за потреби – CIoU, що краще поводиться на дрібних об'єктах). Для "об'єктності" за замовчуванням використовується BCE; у випадку дисбалансу позитивів/негативів увімкнено Focal Loss (типові параметри γ≈2, α≈0,25). Для класифікації трьох класів { Stack, Heap, Off–by–One } використовується крос–

ентропія з вагами класів w та label smoothing (наприклад, ε ≈ 0,1) для зменшення перенавчання.

Призначення позитивів/негативів виконується за стандартним правилом порогів IoU, узгодженим із системою “якорів” та stride (типово позитив при IoU≥0,5; зона ігнорування 0,4–0,5). Щоб зменшити вплив “легкого” фону, обмежується частка негативів у кожному міні–батчі та/або застосовується фокусувальна модифікація втрати. Оптимізація – AdamW із початковою швидкістю навчання, яку задає warmup (приблизно 3 епохи) та подальший cosine–decay (плавне зниження до кінця навчання). Для контролю перенавчання використовується early stopping за валідаційною метрикою (mAP@0.5) із “терпимістю”~15 епох. З метою відтворюваності фіксуються початкові зерна випадковості (seed), версії даних і конфігурацій; увімкнено змішану точність (AMP), L2–регуляризацію (weight decay) та кліпування градієнта.

Аугментації застосовуються лише такі, що не спотворюють топологічний зміст кадру X: масштабування; зсув; помірний поворот (до 10 градусів); випадковий кроп/паддинг у межах; сумісних із укладкою; кольорові перетворення не використовуються, оскільки канали є семантичними картами, а не RGB. Будь–які просторові трансформації виконуються синхронно для всіх каналів кадру. Баланс класів у батчах підтримується через вибірку “за класом” і ваги w, а також, за потреби, через oversampling Off–by–One на рівні кадрів/проєктів (узгоджено з Етапом 10 Методу 1). Вибір “якорів” здійснюється за статистикою типових розмірів локалізованих фрагментів (k–means, мінімум 9 якорів на трьох масштабах), а stride піраміди погоджується з обраним розміром кадру H×W.

Гіперпараметри, що використовуються у всіх експериментах (архітектура, batch, кількість епох, початкова швидкість навчання та графік її зміни, вибір компонент втрат і їх ваг, аугментації, ваги класів w, seed, апаратна платформа), зведено у табл. 3.6.

Етап 3. Постоброблення результатів детекції з урахуванням графових індикаторів.

Таблиця 3.6 – Гіперпараметри навчання

Параметр	Значення
Архітектура (backbone/head)	Backbone: CSP–Dark; Neck: PAN–FPN; Head: decoupled (YOLO–тип)
Розмір кадру (H\times W)	608×608 (основний), 416×416 (для абляцій)
Кількість вхідних каналів (C)	18
Batch size	32
Epochs / Early stopping (patience)	120 / 15
Optimizer / Weight decay	AdamW / 1×10^{-4}
LR schedule (warmup + …)	warmup 3 епох; cosine: $1 \times 10^{-3} \rightarrow 1 \times 10^{-4}$;
$\lambda_{box}, \lambda_{obj}, \lambda_{cls}$	0.05 / 1.00 / 0.50
Loss choices	GIoU; Focal ($\gamma = 2.0$, $\alpha = 0.25$); CE + label smoothing $\varepsilon = 0.1$
Класові ваги (w)	$w_{Stack} = 1.0, w_{Heap} = 1.0\ w_{Off-by-One} = 1.5$
Аугментації	масштаб, зсув, поворот ≤10 градусів, кроп/паддинг; без кольорових
Seed / Відтворюваність	42 / фіксовані конфігурації та версії даних
Апаратна платформа	NVIDIA RTX 3090 (24 GB); AMP: так
Ініціалізація backbone	pretrained, fine-tuning; перший Conv адаптовано під C каналів

Після інференсу моделі формується початковий пул передбачень у вигляді набору пронормованих рамок із оцінкою об’єктності та вектором класових імовірностей. На першому кроці виконується відсікання надлишкових спрацьовувань методом NMS/Soft–NMS. Така процедура усуває дублювання рамок із високим взаємним перекриттям і забезпечує стабільність подальших розрахунків, зберігаючи для кожної локалізованої області лише представницький елемент із найбільшою довірою.

Вагові коефіцієнти складових функції втрат λ_{box}, λ_{obj} і λ_{cls} обрано на основі рекомендованих базових значень для YOLO-подібних детекторів і уточнено за результатами попередніх прогонів на валідаційній вибірці. Повний grid search не виконували, оскільки метою було отримати відтворюваний режим навчання з типовим балансом між локалізацією, objectness і класифікацією для розмітки (формула (3.14)).

Класові ваги w_k введено для компенсації дисбалансу між класами у навчальній вибірці, тобто рідкісніші класи мають більший внесок у функцію втрат і, відповідно, менший ризик недонавчання. Зокрема, $w_{Off-by-One}$ = 1.5 підвищує внесок цього класу у втрату через його нижчу представленість у даних. Пріоритезацію за "ризиковістю" реалізують окремо на етапі оцінювання та постоброблення, а не шляхом зміни ваг класів у функції втрат.

Базову оцінку довіри кожного передбачення формують як добуток імовірності об’єктності та максимальної класової імовірності; геометрія кадру та координати рамок при цьому не змінюються (інваріантність відносно початкової укладки з Етапу 1). Очищений набір після (Soft–)NMS, можна подати як:

$$P = NMS(\{(\hat{b}^j, s_j)\}), s_j = \hat{p}_{obj}^{\,j} \cdot \max_k \hat{p}_{cls}^{\,j,k}, \quad (3.18)$$

де $\hat{b}^j$ – рамка; $\hat{p}_{obj}^{\,j}$ – імовірність об’єктності; $\hat{p}_{cls}^{\,j,k}$ – імовірність класу k ∈ { Stack, Heap, Off–by–One}.

Другим кроком здійснюється граф-обізнане калібрування довіри, що узгоджує результати нейромережевої локалізації з формальними показниками ризику. Для кожної рамки $\hat{b}^j$, що залишилася після NMS, виконується відображення (проєкція) її області на відповідні елементи графового подання програмного коду (вузли та ребра DFG/CFG). Із цієї області витягуються числові індикатори ризику: локальна оцінка R_x, ланцюговий показник поширення ризику вздовж шляхів графа $P_{CR}(\pi)$ та агреговані оцінки на рівні фрагмента або програми $\bar{R}(G)$ за формулами (3.9)–(3.11). Оскільки наведені показники можуть мати різні шкали, спочатку формується єдиний скалярний показник ризику для рамки $\hat{b}^j$ у

вигляді $\tilde{R}_j$ (вибором одного з індикаторів або їх агрегуванням), після чого він нормується до інтервалу [0;1]. Нормоване значення p_j інтерпретується як "графова" складова довіри для детекції j і використовується для перерахунку початкового нейромережевого скору s_j. Таким чином, остаточна довіра s_j^* визначається як комбінація нейромережевої та графової складових:

$$s_j^* = s_j^{\alpha} \cdot p_j^{\beta}, p_j = norm(\tilde{R}_j) = \left(\frac{\tilde{R}_j - R_{min}}{R_{max} - R_{min} + \varepsilon}, 0{,}1\right), \quad \tilde{R}_j \in \{R_x, P_{CR}(\pi), \bar{R}\} \tag{3.19}$$

де j – індекс детекції після NMS; $\tilde{R}_j$ – обраний (або агрегований) показник ризику для області $\hat{b}^j$; norm(·) – фіксована схема нормування до [0,1]; α, β > 0 – гіперпараметри впливу "нейронної" та "графової" складових; $s_j \in [0;1]$ – початковий коефіцієнт довіри нейромережі для детекції j; R_{max}, R_{min} – обчислюються за навчальною (або валідаційною) множиною; ε - мале число для уникнення ділення на нуль.

Мультиплікативна форма s_j^* використовується для м'якого узгодження двох незалежних джерел впевненості, а саме оцінки детектора s_j та структурно-аналітичного ризикового індикатора p_j. Така комбінація зменшує вагу кандидатів, у яких високою є лише одна зі складових, і не дає одному сигналу домінувати над іншим, що можливе в адитивній схемі. У лог-просторі це відповідає зваженій сумі, що мають прозору інтерпретацію вагових коефіцієнтів. Навчене злиття (логістична регресія або MLP) не застосовується, щоб не вводити додатковий навчальний модуль і не підвищувати ризик перенавчання та залежності результатів від конкретної вибірки; натомість використано параметричне калібрування з малою кількістю ступенів свободи.

Третім кроком застосовується порогування скоригованих оцінок довіри з урахуванням класової специфіки. Для переповнень стеку, що потенційно спричиняють перезапис керуючих структур (адрес повернення), доцільно встановлювати суворіший поріг прийняття рішення, ніж для off–by–one, де чимало випадків мають обмежені наслідки. Порогові значення для класів фіксуються на

етапі валідації та не змінюються впродовж серій експериментів. У випадках, коли в зоні рамки підтверджено наявність захисних конструкцій (перевірки меж, canary), дозволяється додаткове послаблення скоригованого балу або повне відхилення спрацьовування за умови низького локального ризику, що узгоджується з безпековою практикою “пріоритезації” знахідок.

$$P = \{(\hat{b}^j, k_j, s_j^*) | s_j^* \geq t_{k1}\},\ k1 = arg \max_k \hat{p}_{cls}^{j,k}, \quad (3.20)$$

де t_{k1} – класові пороги (Stack/Heap/Off–by–One), узгоджені з вимогами безпеки.

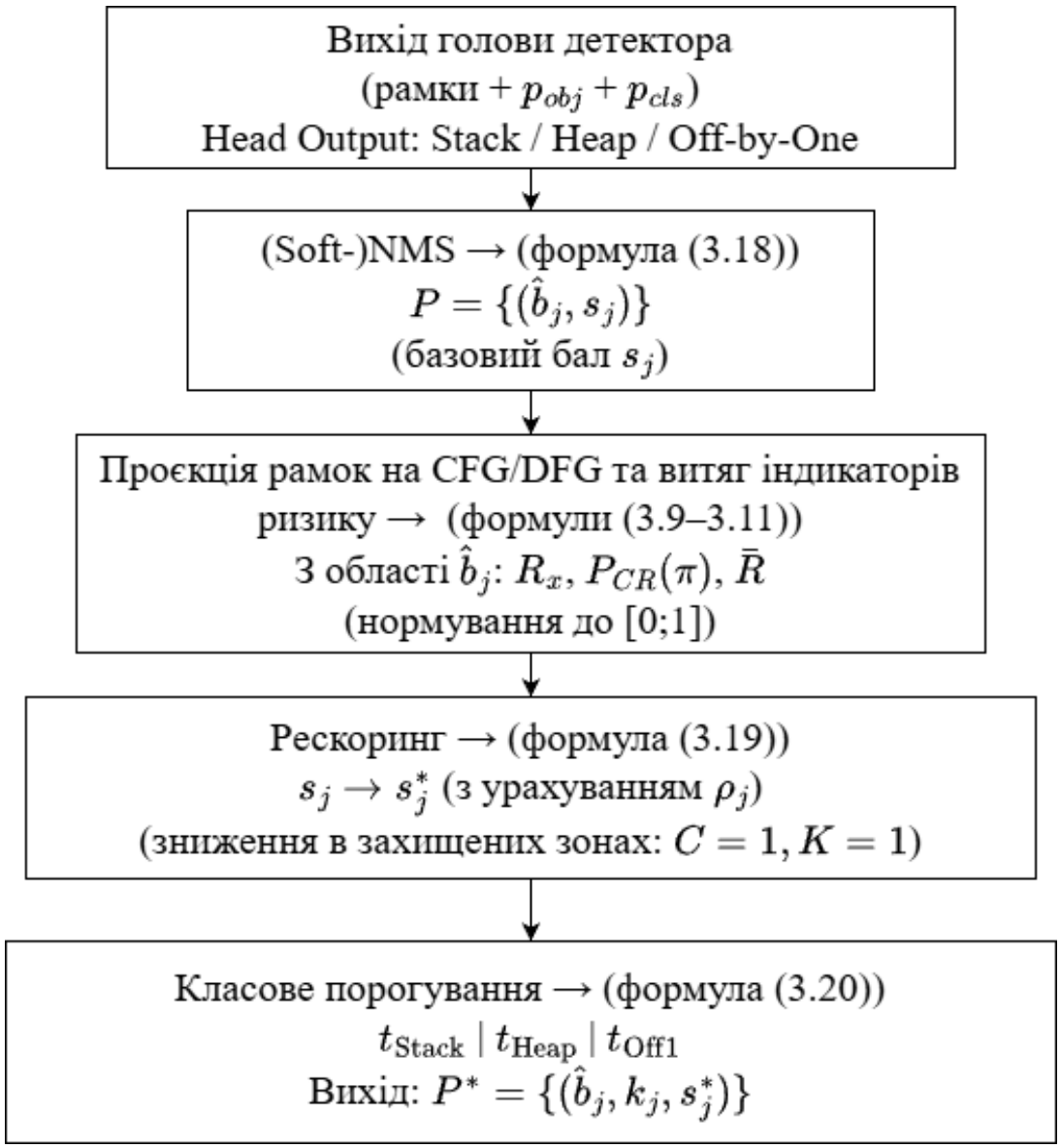


Рисунок 3.4 – Схема обчислень YOLO з багатоканальним входом і трьохкласовим виходом.

Етап 4. Оцінювання якості.

Оцінювання здійснюється в постановці детекції з локалізацією та трикласовою класифікацією (Stack / Heap / Off–by–One). Використовуються показники mAP@0.5 та mAP@0.5:0.95 по кожному класу і в середньому, а також mAR (mean Average Recall), PR–криві та матриці невідповідностей (confusion) для аналізу перетинів між класами. Окремо вводиться “ризик–обізнаний” показник mAP при risk-aware ранжуванні детекцій, у якому ранжування детекцій проводиться з урахуванням графових індикаторів ризику ρ (Етап 3). Протокол валідованого порівняння фіксується: спліт 70/15/15 формується на рівні проєктів (як у Методі 1), зерна випадковості (seed), версії даних і конфігурацій – незмінні між запусками.

Інтерпретація результатів проводиться в розрізі класів. Типові помилки включають:

1) перетини Stack/Heap на ділянках, де невірно інтерпретовано тип пам’яті (динамічні буфери в стекових обгортках);

2) злипання з Off–by–One, коли дрібні індексні зміщення дають слабкий просторовий сигнал;

3) хибні позитиви в присутності захисних гвардів (перевірки меж, canary), які базова модель без постоброблення сприймає як уразливі шаблони.

Застосування граф–обізнаного калібрування балу (Етап 3) і диференційованих порогів зменшує ці похибки, що відбивається на підвищенні mAP при risk-aware ранжуванні детекцій та F1:

$$mAP = \frac{1}{K}\sum_{k1}^{K}\int_{0}^{1} P_k(r)dr, \tag{3.21}$$

де K = 3 – кількість класів; $P_k(r)$ – прецизійність класу k як функція повноти r (mAP@0.5:0.95 обчислюється усередненням за порогами IoU); $\int_{0}^{1} P_k(r)dr$ є узагальненим записом площі під PR-кривою на відрізку $r \in [0;1]$.

Після введення граф-обізнаного калібрування довіри (формула (3.19)) змінюється ключовий етап побудови PR-кривої — ранжування детекцій. У стандартному mAP детекції впорядковуються за нейромережевим скором s_j. У risk-aware варіанті детекції впорядковуються за скоригованим скором s_j^* (або еквівалентною схемою, що враховує ризиковий множник p_j), тобто змінюється порядок проходження по детекціях і, як наслідок, змінюється сама PR-крива. Відповідно, risk-aware середня точність визначається аналогічно (формула (3.21)), але з використанням прецизійності, отриманої після ризик-орієнтованого ранжування:

$$mAP^{risk} = \frac{1}{K}\sum_{k1}^{K}\int_0^1 P_k^p(r)dr, \tag{3.22}$$

де P_k^p – прецизійність при ранжуванні детекцій із урахуванням ризикової ваги ρ (3.19), у тексті фіксуємо схему зважування (наприклад, сортування за s^* замість s).

Формули (3.21) та (3.22) відповідають стандартній процедурі обчислення mAP, тоді як "risk-aware" у цьому випадку полягає не в зміні визначення метрики, а у стратегії ранжування детекцій. Зокрема, сортування виконують за каліброваним скором s^*, який додатково враховує структурно-аналітичний індикатор ризику. Відповідно коректно розглядати наведене як протокол обчислення mAP за умови risk-aware ранжування, що дає змогу оцінити вплив калібрування $s \to s^*$ на якість ранжування детекцій і на підсумкові криві Precision–Recall.

Етап 5. Абляція та чутливість

Метою абляційного аналізу є кількісна оцінка внеску окремих компонентів Методу 2 та стійкості моделі до змін налаштувань. Оцінювання проводилося на валідаційному піднаборі при фіксованих seed, конфігураціях даних і незмінній процедурі підрахунку показників (той самий протокол, що на Етапі 4). За “базовий” варіант приймається модель після тюнінгу, із граф–обізнаним калібруванням балу, класовими вагами w, Focal–втратою для об’єктності, GIoU для рамок і розміром кадру 608×608.

Найбільший приріст якості забезпечує саме граф–обізнане постоброблення: вимкнення калібрування балу призводить до помітного зменшення mAP при risk-

aware ранжуванні детекцій і F1 (головно через хибні позитиви у “захищених” зонах та “злипання” Off–by–One). Внесок каналів безпеки (індикатори перевірок меж та canary) проявляється у зниженні FP для класу Stack; вилучення цих каналів нівелює перевагу, набуту на Етапі 3. Параметри калібрування балу впливають на баланс “нейронної” та “графової” складових: збільшення α робить рішення чутливішими до сирого балу моделі, тоді як збільшення β – до ризикового контексту; оптимальна пара ($\alpha = 1.0$, $\beta = 0.7$) була підібрана на валідаційній вибірці шляхом грубого перебору по сітці малої роздільності, де $\alpha \in \{ 0.7, 1.0, 1.3 \}$ та $\beta \in \{ 0.3, 0.5,0.7, 1.0 \}$, після чого обрано пару, що максимізує підсумкову метрику якості детекції (F1 або mAP) за фіксованих порогів IoU та впевненості. Перехід з GIoU на CIoU для рамок помірно покращує локалізацію дрібних фрагментів. Зменшення розміру кадру до 416×416 знижує mAP, що очікувано для локалізації дрібних структур. Вплив класових ваг w відповідає дисбалансу вибірки: відключення ваг погіршує показники Off–by–One та середні F1.

Таким чином, у межах методу автоматичного виявлення вразливостей на основі допрацьованої нейронної мережі YOLO розроблено та верифіковано детектор, що працює на багатоканальних кадрах X(H×W) і забезпечує одночасну локалізацію фрагментів коду та їхню трикласову класифікацію (Stack/Heap/Off–by–One). Запропонована процедура постоброблення з урахуванням графових індикаторів ризику приводить до пріоритезації результатів за скоригованим балом згідно формули (3.19), що зменшує частку хибнопозитивних спрацьовувань у “захищених” контекстах та підсилює значущі знахідки на критичних шляхах даних. Якість методу підтверджено емпірично за стандартними показниками детекції–mAP@0.5, mAP@0.5:0.95, mAR, F1–а також за “ризик–обізнаною” метрикою згідно формул (3.21) – (3.22), яка відображає корисність детекцій з огляду на безпековий контекст. Сукупно це забезпечує методологічно узгоджене й практично придатне рішення для включення в конвеєри статичного/гібридного аналізу та подальшої інтеграції у процеси забезпечення безпеки ПЗКС.

3.3 Метод захисту програмного коду комп’ютерних систем

Цей метод формалізує перехід від результатів детекції, отриманих попередніми методами, до керованого захисного рішення на основі композитної оцінки ризику. Його метою є не опис DevSecOps-процедур, а задання однозначного правила прийняття рішення у вигляді ланцюжка "виявлення → ризиковий бал → рівень критичності → дія". Формули (3.23)–(3.32) фіксують інтерпретовані критерії агрегування та порогування ризику, що забезпечує відтворюваність і порівнюваність результатів. Тому метою цього методу є перетворення результатів аналізу (отриманих у межах методів підготовки та опрацювання даних) на цілісну процедуру захисту програмного коду комп’ютерних систем. Йдеться про поєднання виявлених локальних та агрегованих індикаторів ризику з практиками виправлення коду, увімкненням превентивних механізмів на рівні компілятора й рантайму, а також із вбудованим контролем у процеси збірки та постачання ПЗ. Запропонований підхід орієнтований на класи переповнень пам’яті (Stack, Heap, Off–by–One), які притаманні системним компонентам, драйверам і RTOS, та забезпечує як зниження ймовірності виникнення помилки, так і обмеження наслідків її можливої експлуатації.

Метод ґрунтується на формальних критеріях (умова перевищення обсягу введення над місткістю буфера; індикатори граничних індексів; перевірка меж і наявність захисних “canary” для стеку) та використовує оцінки ризику й пріоритезації (локальні й ланцюгові показники, нормовані агрегати). На практичному рівні це дає змогу задати порогові правила реагування: де потрібне негайне виправлення коду (вставлення гвардів, перехід на безпечні API, корекція меж циклів), де – увімкнення або посилення компіляторних захистів (stack protector, Fortify–джерела, ASLR/DEP, UBSan/MSan/ASan), а де – процесні запобіжники в конвеєрі збірки (блокування злиття гілок, автоматичні тікети, вимога додаткового рев’ю).

Окрему увагу приділено інтеграції заходів захисту в життєвий цикл розроблення: автоматичні перевірки під час комітів і запитів на злиття, регресійні

тести для типових патернів переповнень, генерація звітів і показників зрілості безпеки для команд розробки. Таким чином, метод перетворює результати інтелектуального аналізу коду на відтворювану “сітку безпеки”, що поєднує виправлення на рівні вихідних текстів, параметри збірки та політики контролю якості у середовищах CI/CD. У підсумку забезпечується зменшення імовірності критичних відмов і підвищення стійкості системного ПЗ до спроб експлуатації переповнень пам’яті.

Етап 1. Вхідні дані.

Вихідні артефакти методу підготовки даних для виявлення переповнень пам’яті і методу автоматичного виявлення вразливостей на основі допрацьованої нейронної мережі YOLO приймаються як вхід до методу захисту. Йдеться про впорядкований набір локалізованих фрагментів коду, для кожного з яких визначено клас переповнення { Stack, Heap, Off–by–One }, координати у кадрі та прив’язку до елементів графового подання програми (вузлів/ребер DFG/CFG). Для кожного фрагмента наявні базові оцінки довіри детекції та їх граф–обізнані корекції, а також числові індикатори ризику. Сукупно ці дані зберігають відповідність умові перевищення обсягу введення над місткістю буфера згідно формул (2.5)–(2.6) та узгоджені з графовими залежностями програми. До складу вхідних даних входять:

1. Локалізовані фрагменти, тобто множина детекцій з класами { Stack, Heap, Off–by–One}, координатами рамок та посиланнями на відповідні функції/буфери/операції в коді.

2. Оцінки довіри, тобто базові бали детектора і скориговані (постобробкою) бали для пріоритезації.

3. Графові індикатори ризику, тобто локальні оцінки R_x, ланцюгові показники поширення ризику $P_{CR}(\pi)$ та агреговані величини $\bar{R}$, використані далі для вибору захисної реакції.

4. Прив’язка до графа програми: для кожної детекції задано відображення на елементи DFG/CFG, що забезпечує подальше застосування захисних дій у правильному контексті коду (даний/виклик).

Для відтворюваності фіксуються службові метадані: ідентифікатор коміту/версії коду, конфігурації інструментів аналізу, параметри побудови графів і нормування ризикових показників. Така постановка гарантує, що подальші кроки методу захисту діятимуть над строго визначеними об’єктами, пов’язаними з формальними критеріями (зокрема, формули (2.5)–(2.6)), і не виходитимуть за межі прийнятої моделі.

Для забезпечення відтворюваності, фіксують службові метадані, такі як ідентифікатор версії коду або коміту, налаштування інструментів аналізу, параметри побудови графів і нормалізації ризикових показників. Такий підхід забезпечує, що наступні етапи методу захисту будуть спрямовані на чітко визначені об'єкти, які відповідають формальним критеріям (зокрема, згідно з формулами (2.5) і (2.6)), та не перевищуватимуть межі затвердженої моделі. Взаємозв’язок артефактів трьох методів і послідовність їх перетворень (детекції → прив’язка до графів → ризикові індикатори → калібрування балу → рішення в CI/CD) зображено на рис. 3.5.

Етап 2. Класифікація критичності та вибір захисної реакції.

На цьому етапі виконується нормалізоване агрегування ризику для кожного виявленого фрагмента коду із подальшим присвоєнням рівня критичності та визначенням захисної дії. Агрегація спирається на локальну оцінку ризику та показники передавання ризику графом залежностей, що узгоджуються з формальними засадами (ланцюгова передача ризику, сумарні/нормовані індикатори). Відповідно, визначимо композитний ризик як опуклу комбінацію складників, після чого врахуємо клас вразливості (Stack/Heap/Off–by–One) через вагові коефіцієнти й застосуємо порогові правила для присвоєння рівня критичності:

$$R^* = y_1 R_x + y_2 \max_{\pi} P_{CR}(\pi) + y_3 \bar{R}\,, \qquad y_i \geq 0, \sum_{i=1}^{3} y_i = 1, \tag{3.23}$$

де R_x – локальний ризик фрагмента (визначений у Методі 1 на основі співвідношення введення/місткості, перевірок меж та наявності типових патернів), $P_{CR}(\pi)$ – максимальна по всіх релевантних шляхах умовна імовірність поширення

ризику за графом даних/викликів (узгоджено з формулою (2.33)), $\bar{R}$ – нормований агрегований ризик на рівні компонента/модуля коду (узгоджено з формулою (2.36)).

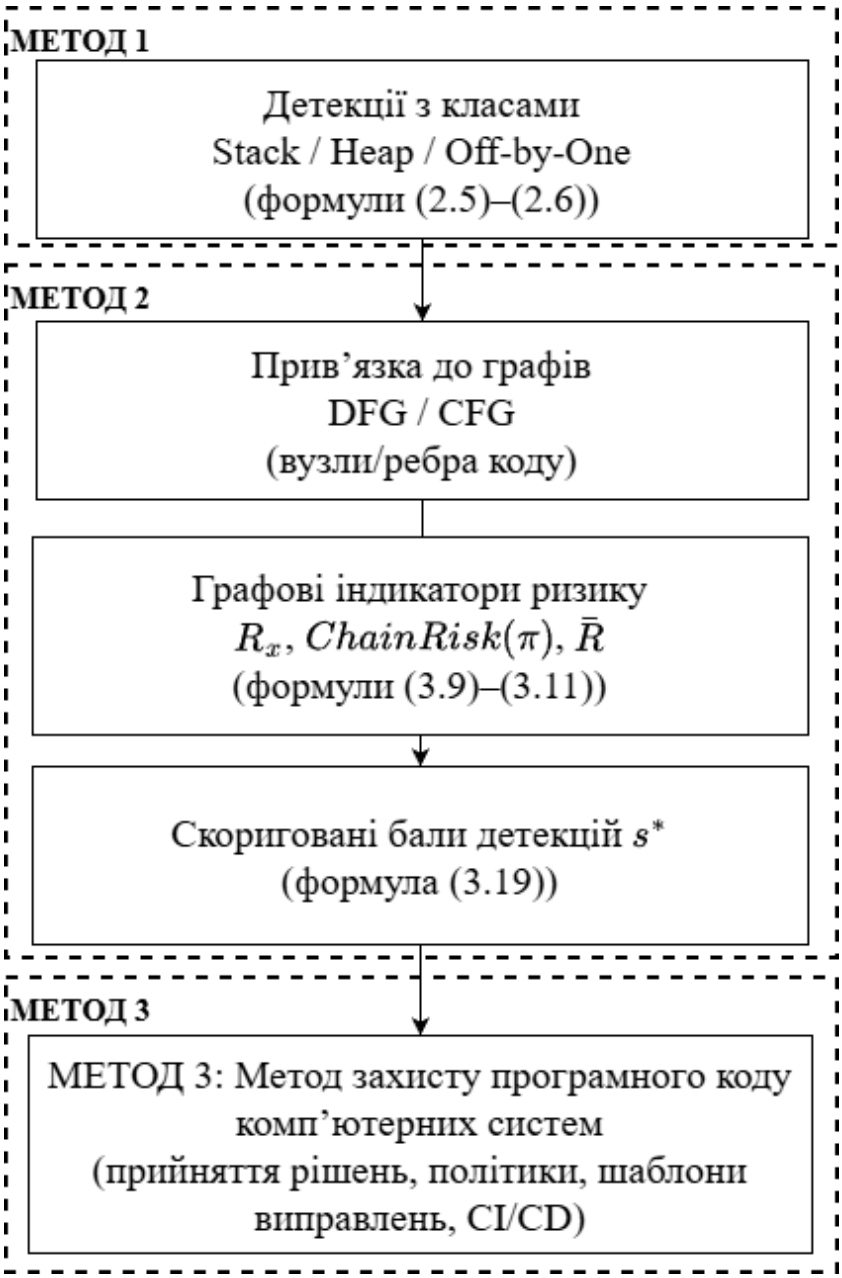


Рисунок 3.5 – Схема зв'язку трьох методів

Коефіцієнти y_1, y_2, y_3 задають відносну вагу локального фактора, шляхової передавальності та загального контексту ризику.

$$\tilde{R} = w_k \cdot R^*, k \in \{ Stack;\ Heap;\ Off - by - One\}, \tag{3.24}$$

де w_k – класова вага, що відображає різну безпекову значущість: як правило, $w_{Stack} > w_{Heap} > w_{Off-by-One}$ з огляду на підвищений ризик перезапису адреси повернення та прямі наслідки для потоку керування у випадку стекових переповнень.

Оскільки підсумковий ризиковий бал R, отриманий за формулами (3.23)–(3.24), є неперервною величиною, то для подальшого автоматизованого прийняття рішень у CI/CD його доцільно відобразити у дискретні рівні критичності. Це забезпечує однозначний вибір сценарію реагування (блокування, попередження або планове виправлення), тому введемо функцію критичності $F_S(x)$ за нормативними порогами, зокрема:

$$F_S(x) = \begin{cases} High, & \tilde{R} \geq T_{high} \\ Medium, & T_{med} \leq \tilde{R} < T_{high}, \\ Low, & \tilde{R} < T_{med} \end{cases} \quad (3.25)$$

де T_{high}, T_{med} – нормативні пороги для присвоєння рівня критичності.

Зважена агрегація у формулі (3.23) забезпечує монотонність підсумкової оцінки ризику, тобто зростання будь-якого чинника не зменшує результат, а також дає змогу інтерпретувати внесок окремих складових через відповідні ваги. Дискретизацію до рівнів High/Medium/Low у формулі (3.25) застосовано для однозначного вибору сценарію реагування, і вона узгоджується з подальшими критеріями прийнятності (PASS), наведеними у формулах (3.30)–(3.32).

Після визначення F_S(x) встановлюються рекомендовані дії:

1. High - обов’язкове блокування коміту/PR, автоматична генерація тікета з прив’язкою до вузлів DFG/CFG, вимога термінового виправлення (вставлення гвардів, перехід на безпечні API, корекція меж індексації) до повторного проходження перевірок.

2. Medium - попередження з обов’язковими рекомендаціями щодо виправлення; дозвіл на злиття лише після коду–рев’ю або підтвердження тестами; фіксація у звіті.

3. Low - фіксація у звіті та планування виправлень у межах регулярного обслуговування (без блокування злиття).

Для практичного застосування параметри $y, w_k, T_{high}, T_{med}$ задаються в конфігурації засобу контролю якості коду та документуються у табл. 3.7.

Таблиця 3.7 – Пороги та ваги для критичності

Клас k	Вага w_k	T_{high}	T_{med}	Коментар
Stack	1.00	0.70	0.40	Найвища критичність; ризик перезапису адреси повернення; пріоритет блокування
Heap	0.85	0.70	0.40	Середня критичність; корупція купи/керованих структур; вимога рев'ю
Off–by–One	0.65	0.70	0.40	Найнижча серед трьох; здебільшого індексні помилки; планове виправлення

Етап 3. Генерування рекомендацій і шаблонів виправлень.

На цьому етапі результати класифікації та оцінювання ризику перетворюються на конкретні дії щодо внесення захисних змін у код. Вибір шаблону виправлення залежить від класу вразливості (Stack/Heap/Off–by–One), типу пам'яті, співвідношення обсягу введення та місткості буфера, а також від наявності/відсутності перевірок меж і механізмів контролю ((2.5)–(2.6) для умови "ввід > місткість", (2.12) для stack canary, (2.18)–(2.20) для off–by–one). Пропоновані нижче формалізовані "вимикачі" керують застосуванням відповідних шаблонів:

$$G_{bounds}(x) = \mathbb{I}[(2.5)\text{або}(2.6) \wedge C(x) = 0], \tag{3.26}$$

де $C(x)$ – індикатор наявності коректної перевірки меж для операції/буфера x.

Якщо $G_{bounds}(x) = 1$, обов'язкове додавання гварда меж або перехід на безпечний API.

$$G_{off-by-one}(x) = \mathbb{I}[\text{порушено } (2.18) - (2.20)], \tag{3.27}$$

де порушення означає використання індексів −1 або n для буфера довжини n, або еквівалентні помилки індексації. Якщо $G_{off-by-one}(x) = 1$), обов'язкова корекція меж циклів/індексів.

У межах цього методу захисту ключовим є перетворення виявленого ризику на конкретні зміни в коді, які гарантовано нівелюють умови експлуатації. Розглянемо типову ситуацію для стекового переповнення. Нехай у фрагменті присутній виклик копіювання рядка до фіксованого буфера на стеці, причому під час підготовки даних та опрацювання зафіксовано порушення співвідношення “обсяг введення > місткість буфера” згідно з формул (2.5)–(2.6) та відсутність коректного гварда меж. У такому разі тригер (формула (3.26)) набуває одиничного значення, і метод генерує рекомендацію: перед виконанням операції копіювання необхідно явним чином перевірити довжину вхідних даних або перейти на безпечний API, що імпліцитно виконує контроль. З погляду захисту стеку доцільним є додаткове увімкнення механізмів контролю цілісності кадру виклику (stack protector), що корелює з формули (2.12) й знижує імовірність успішного перезапису адреси повернення у разі залишкових дефектів.

Ілюстративно наведемо мінімальне перетворення коду. Якщо вихідний фрагмент має вигляд:

```
strcpy(dst, src);
```

то патч, згенерований методикою, або перевіряє довжину явним чином, або замінює виклик на безпечний еквівалент. У першому випадку інваріант “ввід не перевищує S_b буфера” забезпечується безпосередньо, у другому – через обмежувальну семантику функції форматованого запису:

```
size_t cap  = sizeof(dst);
size_t need = strlen(src);

if (need >= cap) {
   /* Обробка помилки / усічення */
   snprintf(dst, cap, "%s", src); /* контроль меж і термінації */
} else {
   memcpy(dst, src, need + 1);
}
```

Таким чином, умови згідно формул (2.5)–(2.6) стають хибними вже на рівні вихідного тексту, а ризик, який було оцінено як високий з огляду на клас “Stack”, знижується до прийнятного рівня.

Для переповнення купи логіка аналогічна, однак увага зосереджується на коректності розрахунків виділення пам’яті та на відсутності арифметичних переповнень, що прямо відповідає інтерпретації ваги ребер і місткості приймача в формулі (2.8) у поєднанні з формул (2.5) та (2.6). Припустимо, що виявлено фрагмент, де обсяг копіювання може перевищити реально виділений блок з купі. У цьому випадку згідно методу необхідно виконати два послідовних кроки: верифікація розрахунку розміру до виділення (зокрема, перевірка добутків параметрів на відсутність переповнення цілих); співвіднесення фактичної довжини копіювання з відомою місткістю цільового буфера. Приклад такого виправлення ілюструє наступний фрагмент:

```
/* До:
   char* p = malloc(n);
   memcpy(p, in, len);   // len може перевищувати n
*/

/* Після: перевірки відповідають (2.5)–(2.6) у куповому контексті */
if (n == 0 || len > n) {
   /* Обробка помилки або обмеження len до n */
   /* ... */
}
char* p = (char*)malloc(n);
if (!p) {
   /* Обробка помилки виділення */
   /* ... */
}
memcpy(p, in, len);  /* інваріант: len <= n (узгоджено з S_b) */
```

Застосування такого патча безпосередньо розриває ланцюг поширення ризику у графі (зменшує внесок у $P_{CR}(\pi)$), оскільки порушення “джерело → буфер–приймач” на ребрі копіювання більше не виконується.

Окремої уваги потребують помилки класу off–by–one. У підготовленому наборі даних вони відмічались за індикаторами граничних індексів, формалізованих у формулах (2.18)–(2.20). Якщо виявлено доступ до елементів за межі [0, n−1] – зокрема, унаслідок використання умови i <= n у голові циклу – метод згідно формули (3.27) пропонує мінімальне втручання у верхню межу. Наприклад:

```
/* До: при i == n відбувається вихід за межі */
for (size_t i = 0; i <= n; ++i) {
    dst[i] = src[i];
}

/* Після: верхню межу зведено до інтервалу [0, n–1] згідно з (2.18)–(2.20) */
for (size_t i = 0; i < n; ++i)  {
    dst[i] = src[i];
}
```

У такий спосіб відновлюється інваріант індексації й усувається сама можливість доступу до n–го елемента, якого в масиві немає. Ефект від такого виправлення відображається не лише на локальному показнику R_x, а й на агрегованій оцінці $\bar{R}$, оскільки усувається типова причина хибних спрацьовувань у подібних шаблонах.

Етап 4. Інтеграція у CI/CD (блокування/допуск, звіти, тікети).

Інтеграційний підмодуль розгортається у конвеєрі збирання як автоматизований крок, що викликає послідовно метод підготовки даних для виявлення переповнень пам’яті та метод автоматичного виявлення вразливостей на основі допрацьованої нейронної мережі YOLO для кожного коміту або pull request. На вході він отримує фіксований знімок коду (ідентифікатор коміту/гілки), будує графові подання, формує багатоканальні кадри та запускає модель

локалізації/класифікації. Далі результати передаються до методу захисту програмного коду комп’ютерних систем, де обчислюється композитний ризик, враховується клас вразливості та призначається рівень критичності. Після цього застосовується політика допуску, формується звіт (SARIF/HTML) і, за потреби, автоматично створюється тікет на виправлення. Така постановка забезпечує відтворюваність (усі параметри фіксуються у конфігурації конвеєра), мінімізує ручні дії та унеможливлює випадкове ігнорування ризикових змін.

Рішення щодо допуску змін приймається на основі уніфікованих правил. У разі високої критичності злиття блокується до усунення дефекту; для середньої критичності генерується попередження, додається коментар до PR із координатами, класом, графовими індикаторами й рекомендованим патчем; у разі низької критичності система фіксує зауваження й пропонує внести зміни в планове обслуговування. Всі рішення та артефакти зберігаються у вигляді машиночитаних звітів, що спрощує аудит і регресійний контроль:

$$G(RP) = \begin{cases} BLOCK, & \exists x: severity(x) = High \\ WARN, & \neg\exists High \wedge \exists Medium \\ ALLOW, & \text{інакше} \end{cases}, \tag{3.28}$$

де severity(x) – рівень критичності, визначений у межах Метод захисту програмного коду комп’ютерних систем.

Попереднє правило встановлює рішення "тут і зараз". Проте для ефективного управління процесом усунення дефектів у рамках DevSecOps потрібно також визначити терміни реагування відповідно до рівня критичності проблеми. Запроваджується функція SLA(x), яка асоціює з кожним виявленим фрагментом x певний часовий норматив для його виправлення. У цій роботі ми використовуємо трирівневу шкалу термінів, задану формулою:

$$SLA(x) = \begin{cases} 24 \text{ год}, & severity(x) = High \\ 7 \text{ днів}, & severity(x) = Medium, \\ 30 \text{ днів}, & severity(x) = Low \end{cases} \tag{3.29}$$

де $SLA(x)$ – цільовий термін усунення виявленого фрагмента x.

Інтеграція рішень (формула (3.28)) та стандартів (формула((3.29)) у процес розробки, а також розташування методів аналізу і створення артефактів (звітів, коментарів до PR, задач на виправлення), проілюстровано на рис. 3.6.

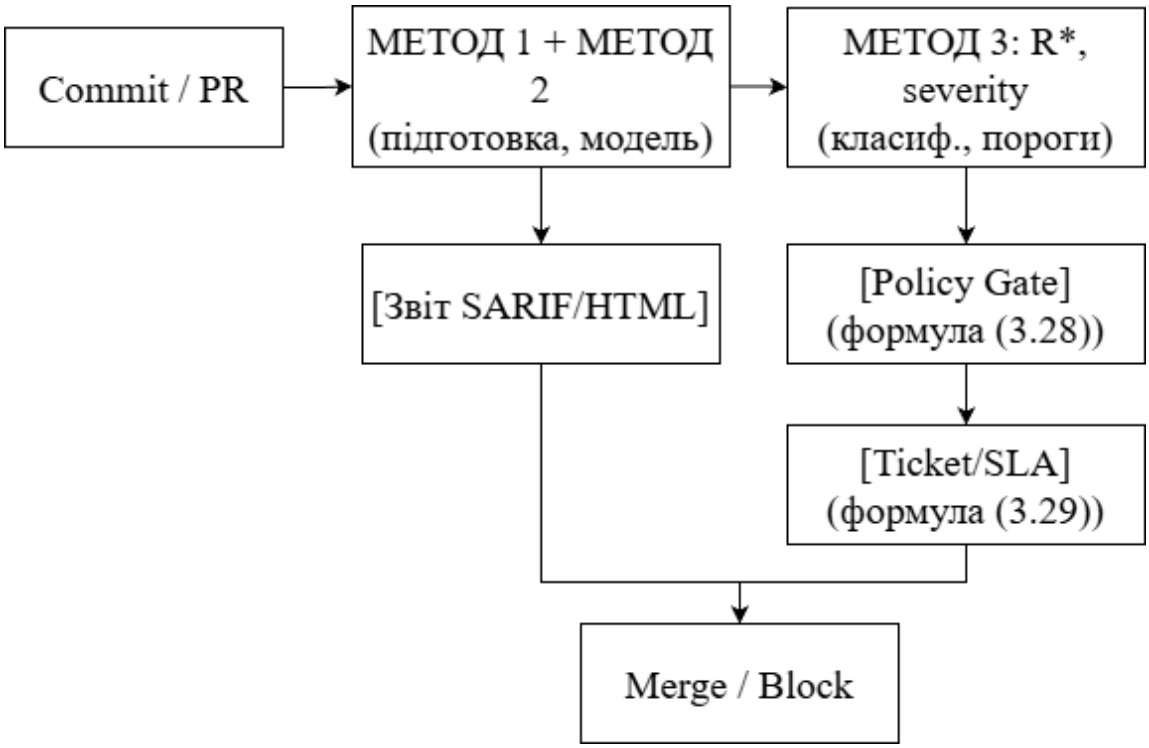


Рисунок 3.6 – CI/CD конвеєр з гейтами безпеки

Етап 5. Контроль ефективності (після виправлень) і зворотний зв’язок.

Після внесення змін у початковий код виконується повторний прогін методу підготовки даних для виявлення переповнень пам’яті та методу автоматичного виявлення вразливостей на основі допрацьованої нейронної мережі YOLO з подальшим застосуванням методу захисту програмного коду комп’ютерних систем. Метою є підтвердження, що порушення умов згідно формул (2.5) і (2.6) усунено, та кількісно зафіксувати зменшення ризику як на локальному рівні (для виправлених фрагментів), так і на агрегованому рівні (для модуля/проєкту). Результати повторної оцінки зберігаються в історії репозиторію разом із ідентифікатором коміту, версіями конфігурацій і параметрами аналізу, що забезпечує відтворюваність аудиту.

Для кількісної оцінки впливу виправлень використовують показники, що відображають зниження загального та нормованого агрегованого ризику. Ці показники розраховуються як різниця між оцінками «до» і «після» внесення змін:

$$\Delta R = R_{\text{до}} - R_{\text{після}},\ \Delta \bar{R} = \bar{R}_{\text{до}} - \bar{R}_{\text{після}}, \tag{3.30}$$

де $R_{\text{до}}, R_{\text{після}}$ – відповідно сумарний ризик до та після виправлень; $\bar{R}_{\text{до}}$, $\bar{R}_{\text{після}}$ – нормовані агреговані оцінки (узгоджено згідно формули (2.41)).

Оскільки регресійна перевірка безпеки повинна забезпечувати не тільки формальне "усунення" детекції, але й реальне покращення ризикового профілю, встановлюється пороговий критерій для оцінки прийнятності результатів. Зміни відносять до категорії, що пройшла регресійну перевірку безпеки (PASS), якщо було досягнуто мінімально необхідного зниження як загальної, так і нормованої оцінок:

$$PASS \Leftrightarrow \Delta R \geq \varepsilon_R \ \wedge\ \Delta \bar{R} \geq \varepsilon_{\bar{R}}, \tag{3.31}$$

де $R, \bar{R}$ – відповідно сумарна та нормована оцінки ризику; $\varepsilon_R, \varepsilon_{\bar{R}}$ – проєктні пороги прийнятності поліпшення (залежні від класу ПЗ).

Критерій PASS сигналізує успішне проходження регресійної перевірки безпеки).

Критерій PASS вказує на успішне завершення регресійного тестування безпеки і підтверджує, що внесені виправлення зменшили ризик до рівня, який відповідає прийнятним стандартам. Для забезпечення дотримання термінів усунення критичних знахідок (SLA) та оцінки дисципліни реагування запроваджується показник ефективності закриття висококритичних фрагментів у визначені строки:

$$KPI_{SLA} = \frac{\#\{x :\ High \text{ закрито вчасно}\}}{\#\{x : High\}}, \tag{3.32}$$

де High – рівень критичності за правилами Методу 3, закрито вчасно – виправлено і повторно підтверджено PASS до спливу терміну SLA.; KPI_{SLA} – частка висококритичних знахідок (High), закритих у межах встановленого SLA згідно формули (3.29) за обраний період спостереження.

Місце проведення регресійного тестування в процесі розробки та його зв'язок з гейтами безпеки ілюструється на рис. 3.7.

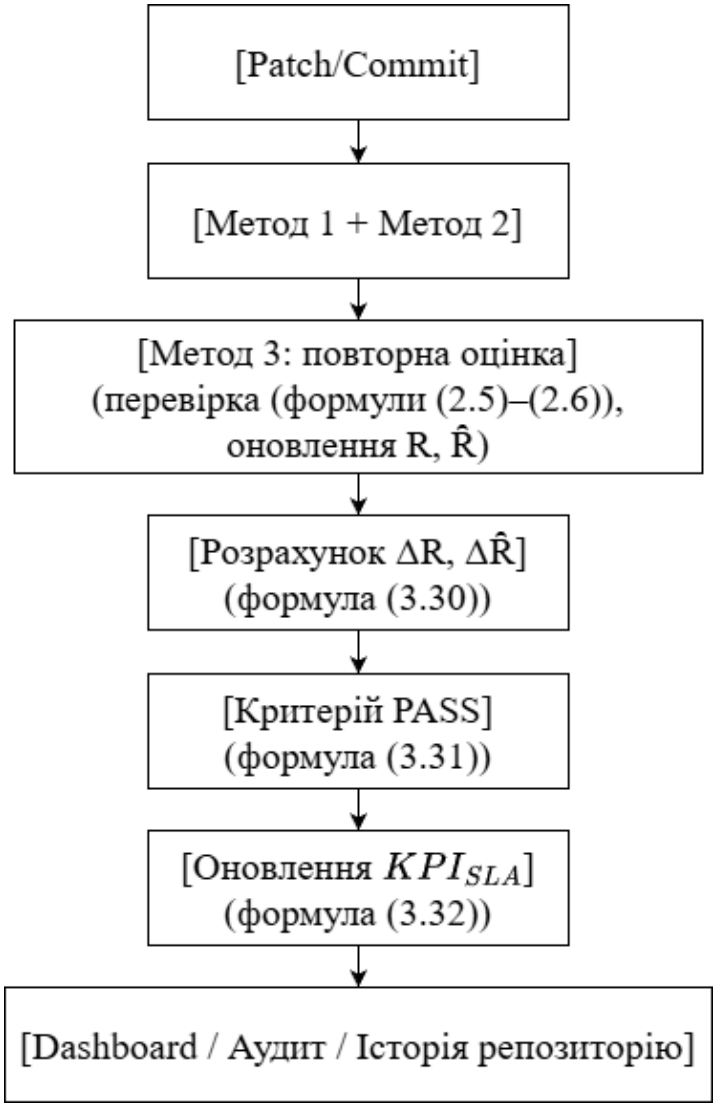


Рисунок 3.7 – Цикл зворотного зв’язку

Запропонований спосіб завершуватиме логічний цикл: від формалізованого підходу до підготовки представлень коду та виявлення переповнень пам’яті до їх системної нейтралізації в процесі розроблення. Ключовим результатом є перетворення оцінених показників ризику на керовані інженерні дії: присвоєння рівня критичності на підставі локальної та ланцюгової компонент ризику (узгоджено з апаратом розд. 2), генерація шаблонів виправлень для конкретних місць у коді, автоматизоване застосування політик допуску у конвеєрах автоматизованого збирання та розгортання та подальший контроль ефективності змін. Така побудова не вводить нових понять чи формальних залежностей понад

використані у Методу 1 та Методу 2: використовуються ті самі індикатори ризику, їх агрегування та правила прийняття рішень, що забезпечує методологічну цілісність і простежуваність.

Практична значущість методу полягає у превентивному характері захисту: переповнення пам’яті виявляються та нейтралізуються на етапі коміту або перевірки запиту на злиття, до потрапляння змін у продукційні збірки. Це забезпечується інтеграцією у конвеєр розроблення з чітко визначеними “гейтами” (блокування, попередження, допуск) і стандартизованою звітністю (SARIF/HTML) з координатами фрагментів, класами (Stack/Heap/Off–by–One), графовими індикаторами та рекомендованими патчами. Додатково фіксуються терміни усунення дефектів за рівнем критичності, що переводить вимоги безпеки у вимірювані операційні зобов’язання (SLA) і підвищує технологічну дисципліну команди.

Метод забезпечує відтворюваність та аудитованість: усі параметри (ваги, пороги, конфігурації), версії моделей і результати повторних вимірювань зберігаються в історії репозиторію, що дозволяє відслідковувати динаміку показників, порівнювати ефективність виправлень та підтверджувати стабільність детектора. Окремо підкреслимо, що зменшення ризиків оцінюється як на локальному рівні виправлених фрагментів, так і на агрегованому рівні модулів/проєкту, що унеможливлює “косметичні” зміни, які не впливають на загальний стан безпеки.

Запропонована процедура є масштабованою: вона однаково застосовна до системного ПЗ (ядра ОС, драйвери, RTOS, бібліотеки) та до вбудованих рішень на C/C++/Arduino, оскільки спирається на графові моделі та універсальні критерії “ввід/місткість” і індексні інваріанти. Водночас метод природно узгоджується з процесами забезпечення якості: результати можна включати до регресійних перевірок, контрольних списків рев’ю та внутрішніх стандартів кодування без необхідності перегляду математичних засад, викладених у попередніх підрозділах.

Нарешті, метод задає основу для подальшого розширення: автоматизоване накладання патчів (із напівавтоматичним підтвердженням), розширення словника

безпечних замін API, поглиблена кореляція з метриками продуктивності та надійності, а також уточнення порогів і ваг на підставі накопиченої статистики. Таким чином, Метод 3 не лише завершує запропонований цикл аналізу та реагування, а й утворює практичну рамку для безперервного підвищення зрілості процесів захисту програмного коду.

3.4 Висновки до третього розділу

Таким чином, представлено та обґрунтовано прикладну частину запропонованого методу для виявлення вразливостей, пов'язаних з переповненням буфера в програмному забезпеченні комп’ютерних систем. Було реалізовано метод підготовки даних. У цьому процесі початковий код формалізується за допомогою графових представлень програми (CFG/DFG), які включають атрибути для вузлів і ребер, а також індикатори ризику, що дає можливість чітко пов'язувати потенційно небезпечні частини коду з елементами графа і надалі застосовувати ці зв’язки в аналізі. Згідно з результатами обробки даних, створюється організований набір локалізованих фрагментів, які мають такі класи: {Stack, Heap, Off–by–One}. Кожен фрагмент містить координати його розташування у кадрі та прив'язку до вузлів або ребер DFG/CFG, а також з основними та уточненими оцінками довіри до виявлення і числовими показниками ризику, узгодженими з формалізаціями попереднього розділу.

У розробленому другому методі представлено інформацію про використання та налаштування нейромережевого детектора, що заснований на архітектурі YOLO. Також описано процеси навчання і валідації, а також етапи постобробки результатів і оцінювання їх якості, включаючи аналіз чутливості та абляційні дослідження. Для гарантії відтворюваності експериментів було зафіксовано набір гіперпараметрів для навчання, визначено функцію втрат і аугментації, а також інформативно описано апаратну платформу, на якій проводилося навчання (зокрема конфігурацію GPU).

Третій метод описує порядок застосування отриманих детекцій для підкріплення захисних рішень: починаючи від створення вхідних артефактів і оцінки їх критичності до розробки рекомендацій та їх інтеграції у виробничий процес, зокрема шляхом впровадження в CI/CD у формі «гейтів» (блокування або дозволу), звітів та завдань у системах управління розробкою, а також із визначеним механізмом зворотного зв'язку для оцінки ефективності після виконання корекцій.

Тоді, створюється єдиний ланцюг: "графове представлення → підготовка даних → виявлення за допомогою YOLO → уточнення/пріоритезація з огляду на графи → інтеграція у процес розробки", що гарантує практичне використання запропонованих моделей та їх націленість на впровадження в реальних процесах розробки програмного забезпечення комп’ютерних систем.

Основні результати розділу опубліковані у працях [120; 121; 126; 127; 152 – 154; 156].

РОЗДІЛ 4
ВИЗНАЧЕННЯ ЕФЕКТИВНОСТІ МОДЕЛЕЙ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНИХ СИСТЕМ

4.1 Опис системи та місця кожного методу

Початковими даними для системи є вихідні тексти ПЗКС (ядра ОС реального часу, драйвери пристроїв, низькорівневі утиліти). Одиницю аналізу задаємо як фрагмент коду, що відповідає функції/процедурі або локалізованому вікну коду, прив'язаному до підграфів керування та даних. Для забезпечення відтворюваності фіксуються ідентифікатор коміту, конфігурація препроцесора та версії інструментів. Структуру експериментального конвеєра та взаємодію реалізованих модулів наведено на рис. 4.1.

Рисунок 4.1 – Експериментальний конвеєр: від коду ПЗКС до рішення

На першому етапі виконується статичний аналіз коду: побудова абстрактного синтаксичного дерева (AST); графа потоку керування (CFG); графа потоку даних (DFG) з анотаціями вузлів і дуг. У процесі аналізу застосовуються формалізовані шаблони трьох узагальнених класів вразливостей (Stack/Heap/Off–by–One). Вони визначають характерні сигнатури підграфів та використовуються для первинного відбору кандидатних фрагментів без переозначення критеріїв (формула (2.3)).

На другому етапі формується ознакове подання підграфів у двох альтернативних форматах, сумісних із подальшим автоматизованим аналізом.

1. Зображальне представлення розміру 416×416.

Підграфи перетворюються на тензор фіксованого розміру 416×416×18. Канальний склад містить: маски вузлів і дуг; типи операцій у вузлах (операції доступу/копіювання в пам’ять, арифметика індексів, перевірки меж); типи дуг (керування/дані) та їх напрям; локальні топологічні індикатори (ступінь, належність до циклу, відстані до точок входу/виходу); службові позиційні коди (ранг у топологічному порядку, рівень домінаторного дерева); числові індикатори, що узагальнюють локальні критерії ризику для кожного з трьох класів. Перед передаванням у подальші модулі тензор нормалізується по каналах до узгоджених діапазонів; для розріджених реброві масок застосовується легке згладжування, що не змінює топологічної коректності відображення.

2. Векторно–послідовне представлення.

Ознаки вузлів/дуг та лінійні фрагменти AST пакуються у послідовності з масками заповнення; за потреби вони можуть укладатися у псевдозображення (channel–wise) для однакового інтерфейсу з модулем виділення ознак. Нормалізація здійснюється per–feature з фіксацією статистик.

Маркування фрагментів виконується згідно з правилами: для класифікації призначається мітка $y \in \{$ Stack, Heap, Off–by–One $\}$. Якщо використовується просторове зображальне подання, додатково фіксуються вікна локалізації у координатах 416×416 (або текстові інтервали для послідовного варіанта), що уможливлює спільну локалізацію та класифікацію.

Параметри “якорів” для детектора ініціалізуються за статистикою розмірів вразливих фрагментів: на навчальній підвибірці виконується кластеризація методом k–means; отримані центри інтерпретуються як базові розміри якорів для багатомасштабних карт ознак (формула (2.4)). Прив’язування фрагмента до якоря надалі здійснюється за критерієм перетину У(IoU), причому остаточна регресія зсувів і розмірів, а також оцінка “об’єктності” та класу виконуються на етапі детектування. Для попередження витоку даних під час формування навчальних/валідаційних/тестових підвибірок застосовується проектно–орієнтоване розшарування: фрагменти з одного проекту або модуля не розподіляються між різними підвибірками. У CI/CD–сценаріях додатково формується службова потокова підвибірка для оцінювання часових характеристик (затримка, пропускна здатність).

На межі етапів фіксуються інтерфейси даних. Вихід статичного аналізу подається у вигляді структурованих файлів з ідентифікаторами фрагментів, посиланнями на джерельні рядки та серіалізованими підграфами; вихід модуля формування представлень – у вигляді тензорних пакетів (для зображального варіанта) або послідовних матриць ознак (для векторного варіанта) разом із метаданими нормалізації. Для подальшого аудиту система веде журнал параметрів препроцесора, версій аналізаторів, статистик розрідженості каналів, частотних профілів класів і результатів k–means. Таким чином, підсистема підготовки даних виконує перехід від початкового коду до стандартизованих вхідних представлень, придатних для єдиного конвеєра локалізації–класифікації вразливостей.

У системі використовуються три узагальнені класи вразливостей: Stack Overflow, Heap Overflow та Off–by–One Error. Їх побудову виконано на основі аналізу відкритих репозиторіїв уразливостей (зокрема, CVE/NVD) та формалізованих критеріїв. Було сформовано єдиний процес: від підготовки даних (AST/CFG/DFG) до класифікації саме цих трьох класів детектором, причому вибір класів і їх специфіка прямо зафіксовані (див. “три узагальнені класи” та їх оброблення у конвеєрі). Також уточнено, що класифікація виконується на рівні мітки $y \in \{ 0, 1, 2 \}$ і метрик для кожного класу окремо.

Формальні моделі виконують дві функції в системі:

1. Еталонні шаблони (ground truth) для маркування навчальних даних.

На етапі підготовки даних шаблони трьох класів застосовуються до підграфів CFG/DFG для відбору “кандидатних” фрагментів і присвоєння їм міток класу. У разі зображального подання додатково фіксуються вікна локалізації (або текстові інтервали). Це забезпечує узгодженість між джерелами CVE/NVD, формальними критеріями та тими представленнями, які споживає детектор.

2. Інтеграція в конвеєр виявлення для валідації спрацювань і пріоритезації.

Після інференсу детектора результати локалізації–класифікації звіряються з формальними шаблонами як з нормативною “маскою” відповідності: по–перше, для відсіву хибнопозитивних спрацювань; по–друге, для обчислення інтегральних ризикових показників з метою подальшої пріоритезації виправлень.

Місце у системі та інтерфейси.

1. На вході етапу маркування доступні: (а) підграфи CFG/DFG з анотаціями; (б) правила відповідності класам; (в) службові карти відповідності фрагментів коду.

2. На виході формується Labels–пакет (клас у, локалізація), який використовується: для навчання/валідації детектора; як базис для післяпроцесинґу спрацювань і звітності.

3. Будь–які деталі обчислювальних критеріїв, їхні індекси та доведення не переозначаються, що унеможливлює самоповтор.

Тому, формальні моделі класів є основою як для коректного маркування навчальних даних, так і для подальшої валідації та агрегування результатів інференсу в єдиному конвеєрі.

У запропонованій системі застосовано одноетапний детектор типу YOLO, адаптований до вхідних представлень, отриманих із графів потоку керування та потоку даних. Таке рішення уможливлює спільну локалізацію підозрілих фрагментів коду та їх класифікацію на три узагальнені класи вразливостей у єдиному контурі оброблення. На вхід детектора подається тензор фіксованого розміру, що відображає структуру підграфів та пов’язані з ними ознаки; вихід

містить координати локалізованих фрагментів (у просторі вхідного подання або у вигляді інтервалів індексів), оцінку наявності об’єкта та імовірнісний розподіл по класах.

Розглянемо послідовність перетворень усередині моделі. На першому етапі працює модуль виділення ознак (backbone), який формує багатомасштабні карти ознак зі зменшенням просторової роздільності шляхом згорток із нормалізацією та нелінійністю SiLU. Крок згортки дорівнює двом, завдяки чому послідовно утворюються три рівні просторових карт із типовими розмірами 52×52, 26×26 та 13×13. Саме на цих рівнях акумулюються локальні структурні закономірності, що притаманні помилкам доступу до пам’яті й індексації: операції читання/запису буферів, модифікації індексів, перевірки меж, характерні переходи керування. Таким чином забезпечується чутливість як до дрібних патернів (зокрема off–by–one), так і до контексту, необхідного для коректного розпізнавання переповнень стеку та купи.

Перейдемо до етапу багатомасштабного злиття. Для покращення стійкості до різнорозмірних фрагментів та зменшення пропусків застосовується комбінація спрямованих з’єднань “згори–вниз” і “знизу–вгору” (PAN–FPN). Суть підходу полягає у перенесенні контекстної інформації з грубших рівнів на дрібні, а також у збагаченні грубших рівнів деталями, характерними для локальних структурних відхилень. Унаслідок такого злиття кожен масштаб отримує збалансоване поєднання локальних і глобальних ознак, що підвищує вірогідність коректного виділення фрагментів різної довжини та складності.

Розглянемо блок прийняття рішень. На кожній карті ознак застосовується детектор, який для кожної комірки сітки та кожного попередньо визначеного “якоря” оцінює параметри локалізованого фрагмента, рівень об’єктності та вектор імовірностей по трьох класах. Ініціалізація якорів виконується за статистикою характерних розмірів фрагментів, отриманою з навчальної підвибірки; така ініціалізація полегшує подальшу регресію зсувів і розмірів, зменшуючи похибку на старті навчання. Після обчислення виходів для всіх масштабів здійснюється

відбір неперекриваючих спрацьовувань за допомогою відсіювання перекривних спрацьовувань, що формує остаточний перелік кандидатів.

Зупинимось на критеріях оптимізації. Під час навчання використовується сумарна функція втрат, що поєднує три компоненти: для регресії координат локалізації – IoU–орієнтовану втрату (зокрема її варіанти на кшталт CIoU); для ознаки “об’єктності” – бінарну крос–ентропію; для класифікації за трьома класами – крос–ентропію багатокласового розподілу. З урахуванням дисбалансу частот класів до класифікаційної складової вводяться вагові коефіцієнти, що зменшують перевагу більш поширених випадків і стабілізують збіжність. Така комбінація забезпечує одночасне налаштування точності локалізації та надійності віднесення до відповідного класу.

Нарешті, зазначимо організацію вихідних даних. Для кожного вхідного фрагмента формується набір передбачень із координатами локалізованого підграфа (або інтервалів індексів), оцінками імовірностей по класах та числовою оцінкою об’єктності. У ході післяоброблення результати впорядковуються за впевненістю, дублікати усуваються, а кожен відібраний елемент збагачується посиланнями на початковий код і службовими ознаками, потрібними для подальшого аналізу та звітності.

У підсумку описана конфігурація забезпечує необхідну для задачі здатність до спільної локалізації та класифікації, зберігаючи багатомасштабний баланс контексту і деталей, а також враховуючи практичні аспекти навчання за умов дисбалансу класів і варіативності розмірів цільових фрагментів.

Після локалізації та первинної екстракції ознак одноетапним детектором типу YOLO відповідні фрагменти коду подаються на послідовну обробку комбінованою архітектурою, яка поєднує згорткові шари, рекурентні шари типу LSTM та механізм уваги. Така побудова дає змогу одночасно врахувати локальні структурні патерни та довгострокові залежності у межах фрагмента.

Спочатку здійснюється згорткова обробка. Розглянемо, як вона функціонує. Вхідні тензори ознак, які отримані на етапі детекції, проходять крізь компактний каскад згорток з нормалізацією та нелінійністю. Згортки агрегують локальні

патерни, характерні для помилок роботи з пам’яттю та індексації (послідовності читання/запису, перевірки меж, модифікації індексів), і водночас зменшують розмірність простору ознак. На цьому етапі формується послідовне представлення – впорядкований ряд векторів ознак, що відповідають підфрагментам коду (наприклад, базовим блокам або токен–окнам), із зафіксованими масками паддінгу для вирівнювання довжин.

Перейдемо до моделювання довгострокових залежностей. Послідовність ознак подається до двонапрямленого LSTM, який акумулює контекст як у прямому, так і в зворотному напрямках. Завдяки цьому мережа коректно враховує залежності між віддаленими операціями (наприклад, обчисленням індексу та його подальшим використанням, чи підготовчими впливами на структури керування пам’яттю). Вихід LSTM подається у вигляді ряду векторів прихованого стану, узгоджених із масками, щоб виключити внесок службового паддінгу.

Розглянемо застосування механізму уваги. Над послідовністю прихованих станів обчислюються ваги важливості, які відтіняють елементи, що найбільше впливають на прийняття рішення. Увага зосереджується на підфрагментах із типово ризиковими ознаками (наприклад, операції запису в буфер із неузгодженим обчисленням довжини або перевищенням меж). Зважена агрегація (наприклад, контекстний вектор як зважена сума станів) підсилює корисний сигнал і зменшує вплив другорядних ділянок коду, що безпосередньо не визначають клас.

Підсумковий шар класифікації формує тривимірний вектор оцінок належності до узагальнених класів (Stack, Heap, Off–by–One). Для цього використовується повнозв’язний перетворювач над контекстним вектором, після чого застосовується Softmax–шар. Отримані імовірності є узгодженими з попередньою локалізацією: кожному локалізованому фрагменту відповідає власний вектор імовірностей, що полегшує подальше ранжування та інтеграцію результатів у звіт. Зазначена комбінація CNN–LSTM–Attention забезпечує баланс між чутливістю до локальних структурних порушень і здатністю враховувати довготривалі залежності у межах фрагмента, завдяки чому зменшується частка хибних спрацювань і підвищується точність класифікації. Технічно це досягається

через послідовний перехід від локальної згорткової агрегації до контекстного узагальнення рекурентним модулем та подальшого фокусування увагою на найбільш інформативних підпослідовностях.

Процес навчання організовано як відтворювану послідовність етапів із фіксацією джерел даних, параметрів і артефактів. Спершу формується навчальний набір. Розмітка фрагментів коду виконується за формальними шаблонами трьох узагальнених класів. Для зображального подання додатково задаються вікна локалізації у координатах вхідного тензора, а для послідовного – інтервали індексів. Розбиття на train/val/test здійснюється на рівні проєктів (ізоляція модулів між підвибірками). Дисбаланс класів компенсується ваговими коефіцієнтами у класифікаційній складовій функції втрат; альтернативно застосовується помірна повторна вибірка рідкісних прикладів лише в train без змін складу test. Аугментації генеруються так, щоб зберігати коректність відображення графів: для зображального варіанта – ізометричні перетворення в межах тензора, слабке “drop–edge/node” із контролем зв’язності, шумові варіації каналів; для послідовного – масковані підстановки токен–векторів, згладження та випадкове усічення в допустимих межах. Для кожного запуску фіксуються зерна ініціалізації та версії інструментів.

Налаштування гіперпараметрів охоплює конфігурацію якорів, геометрію входу, вибір функцій втрат і режим оптимізації. Кількість якорів визначається попереднім кластеруванням характерних розмірів фрагментів методом k–means; за замовчуванням використовується дев’ять якорів на трьох масштабах, узгоджених із пірамідою ознак. Розмір навчального зображення фіксується як 416×416 (за потреби – мульти–scale у вузькому діапазоні для підвищення узагальнюваності). Сумарна функція втрат поєднує IoU/CIoU–складову для регресії рамок, бінарну крос–ентропію для ознаки об’єктності та багатокласову крос–ентропію для класифікації; ваги класів у класифікаційній частині обчислюються з частот на train. Як оптимізатор застосовується AdamW або SGD із моментом. Початкова швидкість навчання калібрується під розмір батчу (лінійне масштабування). Використовується короткий warm–up і ступінчасте або косинусне зменшення, а

weight decay задається сталим малим значенням. Для стабілізації LSTM–компонент найкраще використовувати градієнт–кліпінг. Розмір батчу обирається за обмеженнями пам’яті (при потребі – градієнтне акумулювання). Кількість епох встановлюється експериментально з ранньою зупинкою за валідаційним macro–F1. Усі гіперпараметри та метадані фіксуються у конфігураційних файлах і журналах експериментів.

Фінальний етап включає післяоброблення детекцій і складання звіту. Застосовується NMS (Non–Maximum Suppression) для відсікання перекривних спрацьовувань. Для цього використовується поріг впевненості для первинної фільтрації кандидатів, а далі – поріг IoU для пригнічення дублювальних рамок (клас–агностичний або покласовий режим залежно від сценарію). Результати формуються як впорядкований перелік локалізованих фрагментів із координатами або індексами, оцінками імовірностей за класами, ознакою об’єктності та посиланнями на початковий код. Додатково включаються службові поля (ідентифікатори версій, параметри порогів, статистики аугментацій), що забезпечує аудит і відтворюваність обчислень.

На рівні системи (табл. 4.1) процес організовано як послідовний конвеєр від початкового коду до підсумкового звіту з локалізованими вразливими фрагментами та їх класами. На вході надходять вихідні тексти ПЗКС. Спершу здійснюється статичний аналіз. Для цього будується AST, формуються графи потоку керування (CFG) і потоку даних (DFG) з анотаціями вузлів і дуг. На цій стадії застосовуються формалізовані шаблони трьох узагальнених класів вразливостей, що відсіюють кандидатні фрагменти для подальшої обробки та задають первинні мітки.

Далі виконується формування вхідних представлень. Кандидатні підграфи перетворюються або на зображальні тензори фіксованого розміру, або на векторно–послідовні подання з масками. Для зображального варіанту забезпечується узгодження каналів (вузли, дуги, типи операцій, позиційні та топологічні індикатори), нормалізація та підготовка якорів (k–means) під багатомасштабні карти ознак.

Таблиця 4.1 – Реєстр кроків конвеєра

Крок	Метод/технологія	Вхід → Вихід
Статичний аналіз	AST, CFG/DFG, шаблони класів	Код → підграфи + мітки
Формування подань	Тензор 416×416 або послідовність; k–means (якорі)	Підграфи → тензори/послідовності
Локалізація–класифікація	YOLO (backbone–neck–head)	Тензор → рамки/інтервали + оцінки
Узгодження детекцій	NMS	Кандидати → неперекривні спрацьовування
Послідовна обробка	CNN–LSTM–Attention	Фрагменти → контекстні ознаки
Остаточна оцінка	Softmax	Ознаки → імовірності класів
Формування звіту	Експорт + метадані	Локалізації + класи → звіт системи

Наступною ланкою є одноетапний детектор типу YOLO. На багатомасштабних картах ознак здійснюється спільна локалізація та класифікація. Для кожної комірки сітки та якоря регресуються параметри рамок/інтервалів і обчислюються оцінки належності до трьох класів. Після інференсу застосовується процедура відсікання перекривних спрацьовувань (NMS), внаслідок чого формується узгоджений перелік локалізацій із оцінками впевненості.

Локалізовані фрагменти передаються на послідовну обробку комбінованою CNN–LSTM–Attention архітектурою. Згорткові шари виділяють локальні структурні патерни, двонапрямний LSTM акумулює довгострокові залежності, а механізм уваги фокусується на найбільш інформативних підпослідовностях. Підсумковий шар формує імовірнісний розподіл (Softmax) за трьома узагальненими класах для кожного локалізованого фрагменту.

На виході генерується звіт. Для кожного фрагменту подаються координати у зображальному поданні або у вигляді інтервалів індексів у коді, клас, ймовірності, оцінка об’єктності та службові метадані, що забезпечують відтворюваність (ідентифікатори комітів, конфігурації інструментів, параметри порогів). Таким

чином, кожен метод займає чітко визначене місце. Шаблони визначені на рівні відбору й маркування, YOLO – на рівні локалізації–класифікації, NMS – на рівні узгодження детекцій, CNN–LSTM–Attention – на рівні уточнення класифікації, а Softmax – на рівні формування підсумкових значень ймовірності.

Таким чином, сформовано цілісне подання про систему виявлення вразливостей у ПЗКС як про конвеєр взаємодіючих компонент: від статичного аналізу й графових подань коду до багатомасштабної локалізації та уточнювальної класифікації. Ключовим є поєднання формалізованих моделей вразливостей для коректної розмітки й контролю відповідності з глибинним навчанням для узагальнення й стійкості до варіативності коду. Архітектура детектора типу YOLO забезпечує спільну локалізацію і первинну класифікацію на основі піраміди ознак, що поєднує локальні та глобальні характеристики фрагментів. Подальша CNN–LSTM–Attention обробка враховує довгострокові залежності, фокусуючи увагу на інформативних підпослідовностях. Модульна побудова з чіткими інтерфейсами (дані, метадані, пороги) уможливлює інтеграцію з інженерними процесами та адаптацію до різних типів коду (RTOS–ядра, драйвери, утиліти) без зміни загальної логіки.

У підсумку, узгоджено формальних критерії і нейромережева обробка, що зменшує хибнопозитивні спрацьовування і підвищує інтерпретованість, використано багатомасштабні ознаки і якорі, що підвищує чутливість до різнорозмірних та різноконтекстних фрагментів, отримано здатність опрацьовувати як зображальні тензори на основі CFG/DFG, так і з послідовними представленнями, що підсилює переносимість між проєктами. Наявні чіткі точки збору метрик та журналювання, що забезпечує відтворюваність і контроль якості у виробничих сценаріях.

Встановлено такі обмеження: потрібна достатня кількість маркованих прикладів для усіх трьох класів і їхніх підтипів, бо дефіцит або дисбаланс даних призводить до перекосів у класифікації; складність навчання зумовлена багатокомпонентною функцією втрат і чутливістю до гіперпараметрів (якорі, пороги NMS, режим аугментацій), що потребує ретельного налаштування; наявний

ризик доменного зсуву між кодовими базами (інший стиль, макроси, профілі компіляції), що знижує узагальнювальну здатність; обчислювальні витрати (пам’ять, латентність) можуть бути критичними для інтеграції у CI/CD із жорсткими обмеженнями часу. Тому, для подолання зазначених обмежень потрібно збільшити навчальний набір за рахунок слабкої та напівавтоматичної розмітки (weak–/self–training), активного навчання та генерації синтетичних прикладів на рівні графових патернів, здійснювати балансування класів комбінацією ваг у функції втрат, таргетованих аугментацій та контрольованого oversampling лише на train, здійснити доменну адаптацію й перенавчання на проєкт–специфічних підвибірках, включно з регулярною валідацією на “холодних” репозиторіях, виконати оптимізацію моделей через дистиляцію знань, прунінг і квантизацію, а також калібрування ймовірностей і адаптивні пороги для стабільної роботи в потоці.

4.2 Методика отримання та відтворюваність результатів експерименту

Постановка експерименту. Розглянемо опис умов, за яких отримуються числові значення метрик, які включають склад інструментів, порядок вибірки та режиму навчання, щоб при необхідності, можна було повторити експерименти та перевірити стабільність результатів. Цей підхід відповідає вимогам до сучасних систем статичного аналізу, які завжди є виробничими конвеєрними, адже від результатів детектора залежать рішення щодо захисту безпечності ПЗКС.

Експерименти виконувалися на фіксованому обчислювальному стенді, конфігурація якого зберігалася незмінною протягом усіх серій запусків. Апаратна платформа включала центральний процесор серверного класу з багатоядерною архітектурою, графічний прискорювач із підтримкою тензорних обчислень, оперативну пам’ять обсягом не менше 32 ГБ та швидкий твердотільний накопичувач для розміщення кодових баз і проміжних артефактів. Програмне середовище будувалося на основі операційної системи сімейства Linux із фіксованими версіями ядра та драйверів графічного адаптера, а також бібліотеки

глибинного навчання. Компілятори та інструменти статичного аналізу встановлювалися один раз перед початком серій і надалі не оновлювалися. Для кожного компонента (фреймворк глибинного навчання, інструментарій побудови AST/CFG/DFG, SAST–засоби для порівняння) фіксувалися точні версії, які надалі зафіксовано в журналі експериментів. Така політика «замороженого» середовища дає змогу уникнути прихованих ефектів, пов'язаних із оновленням бібліотек чи зміненням поведінки компілятора, і розглядати розбіжності результатів як наслідок саме зміни архітектури моделі або складу даних, а не інфраструктури. Послідовність перетворення графового подання у зображувальний формат для подальшого аналізу комп'ютерним зором показано на рис. 4.2.

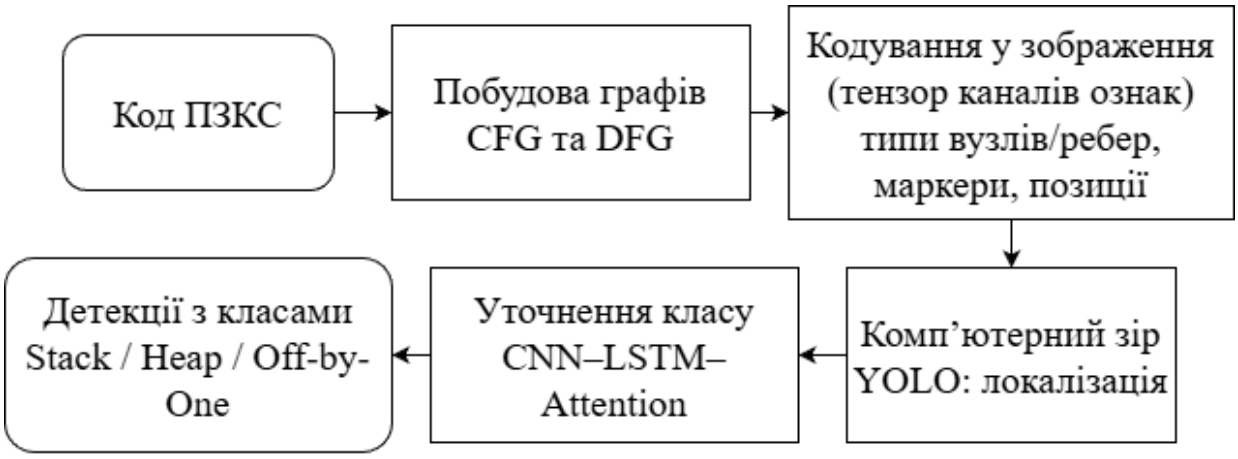


Рисунок 4.2 – Формування вхідних подань: граф як зображення для комп'ютерного зору

Для збірки використовувався єдиний протокол, який охоплював вибір, реконструкцію фрагментів та розподіл на підмножини навчання, валідації й тестування. Інциденти, що містили вразливості з таких відкритих репозиторіїв як CVE/NVD, були початковими джерелами. Для кожного запису, що потрапив під категорії, помилка на одну позицію або переповнення буфера, відтворювався мінімальний самостійний фрагмент коду з репозиторію постачання, збережений із контекстними залежностями (оголошення структур, допоміжні функції, макроси). Відкриті репозиторії систем реального часу та драйверів пристроїв, які першою дією автоматично проходились введеними шаблонами, а потім здійснювалась верифікація щодо наявності в загальних класах Stack Overflow, Heap Overflow та

Off–by–One, яка є другою групою джерел. Додатково операція формування була проведена шляхом генерації синтетичних прикладів коду, тобто створення патернів рідкісних випадків, за допомогою контрольованих трансформацій коду (зміна межових умов, довжин буферів, індексів циклів) на фрагменти, у яких інваріанти навмисно порушувалися, створювалися, зберігаючи при цьому загальну структуру алгоритму.

Розмітка виконувалася у двох взаємодоповнюваних аспектах. Спочатку кожному фрагменту призначався клас вразливості згідно з формалізованими критеріями. У випадку сумнівів фрагмент або вилучався, або переносився до службової вибірки для подальшого аналізу. Для підтримки локалізації фіксувалися точні межі вразливої ділянки. У випадку зображувального подання використовувались вікна в координатах тензора, які пов’язані з відповідним підграфом CFG/DFG. А у випадку послідовних ознак – інтервали індексів у початковому коді, тобто рядки та позиції, що відповідають операціям доступу до пам’яті, індексації або модифікації меж. Розбиття на навчальну, валідаційну та тестову підвибірки здійснювалося на рівні проєктів. Один і той самий репозиторій не міг одночасно бути джерелом для двох різних підмножин, що унеможливлює витік специфічних артефактів стилю коду чи макросистеми між train, val і test. Синтетичні приклади використовувалися виключно в навчальній підвибірці. На валідації та тесті застосовувався лише реальний код, що дозволяє об’єктивно оцінити узагальнювальну здатність моделі.

Подальший препроцесинг залежав від обраного типу подання, але підпорядковувався загальним принципам нормалізації каналів та контролю розрідженості. Для зображувального варіанту кожен відібраний підграф CFG/DFG відображався у тензор фіксованого розміру. При цьому, окремі канали відповідали типам вузлів (операції читання, запису, перевірки меж), типам дуг (потік керування, потік даних, залежності за адресою) та додатковим числовим атрибутам (оцінки місткості буфера, індексів, інваріантів циклів). Значення в каналах масштабувалися до уніфікованих діапазонів, а розрідженість контролювалася шляхом відсікання малозначущих елементів та, за потреби, локальної агрегації.

Для послідовного подання ознаки формувалися як впорядковані вектори, де кожному елементу відповідали локальні атрибути операцій над пам’яттю й індексацією. Послідовності вирівнювалися до допустимої довжини з використанням масок паддінгу, щоб мережа могла відокремити реальні елементи від службових.

Обґрунтування коректності та валідності експерименту. Політика аугментацій була відмінною для тензорних і векторних представлень, але в обох випадках зберігала коректність топології та семантики. Для зображувального варіанта застосовувалися допустимі ізометричні перетворення у межах тензора, випадкове “drop–edge/node” з контролем зв’язності та слабкий шум у каналах, який не порушував відповідність між координатами і структурами коду. Для послідовних ознак використовувалися випадкові маскування та підстановки векторів окремих токенів у межах одного типу операцій, помірне усічення неінформативних префіксів або суфіксів та згладжування числових атрибутів. Усередині детектору ініціалізація «якорів» для багатомасштабної сітки виконувалася методом k–means на статистиці розмірів еталонних локалізацій у навчальній підвибірці, що дозволяло узгодити форми й масштаби рамок із реальними патернами вразливостей. На кожному етапі препроцесингу збиралися агреговані статистики (розподіли довжин, діапазони значень у каналах, частка аугментованих прикладів), які зберігалися в журналі й використовувалися для контролю стабільності даних між різними серіями експериментів.

Також стандартизувався режим навчання моделі для всіх типів архітектури, які порівнювалися, щоб відмінність у результатах було однозначно асоційовано зміною мережі, а не зміною гіперпараметрів. Перед запуском кожної серії навчання фіксувався псевдовипадкове зерно у фреймворку глибинного навчання, бібліотеках лінійної алгебри та засобах формування батчів для послаблення варіативності, що пов’язана зі стирлізацією різних порядків вибірки. Для оптимізації використовувався стохастичний оптимізатор типу Adam або SGD із моментом у всіх випадках, застосовувався один і той же графік зміни лр – коротка фаза прогріву кроку градієнта у перші епохи із поступовим збільшення кроку

градієнта до номінального значення та подальшим поетапним зниженням при досягненні плато на валідаційній вибірці. Розмір батчу та, при необхідності, градієнт–акумуляція, обиралися так, щоб забезпечити стабільність збіжності за об’ємом пам’яті графічного прискорювача; критерій ранньої зупинки спирався на відстеженні macro–F1 на валідаційній підвибірці з заданим «вікном допустимості». Щоб контролювати доменний зсув, окрема частина проєктів залишалася холодною і використовувалася тільки на фінальній стадії оцінювання. Ці репозиторії не мали встановлення порогів та налаштування, забезпечуючи сприятливе оцінювання для реального узагальнення.

Після завершення навчання всі моделі оцінювалися за єдиним протоколом порогування та післяобробки. На першому етапі для кожної конфігурації будувалися валідаційні криві Precision–Recall, за якими підбиралися робочі значення порогу впевненості для класів. Для базових експериментів використовувався єдиний поріг, тоді як у розширених дослідженнях розглядався варіант покласового налаштування. Далі застосовувалася процедура супресії неперекривних рамок (Non–Maximum Suppression). У її реалізації розглядалися як клас–агностичний варіант, коли конкуруючими вважалися всі детекції незалежно від класу, так і клас–залежний, де рамки різних класів могли співіснувати в одному регіоні. Пороги перетину (IoU) для NMS також визначалися за результатами валідації, причому для локалізації коротких фрагментів використовувалися дещо менші значення, що дозволяло не «гасити» тісно розміщені, але різні прояви. Відповідність прогнозу еталону визначалася за поєднанням умов. Перетин інтервалів коду або досягнення достатнього IoU у зображувальному поданні, а також збіг класу. Лише за виконання обох умов детекція вважалася правильною.

Система метрик і статистичних оцінок була узгодженою з формальними визначеннями і застосовувалася однаково для всіх експериментальних сценаріїв. Для кожного з трьох узагальнених класів обчислювалися точність і повнота, на основі яких формувалася покласова F1–міра, додатково визначався macro–F1 як середнє арифметичне F1 за класами без зважування, що дозволяло уникнути домінування більш чисельного класу. Для оцінювання якості локалізації в частині

детектора використовувалися mAP@0.50 та mAP@0.50:0.95. Перша метрика відображала чутливість до правильного виявлення уразливих зон при фіксованому порозі перекриття, а друга – інтегральну якість за діапазоном порогів. Окремо фіксувалися часові характеристики (латентність інференсу на один фрагмент та на проєкт) і пікове споживання пам'яті, а також нормований показник хибнопозитивних спрацювань FP/1kLOC, який дає змогу оцінювати придатність системи до промислового використання з погляду трудомісткості ручної перевірки. Для перевірки стабільності результатів застосовувався бутстреп над множиною фрагментів. На кожній ітерації випадковим чином формувався піднабір тестових прикладів із поверненням, для якого повторно обчислювалися основні метрики. Таким чином оцінювалися довірчі інтервали й варіативність показників. Агрегація проводилася на рівні фрагментів і проєктів, а отримані значення використовувалися надалі для обговорення компромісів між точністю, повнотою, швидкодією та стійкістю до доменного зсуву.

Відтворюваність результатів експерименту. Окремим елементом методики стала можливість відтворення та аудиту результатів. Для цього після кожного запуску навчання або тестування записувався повний набір службової інформації в лог–файл експерименту. Це включало в себе ідентифікатори стану кодової бази – хеші комітів в СХВ для кожного репозиторію, що брав участь у формуванні вибірок, активні профілі пре–процесора зі списком певних макросів і конфігураційних прапорців, а також версії тулчейну, компіляторів, статичних аналізаторів, бібліотек глибокого навчання та допоміжних утиліт. В окремі файли виносилися і конфігураційні файли моделі та pipeline обробки, включаючи архітектуру мережі, гіперпараметри оптимізації, типи аугментацій та порядок їх застосування. Для етапу детекції та пост–обробки в лог–файл записувалися значення робочих порогів впевненості для кожного класу, параметри NMS, будь то тип, пороги IoU, та агреговані статистики аугментацій і розподілу ознак. Їх сукупність, що могла включати «сирі» коди, зафіксовані коміти, конфігураційні файли, логи запусків і зафіксоване оточення, була б достатньою для повного

повторення будь–якої серії експериментів на обраному стенді або на сумісній платформі.

Запропонована методика є необхідною і достатньою для відтворення всіх експериментів в умовах, близьких до вихідних. Жорстка фіксація середовища, дисциплінований протокол формування вибірок, уніфіковані режими навчання й порогування, а також детальне журналювання конфігурацій і результатів забезпечують прозорість процедур та дозволяють оцінювати зміни метрик виключно як наслідок модифікацій моделі або даних. Разом із тим межі валідності цієї методики визначаються характеристиками доступного стенду та складу даних. Результати безпосередньо стосуються аналізу системного C/C++–коду з близькими до розглянутих патернами вразливостей і можуть змінюватися за переходу до інших мов, вкрай обмежених апаратних ресурсів або закритих промислових кодових баз з нетиповою структурою. Зазначені обмеження слід враховувати при перенесенні підходу на нові домени. Саме тому у подальших підрозділах розглядаються питання стійкості до доменного зсуву та адаптації моделі до альтернативних контекстів.

4.3 Експерименти і аналіз результатів

Метою експериментальних досліджень є верифікація здатності розробленої системи коректно локалізувати та класифікувати вразливості трьох узагальнених класів, тобто Stack Overflow, Heap Overflow та Off–by–One, у початкового коді ПЗКС, а також з’ясування впливу архітектурних конфігурацій моделі на показники якості та обчислювальну вартість. У межах цієї мети необхідно порівняльно оцінити запропонований конвеєр із репрезентативними базовими підходами, що відображають практику виявлення вразливостей:

1) класичним статичним аналізом без використання глибинних моделей;

2) послідовними нейромережами типу LSTM без блоку просторової локалізації;

3) спрощеними згортковими класифікаторами, які працюють лише на агрегованих ознаках.

Таким чином, експериментальний план орієнтовано не на ізольоване вимірювання окремих метрик, а на дослідження переваги саме від інтеграції формальних моделей та глибинних методів локалізації–класифікації.

Дослідження проводемо у двох взаємодоповнюваних напрямах. Перший спрямовано на оцінювання детекційної спроможності системи за класами з урахуванням локалізації фрагментів коду. При цьому аналізуватимемо точність класифікації та повноту виявлення у межах кожного класу, а також інтегральні показники на сукупності прикладів із різних проєктів. Другий напрям стосується архітектурних абляцій. Послідовно розглядатимемо внесок окремих компонентів у кінцевий результат, причому якість інтерпретуватимемо разом із витратами ресурсів під час навчання та інференсу: багатомасштабного злиття ознак (PAN–FPN); рекурентного блоку LSTM; механізму уваги. Усі порівняння будемо виконувати за єдиним протоколом відтворюваності зі сплітом на рівні проєктів, щоб виключити витік унікальних артефактів між підмножинами та перевірити здатність моделі узагальнювати знання на нові репозиторії. Зведений план експериментів, групи порівнянь та набір метрик оцінювання наведено на рис. 4.3.

Сформулюємо чіткі робочі гіпотези, для яких експерименти повинні їх підтвердити або спростувати. Використання графових представлень коду та каналів, узгоджених із формальними критеріями забезпечує істотний приріст якості порівняно з підходами, що оперують лише послідовними або токенними ознаками. Ця гіпотеза перевіряється у зіставленні із “пласким” CNN–класифікатором та LSTM–мережею без просторової локалізації. Припустимо, що багатомасштабне злиття ознак у модулі PAN–FPN та механізм уваги в послідовній частині зменшують кількість пропусків насамперед для дрібних і коротких проявів типу Off–by–One та для фрагментів із розірваним контекстом. Відповідні абляції дозволяють кількісно відокремити внесок кожного компонента. Гіпотеза полягає в тому, що комбінована архітектура “YOLO для локалізації + CNN–LSTM–Attention для уточнення класу” забезпечує кращий баланс між precision і recall, ніж окремо

взяті модулі, і водночас утримує прийнятний час інференсу на практично значущих розмірах проєктів.

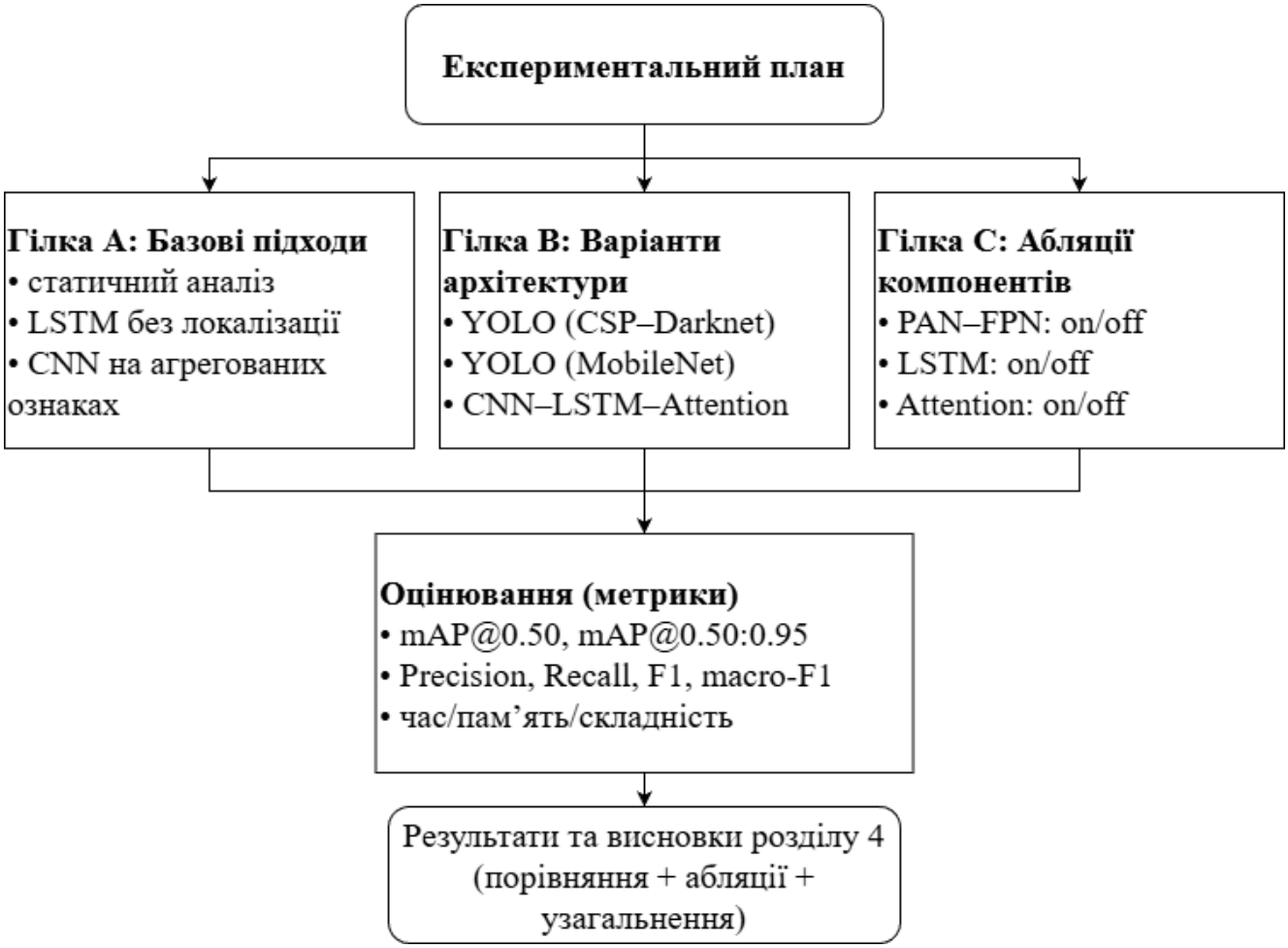


Рисунок 4.3 – План експериментів і групи порівнянь

Для інтерпретації результатів застосовуються узгоджені критерії оцінювання: середня точність детекції за порогами перекриття (mAP@0.50 та mAP@0.50:0.95) покласові Precision, Recall і F1 із довірчими інтервалами; оперативні характеристики (латентність інференсу на один файл/проєкт, пікове споживання пам'яті, кількість параметрів та умовних операцій (FLOPs) для різних конфігурацій). Отримані дані аналізуються як покласово, так і в агрегованому вигляді з урахуванням дисбалансу прикладів. Особлива увага приділяється стабільності показників на міжпроєктних тестах, що імітують реальне включення нових кодових баз у конвеєр. Підсумкове зіставлення з базовими методами дає

змогу обґрунтувати доцільність запропонованих рішень та виявити межі застосовності архітектури залежно від класу вразливості та ресурсних обмежень.

Набір даних сформуємо поєднанням трьох джерел, що відображають реальні сценарії виникнення вразливостей у ПЗКС. Для цього використаємо фрагменти коду, які отримані з описів інцидентів у загальнодоступних базах вразливостей (CVE/NVD). Для кожного запису відтворено мінімальний самодостатній фрагмент із відповідними контекстними залежностями, щоб коректно відобразити причину помилки. У залучених відкритих репозитаріях систем реального часу та драйверів пристроїв, як кодових базах, виділялятимемо кандидатів на вразливості за формальними шаблонами і після цього будемио виконувати ручну верифікацію відповідності класу. Для охоплення рідкісних патернів створено синтетичні приклади. У цих прикладах переповнення стеку або купи та помилки на одну позицію моделювалися за допомогою контрольованих змін в індексації, перевірок меж і копіювань у буфер. Ці приклади необхідні в якості допоміжних і включимо тільки до вибірки для тренування, тоді як основний корпус складається з реальних фрагментів системного програмного забезпечення, написаного мовами C/C++. Ці фрагменти були відібрані та марковані відповідно до формальних критеріїв трьох узагальнених класів: переповнення стека; переповнення купи; помилка на одиницю. Розмітку проводимо згідно з вказаними критеріями. Для графічного подання фіксуватимемо координати розташування вхідного тензора у просторі, а для послідовного – визначатимемо інтервали індексів у початковому коді. Ділення на підвибірки будемо здійснювати на рівні проєктів, що унеможливлює перетікання однакових фрагментів між тренувальними, валідаційними та тестовими наборами. Це гарантує перевірку здатності до узагальнення на нових репозиторіях. Крім того, ризик перенавчання знижується завдяки використанню тільки топологічно правильних аугментацій під час навчання, а також завдяки ранній зупинці за валідаційною метрикою (patience 15). Для реплікації наведемо зведений розподіл прикладів за класами (включаючи всі підвибірки) і обсягами підвибірок, які не перетинають проєкти. Розподіл прикладів за класами подано в табл. 4.2.

Таблиця 4.2 – Розподіл прикладів за класами

Клас	Кількість
Stack Overflow	1 260
Heap Overflow	930
Off–by–One	1 410
Разом	3 600

Окремо вибілимо обсяг підвибірок без врахування перетинів між проєктами: тренувальна вибірка складає 2 520 фрагментів (70 %); валідаційна – 360 (10 %); тестова – 720 (20 %). Частки класів у валідаційній та тестовій підвибірках відповідали тренувальній вибірці в межах ±1,5 %. Синтетичні приклади використовувалися лише для тренувальної підвибірки, щоб не впливати на незалежну оцінку. Узагальнювальна здатність підтверджується підсумковими показниками на тесті. Для конфігурації YOLO (CSP–Darknet) отримано macro–F1 = 0,82 та mAP@0.50 = 0,86. Обсяг підвибірок і частка від загалу подано в табл. 4.3.

Таблиця 4.3 – Обсяг підвибірок і частка від загалу

Підвибірка	Кількість	Частка
Train	2 520	70 %
Val	360	10 %
Test	720	20 %

Методика оцінювання включає як покласові, так і агреговані показники. Для кожного з трьох класів обчислюємо точність (Precision), повноту (Recall) і F1–міру. Додатково подається macro–F1 на сукупності класів. Для частини, що виконує локалізацію, використаємо середню точність детекції згідно mAP@0.50 і mAP@0.50:0.95, як стандартних інтегральних показників відповідності прогнозованих та еталонних інтервалів/“рамок”. Часові характеристики подамо як латентність інференсу на один фрагмент і на проєкт (медіана за серією запусків після короткого розігріву), а також як пікове споживання пам’яті під час інференсу. Формальні визначення зазначених метрик застосовуються як методологічна основа для порівняння конфігурацій і узгодженого підбиття підсумків.

У межах першого експерименту виконаємо навчання й тестування чотирьох конфігурацій класифікаторів та двох варіантів одноетапного детектора. Розглянемо чистий згортковий класифікатор як його послідовне розширення за рахунок LSTM, а також поєднання CNN, LSTM і механізму уваги. Окремо оцінимо детектор типу YOLO з опорними мережами CSP–Darknet і MobileNet. Для коректності зіставлення використаємо однакові підвибірки даних, уніфіковані аугментації та ідентичний протокол вибору порогів за валідаційними кривими “Precision–Recall”. Часові характеристики вимірюватимемо на тому самому стенді, а затримку інференсу визначимо як медіану серії запусків після короткого “розігріву”.

За результатами експерименту встановлено, що додавання LSTM і механізму уваги стабільно підвищує якість порівняно з базовим згортковим класифікатором. Комбінація CNN–LSTM–Attention демонструє помітний приріст для класу Off–by–One, де важливу роль відіграють довготривалі залежності й фокусування на ключових підпослідовностях. Разом із тим найвищі інтегральні результати забезпечує саме локалізаційно–класифікаційний підхід. Обидва варіанти YOLO перевершують суто класифікаційні моделі за macro–F1 та AUPRC, причому CSP–Darknet має невелику перевагу в якості, а MobileNet – у швидкодії. За ресурсних обмежень доцільно надавати перевагу варіанту з MobileNet як більш збалансованому; за пріоритету максимальної точності варто обирати CSP–Darknet.

Згідно експериментів аналізуються два різні підходи:

1) класифікація, що включає базові моделі, такі як CNN, CNN+LSTM;

2) CNN+LSTM+Attention.

Другий підхід представлений локалізацією з елементами класифікації (YOLO), який орієнтований на практичне застосування й дозволяє отримувати координати або інтервали фрагментів коду. Основною конфігурацією системи є каскад: YOLO → NMS → уточнювальна модель CNN–LSTM–Attention. Після цього проводять абляційний аналіз, щоб оцінити внесок FPN/PAN та attention при дотриманні одного й того ж протоколу навчання і тестування. Підсумкові метрики класифікації подано в табл. 4.4.

Таблиця 4.4 – Підсумкові метрики класифікації

Метод (Backbone)	macro–F1	AUPRC
CNN	0,69	0,74
CNN+LSTM	0,73	0,78
CNN+LSTM+Attention	0,77	0,82
YOLO (CSP–Darknet)	0,82	0,88
YOLO (MobileNet)	0,79	0,85
YOLO (MobileNet–Small)	0,77	0,82

Таким чином, локалізаційні характеристики підтверджують перевагу одноетапного детектора. Середня точність детекції для CSP–Darknet є найвищою з випробуваних варіантів, тоді як MobileNet забезпечує меншу затримку при незначній втраті якості. Це дозволяє адаптувати вибір конфігурації під конкретні вимоги – від офлайн–аналізу з акцентом на точність до інтеграції у CI/CD, де критичним є час спрацювання. Результати локалізації та швидкодії (YOLO) подано в табл. 4.5.

Таблиця 4.5 – Локалізація та швидкодія (YOLO)

Метод (Backbone)	mAP@0.50	GPU, мс
YOLO (CSP–Darknet)	0,86	9,4
YOLO (MobileNet)	0,83	6,1
YOLO (MobileNet–Small)	0,80	4,8

Для повноти інтерпретації подано модельні витрати. Зі зростанням глибини й ширини опорної мережі підвищуються кількість параметрів і обчислювальне навантаження. Відповідно, досягається дещо вищий рівень якості. Саме це й формує компроміс “якість–швидкодія”, який надалі використовується при виборі робочої конфігурації.

Порівняння складності моделей подано в табл. 4.6.

PR–криві підтверджують покращення якості при переході від CNN до моделей CNN+LSTM та CNN+LSTM+Attention. Водночас, варіанти YOLO показують найвищу площу під кривою для класів Stack і Heap. Для практичних випадків доцільніше вибрати YOLO (MobileNet), оскільки він є оптимальним

компромісом між якістю та швидкістю. Якщо ж необхідно досягти максимальної якості, слід звернути увагу на YOLO (CSP–Darknet).

Таблиця 4.6 – Складність моделей

Метод (Backbone)	Параметри, M	FLOPs, Г
CNN	5,2	8,1
CNN+LSTM	8,7	11,5
CNN+LSTM+Attention	11,3	14,2
YOLO (CSP–Darknet)	27,5	41,0
YOLO (MobileNet)	9,2	14,8
YOLO (MobileNet–Small)	4,6	7,2

У другому експерименті проведено абляційне дослідження. В якості базової конфігурації використовували повну каскадну модель, яка включає YOLO з багатомасштабним злиттям ознак та уточнювальну архітектуру CNN–LSTM–Attention, потім окремо виділялися FPN, PAN та attention, після чого навчання або тестування проходило за тією ж схемою.

Базова конфігурація (табл. 4.7) забезпечила узгоджений баланс між точністю і повнотою, а також найвищу середню точність детекції за порогами перекриття. Саме від цих значень відраховувалися зміни при вилученні компонентів.

Таблиця 4.7 – Базові метрики повної моделі

Показник	Значення
macro–F1	0,82
mAP@0.50	0,86
Recall (усереднений)	0,81
FP/1kLOC	7,2

Вилучення багатомасштабних гілок вплинуло насамперед на повноту для коротких і дрібномасштабних проявів (табл. 4.8). Втрата FPN призвела до помітного зниження виявлення класу Off–by–One, тоді як відсутність PAN негативно позначилася на узгодженні контексту для довших фрагментів, що характерно для переповнень стеку та купи. Механізм уваги в послідовній частині,

у свою чергу, істотно вплинув на відсікання другорядних підпослідовностей: без нього зросла частота хибних спрацьовувань за незначного просідання повноти.

Таблиця 4.8 – Зміна повноти порівняно з повною моделлю

Варіант	Stack	Heap	Off–by–One
–FPN	−1,9	−2,4	−3,8
–PAN	−2,7	−2,1	−1,2
–Attention	−0,8	−0,9	−1,4

Поведінка за рівнем хибнопозитивних детекцій (табл. 4.9) була найчутливішою до наявності уваги. Саме вона стабілізує рішення класифікатора на локалізованих фрагментах, “приглушуючи” слабкоінформативні відрізки коду. Водночас вилучення PAN безпосередньо погіршило узгодження багатомасштабного контексту, що проявилося у додаткових перекриттях та частіших спрацьовуваннях на шумових структурах.

Таблиця 4.9 – Хибні спрацьовування та агреговані метрики

Варіант	FP/1kLOC	Δmacro–F1 (п.п.)
–FPN	8,9	−2,1
–PAN	8,0	−1,6
–Attention	9,6	−2,5

З урахуванням результатів доцільність кожного компонента обгрунтовується так (табл. 4.10). FPN варто зберігати як обов’язковий елемент для стабільного виявлення дрібномасштабних аномалій індексації. Без нього систематично погіршується повнота саме для Off–by–One. PAN доцільно використовувати в конфігураціях, орієнтованих на максимально можливу повноту за збереження прийнятної швидкодії. Його внесок особливо відчутний для більш протяжних шаблонів переповнення з розірваним контекстом. Увага у послідовній частині є ключовим засобом зниження рівня хибних спрацьовувань. Її вилучення приводить до помітного зростання FP/1kLOC навіть за незначної втрати повноти, що погіршує

загальну придатність системи для інтеграції у конвеєри з обмеженнями на кількість ручних перевірок.

Таблиця 4.10 – Узагальнення доцільності компонентів

Компонент	Основний ефект	Рішення
FPN	Підтримка дрібномасштабних детекцій	Залишати як базовий
PAN	Узгодження глобального контексту	Рекомендовано для точності
Attention	Зменшення FP, стабілізація рішень	Критично з погляду якості

Отримані дані підтверджують, що багатомасштабна агрегація ознак і механізм уваги мають доповнювальний характер. Перша підвищує повноту, а друга знижує хибні спрацьовування. Саме їх поєднання у повній моделі забезпечує найкраще співвідношення “якість–швидкодія–трудомісткість ручної валідації” та обґрунтовує вибір повної конфігурації як базової.

Метою експерименту впливу способу подання даних є з’ясування, як вибір вхідного подання, тобто компактний вектор ознак чи двовимірне “зображення” з багатьма каналами, яке сформоване з CFG/DFG, впливає на детекційну спроможність та ресурсоємність системи. Для коректності зіставлення використано однаковий навчально–тестовий розклад і спільні процедури нормалізації. Векторний варіант подає послідовність ознак у комбіновану CNN–LSTM–Attention–мережу з легким локалізаційним head’ом по індексах коду. Зображувальний варіант подає багатоканальний тензор 416×416 до YOLO–детектора (MobileNet–класу) з подальшим уточненням класу тим самим послідовним блоком, що дозволяє ізолювати вплив саме подання.

Отримані результати (табл. 4.11) свідчать, що зображувальне подання систематично покращує локалізацію уразливих зон. Середня точність детекції зростає насамперед за рахунок коротких та “дрібних” проявів (типу Off–by–One), де просторове кодування графових зв’язків і “якорі” багатомасштабної сітки полегшують коректне прив’язування рамок. За класифікаційними показниками перевага також на боці зображувального варіанта, хоча розрив менший, ніж для локалізації. Додатковий контекст із каналів покращує як precision, так і recall, проте

ці виграші супроводжуються помірним приростом затримки та споживанням пам’яті. Ресурсні показники подано в табл. 4.12.

Таблиця 4.11 – Порівняння якості (векторне проти зображувального подання)

Подання	macro–F1	AUPRC	mAP@0.50	mAP@0.50:0.95
Вектор ознак (послідовне)	0,76	0,81	0,78	0,55
Зображення 416×416 (багатоканальне)	0,81	0,86	0,85	0,61

Таблиця 4.12 – Ресурсні показники (інференс на однаковому стенді)

Подання	GPU, мс	CPU, мс	Пам’ять, MB
Вектор ознак (послідовне)	5,2	165	430
Зображення 416×416 (багатоканальне)	6,4	198	560

У табл. 4.12 наведено часові витрати інференсу та обробки подання на стенді, тоді як повна end-to-end затримка конвеєру (парсинг → побудова CFG/DFG → растрування → інференс → постоброблення) додатково визначається вибраним фронтендом або парсером і розміром коду. Вимірювання end-to-end латентності з розподілом за етапами розглядається як практичне розширення експериментальної частини для конкретних CI/CD-стендів.

Інтерпретація цих відмінностей є узгодженою з очікуваннями від архітектури. Просторове подання дає детектору можливість безпосередньо “бачити” топологію залежностей у вигляді каналів і працювати з багатомасштабною сіткою якорів. У результаті mAP@0.50 зростає з 0,78 до 0,85, а узагальнений mAP@0.50:0.95 – з 0,55 до 0,61. Перевага у macro–F1 (з 0,76 до 0,81) проявляється найсильніше на класі Off–by–One, де локальні “зсуви на один” краще виявляються після просторової агрегації ознак. Водночас додаткова обробка багатоканального тензора потребує більше пам’яті й додає ≈1,2 мс до затримки на GPU та близько 30 мс на CPU порівняно з послідовним векторним варіантом.

З практичного погляду вибір подання залежить від обмежень застосування. Якщо пріоритетом є максимально точна локалізація з мінімальною кількістю пропусків, тобто зображувальний шлях є кращим. Якщо ж ключовим є бюджет

часу/пам’яті (наприклад, інтеграція у жорсткі CI/CD–конвеєри або аналіз на обмежених вузлах), то векторний варіант залишається привабливим, забезпечуючи прийнятну якість при нижчих витратах. У розглядуваному випадку виграш у mAP та macro–F1 виправдовує помірні додаткові витрати, тож зображувальне подання доцільно приймати як основне, а векторне – як легковагову альтернативу для ресурсно обмежених сценаріїв.

Метою наступного експерименту з порівняння аналогів є оцінка результатів розробленої системи не тільки в контексті класичних методів статичного аналізу, а й в аспекті сучасних нейромережевих технологій, що відображає сучасні тенденції у виявленні вразливостей у коді за допомогою машинного навчання та глибокого навчання протягом останніх років. Для представлення були обрані два поширених SAST–інструменти Clang Static Analyzer і Cppcheck, а також трансформерна модель GraphCodeBERT, яка була адаптована через доопрацювання для задачі багатокласової класифікації типів переповнення буфера (Stack/Heap/Off–by–one) на сформованій у роботі вибірці. Оцінка проводилася на тій же тестовій підвибірці, що й у попередніх випадках. Для SAST–інструментів і запропонованого детектора відповідність перевірялась за однаковим протоколом, що включав оцінку спрацювань з еталонними позначками. Це охоплювало відповідність класу та узгодження локалізації з референтним стандартом, тоді як для GraphCodeBERT, який не має чіткої просторової локалізації, метрики обчислювалися на рівні фрагмента. Прогноз типу вразливості для певного фрагмента визнавався правильним.

Обрані засоби для порівняння відображають різні категорії підходів, що дає змогу не лише чисельно аналізувати результати, а й оцінювати їх з методологічної точки зору. Clang Static Analyzer та Cppcheck є прикладами статичних засобів аналізу безпеки коду (SAST), які генерують конкретні повідомлення про помилки на основі шаблонних правил, символьного виконання і евристичних методів. Таке порівняння за показниками Precision, Recall та macro–F1 є адекватним способом оцінки реальної різниці. Статичні аналізатори зазвичай мають високу вибірковість щодо канонічних патернів, але можуть втрачати точність у випадках із складними

залежностями даних та управлінням, тоді як освітній підхід може заповнити деякі з цих прогалин, але за умови помірного рівня варіативності та необхідності у позначених даних. Для практичного контексту також представлене порівняння (табл. 4.13) за FP/1kLOC і часткою пропусків (FN). Ці показники суттєво впливають на обсяг ручної валідації та загальні витрати впровадження методу у виробничому процесі.

Таблиця 4.13 – Зведені метрики (агреговані за класами)

Підхід	Precision	Recall	F1 (macro)
Наша система (YOLO MobileNet)	0,82	0,78	0,80
GraphCodeBERT	0,78	0,75	0,77
Clang Static Analyzer	0,83	0,54	0,65
Cppcheck	0,75	0,49	0,59

У наукових публікаціях широко представлені графові нейронні мережі для виявлення вразливостей у програмному коді, зокрема Devign, ReVeal, IVDetect та LineVD. У цих роботах вихідні дані подаються у вигляді графів керування та залежностей (CFG/DFG/PDG), а модель формує графове або вузлове подання для подальшого прийняття рішення. Водночас більшість таких підходів орієнтована на класифікацію на рівні функції або фрагмента ("вразливий або невразливий", тип вразливості) чи на ранжування вузлів. На відміну відомих підходів було здійснено спочатку локалізацію підозрілих областей у представленні R(G), що за форматом результату відповідає задачам детектування об'єктів. Після цього локалізована область прив'язується до відповідного підграфа. Тому, враховуючи різний тип виходу (локалізація проти класифікації) та відмінні протоколи оцінювання, прямого зіставлення з GNN-підходами немає.

Як репрезентативний базовий метод обрано GraphCodeBERT, оскільки він є поширеним трансформерним підходом для представлення програмного коду та підтримує протоколи оцінювання, сумісні з використаними даними. Розширення експериментальної частини шляхом адаптації графових моделей (GCN/GAT) до задачі локалізації або, навпаки, приведення запропонованого підходу до

постановки graph-level класифікації розглядається як напрям подальших досліджень.

Застосування графових нейронних мереж (GCN/GAT) та спеціалізованих моделей, зокрема Devign, ReVeal, IVDetect і LineVD, які працюють безпосередньо з поданнями CFG/DFG/PDG, дають змогу розв’язувати задачі класифікації на рівні графа або функції, а також ранжування чи класифікації вузлів. Запропоновані методи орієнтовані на локалізацію підозрілих областей (детекцію рамок) з подальшою прив’язкою результату до відповідного підграфа. Для оцінювання стійкості метрик до випадкових чинників, зокрема ініціалізації та порядку мініпакетів, зазвичай наводять mean±std за кількома запусками або довірчі інтервали, отримані бутстреп-перевибіркою тестового набору. Отримані результати подано для фіксованої конфігурації та параметрів з метою відтворюваності, а статистичне підтвердження відмінностей (95% CI або p-value) розглядається як напрям подальшого розширення експериментальної частини. Вищий показник Precision для Clang Static Analyzer (0.83) поєднується з істотно нижчим Recall (0.54), унаслідок чого його macro-F1 (0.65) є нижчим, ніж у запропонованої системи (0.80). Це відображає типовий компроміс між вибірковістю та повнотою: правил-орієнтовані SAST-інструменти краще відсіюють канонічні патерни, проте частіше пропускають нетривіальні залежності.

Інтерпретація відмінностей узгоджується з природою методів. Статичні аналізатори ґрунтуються на наборах правил і евристик. Вони надійно сигналізують за чіткими патернами (на зразок небезпечних викликів копіювання в буфер без перевірки меж), але втрачають повноту там, де вирішальним є глобальний контекст (ланцюжки використання–визначення, неочевидні проходи через гілки керування, крос–модульні залежності). Комбінований підхід, який оперує багатомасштабними картами ознак і послідовною увагою, здатен “зібрати” такі залежності, тому зменшує пропуски і водночас утримує прийнятну вибірковість. Помірний виграш у Precision пояснюється післяобробленням (NMS та калібрування порогів), що відсікає другорядні кандидати. Результати хибних спрацьовувань та пропусків подано в табл. 4.14.

Таблиця 4.14 – Хибні спрацьовування та пропуски

Підхід	FP/1kLOC	Пропуски (FN), %	Підхід
Наша система (YOLO MobileNet)	8,1	22	Наша система (YOLO MobileNet)
Clang Static Analyzer	11,5	46	Clang Static Analyzer
Cppcheck	14,8	51	Cppcheck

Якісний аналіз помилок показує, що пропуски (FN) найчастіше виникають за наявності нетипових обгорток над операціями копіювання або перевірками меж, складної арифметики індексів, яка залежить від кількох гілок керування, міжпроцедурних переходів через виклики без явних маркерів у локальному фрагменті, а також у випадках макросів чи генерованого коду, коли інформативний сигнал розподіляється між вузлами графа і локально виражений слабко. Хибні спрацьовування (FP) частіше пов'язані з тим, що в межі рамки потрапляють захисні конструкції, зокрема перевірки меж або canary, або з тим, що умови гілкування в локальному підграфі відображені неповно.

Переваги та обмеження кожного підходу проявляються у практичних сценаріях. Статичні аналізатори мають низький поріг входу (легка інтеграція у CI, пояснювані правила), швидко знаходять канонічні помилки й забезпечують детерміновані звіти, однак генерують більше FP на складних кодових базах і мають нижчу Recall на нетривіальних випадках, що збільшує витрати на ручну перевірку і ризик пропусків. Розроблена система демонструє вищий macro–F1 і нижчі FP/1kLOC, краще працює на міжпроєктних тестах, але потребує навчання на маркованих даних і має дещо більшу латентність через нейромережевий етап.

З огляду на наведені результати доцільним є поєднання двох підходів у виробничому конвеєрі. Статичні аналізатори як перша лінія для швидкого “грубого” відсіву типових дефектів та політик безпеки. Нейромережна локалізаційно–класифікаційна система як друга лінія для зменшення FP, відновлення пропусків і пріоритезації знахідок за ймовірністю та класом. Така композиція мінімізує ризики й сумарні витрати на рев’ю, забезпечуючи кращий баланс між якістю виявлення та операційною ефективністю.

Сукупність експериментів дозволяє узгоджено оцінити вплив архітектури, способу подання даних і типу базового підходу на якість виявлення та локалізацію вразливостей. Узагальнені показники наведено у підсумковій таблиці: порівнюються суто класифікаційні моделі, локалізаційно-класифікаційні варіанти на основі YOLO з різними опорними мережами, варіанти з векторним та зображальним поданням, а також традиційні статичні аналізатори. Показники подано у макро–усередненні; час інференсу наведено як медіану вимірювань на єдиному стенді. Значення підсумкових метрик за сценаріями наведено в табл. 4.15.

Таблиця 4.15 – Підсумкові метрики за сценаріями

Сценарій	Precision	Recall	F1	mAP@0.50
CNN	0,73	0,66	0,69	н/з
CNN + LSTM	0,75	0,71	0,73	н/з
CNN + LSTM + Attention	0,79	0,75	0,77	н/з
YOLO (CSP–Darknet)	0,84	0,80	0,82	0,86
YOLO (MobileNet)	0,82	0,76	0,79	0,83
YOLO (MobileNet–Small)	0,80	0,74	0,77	0,80
Подання: вектор (послідовне)	0,78	0,74	0,76	0,78
Подання: зображення 416×416	0,83	0,79	0,81	0,85
Clang Static Analyzer	0,83	0,54	0,65	н/з

Отримані значення демонструють послідовну динаміку. Перехід від чисто згорткових класифікаторів до поєднання з LSTM і далі до включення механізму уваги приводить до зростання точності та повноти, причому особливо відчутно це для випадків з розірваним контекстом. Локалізаційно-класифікаційна архітектура на основі YOLO перевершує суто класифікаційні моделі не лише за F1, а й за mAP, що підтверджує вагомість багатомасштабної агрегації ознак. Зображувальне подання з багатоканальним кодуванням CFG/DFG систематично підвищує mAP у порівнянні з векторним, передусім на коротких аномаліях типу Off–by–One.

Водночас зростання якості супроводжується помірним збільшенням латентності та споживання пам’яті. Порівняння з традиційними аналізаторами засвідчує вищу вибірковість у сигнатурних випадках у правил-орієнтованих інструментів, однак нижчу повноту на нетривіальних графових залежностях. Інтегрована модель краще відновлює пропуски, що відбивається у вищому macro-F1 та нижчій кількості хибнопозитивних спрацювань на одиницю коду.

Компроміси зводяться до вибору опорної мережі та способу подання. Конфігурація з CSP–Darknet забезпечує максимальні F1 та mAP і доцільна у сценаріях, де пріоритет надано якості; варіанти з MobileNet і MobileNet–Small зменшують час інференсу на 30–50 % із помірною втратою якості, що робить їх придатними для швидких перевірок у CI/CD з жорсткими SLA. У межах послідовної частини ключову роль відіграє увага, яка стабілізує рішення й знижує хибнопозитивні спрацювання без відчутної втрати повноти. Збереження багатомасштабного злиття у neck–модулі є критичним для локалізації дрібних проявів та узгодження контексту.

З практичного погляду оптимальною слід вважати схему, яка поєднує зображувальне подання та одноетапний детектор з мобільним опорним блоком, доповнений послідовною CNN–LSTM–Attention–обробкою для уточнення класу. Така конфігурація забезпечує прийнятний час інференсу і водночас утримує високі значення macro–F1 і mAP; у задачах, де вимога до якості максимальна, доцільним є перехід на CSP–Darknet. Для повноти сприйняття результати рекомендовано супроводити кривими Precision–Recall для кожного класу та діаграмою “якість–швидкодія”, а також кривою залежності F1 від розміру навчальної вибірки, що наочно ілюструє стабільнішу збіжність комбінованої архітектури порівняно з базовими варіантами.

Окремий етап методики пов’язаний не стільки з навчанням моделі, скільки з налаштуванням порогів прийняття рішень на виході детектора. Навіть за фіксованої архітектури й незмінних ваг остаточні значення метрик суттєво залежать від обраного порогу впевненості та параметрів неперекривної супресії, зокрема граничного значення IoU. Тому для базової конфігурації локалізаційно-

класифікаційної моделі було проведено окрему серію вимірювань, метою якої було дослідити, як зміна цих параметрів зсуває криві Precision–Recall, відбивається на macro–F1 та впливає на кількість хибнопозитивних спрацювань у розрахунку на одиницю коду.

Калібрування порогу впевненості виконувалося на валідаційній підвибірці з використанням фіксованої конфігурації NMS та IoU. Для кожного можливого значення порогу у діапазоні від 0,3 до 0,7 будувалася крива Precision–Recall із подальшим обчисленням інтегральних показників. При низькому порозі, близько 0,3, детектор генерує більшу кількість кандидатів; це призводить до зростання повноти, особливо для слабких або прикордонних проявів вразливостей, але одночасно збільшує частку хибнопозитивних спрацювань, що знижує точність. На протилежному кінці діапазону, при значенні порогу 0,7, спостерігається зворотна картина. Precision зростає, оскільки відсікаються майже всі «невпевнені» спрацьовування, але частина справжніх вразливостей, зокрема класу Off–by–One, втрачається, що призводить до суттєвого падіння recall. В околі проміжних значень, близьких до 0,5, форма PR–кривих набуває найбільш збалансованого вигляду. Повнота ще не деградує до неприйнятного рівня, а точність уже компенсує надлишок помилкових сповіщень. Саме на цьому інтервалі спостерігалося максимальне значення macro–F1, що й було покладено в основу вибору робочого порогу для «загального» сценарію.

Аналогічний аналіз було проведено щодо порогу перекриття, який використовується в процедурі NMS. Для фіксованого порогу впевненості вивчався вплив зміни граничного IoU у діапазоні від 0,45 до 0,7. Якщо вимогу до перекриття послабити (IoU≈0,45), то NMS агресивніше зберігає альтернативні рамки, що частково допомагає у випадках, коли вразливий фрагмент коду має кілька близьких за розташуванням проявів; у таких ситуаціях детектор може залишити декілька коректних спрацьовувань, і локалізація виглядає більш «щільною». Однак у більшості проєктів це обертається збільшенням числа дублювань і, як наслідок, зростанням FP/1kLOC. При жорсткому IoU≈0,7 супресія, навпаки, відкидає більшість конкуруючих рамок, у тому числі такі, що частково перекриваються з

істинною локалізацією, але недостатньо добре збігаються з еталоном. У результаті precision зростає за рахунок скорочення числа виявлень, проте частина правильних спрацювань також втрачається, і recall помітно зменшується. Найкраще співвідношення між цими ефектами виявилося для проміжних значень IoU поблизу 0,5–0,55, які забезпечують прийнятний рівень усунення дубльованих рамок без істотної втрати справжніх детекцій.

У контексті практичного застосування було розглянуто дві робочі конфігурації порогів, орієнтовані на різні сценарії використання. Для інтеграції в CI/CD, де надмірна кількість хибних сповіщень прямо впливає на продуктивність розробників, поріг впевненості доцільно піднімати до верхньої частини «оптимального» інтервалу, а поріг IoU для NMS обирати ближчим до 0,55–0,6. Така комбінація зменшує потік повідомлень, зберігаючи при цьому більшість дійсних вразливостей, і виявляється зручною для автоматизованого фільтрування. Для офлайн–аудиту, навпаки, більш прийнятною є конфігурація з дещо нижчим порогом впевненості та менш жорсткими налаштуваннями NMS. Це дозволяє свідомо змістити баланс у бік повноти, прийнявши більшу кількість кандидатів, які вже верифікуються експертом. Обидва режими спираються на одну й ту саму модель, але різне порогування дає змогу адаптувати її поведінку до вимог конкретного процесу, що й демонструє важливість калібрування порогів як невід'ємної частини «засобу отримання» експериментальних та виробничих результатів.

Експериментальна частина підтвердила висунуті гіпотези. Залучення графових представлень коду (CFG/DFG) у поєднанні з багатоканальним зображувальним поданням істотно підвищує якість локалізації та класифікації, зокрема для коротких і дрібномасштабних проявів типу off–by–one, тоді як суто послідовні векторні ознаки поступаються за інтегральними показниками. Багатомасштабне злиття ознак у neck–модулі забезпечує приріст повноти, а механізм уваги в послідовній частині стабілізує рішення та зменшує кількість хибнопозитивних спрацьовувань. На цій основі комбінована схема "YOLO для локалізації та CNN–LSTM–Attention для уточнення класу" демонструє найкращий

баланс precision/recall і перевершує традиційні правил-орієнтовані аналізатори на міжпроєктних тестах, зберігаючи прийнятну швидкодію.

Найвищі результати досягаються за умов використання зображувального подання з пірамідою ознак і правильно ініціалізованими “якорями”, каліброваних порогів прийняття рішень і відтворюваного протоколу навчання з фіксацією конфігурацій. Для практичного впровадження в CI/CD необхідні марковані приклади для початкового навчання, періодична валідація на “холодних” репозиторіях, а також легкі процедури адаптації до нового домену (переініціалізація якорів, перекалібровка порогів, коротке тонке налаштування класифікаційного блоку). За пріоритету максимальної точності доцільно використовувати опорну мережу класу CSP-Darknet; для жорстких обмежень часу й пам’яті, тобто мобільні варіанти з помірною втратою якості, що забезпечують стабільну роботу у потокових сценаріях.

4.4 Висновки до четвертого розділу

Розроблено методику визначення ефективності моделей виявлення вразливостей типу “переповнення буферу” у ПЗКС. В ній враховано специфіку трьох узагальнених класів вразливостей (Stack Overflow, Heap Overflow, Off–by–One), побудову навчальної та тестової вибірок на рівні проєктів, поєднання векторного й зображувального подання коду на основі графів AST/CFG/DFG, а також використання узгодженого набору показників якості, що включає покласові значення точності, повноти, F1–міри, інтегральні mAP–показники та часові характеристики інференсу. Запропонована методика орієнтована на комплексне оцінювання якості моделей з урахуванням відтворюваності експериментів і придатності результатів до інтеграції в конвеєри CI/CD.

На основі розробленої методики сформовано репрезентативний набір даних із реальних та синтетично згенерованих прикладів, побудовано та досліджено низку конфігурацій моделей, зокрема послідовні нейромережеві класифікатори, локалізаційно-класифікаційні архітектури на основі детектора типу YOLO та їх

комбінування з модулем CNN–LSTM–Attention, а також виконано порівняння із традиційними засобами статичного аналізу коду. Експериментальні результати показали перевагу запропонованого методу, що поєднує графові представлення коду, багатоканальне зображальне подання та локалізаційно-класифікаційний підхід, і підтвердили достатність отриманих характеристик ефективності для його практичного застосування в задачах автоматизованого виявлення вразливостей та їх інтеграції в процеси CI/CD.

Основні наукові результати четвертого розділу опубліковані в [126–128; 152–156].

ВИСНОВКИ

У результаті виконання дисертаційного дослідження було розв’язано актуальну науково-прикладну задачу підвищення ефективності виявлення та оцінювання ризику експлуатації вразливостей типу переповнення буферу у програмному забезпеченні мов C/C++ із наступною інтеграцією розроблених методів у процеси DevSecOps і конвеєри CI/CD. У роботі отримано такі наукові та практичні результати:

1. Здійснено системний аналіз існуючих підходів до статичного й динамічного виявлення вразливостей у програмному забезпеченні, визначено їх переваги й недоліки та сформовано вимоги до автоматизованих засобів детектування переповнення буфера, придатних для безперервної інтеграції та доставляння програмних продуктів.

2. Створено формальну модель вразливості переповнення буфера у вигляді орієнтованого графа з атрибутами вузлів і ребер, що відображають залежності між даними й керуванням, параметри доступу до буферів та інші характеристики програмного коду. Модель забезпечує основу для виявлення вразливих фрагментів та кількісної оцінки ризику їх експлуатації.

3. Удосконалено модель процесу виявлення вразливостей типу переповнення буфера шляхом інтеграції графової моделі представлення коду, нейромережевого локалізаційно-класифікаційного детектора та модуля композитної оцінки ризику в конвеєри автоматизованого збирання та розгортання (CI/CD). Запропонована модель процесу забезпечує підтримку повного циклу аналізу коду та підвищує практичну придатність застосування методів у підходах DevSecOps.

4. Розроблено метод машинного виявлення переповнення буфера з використанням нейромережевої архітектури YOLO/Transformer, у якому визначено правила сегментації орієнтованих графів та формування навчальних вибірок. Реалізація у вигляді локалізаційно-класифікаційного детектора дала змогу підвищити точність і повноту виявлення у порівнянні з відомими статичними та нейромережевими засобами.

5. Розроблено метод підготовки та обробки даних для тренування нейронних детекторів, що передбачає розмітку початкового коду, побудову та сегментацію орієнтованих графів і перетворення підграфів у багатоканальні зображення з класами Stack/Heap/Off-by-one. Методика дала змогу сформувати репрезентативні навчальні вибірки, збалансовані за класами, забезпечити сумісність з архітектурами обробки зображень та покращити узагальнювальну здатність моделей.

6. Розроблено метод композитної оцінки ризику експлуатації виявлених вразливостей і алгоритм інтеграції цього методу в конвеєрах автоматизованого збирання та розгортання. Оцінювання враховує критичність, складність виправлення та ймовірність виникнення, що забезпечує автоматизоване визначення пріоритетів усунення, блокування небезпечних збірок та управління процесом розгортання. Експериментально підтверджено, що запропонований метод на основі YOLO забезпечує macro-F1 = 0,82 та AUPRC = 0,88. Порівняно з базовим CNN-класифікатором досягнуто приросту якості на 0,13 за macro-F1. Для інтеграції у конвеєрах автоматизованого збирання та розгортання доцільною є конфігурація YOLO (MobileNet), яка забезпечує mAP@0.50 = 0,83 за латентності 6,1 мс на GPU.

7. Реалізовано прототипи програмних засобів та проведено експериментальні дослідження, виконано інтеграцію розроблених моделей і методів у середовища розробки та у конвеєрах автоматизованого збирання та розгортання, оцінено точність і швидкодію порівняно з існуючими статичними сканерами. Показано, що застосування нейромережевого детектора на основі графового представлення коду забезпечує вищі значення точності та повноти виявлення переповнення буфера за меншого часу аналізу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Abichandani P., Lobo D., Ford G. et al. Wind measurement and simulation techniques in multi-rotor small unmanned aerial vehicles. *IEEE Access*. 2020. Vol. 8. P. 54910–54927. DOI: https://doi.org/10.1109/ACCESS.2020.2977693.

2. Aggarwal S., Kumar N. Path planning techniques for unmanned aerial vehicles: A review, solutions, and challenges. *Computer Communications*. 2020. Vol. 149. P. 270–299. DOI:https://doi.org/10.1016/j.comcom.2019.10.014.

3. AGRAS T20 – DJI. DJI Technology Co. Ltd. URL: https://www.dji.com/t20 (дата звернення: 02.01.2026).

4. Ahmad F., Abidin S., Qureshi I., Ishrat M. Big Data and Its Role in Cybersecurity. 2023. DOI:https://doi.org/10.1007/978-981-99-0550-8_10.

5. Allodi L., Cremonini M., Massacci F. et al. Measuring the accuracy of software vulnerability assessments: experiments with students and professionals. *Empirical Software Engineering*. 2020. Vol. 25. P. 1063–1094. DOI:https://doi.org/10.1007/s10664-019-09797-4.

6. Amami M. Fast and reliable vision-based navigation for real time kinematic applications. *International Journal for Research in Applied Science and Engineering Technology*. 2022. Vol. 10, No. 2. P. 922–932. DOI:https://doi.org/10.22214/ijraset.2022.40395.

7. Amankwah R., Kudjo P., Yeboah S. Evaluation of Software Vulnerability Detection Methods and Tools: A Review. *International Journal of Computer Applications*. 2017. Vol. 169. P. 22–27. DOI:https://doi.org/10.5120/ijca2017914750.

8. Anagnostis A., Tagarakis A. C., Asiminari G. et al. A deep learning approach for anthracnose infected trees classification in walnut orchards. *Computers and Electronics in Agriculture*. 2021. Vol. 182. Art. 105998. DOI:https://doi.org/10.1016/j.compag.2021.105998.

9. Anwar A., Khormali A., Choi J. et al. Measuring the Cost of Software Vulnerabilities. *ICST Transactions on Security and Safety*. 2018. Vol. 7. Art. 164551. DOI:https://doi.org/10.4108/eai.13-7-2018.164551.

10. Aslan Ö., Aktuğ S., Ozkan M., Yılmaz A., Akin E. A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions. *Electronics*. 2023. Vol. 12. P. 1–42. DOI:https://doi.org/10.3390/electronics12061333.

11. Aslan Ö. Computer system and third-parties vulnerabilities increases the risk of cyber attacks // 7. Uluslararası Akademik Araştırmalar Kongresi : Proceedings. Ankara, 2022. P. 124. URL: https://www.researchgate.net/publication/358883187_Computer_System_and_Third-Parties_Vulnerabilities_Increases_the_Risk_of_Cyber_Attacks (дата звернення: 02.01.2026).

12. Azman M., Marhusin M. F., Sulaiman R. Machine Learning-Based Technique to Detect SQL Injection Attack. *Journal of Computer Science*. 2021. Vol. 17. P. 296–303. DOI:https://doi.org/10.3844/jcssp.2021.296.303.

13. Bailey J. P., Nash A., Tovey C. A. et al. Path-length analysis for grid-based path planning. *Artificial Intelligence*. 2021. Vol. 301. Art. 103560. DOI:https://doi.org/10.1016/j.artint.2021.103560.

14. Benkhelifa E., Welsh T., Hamouda W. A Critical review of practices and challenges in intrusion detection systems for IoT: towards universal and resilient systems. *IEEE Communications Surveys & Tutorials*. 2018. Vol. 20, No. 4. P. 3496–3515. DOI:https://doi.org/10.1109/COMST.2018.2844742.

15. Bertoglio D., Zorzo A. Overview and open issues on penetration test. *Journal of the Brazilian Computer Society*. 2017. Vol. 23. DOI:https://doi.org/10.1186/s13173-017-0051-1.

16. Bhandari G. P., Naseer A., Moonen L. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software // PROMISE '21 : Proceedings. 2021. DOI:https://doi.org/10.1145/3475960.3475985.

17. Bouguettaya A., Zarzour H., Taberkit A. M. et al. A review on early wildfire detection from unmanned aerial vehicles using deep learning-based computer vision algorithms. *Signal Processing*. 2022. Vol. 190. Art. 108309. DOI:https://doi.org/10.1016/j.sigpro.2021.108309.

18. Cai K., Dai T., Lin Q. et al. Route planning based on parallel optimization in the air-ground integrated network. *IEEE Transactions on Intelligent Transportation Systems*. 2023. Vol. 24, No. 12. P. 15762–15773. DOI:https://doi.org/10.1109/TITS.2023.3234936.

19. Cai W., Chen J., Yu J., Gao L. A software vulnerability detection method based on deep learning with complex network analysis and subgraph partition. *Information and Software Technology*. 2023. Vol. 164. Art. 107328. DOI:https://doi.org/10.1016/j.infsof.2023.107328.

20. Cao D., Hu W., Zhao J. et al. Reinforcement learning and its applications in modern power and energy systems: A review. *Journal of Modern Power Systems and Clean Energy*. 2020. Vol. 8, No. 6. P. 1029–1042. DOI:https://doi.org/10.35833/MPCE.2020.000552.

21. Vassallo C., Panichella S., Palomba F., Gall H. C., Zaidman A. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*. 2020. Vol. 25, No. 2. P. 1419–1457. DOI:https://doi.org/10.1007/s10664-019-09750-5.

22. Cavelty M. D. The Politics of Cyber-Security. London : Routledge, 2024. 224 p. DOI:https://doi.org/10.4324/9781003497080.

23. Chaabouni N., Mosbah M., Zemmari A., Sauvignac C., Faruki P. Network intrusion detection for IoT security based on learning techniques. *IEEE Communications Surveys & Tutorials*. 2018. Vol. 21. P. 2671–2701. DOI:https://doi.org/10.1109/COMST.2019.2896380.

24. Chandrashekhar R., Mardithaya M., Thilagam S., Saha D. SQL Injection Attack Mechanisms and Prevention Techniques // Advanced Computing, Networking and Security. ADCONS 2011 : Proceedings. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer, 2012. Vol. 7135. DOI:https://doi.org/10.1007/978-3-642-29280-4_61.

25. Charoenwet W., Thongtanunam P., Pham V., Treude C. An empirical study of static analysis tools for secure code review // Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis : Proceedings. New York : ACM, 2024. 13 p. DOI:https://doi.org/10.48550/arXiv.2407.12241.

26. Chen Z., Cao J. VMCTE: visualization-based malware classification using transfer and ensemble learning. *Computers, Materials & Continua*. 2023. Vol. 75, No. 2. P. 4445–4465. DOI:https://doi.org/10.32604/cmc.2023.038639.

27. Chirillo J. Hack Attacks Testing: How to Conduct Your Own Security Audit. Hoboken, NJ : Wiley, 2003. 576 p. URL: https://dl.acm.org/doi/10.5555/1077293.1077297 (дата звернення: 02.01.2026).

28. Claid. ai: Stunning Product Photos for CPG and Marketplaces. Let's Enhance. URL: https://claid.ai (дата звернення: 02.01.2026).

29. CUDA Toolkit – Free Tools and Training. NVIDIA Developer. NVIDIA Corp. URL: https://developer.nvidia.com/cuda-toolkit (дата звернення: 02.01.2026).

30. Czekster R. Continuous risk assessment in secure DevOps. 2024. DOI:https://doi.org/10.48550/arXiv.2409.03405.

31. Dempsey P. Reviews – Consumer Technology. The Teardown – Apple iPhone Pro 13 smartphone. *Engineering & Technology*. 2021. Vol. 16, No. 11. P. 68–69. DOI:https://doi.org/10.1049/et.2021.1122.

32. Depcik C., Cassady T., Collicott B. et al. Comparison of lithium ion batteries, hydrogen fueled combustion engines, and a hydrogen fuel cell in powering a small unmanned aerial vehicle. *Energy Conversion and Management*. 2020. Vol. 207. Art. 112514. DOI:https://doi.org/10.1016/j.enconman.2020.112514.

33. Dhillon A., Verma G. K. Convolutional neural network: A review of models, methodologies and applications to object detection. *Progress in Artificial Intelligence*. 2020. Vol. 9, No. 2. P. 85–112. DOI:https://doi.org/10.1007/s13748-019-00203-0.

34. DJI Phantom 4 RTK – DJI. DJI Technology Co. Ltd. URL: https://www.dji.com/phantom-4-rtk (дата звернення: 02.01.2026).

35. DJI GS Pro – DJI. DJI Technology Co. Ltd. URL: https://www.dji.com/ground-station-pro (дата звернення: 02.01.2026).

36. Do L. N. Q., Wright J. R., Ali K. Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations. *IEEE Transactions on Software Engineering*. 2022. Vol. 48, No. 3. P. 835–847. DOI:https://doi.org/10.1109/TSE.2020.3004525.

37. eBee×mapping drone – Drones | AgEagle Aerial Systems. AgEagle Aerial Systems. URL: https://ageagle.com/drones/ebee-x (дата звернення: 02.01.2026).

38. Esposito M., Falaschi V., Falessi D. An Extensive Comparison of Static Application Security Testing Tools // Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering : Proceedings. Salerno, 2024. DOI:https://doi.org/10.1145/3661167.3661187.

39. Esposito M., Moreschini S., Lenarduzzi V., Hästbacka D., Falessi D. Can we trust the default vulnerabilities severity? // *IEEE 23rd International Working Conference on Source Code Analysis and Manipulation : Proceedings*. 2023. P. 265–270. DOI:https://doi.org/10.1109/SCAM59687.2023.00037.

40. Feng L., Chen S., Zhang C. et al. A comprehensive review on recent applications of unmanned aerial vehicle remote sensing with various sensors for high-throughput plant phenotyping. *Computers and Electronics in Agriculture*. 2021. Vol. 182. Art. 106033. DOI:https://doi.org/10.1016/j.compag.2021.106033.

41. Feng X., Murray A. T. Allocation using a heterogeneous space Voronoi diagram. *Journal of Geographical Systems*. 2018. Vol. 20, No. 3. P. 207–226. DOI:https://doi.org/10.1007/s10109-018-0274-5.

42. Fernández A., Usamentiaga R., Arquer P. et al. Robust detection, classification and localization of defects in large photovoltaic plants based on unmanned aerial vehicles and infrared thermography. *Applied Sciences*. 2020. Vol. 10, No. 17. Art. 5948. DOI:https://doi.org/10.3390/app10175948.

43. Fernández E. A Methodology for Secure Software Design // Proceedings of the International Conference on Software Engineering Research and Practice, SERP'04 : Proceedings. Las Vegas, 2004. Vol. 1. P. 130–136. URL: https://www.researchgate.net/publication/221610247_A_Methodology_for_Secure_Software_Design (дата звернення: 02.01.2026).

44. Fesenko H., Kharchenko V., Sachenko A., Hiromoto R., Kochan V. An internet of drone-based multi-version post-severe accident monitoring system: Structures and reliability // Dependable IoT for Human and Industry: Modeling, Architecting,

Implementation. Gistrup : *River Publishers*, 2018. P. 197–217. URL: https://ieeexplore.ieee.org/document/9226731 (дата звернення: 02.01.2026).

45. Gollapudi S. OpenCV with Python. Learn computer vision using OpenCV: With deep learning CNNs and RNNs. Berkeley, CA : Apress, 2019. P. 31–50. DOI:https://doi.org/10.1007/978-1-4842-4261-2.

46. Gonzalez-De-Santos P., Fernández R., Sepúlveda D. et al. Unmanned ground vehicles for smart farms // Agronomy – Climate Change, Food Security. London : IntechOpen, 2020. DOI:https://doi.org/10.5772/intechopen.90683.

47. Gonçalves G., Andriolo U. Operational use of multispectral images for macro-litter mapping and categorization by unmanned aerial vehicle. *Marine Pollution Bulletin*. 2022. Vol. 176. Art. 113431. DOI:https://doi.org/10.1016/j.marpolbul.2022.113431.

48. Guo Y., Yang H., Chen M. et al. Grid-based dynamic robust multi-objective brain storm optimization algorithm. *Soft Computing*. 2020. Vol. 24, No. 10. P. 7395–7415. DOI:https://doi.org/10.1007/s00500-019-04365-w.

49. Guo Z., Wang T., Liu S. et al. Biomass and vegetation coverage survey in the Mu Us sandy land – based on unmanned aerial vehicle RGB images. *International Journal of Applied Earth Observation and Geoinformation*. 2021. Vol. 94. Art. 102239. DOI:https://doi.org/10.1016/j.jag.2020.102239.

50. Guo D., Ren S., Lu S. et al. GraphCodeBERT: pre-training code representations with data flow. arXiv preprint arXiv:2009.08366. 2020. URL: https://arxiv.org/abs/2009.08366 (дата звернення: 02.01.2026).

51. Hajraoui A., El Merabet H. Classification of Anti-Malware Methods // International Conference on Intelligent Systems and Computer Vision : Proceedings. Fez, 2018. P. 1–4. URL: https://www.academia.edu/37161042/Classification_of_Anti_Malware_Methods (дата звернення: 02.01.2026).

52. Harder P., Pomeroy J. W., Helgason W. D. Improving sub-canopy snow depth mapping with unmanned aerial vehicles: Lidar versus structure-from-motion techniques. *The Cryosphere*. 2020. Vol. 14, No. 6. P. 1919–1935. DOI:https://doi.org/10.5194/tc-14-1919-2020.

53. Hong Z., Yang F., Pan H. et al. Highway crack segmentation from unmanned aerial vehicle images using deep learning. *IEEE Geoscience and Remote Sensing Letters*. 2022. Vol. 19. P. 1–5. DOI:https://doi.org/10.1109/LGRS.2021.3129607.

54. Hu P., Chapman S. C., Zheng B. et al. Coupling of machine learning methods to improve estimation of ground coverage from unmanned aerial vehicle (UAV) imagery for high-throughput phenotyping of crops. *Functional Plant Biology*. 2021. Vol. 48, No. 8. P. 766–779. DOI:https://doi.org/10.1071/FP20309.

55. Huan W., Shcherbakova G., Sachenko A. et al. Haar wavelet-based classification method for visual information processing systems. *Applied Sciences*. 2023. Vol. 13. Art. 5515. DOI:https://doi.org/10.3390/app13095515.

56. Huang Y., Wang Z., Ou H., Chi Y. Fuzzing-Based Office Software Vulnerability Mining on Android Platform // Proceeding of 2021 International Conference on Wireless Communications, Networking and Applications. WCNA 2021 : Proceedings. Lecture Notes in Electrical Engineering. Singapore : Springer, 2022. DOI:https://doi.org/10.1007/978-981-19-2456-9_114.

57. Humayun M., Niazi M., Jhanjhi N. et al. Cyber Security Threats and Vulnerabilities: A Systematic Mapping Study. *Arabian Journal for Science and Engineering*. 2020. Vol. 45. DOI:https://doi.org/10.1007/s13369-019-04319-2.

58. Im S.-young, Shin S.-Hun R., Ki R., Byeong-hee. Performance evaluation of network scanning tools with operation of firewall. 2016. P. 876–881. DOI:https://doi.org/10.1109/ICUFN.2016.7537162.

59. Israr A., Ali Z. A., Alkhammash E. H. et al. Optimization methods applied to motion planning of unmanned aerial vehicles: A review. *Drones*. 2022. Vol. 6, No. 5. Art. 126. DOI:https://doi.org/10.3390/drones6050126.

60. Jang B., Kim M., Harerimana G. et al. Q-learning algorithms: A comprehensive classification and applications. *IEEE Access*. 2019. Vol. 7. P. 133653–133667. DOI:https://doi.org/10.1109/ACCESS.2019.2941229.

61. Jiang X., Hadid A., Pang Y. et al. Deep learning in object detection and recognition. 1st ed. Singapore : Springer, 2019. 240 p. DOI:https://doi.org/10.1007/978-981-10-5152-4.

62. Kang H., Chen C. Fruit detection, segmentation and 3D visualisation of environments in apple orchards. *Computers and Electronics in Agriculture*. 2020. Vol. 171. Art. 105302. DOI:https://doi.org/10.48550/arXiv.1911.12889.

63. Kariuki P., Ofusori L. O., Subramaniam P. R. Cybersecurity threats and vulnerabilities experienced by small-scale African migrant traders in Southern Africa. *Security Journal*. 2024. Vol. 37. P. 292–321. DOI:https://doi.org/10.1057/s41284-023-00378-1.

64. Kasmawi N., Hidayasari N. Vulnerability analysis using OWASP ZAP on higher education websites // AIP Conference Proceedings. 2023. Vol. 2665. Art. 030015. DOI:https://doi.org/10.1063/5.0153145.

65. Kasturi S., Li X., Li P., Pickard J. A Proposed Approach to Integrate Application Security Vulnerability Data with Incidence Response Systems. *American Journal of Networks and Communications*. 2024. Vol. 13. P. 19–29. DOI:https://doi.org/10.11648/j.ajnc.20241301.12.

66. Kasturi S., Li X., Li P., Pickard J. Predicting attack paths from application security vulnerabilities using a multi-layer perceptron. *American Journal of Software Engineering and Applications*. 2024. Vol. 12, No. 1. P. 23–35. DOI:https://doi.org/10.11648/j.ajsea.20241201.14.

67. Kasturi S., Li X., Pickard J., Li P. Understanding Statistical Correlation of Application Security Vulnerability Data from Detection and Monitoring Tools // 2023 33rd International Telecommunication Networks and Applications Conference : Proceedings. Melbourne, 2023. P. 289–296. DOI:https://doi.org/10.1109/ITNAC59571.2023.10368476.

68. Kavianpour A., Anderson M. C. An overview of wireless network security // Proceedings of the 2017 IEEE 4th International Conference on Cyber Security and Cloud Computing : Proceedings. New York, 2017. P. 306–309. DOI:https://doi.org/10.1109/CSCloud.2017.45.

69. Khraisat A., Gondal I., Vamplew P., Kamruzzaman J. Survey of intrusion detection systems: techniques, datasets and challenges. Cybersecurity. 2019. Vol. 2. Art. 20. DOI:https://doi.org/10.1186/s42400-019-0038-7.

70. Koch W., Bestavros A. PROVIDE: Hiding from Automated Network Scans with Proofs of Identity. 2016. P. 66–71. DOI:https://doi.org/10.1109/HotWeb.2016.20.

71. Kravchenko Y., Bondarenko V., Tyshchenko M. Model of information protection system database of the mobile terminals information system on the territory of Ukraine // 2020 IEEE International Conference on Problems of Infocommunications. Science and Technology : Proceedings. Kharkiv, 2020. P. 785–790. DOI:https://doi.org/10.1109/PICST51311.2020.9468092.

72. Krombholz K., Hobel H., Huber M., Weippl E. Advanced social engineering attacks. *Journal of Information Security and Applications*. 2015. Vol. 22. P. 113–122. DOI:https://doi.org/10.1016/j.jisa.2014.09.005.

73. Li D., Yin W., Wong W. E. et al. Quality-oriented hybrid path planning based on A* and Q-learning for unmanned aerial vehicle. *IEEE Access*. 2022. Vol. 10. P. 7664–7674. DOI:https://doi.org/10.1109/ACCESS.2021.3139534.

74. Li M., Zhu X., Li W. et al. Retrieval of nitrogen content in apple canopy based on unmanned aerial vehicle hyperspectral images using a modified correlation coefficient method. *Sustainability*. 2022. Vol. 14, No. 4. Art. 1992. DOI:https://doi.org/10.3390/su14041992.

75. Li X., Wu H., Yang X. et al. Multiview machine vision research of fruits boxes handling robot based on the improved 2D kernel principal component analysis network. *Journal of Robotics*. 2021. Art. 3584422. DOI:https://doi.org/10.1155/2021/3584422.

76. Li X., Moreschini S., Zhang Z., Palomba F., Taibi D. The anatomy of a vulnerability database: A systematic mapping study. *Journal of Systems and Software*. 2023. Vol. 201. Art. 111679. DOI:https://doi.org/10.1016/j.jss.2023.111679.

77. Li Z., Zou D., Xu S. et al. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection // Network and Distributed System Security Symposium : Proceedings. San Diego, 2018. DOI:https://doi.org/10.14722/ndss.2018.23158.

78. Liang C., Miao M., Ma J. et al. Detection of GPS spoofing attack on unmanned aerial vehicle system // Machine Learning for Cyber Security : Proceedings. Cham : Springer, 2019. P. 123–139. DOI:https://doi.org/10.1007/978-3-030-30619-9_10.

79. Lin G., Wen S., Han Q.-L. et al. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*. 2020. Vol. 108, No. 10. P. 1825–1848. DOI:https://doi.org/10.1109/JPROC.2020.2993293.

80. Luo C., Li P., Meng W. TChecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications // Proceedings of the ACM SIGSAC Conference on Computer and Communications Security : Proceedings. 2022. P. 2175–2188. DOI:https://doi.org/10.1145/3548606.3559391.

81. Lv J., Xu H., Han Y. et al. A visual identification method for the apple growth forms in the orchard. Computers and Electronics in Agriculture. 2022. Vol. 197. Art. 106954. DOI:https://doi.org/10.1016/j.compag.2022.106954.

82. Lysenko S., Lysenko S., Bobrovnikova K. et al. IoT multi-vector cyberattack detection based on machine learning algorithms: traffic features analysis, experiments, and efficiency. *Algorithms*. 2022. Vol. 15, No. 7. Art. 239. DOI:https://doi.org/10.3390/a15070239.

83. Gopinath M., Chakkaravarthy S. A comprehensive survey on deep learning based malware detection techniques. *Computer Science Review*. 2023. Vol. 47. Art. 100529. DOI:https://doi.org/10.1016/j.cosrev.2022.100529.

84. Ma Z., Li N. Improving apple detection using RetinaNet // The International Conference on Image, Vision and Intelligent Systems : Proceedings. 2022. P. 131–141. DOI:https://doi.org/10.1007/978-981-16-6963-7_12.

85. Mai X., Zhang H., Jia X. et al. Faster R-CNN with classifier fusion for automatic detection of small fruits. *IEEE Transactions on Automation Science and Engineering*. 2020. Vol. 17, No. 3. P. 1555–1569. DOI:https://doi.org/10.1109/TASE.2020.2964289.

86. Mandal N., Jadhav S. A survey on network security tools for open source // 2016 IEEE International Conference on Current Trends in Advanced Computing (ICCTAC): Proceedings. 2016. P. 1–6. DOI: https://doi.org/10.1109/ICCTAC.2016.7567330.

87. Mao D., Sun H., Li X. et al. Real-time fruit detection using deep neural networks on CPU (RTFD): An edge AI application. *Computers and Electronics in*

Agriculture. 2023. Vol. 204. Art. 107517. DOI:https://doi.org/10.1016/j.compag.2022.107517.

88. Medeiros I., Fonseca J., Neves N. et al. Benchmarking Static Analysis Tools for Web Security. *IEEE Transactions on Reliability*. 2018. Vol. 67, No. 3. P. 1159–1175. DOI:https://doi.org/10.1109/TR.2018.2839339.

89. Mehdi H., Jaïdi F., Bouhoula A. A Systematic Approach for IoT Cyber-Attacks Detection in Smart Cities Using Machine Learning Techniques. 2021. P. 215–228. DOI:https://doi.org/10.1007/978-3-030-75075-6_17.

90. Mehrpour S., LaToza T. D. Can static analysis tools find more defects? *Empirical Software Engineering*. 2023. Vol. 28, No. 5. DOI:https://doi.org/10.1007/s10664-022-10232-4.

91. Mirbod O., Choi D., Heinemann P. H. et al. On-tree apple fruit size estimation using stereo vision with deep learning-based occlusion handling. *Biosystems Engineering*. 2023. Vol. 226. P. 27–42. DOI:https://doi.org/10.1016/j.biosystemseng.2022.12.008.

92. Mishra A. Amazon SageMaker // Machine learning in the AWS cloud: Add intelligence to applications with Amazon SageMaker and Amazon Rekognition. Hoboken, NJ : John Wiley & Sons, 2019. P. 353–385. DOI:https://doi.org/10.1002/9781119556749.

93. Moghadam P., Lowe T., Edwards E. J. Digital twin for the future of orchard production systems. *Proceedings*. 2020. Vol. 36, No. 1. Art. 92. DOI:https://doi.org/10.3390/proceedings2019036092.

94. Mohaidat A. I., Al-Helali A. Web Vulnerability Scanning Tools: A Comprehensive Overview, Selection Guidance, and Cyber Security Recommendations. *International Journal of Research Studies in Computer Science and Engineering*. 2024. Vol. 10, No. 1. P. 8–15. DOI:https://doi.org/10.20431/2349-4859.1001002.

95. Mohasseb A., Aziz B., Jung J. et al. Cyber security incidents analysis and classification in a case study of Korean enterprises. *Knowledge and Information Systems*. 2020. Vol. 62. P. 2917–2935. DOI:https://doi.org/10.1007/s10115-020-01452-5.

96. Morgan G. R., Wang C., Morris J. T. RGB indices and canopy height modelling for mapping tidal marsh biomass from a small unmanned aerial system. *Remote Sensing*. 2021. Vol. 13, No. 17. Art. 3406. DOI:https://doi.org/10.3390/rs13173406.

97. Patnaik N., Hallett J., Rashid A. Usability smells: An analysis of developers' struggle with crypto libraries // Fifteenth Symposium on Usable Privacy and Security : Proceedings. Santa Clara, 2019. P. 245–257. URL: https://www.usenix.org/system/files/soups2019-patnaik.pdf (дата звернення: 02.01.2026).

98. Naiakshina A., Danilova A., Gerlitz E., Smith M. On conducting security developer studies with CS students // Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems : Proceedings. Honolulu, 2020. P. 1–13. DOI:https://doi.org/10.1145/3313831.3376791.

99. Nguyen Quang Do L., Bodden E. Explaining Static Analysis With Rule Graphs. *IEEE Transactions on Software Engineering*. 2020. Vol. 48, No. 2. P. 719–733. DOI:https://doi.org/10.1109/TSE.2020.2999534.

100. NIST. National Institute of Standards and Technology. URL: https://www.nist.gov (дата звернення: 02.01.2026).

101. NIST. Recommendations for Federal Vulnerability Disclosure Guidelines. Special Publication 800-216. 2023. DOI:https://doi.org/10.6028/NIST.SP.800-216.

102. NVD. National Vulnerability Database. URL: https://nvd.nist.gov (дата звернення: 02.01.2026).

103. NVIDIA cuDNN. NVIDIA Developer. NVIDIA Corp. URL: https://developer.nvidia.com/cudnn (дата звернення: 02.01.2026).

104. Oyetoyan T. D., Milosheska B., Grini M., Cruzes D. Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital // Agile Processes in Software Engineering and Extreme Programming : Proceedings. Cham : Springer, 2018. P. 86–103. DOI:https://doi.org/10.1007/978-3-319-91602-6_6.

105. Pan L., Rashid T., Peng B. et al. Regularized softmax deep multi-agent Q-learning // Advances in Neural Information Processing Systems : Proceedings. 2021. Vol. 34. P. 1365–1377. DOI:https://doi.org/10.48550/arXiv.2103.11883.

106. Panjwani S., Tan S., Jarrin K. M., Cukier M. An experimental evaluation to determine if port scans are precursors to an attack // Proceedings of the International Conference on Dependable Systems and Networks : Proceedings. Yokohama, 2005. P. 602–611. DOI:https://doi.org/10.1109/DSN.2005.18.

107. Parrot Bluegrass Fields – AEROMOTUS. AEROMOTUS, LLC. URL: https://www.aeromotus.com/product/parrot-bluegrass (дата звернення: 02.01.2026).

108. Paszke A., Gross S., Massa F. et al. PyTorch: An imperative style, high-performance deep learning library // Advances in Neural Information Processing Systems : Proceedings. Vancouver, 2019. Vol. 32. P. 8024–8035. DOI:https://doi.org/10.48550/arXiv.1912.01703.

109. Pavlova O., Radiuk P., Kravchuk S., Kulbachnyi V. Information system for public places and institutions visualization with opportunities of inclusive access and optimal routing. *Computer systems and information technologies*. 2022. Vol. 1, No. 6. P. 62–68. DOI:https://doi.org/10.31891/CSIT-2022-1-8.

110. Pewny J., Schuster F., Bernhard L., Holz T., Rossow C. Leveraging semantic signatures for bug search in binary programs // Proceedings of the 30th Annual Computer Security Applications Conference : Proceedings. New Orleans, 2014. P. 406–415. DOI:https://doi.org/10.1145/2664243.2664269.

111. Pırtı A. Evaluating the accuracy of post-processed kinematic (PPK) positioning technique. *Geodesy and Cartography*. 2021. Vol. 47, No. 2. P. 66–70. DOI:https://doi.org/10.3846/gac.2021.12269.

112. Qamar A., Karim A., Chang V. Mobile malware attacks: review, taxonomy, future directions. *Future Generation Computer Systems*. 2019. Vol. 97. P. 887–909. DOI:https://doi.org/10.1016/j.future.2019.03.007.

113. Williams R., McMahon E., Samtani S., Patton M., Chen H. Identifying vulnerabilities of consumer Internet of Things (IoT) devices: A scalable approach // 2017 IEEE International Conference on Intelligence and Security Informatics : Proceedings. Beijing, 2017. P. 179–181. DOI:https://doi.org/10.1109/ISI.2017.8004904.

114. Ray A., Ray H. Proposing ε-greedy reinforcement learning technique to self-optimize memory controllers // 2021 2nd International Conference on Secure Cyber

Computing and Communications : Proceedings. Jalandhar, 2021. P. 318–323. DOI:https://doi.org/10.1109/ICSCCC51823.2021.9478147.

115. Reddy Maddikunta P. K., Hakak S., Alazab M. et al. Unmanned aerial vehicles in smart agriculture: Applications, requirements, and challenges. *IEEE Sensors Journal*. 2021. Vol. 21, No. 16. P. 17608–17619. DOI:https://doi.org/10.48550/arXiv.2007.12874.

116. Sara U., Akter M., Uddin M. S. Image quality assessment through FSIM, SSIM, MSE and PSNR-A comparative study. *Journal of Computer and Communications*. 2019. Vol. 7, No. 3. P. 8–18. DOI:https://doi.org/10.4236/jcc.2019.73002.

117. Savenko B., Lysenko S., Bobrovnikova K. et al. Detection DNS tunneling botnets // 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications : Proceedings. Bucharest, 2021. P. 64–69. DOI:https://doi.org/10.1109/IDAACS53288.2021.9660944.

118. Savenko O., Lysenko S., Nicheporuk A. et al. Metamorphic viruses' detection technique based on the equivalent functional block search // *13th International Conference on ICT in Education, Research and Industrial Applications : Proceedings*. *Kyiv*, 2017. P. 555–568. URL: https://ceur-ws.org/Vol-1844/10000555.pdf (дата звернення: 02.01.2026).

119. Savenko O., Gaj P., Sierhieiev Ye. Detection of buffer overflow vulnerabilities in system software based on a graph and transformer model. *AISSLE–2025: The International Workshop on Applied Intelligent Security Systems in Law Enforcement*, October, 30–31, 2025, Vinnytsia, Ukraine. Pp. 292–305. URL: https://ceur–ws.org/Vol–4126/paper17.pdf (дата звернення: 06.02.2026).

120. Savenko O., Lips S., Gaj P., Sierhieiev Y.. Graph-based data preparation for detecting buffer overflow vulnerabilities in code within CI/CD pipelines. (2025) CEUR Workshop Proceedings, 4163, pp. 1–10. The 2nd International Workshop on Advanced Applied Information Technologies: AI & DSS (AdvAIT-2025), December 05, 2025, Khmelnytskyi, Ukraine, Zilina, Slovakia : CEUR-Workshop Proceedings. Vol. 4163. Khmelnytskyi, 2025. Pp. 1–10. URL: https://ceur-ws.org/Vol-4163/paper19.pdf (дата звернення: 06.02.2026).

121. Savenko O., Sierhieiev Y., Gaj P., Balej J. Using artificial intelligence in the context of buffer overflow vulnerabilities // 2nd International Workshop on Intelligent & CyberPhysical Systems : Proceedings. 2025. Vol. 4013. P. 211–220. URL: https://ceur-ws.org/Vol-4013 (дата звернення: 02.01.2026).

122. Shah S., Mehtre B. M. A modern approach to cyber security analysis using vulnerability assessment and penetration testing. *International Journal of Electronics Communication and Computer Engineering.* 2013. Vol. 4, No. 6. P. 47–52. URL: https://ijecce.org/Download/conference/NCRTCST-2/11NCRTCST-13018.pdf (дата звернення: 02.01.2026).

123. Shakya S. Unmanned aerial vehicle with thermal imaging for automating water status in vineyard. *Journal of Electrical Engineering and Automation*. 2021. Vol. 3, No. 2. P. 79–91. DOI:https://doi.org/10.36548/jeea.2021.2.002.

124. Shen Z., Chen S. A Survey of Automatic Software Vulnerability Detection, Program Repair, and Defect Prediction Techniques. *Security and Communication Networks*. 2020. Art. 8858012. 16 p. DOI:https://doi.org/10.1155/2020/8858012.

125. Shen S., Kolluri A., Dong Z., Saxena P., Roychoudhury A. Localizing vulnerabilities statistically from one exploit // Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security : Proceedings. 2021. P. 537–549. DOI:https://doi.org/10.1145/3433210.3437528.

126. Sierhieiev Y., Paiuk V., Nicheporuk A., Kwiecien A., Huralnyk O. Detection and prediction of the vulnerabilities in software systems based on behavioral analysis with machine learning // Proceedings of the 1st International Workshop on Intelligent & CyberPhysical Systems : Proceedings. 2024. Vol. 3736. P. 239–254. URL: https://ceur-ws.org/Vol-3736 (дата звернення: 02.01.2026).

127. Sierhieiev Y., Paiuk V., Sachenko A., Nicheporuk A., Kwiecien A. A graph-based vulnerability detection method // Proceedings of the 5th International Workshop on Intelligent Information Technologies & Systems of Information Security : Proceedings. 2024. Vol. 3675. P. 343–355. URL: https://ceur-ws.org/Vol-3675 (дата звернення: 02.01.2026).

128. Sierhieiev Y., Paiuk V., Savenko O., Drozd A. Improvement of effectiveness for Static Application Security Testing for detection of SQL Injection vulnerabilities. *IEEE 14th International Conference on Dependable Systems, Services and Technologies (DESSERT-2024)* : Proceedings. Athens, Greece, October 11–13, 2024. Pp. 1–6. DOI: https://doi.org/10.1109/DESSERT65323.2024.11122171

129. Smith M., Naiakshina A., Danilova A. et al. Examining a password-storage study with CS students, freelancers, and company developers // Proceedings of the CHI Conference on Human Factors in Computing Systems : Proceedings. 2020. P. 1–12. DOI:https://doi.org/10.1145/3290605.3300370.

130. Softić J., Vejzović Z. Impact of vulnerability assessment and penetration testing (VAPT) on operating system security // 2023 22nd International Symposium INFOTEH-JAHORINA : Proceedings. East Sarajevo, 2023. P. 1–6. DOI:https://doi.org/10.1109/INFOTEH57020.2023.10094095.

131. Spanò S., Cardarilli G. C., Di Nunzio L. et al. An efficient hardware implementation of reinforcement learning: The Q-learning algorithm. *IEEE Access*. 2019. Vol. 7. P. 186340–186351. DOI:https://doi.org/10.1109/ACCESS.2019.2961174.

132. Steenhoek B., Rahman M., Jiles R., Le W. An Empirical Study of Deep Learning Models for Vulnerability Detection. 2023. P. 2237–2248. DOI:https://doi.org/10.1109/ICSE48619.2023.00188.

133. Stott E., Williams R. D., Hoey T. B. Ground control point distribution for accurate kilometre-scale topographic mapping using an RTK-GNSS unmanned aerial vehicle and SfM photogrammetry. *Drones*. 2020. Vol. 4, No. 3. Art. 55. DOI:https://doi.org/10.3390/drones4030055.

134. Sun L., Hu G., Chen C. et al. Lightweight apple detection in complex orchards using YOLOV5-PRE. *Horticulturae*. 2022. Vol. 8, No. 12. Art. 1169. DOI:https://doi.org/10.3390/horticulturae8121169.

135. Tang F., You X., Zhang X. et al. Hexagon-based generalized Voronoi diagrams generation for path planning of intelligent agents. *Mathematical Problems in Engineering*. 2020. Art. 5750739. DOI:https://doi.org/10.1155/2020/5750739.

136. Tharwat A. Classification assessment methods. *Applied Computing and Informatics*. 2020. Vol. 17, No. 1. P. 168–192. DOI:https://doi.org/10.1016/j.aci.2018.08.003.

137. Trickel E., Pagani F., Zhu C. et al. Toss a Fault to Your Witcher: Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities. 2023. P. 2658–2675. DOI:https://doi.org/10.1109/SP46215.2023.10179317.

138. Tufano R., Dabić O., Mastropaolo M. et al. Code review automation: strengths and weaknesses of the state of the art. *IEEE Transactions on Software Engineering*. 2023. P. 1–16. DOI:https://doi.org/10.1109/TSE.2023.3348172.

139. Utamima A., Djunaidy A. Agricultural routing planning: A narrative review of literature // Procedia Computer Science. Sixth Information Systems International Conference : Proceedings. 2022. Vol. 197. P. 693–700. DOI:https://doi.org/10.1016/j.procs.2021.12.190.

140. Valdés-Rodríguez Y., Hochstetter-Diez J., Díaz-Arancibia J., Cadena-Martínez R. Towards the Integration of Security Practices in Agile Software Development: A Systematic Mapping Review. *Applied Sciences*. 2023. Vol. 13. Art. 4578. DOI:https://doi.org/10.3390/app13074578.

141. Vassallo C., Panichella S., Palomba F. et al. Context is king: The developer perspective on the usage of static analysis tools // IEEE 25th International Conference on Software Analysis, Evolution and Reengineering : Proceedings. Campobasso, 2018. P. 38–49. DOI:https://doi.org/10.1109/SANER.2018.8330195.

142. Wang Y., Wang D., Zhao W., Liu Y. Detecting SQL Vulnerability Attack Based on the Dynamic and Static Analysis Technology // IEEE 39th Annual Computer Software and Applications Conference : Proceedings. Taichung, 2015. P. 604–607. DOI:https://doi.org/10.1109/COMPSAC.2015.277.

143. Website of Our Study. Static Application Security Testing (SAST) Tools for Smart Contracts: How Far Are We? URL: https://sites.google.com/view/sc-sast-study-fse2024/home (дата звернення: 02.01.2026).

144. Wu G., Chen C., Yang N. et al. Design of differential GPS system based on BP neural network error correction for precision agriculture // 2019 Chinese Intelligent Automation Conference : Lecture Notes in Electrical Engineering. Singapore : Springer, 2020. Vol. 568. DOI:https://doi.org/10.1007/978-981-32-9050-1_49.

145. Yassine M., Baddi Y., Alazab M., Tawalbeh L., Romdhani I. Artificial Intelligence and Blockchain for Future Cybersecurity Applications. Cham : Springer, 2021. 391 p. DOI:https://doi.org/10.1007/978-3-030-74575-2.

146. Yu T., Hu C., Xie Y. et al. Mature pomegranate fruit detection and location combining improved F-PointNet with 3D point cloud clustering in orchard. *Computers and Electronics in Agriculture*. 2022. Vol. 200. Art. 107233. DOI:https://doi.org/10.1016/j.compag.2022.107233.

147. Yuan Y., Yuliang Lu K. Z., Hui Huang L. Y., Jiazhen Zhao. A Static Detection Method for SQL Injection Vulnerability Based on Program Transformation. *Applied Sciences*. 2023. Vol. 13, No. 21. Art. 11763. DOI:https://doi.org/10.3390/app132111763.

148. Li Z., Zou D., Xu S., Jin H., Zhu Y., Chen Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*. 2022. Vol. 19, No. 4. P. 2244–2258. DOI:https://doi.org/10.1109/TDSC.2021.3051525.

149. Zhou X., Cao S., Sun X., Lo D. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. *ACM Transactions on Software Engineering and Methodology*. 2024. Vol. 34. DOI:https://doi.org/10.1145/3708522.

150. Бучик С. С. Інформаційна безпека : монографія. Житомир : Житомирський військовий інститут імені С. П. Корольова, 2019. 250 с.

151. Журавська І. М. Теоретичні основи, методи та засоби створення та функціонування швидкодинамічних гетерогенних комп'ютерних мереж критичного застосування : дис. ... д-ра техн. наук : 05.13.05. Миколаїв, 2019. 400 с. URL: https://chmnu.edu.ua/wp-content/uploads/2017/12/dis_Zhuravska.pdf (дата звернення: 02.01.2026).

152. Сєргєєв Є. В., Савенко О. С. Авторське свідоцтво №143407. Україна. Комп'ютерна програма «OverflowGuard: система аналізу переповнення буфера та оцінки ризику в CI/CD». Дата реєстрації: 23 лютого 2026 р.

153. Сєргєєв, Є.В., Кльоц Ю.П. Композитна оцінка ризику переповнення буфера і її трансляція в дії CI/CD. *MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES*. 2025. № 84(4), Pp. 89–94. DOI: https://doi.org/10.31891/2219-9365-2025-84-10

154. Сєргєєв, Є.В. Підготовка даних на основі графіків для виявлення вразливостей переповнення буфера в коді в рамках CI/CD-процесів. *Herald Of Khmelnytskyi National University. Technical Sciences*. 2026. № 361(1), Pp. 316–322. DOI: https://doi.org/10.31891/2307-5732-2026-361-45.

155. Сєргєєв Є. В., Каштальян А. С., Ковальчук В. В., Савенко О. С., Іванченко О. О. Ефективність і вдосконалення SAST у контексті SQL Injection вразливостей. Information Technology: Computer Science, Software Engineering and Cyber Security. 2024. № 3. С. 149–158. DOI:https://doi.org/10.32782/IT/2024-3-16.

156. Сєргєєв Є.В., Савенко О.С. Виявлення вразливостей переповнення буфера в системному програмному забезпеченні на основі графа та моделі трансформатора. *Вчені записки ТНУ імені В.І. Вернадського. Серія: Технічні науки.* 2025. № 6. С. 318 – 327. DOI: https://doi.org/10.32782/2663-5941/2025.6.2/43

157. Основні поняття. НД ТЗІ 1.1-003-99: Термінологія в галузі захисту інформації в комп'ютерних системах від несанкціонованого доступу. Київ : Департамент спеціальних телекомунікаційних систем та захисту інформації Служби безпеки України, 1999. 50 с.

ДОДАТКИ

ДОДАТОК А

Список публікацій здобувача

Наукові праці, в яких опубліковані основні наукові результати дисертації:

1. Сєргєєв Є., Каштальян А., Ковальчук В., Савенко О., Іванченко О. Ефективність і вдосконалення SAST у контексті SQL Injection вразливостей. *Information Technology: Computer Science, Software Engineering and Cyber Security*. 2024. № 3. С. 149-158. DOI: https://doi.org/10.32782/IT/2024-3-16

2. Сєргєєв Є.В., Савенко О.С. Виявлення вразливостей переповнення буфера в системному програмному забезпеченні на основі графа та моделі трансформатора. *Вчені записки ТНУ імені В.І. Вернадського. Серія: Технічні науки.* 2025. № 6. С. 318 – 327. DOI: https://doi.org/10.32782/2663-5941/2025.6.2/43

3. Сєргєєв Є.В. Підготовка даних на основі графіків для виявлення вразливостей переповнення буфера в коді в рамках CI/CD-процесів. *Herald Of Khmelnytskyi National University. Technical Sciences*. 2026. № 361(1), Pp. 316–322. DOI: https://doi.org/10.31891/2307-5732-2026-361-45

4. Сєргєєв, Є.В., Кльоц Ю.П. Композитна оцінка ризику переповнення буфера і її трансляція в дії CI/CD. *MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES*. 2025. № 84(4), Pp. 89–94. DOI: https://doi.org/10.31891/2219-9365-2025-84-10

Праці, які засвідчують апробацію матеріалів дисертації:

5. Sierhieiev Y., Paiuk V., Sachenko A., Nicheporuk A., Kwiecien A. A graph-based vulnerability detection method. (2024) *CEUR Workshop Proceedings*, 3675, pp. 388-401. *The 5th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS-2024)* : CEUR-Workshop Proceedings. Vol. 3675. (Khmelnytskyi, March 2024). Khmelnytskyi, 2024. Pp. 343-355. URL: https://ceu’r-ws.org/Vol-3675/

6. Sierhieiev Y., Paiuk V., Nicheporuk A., Kwiecien A., Huralnyk O. Detection and prediction of the vulnerabilities in software systems based on behavioral analysis with machine learning. (2024) *CEUR Workshop Proceedings*, 3736, pp. 239-254. *The 1th* Proceedings of the 1st International Workshop on Intelligent & CyberPhysical Systems (ICyberPhyS 2024) Khmelnytskyi, Ukraine, June 28, 2024 : CEUR-Workshop Proceedings. Vol. 3736. (Khmelnytskyi, Ukraine, June 28, 2024). Khmelnytskyi, 2024. Pp. 239-254. URL: https://ceur-ws.org/Vol-3736/

7. Savenko O., Lips S., Gaj P., Sierhieiev Y.. Graph-based data preparation for detecting buffer overflow vulnerabilities in code within CI/CD pipelines. (2025) CEUR Workshop Proceedings, 4163, pp. 1–10. The 2nd International Workshop on Advanced Applied Information Technologies: AI & DSS (AdvAIT-2025), December 05, 2025, Khmelnytskyi, Ukraine: CEUR-Workshop Proceedings. Vol. 4163. Khmelnytskyi, 2025. Pp. 1–10. URL: https://ceur-ws.org/Vol-4163/paper19.pdf

8. Savenko O., Sierhieiev Y., Gaj P., Balej J. Using artificial intelligence in the context of buffer overflow vulnerabilities. *The 2nd International Workshop on Intelligent & CyberPhysical Systems (ICyberPhyS 2025), CEUR Workshop Proceedings*. Vol. 4013. Khmelnytskyi, Ukraine, 4 July 2025. Pp. 211–220. URL: https://ceur-ws.org/Vol-4013/paper17.pdf

9. Savenko O., Gaj P., Sierhieiev Ye. Detection of buffer overflow vulnerabilities in system software based on a graph and transformer model. *AISSLE-2025: The International Workshop on Applied Intelligent Security Systems in Law Enforcement*, October, 30–31, 2025, Vinnytsia, Ukraine. Pp. 292-305. URL: https://ceur-ws.org/Vol-4126/paper17.pdf

10. Sierhieiev Y., Paiuk V., Savenko O., Drozd A. Improvement of effectiveness for Static Application Security Testing for detection of SQL Injection vulnerabilities. *IEEE 14th International Conference on Dependable Systems, Services and Technologies (DESSERT-2024)* : Proceedings. Athens, Greece, October 11–13, 2024. Pp. 1–6. DOI: https://doi.org/10.1109/DESSERT65323.2024.11122171

Публікації, які додатково відображають наукові результати дисертації:

11. Сєргєєв Є. В., Савенко О. С. Авторське свідоцтво №143407. Україна. Комп’ютерна програма «OverflowGuard: система аналізу переповнення буфера та оцінки ризику в CI/CD». Дата реєстрації: 23 лютого 2026 р.

ДОДАТОК Б

Акти впровадження

«Затверджую»
Технічний директор ПП «Нолт Технолоджис»
Мельниченко О.В.
«16» листопада 2026 р.

АКТ
про впровадження результатів дисертаційної роботи
аспіранта кафедри комп'ютерної інженерії та інформаційних систем
Хмельницького національного університету Сєргєєва Євгеній Віталійовича
«Методи та засоби виявлення вразливостей в програмному забезпеченні комп'ютерних систем»

Результати дисертаційної роботи аспіранта кафедри комп'ютерної інженерії та інформаційних систем Хмельницького національного університету Сєргєєва Є.В. впроваджені на ПП «Нолт Технолоджис».

При розробленні та впровадженні методів і програмних засобів автоматизованого виявлення вразливостей типу «переповнення буфера» (buffer overflow) у початковому коді програмного забезпечення комп'ютерних систем в ПП «Нолт Технолоджис» були використані наступні матеріали досліджень, отримані Сєргєєвим Є.В. особисто:

1) розроблено нову орієнтовану графову модель переповнення буфера, що забезпечує формалізацію структур даних стеку та купи, керуючого графу потоку виконання програми, а також відслідковування виділення, використання та звільнення блоків пам'яті;;

2) розроблено новий метод автоматизованого виявлення вразливостей «переповнення буфера» у початковому коді програмного забезпечення комп'ютерних систем на основі комбінації статичного аналізу та трансформерної архітектури нейронних мереж з графовим представленням коду, який дозволяє детектувати та локалізовувати вразливості за їхніми вхідними та вихідними змінними;

3) удосконалено метод формування навчальної вибірки з початкового коду шляхом розробки методу видобування кодових графів та застосування нового методу розширення й об'єднання їх вузлів, що дозволяє підвищити стійкість моделі виявлення вразливостей;

4) розроблено новий метод композитної оцінки ризику експлуатації виявлених вразливостей на основі їх характеристик, який дозволяє визначати пріоритети виправлень та блокувати небезпечні збірки шляхом інтеграції у конвеєри автоматизованого збирання та розгортання програмного забезпечення

комп'ютерних систем.

За результатами виконаних досліджень розроблено комплекс моделей та методів виявлення вразливостей типу buffer overflow у програмному забезпеченні та програмні засоби для їх застосування. Використання запропонованих методів дозволяє підвищити точність і швидкодію аналізу коду та інтегрувати автоматизоване виявлення вразливостей у конвеєри автоматизованого збирання та розгортання. Ефективність розроблених рішень підтверджено експериментальними дослідженнями.

Цей акт не є підставою для фінансових розрахунків.

Технічний директор Олександр Мельниченко

Директор ТОВ «ІТТ»
Сімогук В.С.
«16» лютого 2026 р.

АКТ

про впровадження результатів дисертаційної роботи
аспіранта кафедри комп'ютерної інженерії та інформаційних систем
Хмельницького національного університету Сєргєєва Євгенія Віталійовича
«Методи та засоби виявлення вразливостей в програмному забезпеченні
комп'ютерних систем»

Результати дисертаційної роботи Сєргєєва Є.В. аспіранта кафедри комп'ютерної інженерії та інформаційних систем Хмельницького національного університету впроваджені на ТОВ «ІТТ».

При розробленні та впровадженні методів і програмних засобів автоматизованого виявлення вразливостей типу «переповнення буфера» (buffer overflow) у початковому коді програмного забезпечення комп'ютерних систем в ТОВ «ІТТ» були використані наступні матеріали досліджень, отримані Сєргєєвим Є.В. особисто:

1) розроблено нову орієнтовану графову модель переповнення буфера, що забезпечує формалізацію структур даних стеку та купи, керуючого графу потоку виконання програми, а також відслідковування виділення, використання та звільнення блоків пам'яті;;

2) розроблено новий метод автоматизованого виявлення вразливостей «переповнення буфера» у початковому коді програмного забезпечення комп'ютерних систем на основі комбінації статичного аналізу та трансформерної архітектури нейронних мереж з графовим представленням коду, який дозволяє детектувати та локалізовувати вразливості за їхніми вхідними та вихідними змінними;

3) удосконалено метод формування навчальної вибірки з початкового коду шляхом розробки методу видобування кодових графів та застосування нового методу розширення й об'єднання їх вузлів, що дозволяє підвищити стійкість моделі виявлення вразливостей;

4) розроблено новий метод композитної оцінки ризику експлуатації виявлених вразливостей на основі їх характеристик, який дозволяє визначати пріоритети виправлень та блокувати небезпечні збірки шляхом інтеграції у конвеєри автоматизованого збирання та розгортання програмного забезпечення комп'ютерних систем.

За результатами виконаних досліджень розроблено комплекс моделей та методів виявлення вразливостей типу buffer overflow у програмному забезпеченні та програмні засоби для їх застосування. Використання запропонованих методів дозволяє підвищити точність і швидкодію аналізу коду та інтегрувати автоматизоване виявлення вразливостей у конвеєри автоматизованого збирання та розгортання. Ефективність розроблених рішень підтверджено експериментальними дослідженнями.

Цей акт не є підставою для фінансових розрахунків.

Заступник директора Іванов О.В.

Системний адміністратор Веремеєнко В.А.

«Затверджую»
Проректор з науково-педагогічної роботи
_____Віктор ЛОПАТОВСЬКИЙ
«25» травня 2026 р.

АКТ

про впровадження в навчальний процес Хмельницького національного університету результатів дисертаційної роботи аспіранта кафедри комп'ютерної інженерії та інформаційних систем Сєргєєва Євгенія Віталійовича «Методи та засоби виявлення вразливостей в програмному забезпеченні комп'ютерних систем»

Ми, комісія в складі: декана факультету інформаційних технологій, професора кафедри комп'ютерної інженерії та інформаційних систем, д.т.н., професора Говорущенко Т. О. (голова комісії), завідувача кафедри комп'ютерної інженерії та інформаційних систем, д.ф., доцента Павлової О. О., доцента кафедри комп'ютерної інженерії та інформаційних систем, к.т.н., доцента Капустян М. В. склала акт про те, що результати дисертаційної роботи Сєргєєва Є.В. впроваджені та використовуються в освітньому процесі Хмельницького національного університету при викладанні дисциплін на кафедрі комп'ютерної інженерії та інформаційних систем для здобувачів спеціальності F7 Комп'ютерна інженерія, зокрема в курсах «Безпека та захист комп'ютерних систем», «Моделювання та методи оптимізації в наукових та експериментальних дослідженнях», «Методології забезпечення якості, надійності, гарантоздатності та безпеки комп'ютерних систем та мереж».

При викладанні цих дисциплін викладачами кафедр використовувалися наступні матеріали досліджень, отримані Сєргєєвим Є.В. особисто:

1) орієнтована графова модель переповнення буфера, що забезпечує формалізацію структур даних стеку та купи, керуючого графу потоку виконання програми, а також відслідковування виділення, використання та звільнення блоків пам'яті;;

2) метод автоматизованого виявлення вразливостей «переповнення буфера» у початковому коді програмного забезпечення комп'ютерних систем на основі комбінації статичного аналізу та трансформерної архітектури нейронних мереж з графовим представленням коду, який дозволяє детектувати та локалізовувати вразливості за їхніми вхідними та вихідними змінними;

3) метод формування навчальної вибірки з початкового коду шляхом розробки методу видобування кодових графів та застосування нового методу

розширення й об'єднання їх вузлів, що дозволяє підвищити стійкість моделі виявлення вразливостей;

4) метод композитної оцінки ризику експлуатації виявлених вразливостей на основі їх характеристик, який дозволяє визначати пріоритети виправлень та блокувати небезпечні збірки шляхом інтеграції у конвеєри автоматизованого збирання та розгортання програмного забезпечення комп'ютерних систем.

За результатами виконаних досліджень розроблено комплекс моделей та методів виявлення вразливостей типу buffer overflow у програмному забезпеченні та програмні засоби для їх застосування. Використання запропонованих методів дозволяє підвищити точність і швидкодію аналізу коду та інтегрувати автоматизоване виявлення вразливостей у конвеєри автоматизованого збирання та розгортання. Ефективність розроблених рішень підтверджено експериментальними дослідженнями.

Говорущенко Т. О.

Павлова О. О.

Капустян М. В.

ДОДАТОК В

Керівництво користувача

Керівництво користувача програмного комплексу «OverflowGuard»

OverflowGuard – програмний комплекс для статичного аналізу початкового коду мов C/C++ з метою **виявлення вразливостей переповнення буфера** та **оцінювання ризику їх експлуатації**. Система допомагає розробникам і фахівцям з інформаційної безпеки інтегрувати контроль безпеки в CI/CD-конвеєри та своєчасно виявляти потенційно небезпечні буфери.

Архітектура програми

OverflowGuard побудовано за модульним принципом. Основні компоненти.

Graph Model – модуль побудови уніфікованого орієнтованого графа програми. Він завантажує JSON-опис, сформований стороннім парсером (Clang/LLVM), і створює графову модель з вузлами (функції, буфери, операції) та ребрами (потоки керування та даних). Граф модель використовується у всіх інших компонентах.

Risk Indicators – модуль розрахунку локальних і шляхових індикаторів ризику. Для кожного буфера визначаються: перевищення кількості записуваних байтів над розміром, наявність перевірки меж, глибина вкладених циклів, використання ненадійних даних та інші фактори. На основі цих показників та ймовірності виконання шляху обчислюється композитна оцінка ризику.

Data Preparation – засоби для підготовки даних до тренування нейронної мережі. Підмодулі реалізують перетворення графів на зображення (растрування підграфів), формування CSV/JSON-метаданих та генерацію вибірок для навчання та тестування.

Model Architecture – модуль із визначенням архітектури нейронної мережі для виявлення переповнення буфера. Для демонстрації використовується спрощена CNN+Transformer модель, але структуру можна замінити на будь-яку, сумісну з PyTorch або TensorFlow.

Analysis Module – функції для аналізу JSON-графа та формування CSV-звіту. Модуль analysis.py читає граф програми, обчислює ризики для всіх буферів і формує файл результатів.

Dataset Module – класи для роботи з вибірками (ImageRiskDataset) та інтеграції з фреймворками машинного навчання. Вони дозволяють завантажувати зображення та числові ознаки, формувати батчі та визначати цільові мітки.

Metrics – допоміжні функції для оцінювання якості моделей (точність, повнота, F1-міра), що можуть бути використані на етапі тестування.

CLI Interface – командний інтерфейс для користувача. Підтримуються підкоманди analyze, dataset, train та predict, які дозволяють виконувати аналіз, будувати вибірки, тренувати модель та робити прогнозування ризику з командного рядка.

Схема архітектури OverflowGuard

На рисунку наведено схематичну блок-схему архітектури OverflowGuard.

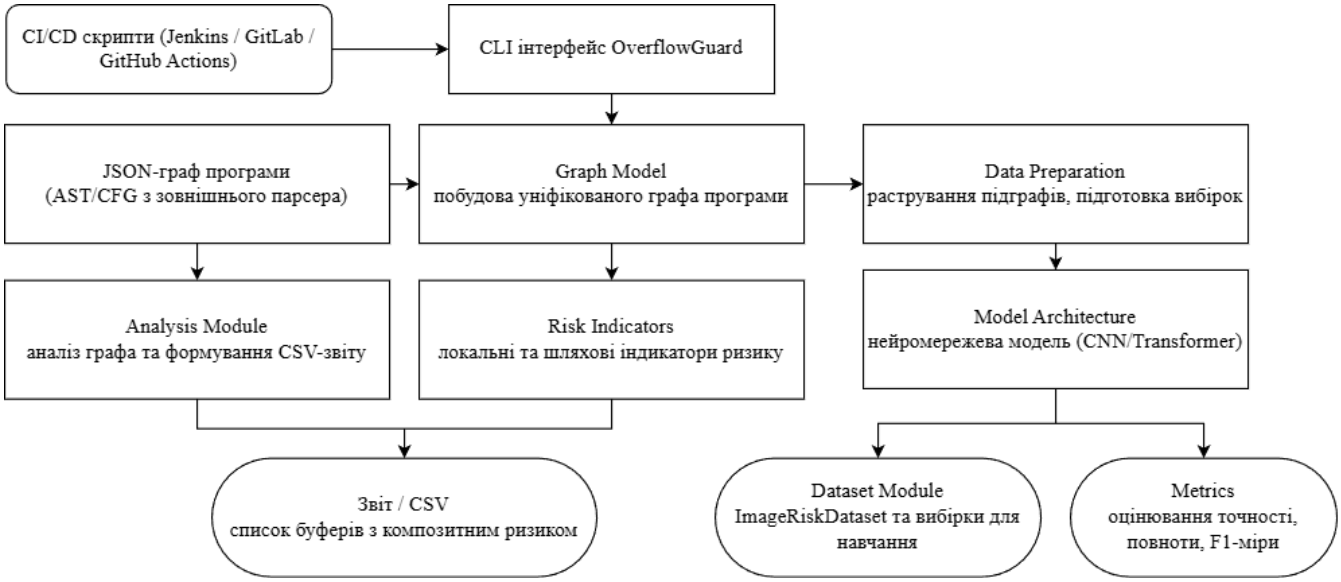


Рисунок В.1 – Архітектура програмного комплексу OverflowGuard: дані з JSON-парсера надходять до модуля графової моделі, далі розділяються між індикаторами ризику, підготовкою даних та аналізом, а через CLI користувач отримує звіт або передає дані до моделі.

Використання OverflowGuard

Нижче наведено покрокову інструкцію для використання програми. Приклади команд припускають, що ви вже встановили необхідні залежності (Python 3.10+, бібліотеки pandas, torch, PIL, argparse тощо) і отримуєте файл *risk_config.yaml* із налаштуваннями ваг.

1. Підготовка вихідного графа

Отримайте репозиторій початкового коду мовою C/C++.

Згенеруйте JSON-опис графа програми за допомогою зовнішнього інструменту. Наприклад, використовуйте Clang/LLVM разом із додатком bear для створення compilation database, а потім власний скрипт для вивантаження AST/CFG у JSON. Структура файлу повинна містити масиви nodes та edges з полями id, type, file, line, kind, size, max_write, has_guard, loop_depth, tainted, origin.

2. Аналіз коду та формування звіту

Використовуйте підкоманду analyze для оцінки ризиків. Приклад виклику:

python overflowguard.py analyze —input path/to/project_graph.json —output risk_report.csv —path_prob 0.5

Параметри:

—input – шлях до JSON-файлу з графом;

—output – файл для виводу CSV-звіту;

—path_prob – значення ймовірності проходження шляху, якщо точна інформація відсутня (за замовчуванням 0.5).

Результатом буде CSV-файл з такими колонками: buffer_id, file, line, kind, size, max_write, local_risk, composite_risk. Це дозволяє легко імпортувати дані у системи відстеження помилок.

3. Побудова датасету для навчання моделі

Щоб підготувати вибірку для тренування нейронної мережі, використовуйте підкоманду dataset. Приклад:

python overflowguard.py dataset —inputs src1.json src2.json src3.json —output_dir data/train

Програма згенерує зображення підграфів у вказаному каталозі та метадані metadata.json. Ці дані можна завантажити через клас ImageRiskDataset для тренування моделі.

4. Навчання та оцінювання моделі

Для тренування нейронної мережі передбачена підкоманда train, яка запускає умовний цикл навчання (скрипт потребує реалізації повного процесу навчання). Приклад виклику:

python overflowguard.py train —dataset_dir data/train —epochs 20 —lr 1e–4

Після навчання модель буде збережено у файл model.pth. Для оцінки ефективності можна використати функції з модуля metrics.py та контрольну вибірку.

5. Прогнозування ризиків за допомогою моделі

Після навчання моделі можна оцінити ризик для конкретного підграфа, надавши зображення та файл моделі:

python overflowguard.py predict —model model.pth —image data/train/file_b1.png

Функція поверне прогнозований клас (0 – низький ризик, 1 – високий ризик). Розширивши скрипт, можна отримувати ймовірності для інтерпретації результатів.

6. Налаштування ваг та порогів у risk_config.yaml

Файл risk_config.yaml містить ваги для розрахунку локального та шляхового ризиків, а також порогове значення risk_threshold, при перевищенні якого збірка повинна бути заблокована. Приклад:

```
weights:
  size_exceed: 0.4
  no_guard: 0.3
  loop_depth: 0.1
  tainted: 0.1

path_weights:
  long_call_chain: 0.05
```

```
  recursion: 0.2

  risk_threshold: 0.6
```

Змінюючи ці значення, ви можете адаптувати інструмент до специфіки проєкту. Наприклад, для більш консервативного аналізу збільште вагу no_guard або зменшіть risk_threshold.

Приклади вхідних та вихідних даних

Фрагмент вхідного JSON

```
{
  "nodes": [
    {"id": "f1", "type": "function", "name": "foo"},
    {"id": "b1", "type": "buffer", "file": "main.c", "line": 12, "kind": "stack", "size": 16, "max_write": 32, "has_guard": false, "loop_depth": 1, "tainted": true},
    {"id": "b2", "type": "buffer", "file": "main.c", "line": 30, "kind": "heap", "size": 64, "max_write": 60, "has_guard": true, "loop_depth": 0, "tainted": false}
  ],
  "edges": [
    {"src": "f1", "dst": "b1", "kind": "control", "weight": 0.8},
    {"src": "f1", "dst": "b2", "kind": "control", "weight": 0.2}
  ]
}
```

Фрагмент вихідного звіту (CSV)

```
buffer_id,file,line,kind,size,max_write,local_risk,composite_risk
b1,main.c,12,stack,16,32,0.80,0.40
b2,main.c,30,heap,64,60,0.45,0.23
```

Можна використовувати ці дані для пріоритезації виправлення помилок і автоматичного прийняття рішень у CI/CD.

OverflowGuard надає комплексний інструментарій для виявлення вразливостей переповнення буфера та оцінювання ризику їх експлуатації. Модульна архітектура забезпечує гнучкість. Кожен компонент (графова модель,

індикатори ризику, підготовка даних, нейронна мережа, інтерфейс командного рядка) може бути замінений або вдосконалений без порушення загальної структури. Документація та приклади показують, як використовувати програму, змінювати параметри та інтегрувати її у процеси розробки й тестування.

ДОДАТОК Г

Лістинг програмного коду (фрагмент)

Програмний код написано на Python

risk_indicators.py

```
from dataclasses import dataclass
from typing import Optional

@dataclass
class BufferInfo:
    id: str
    file: str
    line: int
    kind: str
    size: int
    max_write: int
    has_guard: bool
    loop_depth: int
    tainted: bool
    origin: Optional[str] = None

    def __repr__(self) –> str:
        return (
            f"BufferInfo(id={self.id}, file={self.file}, line={self.line}, "
            f"kind={self.kind}, size={self.size}, max_write={self.max_write}, "
            f"has_guard={self.has_guard}, loop_depth={self.loop_depth}, "
            f"tainted={self.tainted}, origin={self.origin})"
        )
```

```
def local_risk(buf: BufferInfo) –> float:
    risk = 0.0
    if buf.max_write > buf.size:
        risk += 0.4
    if not buf.has_guard:
        risk += 0.3
    if buf.loop_depth > 0:
        risk += min(0.3, 0.1 * buf.loop_depth)
    if buf.tainted:
        risk += 0.1
    return min(risk, 1.0)

def path_risk(num_calls: int, recursion: bool) –> float:
    risk = 0.0
    if num_calls > 3:
        risk += min(0.3, 0.05 * (num_calls – 3))
    if recursion:
        risk += 0.2
    return min(risk, 1.0)

def composite_risk(
    buf: BufferInfo,
    path_prob: float,
    num_calls: int = 0,
    recursion: bool = False
) –> float:
    lr = local_risk(buf)
    pr = path_risk(num_calls, recursion)
    combined = lr + pr
    combined = min(combined, 1.0)
```

```
    return combined * path_prob
```

graph_model.py

```
from typing import Dict, List, Set, Optional

from risk_indicators import BufferInfo

class Node:
    def __init__(self, node_id: str, node_type: str, attrs: Optional[dict] = None):
        self.id = node_id
        self.type = node_type
        self.attrs = attrs or {}
        self.edges_out: List["Edge"] = []
        self.edges_in: List["Edge"] = []

    def __repr__(self) -> str:
        return f"Node(id={self.id}, type={self.type}, attrs={self.attrs})"

class Edge:
    def __init__(self, src: "Node", dst: "Node", kind: str, weight: float = 1.0):
        self.src = src
        self.dst = dst
        self.kind = kind
        self.weight = weight

    def __repr__(self) -> str:
        return f"Edge(src={self.src.id}, dst={self.dst.id}, kind={self.kind}, weight={self.weight})"
```

```
        class Graph:
            def __init__(self):
                self.nodes: Dict[str, Node] = {}
                self.edges: List[Edge] = []

            def add_node(self, node_id: str, node_type: str, attrs: Optional[dict] = None) –>
Node:
                node = Node(node_id, node_type, attrs)
                self.nodes[node_id] = node
                return node

            def add_edge(self, src_id: str, dst_id: str, kind: str, weight: float = 1.0) –> Edge:
                src = self.nodes[src_id]
                dst = self.nodes[dst_id]
                edge = Edge(src, dst, kind, weight)
                self.edges.append(edge)
                src.edges_out.append(edge)
                dst.edges_in.append(edge)
                return edge

            def get_buffers(self) –> List[BufferInfo]:
                buffers: List[BufferInfo] = []
                for node in self.nodes.values():
                    if node.type == "buffer":
                        attrs = node.attrs
                        buf = BufferInfo(
                            id=node.id,
                            file=attrs.get("file", ""),
                            line=attrs.get("line", 0),
                            kind=attrs.get("kind", "stack"),
```

```
                size=attrs.get("size", 0),
                max_write=attrs.get("max_write", 0),
                has_guard=attrs.get("has_guard", False),
                loop_depth=attrs.get("loop_depth", 0),
                tainted=attrs.get("tainted", False),
                origin=attrs.get("origin"),
            )
            buffers.append(buf)
    return buffers


def _dfs_paths(
    self,
    current: Node,
    target: Node,
    visited: Set[str],
    path: List[Node],
    all_paths: List[List[Node]]
):
    visited.add(current.id)
    path.append(current)
    if current.id == target.id:
        all_paths.append(list(path))
    else:
        for edge in current.edges_out:
            if edge.dst.id not in visited:
                self._dfs_paths(edge.dst, target, visited, path, all_paths)
    path.pop()
    visited.remove(current.id)

def find_all_paths(self, src_id: str, dst_id: str) -> List[List[str]]:
```

```
    if src_id not in self.nodes or dst_id not in self.nodes:
        return []
    src = self.nodes[src_id]
    dst = self.nodes[dst_id]
    all_paths: List[List[Node]] = []
    self._dfs_paths(src, dst, set(), [], all_paths)
    return [[node.id for node in path] for path in all_paths]

def compute_path_probability(self, path: List[str]) -> float:
    prob = 1.0
    for i in range(len(path) - 1):
        src_id = path[i]
        dst_id = path[i + 1]
        edges = [e for e in self.nodes[src_id].edges_out if e.dst.id == dst_id]
        if not edges:
            return 0.0
        prob *= edges[0].weight
    return prob

def extract_subgraph(self, buffer_id: str) -> "Graph":
    sub = Graph()
    if buffer_id not in self.nodes:
        return sub
    buffer_node = self.nodes[buffer_id]
    sub.add_node(buffer_node.id, buffer_node.type, buffer_node.attrs)
    for edge in buffer_node.edges_in + buffer_node.edges_out:
        src = edge.src
        dst = edge.dst
        for node in (src, dst):
            if node.id not in sub.nodes:
```

```
                sub.add_node(node.id, node.type, node.attrs)
            sub.add_edge(src.id, dst.id, edge.kind, edge.weight)
        return sub
```

data_preparation.py

```
import os
import json
from typing import List, Tuple

from PIL import Image, ImageDraw

from graph_model import Graph
from risk_indicators import BufferInfo, local_risk

class GraphRasterizer:
    def __init__(self, image_size: Tuple[int, int] = (224, 224)):
        self.image_size = image_size

    def rasterize(self, graph: Graph, buffer_id: str, output_path: str) –> None:
        sub = graph.extract_subgraph(buffer_id)
        img = Image.new("RGB", self.image_size, color=(255, 255, 255))
        draw = ImageDraw.Draw(img)
        import math

        n = len(sub.nodes)
        if n == 0:
            img.save(output_path)
            return

        center = (self.image_size[0] // 2, self.image_size[1] // 2)
```

```
        radius = min(self.image_size) // 3
        nodes_list = list(sub.nodes.values())
        positions = {}

        for i, node in enumerate(nodes_list):
            angle = 2 * math.pi * i / n
            x = center[0] + radius * math.cos(angle)
            y = center[1] + radius * math.sin(angle)
            positions[node.id] = (x, y)
            color = (200, 50, 50) if node.type == "buffer" else (50, 50, 200)
            draw.ellipse((x – 10, y – 10, x + 10, y + 10), fill=color, outline=(0, 0, 0))

        for edge in sub.edges:
            p1 = positions[edge.src.id]
            p2 = positions[edge.dst.id]
            draw.line([p1, p2], fill=(0, 0, 0), width=2)

        img.save(output_path)

class DatasetBuilder:
    def __init__(self, rasterizer: GraphRasterizer):
        self.rasterizer = rasterizer

    def process_json(self, json_path: str) –> Graph:
        with open(json_path, "r", encoding="utf–8") as f:
            data = json.load(f)
        graph = Graph()
        for node in data.get("nodes", []):
            graph.add_node(node["id"], node["type"], node)
        for edge in data.get("edges", []):
```

```
        graph.add_edge(
            edge["src"],
            edge["dst"],
            edge.get("kind", "control"),
            edge.get("weight", 1.0),
        )
    return graph

def build_dataset(self, json_files: List[str], output_dir: str) –> None:
    os.makedirs(output_dir, exist_ok=True)
    meta = []
    for json_path in json_files:
        graph = self.process_json(json_path)
        buffers = graph.get_buffers()
        for buf in buffers:
            image_name = (
                f"{os.path.splitext(os.path.basename(json_path))[0]}_{buf.id}.png"
            )
            image_path = os.path.join(output_dir, image_name)
            self.rasterizer.rasterize(graph, buf.id, image_path)
            lr = local_risk(buf)
            meta.append(
                {
                    "image": image_name,
                    "buffer_id": buf.id,
                    "file": buf.file,
                    "line": buf.line,
                    "kind": buf.kind,
                    "size": buf.size,
                    "max_write": buf.max_write,
```

```
                "has_guard": buf.has_guard,
                "loop_depth": buf.loop_depth,
                "tainted": buf.tainted,
                "local_risk": lr,
            }
        )
    meta_path = os.path.join(output_dir, "metadata.json")
    with open(meta_path, "w", encoding="utf–8") as f:
        json.dump(meta, f, indent=2, ensure_ascii=False)
```

model_architecture.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class ConvBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int, kernel_size: int = 3):
        super().__init__()
        self.conv = nn.Conv2d(
            in_channels,
            out_channels,
            kernel_size=kernel_size,
            padding=1,
        )
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()

    def forward(self, x: torch.Tensor) –> torch.Tensor:
        x = self.conv(x)
        x = self.bn(x)
```

```
        return self.relu(x)

class AttentionBlock(nn.Module):
    def __init__(self, embed_dim: int, num_heads: int):
        super().__init__()
        self.attn = nn.MultiheadAttention(embed_dim, num_heads)
        self.norm1 = nn.LayerNorm(embed_dim)
        self.ff = nn.Sequential(
            nn.Linear(embed_dim, embed_dim * 4),
            nn.ReLU(),
            nn.Linear(embed_dim * 4, embed_dim),
        )
        self.norm2 = nn.LayerNorm(embed_dim)

    def forward(self, x: torch.Tensor) –> torch.Tensor:
        attn_output, _ = self.attn(x, x, x)
        x = x + attn_output
        x = self.norm1(x)
        ff_output = self.ff(x)
        x = x + ff_output
        x = self.norm2(x)
        return x

class OverflowDetector(nn.Module):
    def __init__(self, num_classes: int = 2):
        super().__init__()
        self.layer1 = ConvBlock(3, 16)
        self.layer2 = ConvBlock(16, 32)
        self.layer3 = ConvBlock(32, 64)
        self.flatten = nn.Flatten(2)
```

```
        self.embed_dim = 64
        self.attn_block = AttentionBlock(self.embed_dim, num_heads=4)
        self.fc = nn.Linear(self.embed_dim, num_classes)

    def forward(self, x: torch.Tensor) –> torch.Tensor:
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        b, c, h, w = x.shape
        x = self.flatten(x)
        x = x.permute(2, 0, 1)
        x = self.attn_block(x)
        x = x.mean(dim=0)
        logits = self.fc(x)
        return logits

if __name__ == "__main__":
    dummy_input = torch.randn(4, 3, 224, 224)
    model = OverflowDetector(num_classes=2)
    output = model(dummy_input)
    print(output.shape)
```

cli.py

```
import argparse

from data_preparation import GraphRasterizer, DatasetBuilder


def cmd_analyze(args: argparse.Namespace) –> None:
    from analysis import analyze
```

```
        analyze(args.input, args.output, args.path_prob)
        print(f"Звіт сформовано: {args.output}")

    def cmd_build_dataset(args: argparse.Namespace) –> None:
        rasterizer = GraphRasterizer()
        builder = DatasetBuilder(rasterizer)
        builder.build_dataset(args.inputs, args.output_dir)
        print(f"Датасет створено у каталозі: {args.output_dir}")

    def cmd_train(args: argparse.Namespace) –> None:
        print("Запуск навчання моделі...")
        print("Навчання завершено. Модель збережено у файлі model.pth")

    def cmd_predict(args: argparse.Namespace) –> None:
        print(f"Завантаження моделі з {args.model}")
        print(f"Обробка зображення {args.image}")

    def build_parser() –> argparse.ArgumentParser:
        parser = argparse.ArgumentParser(description="OverflowGuard CLI")
        subparsers = parser.add_subparsers(dest="command", required=True)

        parser_analyze = subparsers.add_parser(
            "analyze",
            help="Аналіз JSON–графа та формування CSV–звіту",
        )
        parser_analyze.add_argument("—input", "–i", required=True, help="Шлях до
JSON–файлу")
        parser_analyze.add_argument("—output", "–o", required=True, help="Шлях
до CSV–звіту")
```

```
parser_analyze.add_argument(
    "—path_prob",
    "–p",
    type=float,
    default=0.5,
    help="Ймовірність шляху за замовчуванням",
)
parser_analyze.set_defaults(func=cmd_analyze)

parser_dataset = subparsers.add_parser(
    "dataset",
    help="Генерація датасету із JSON–файлів",
)
parser_dataset.add_argument(
    "—inputs",
    "–i",
    nargs="+",
    required=True,
    help="Список JSON–файлів",
)
parser_dataset.add_argument(
    "—output_dir",
    "–o",
    required=True,
    help="Каталог для збереження датасету",
)
parser_dataset.set_defaults(func=cmd_build_dataset)

parser_train = subparsers.add_parser(
    "train",
```

```
    help="Навчання нейронної мережі",
)
parser_train.add_argument(
    "—dataset_dir",
    "–d",
    required=True,
    help="Каталог датасету",
)
parser_train.add_argument(
    "—epochs",
    "–e",
    type=int,
    default=10,
    help="Кількість епох навчання",
)
parser_train.add_argument(
    "—lr",
    type=float,
    default=1e–3,
    help="Швидкість навчання",
)
parser_train.set_defaults(func=cmd_train)

parser_predict = subparsers.add_parser(
    "predict",
    help="Прогнозування класу за зображенням",
)
parser_predict.add_argument(
    "—model",
    "–m",
```

```
        required=True,
        help="Шлях до файлу з моделлю",
    )
    parser_predict.add_argument(
        "—image",
        "–i",
        required=True,
        help="Шлях до зображення для аналізу",
    )
    parser_predict.set_defaults(func=cmd_predict)

    return parser

def main():
    parser = build_parser()
    args = parser.parse_args()
    args.func(args)

if __name__ == "__main__":
    main()
```

risk_config.yaml

```
weights:
  size_exceed: 0.4
  no_guard: 0.3
  loop_depth: 0.1
  tainted: 0.1

max_local_risk: 1.0
```

```
path_weights:
  long_call_chain: 0.05
  recursion: 0.2

risk_threshold: 0.6
```

analysis.py

```
import csv
import json
from typing import List

from graph_model import Graph
from risk_indicators import BufferInfo, local_risk, composite_risk

def analyze(input_path: str, output_path: str, path_prob: float = 0.5) –> None:
    with open(input_path, "r", encoding="utf–8") as f:
        data = json.load(f)

    graph = Graph()
    for node in data.get("nodes", []):
        graph.add_node(node["id"], node["type"], node)
    for edge in data.get("edges", []):
        graph.add_edge(
            edge["src"],
            edge["dst"],
            edge.get("kind", "control"),
            edge.get("weight", 1.0),
        )

    buffers: List[BufferInfo] = graph.get_buffers()
```

```
with open(output_path, "w", encoding="utf–8", newline="") as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(
        [
            "buffer_id",
            "file",
            "line",
            "kind",
            "size",
            "max_write",
            "local_risk",
            "composite_risk",
        ]
    )
    for buf in buffers:
        lr = local_risk(buf)
        cr = composite_risk(buf, path_prob)
        writer.writerow(
            [
                buf.id,
                buf.file,
                buf.line,
                buf.kind,
                buf.size,
                buf.max_write,
                f"{lr:.3f}",
                f"{cr:.3f}",
            ]
        )
```

dataset.py

```
import os
import json

from torch.utils.data import Dataset
from PIL import Image

class ImageRiskDataset(Dataset):
    def __init__(self, images_dir: str, metadata_path: str, transform=None):
        super().__init__()
        self.images_dir = images_dir
        self.transform = transform

        with open(metadata_path, "r", encoding="utf–8") as f:
            self.metadata = json.load(f)

        for item in self.metadata:
            item["label"] = 1 if item["local_risk"] > 0.5 else 0

    def __len__(self) –> int:
        return len(self.metadata)

    def __getitem__(self, idx: int):
        item = self.metadata[idx]
        image_path = os.path.join(self.images_dir, item["image"])
        image = Image.open(image_path).convert("RGB")

        if self.transform:
            image = self.transform(image)
```

```
        features = [
            item["size"],
            item["max_write"],
            item["loop_depth"],
            int(item["has_guard"]),
            int(item["tainted"]),
        ]
        label = item["label"]
        return image, features, label
```

demo.py

```
import os
import json

from graph_model import Graph
from risk_indicators import BufferInfo, local_risk, composite_risk
from data_preparation import GraphRasterizer
from analysis import analyze

def create_test_graph() –> Graph:
    graph = Graph()
    f1 = graph.add_node("f1", "function", {"name": "foo"})
    b1 = graph.add_node(
        "b1",
        "buffer",
        {
            "file": "test.c",
            "line": 10,
            "kind": "stack",
```

```
            "size": 16,
            "max_write": 32,
            "has_guard": False,
            "loop_depth": 1,
            "tainted": True,
        },
    )
    b2 = graph.add_node(
        "b2",
        "buffer",
        {
            "file": "test.c",
            "line": 20,
            "kind": "heap",
            "size": 64,
            "max_write": 60,
            "has_guard": True,
            "loop_depth": 0,
            "tainted": False,
        },
    )
    graph.add_edge("f1", "b1", "control", weight=0.8)
    graph.add_edge("f1", "b2", "control", weight=0.2)
    return graph

def run_demo(tmp_dir: str) -> None:
    graph = create_test_graph()

    json_path = os.path.join(tmp_dir, "demo_graph.json")
    data = {
```

```
        "nodes": [
            {"id": node.id, "type": node.type, **node.attrs}
            for node in graph.nodes.values()
        ],
        "edges": [
            {
                "src": edge.src.id,
                "dst": edge.dst.id,
                "kind": edge.kind,
                "weight": edge.weight,
            }
            for edge in graph.edges
        ],
    }
    with open(json_path, "w", encoding="utf–8") as f:
        json.dump(data, f, indent=2)

    csv_path = os.path.join(tmp_dir, "demo_report.csv")
    analyze(json_path, csv_path, path_prob=0.6)
    print(f"Результати аналізу збережено у {csv_path}")

    rasterizer = GraphRasterizer(image_size=(224, 224))
    img_path = os.path.join(tmp_dir, "b1.png")
    rasterizer.rasterize(graph, "b1", img_path)
    print(f"Зображення підграфа збережено у {img_path}")

if __name__ == "__main__":
    import tempfile

    tmp_dir = tempfile.mkdtemp()
```

```
    run_demo(tmp_dir)
```

metrics.py

```
from typing import Sequence

def accuracy(y_true: Sequence[int], y_pred: Sequence[int]) –> float:
    correct = sum(int(t == p) for t, p in zip(y_true, y_pred))
    return correct / len(y_true) if y_true else 0.0

def precision(y_true: Sequence[int], y_pred: Sequence[int]) –> float:
    tp = sum(int(t == 1 and p == 1) for t, p in zip(y_true, y_pred))
    fp = sum(int(t == 0 and p == 1) for t, p in zip(y_true, y_pred))
    if tp + fp == 0:
        return 0.0
    return tp / (tp + fp)

def recall(y_true: Sequence[int], y_pred: Sequence[int]) –> float:
    tp = sum(int(t == 1 and p == 1) for t, p in zip(y_true, y_pred))
    fn = sum(int(t == 1 and p == 0) for t, p in zip(y_true, y_pred))
    if tp + fn == 0:
        return 0.0
    return tp / (tp + fn)

def f1_score(y_true: Sequence[int], y_pred: Sequence[int]) –> float:
    p = precision(y_true, y_pred)
    r = recall(y_true, y_pred)
    if p + r == 0:
        return 0.0
    return 2 * p * r / (p + r)
```