https://doi.org/10.31891/2219-9365-2025-84-10
УДК 004.491.42

СЄРГЄЄВ Євгеній
Хмельницький національний університет
https://orcid.org/0009-0008-9877-9863
e-mail: ysierhieiev@gmail.com
КЛЬОЦ Юрій
Хмельницький національний університет
https://orcid.org/0000-0002-3914-0989
klots@khmnu.edu.ua

КОМПОЗИТНА ОЦІНКА РИЗИКУ ПЕРЕПОВНЕННЯ БУФЕРА І ЇЇ ТРАНСЛЯЦІЯ В ДІЇ CI/CD

У роботі представлено відтворюваний метод управління ризиками переповнення буфера в C/C++ у рамках CI/CD-пайплайнів, який поєднує прозорі, автоматизовані інженерні дії з формальною оцінкою критичності. Відповідно до чотирирівневої стратифікації за ступенем серйозності, запропонований комплексний показник включає індикатори ризику на рівні шляху та локальні індикатори ризику, а також враховує специфічну поведінку класу (стек/купа/один поза межами). Ми визначаємо тригери виправлення для операційної інтеграції: політики часу виправлення (SLA) з чіткими термінами та рішеннями конвеєра (пропустити, попередити з квитком, заблокувати) негайно запускаються при відсутності перевірок меж та порушеннях індексації. Фіксація профілів препроцесора та версій інструментарію, реєстрація маніфестів виконання та збереження артефактів аудиту (звіти SARIF/HTML, параметри середовища та журнали рішень) допомагають гарантувати відтворюваність. Ми порівнюємо цей підхід з cppcheck, flawfinder та базовим рівнем на основі зору (YOLO) у нашому експериментальному дослідженні, яке охоплює шість відкритих проектів C/C++ за двома профілями збірки (Debug/Release). Згідно з оцінками точності, відтворюваності, F1, специфічності, стабільності між запусками та часу аналізу кожного файлу, запропонований метод підтримує затримку, прийнятну для CI/CD, одночасно досягаючи вищого F1 та специфічності, а також найкращої відтворюваності між декількома запусками. Крім того, інтегрований у SLA робочий процес покращує надійність випуску та знижує операційний ризик на етапі PR/commit, збільшуючи відсоток випадків високої/критичної важливості, які вирішуються вчасно. Враховуючи все вищезазначене, формалізація критичності та поєднання її з контрольними точками якості та тригерами виправлення призводить до закритого циклу "виявлення–виправлення–верифікація", який працює з різними конфігураціями збірки, мовами та репозиторіями.

Ключові слова: переповнення буфера; статичний аналіз; критичність; композитний ризик; CI/CD-гейти; SLA.

SIERHIEIEV Yevhenii, KLOTS Yurii
Khmelnytskyi National University

COMPOSITE RISK ASSESSMENT OF BUFFER OVERFLOWS AND ITS TRANSLATION INTO CI/CD ACTIONS

We describe a reproducible method for buffer-overflow risk management in C/C++ within CI/CD pipelines that combines transparent, automatable engineering actions with a formal criticality assessment. Following a four-level severity stratification, the suggested composite metric incorporates path-level and local risk indicators and takes class-specific behavior (Stack/Heap/Off-by-one) into consideration. We define fix triggers for operational integration: time-to-fix policies (SLA) with clear deadlines and pipeline decisions (pass, warn with ticket, block) are immediately triggered by missing boundary checks and indexing violations. Pinning preprocessor profiles and toolchain versions, logging run manifests, and preserving audit artifacts (SARIF/HTML reports, environment parameters, and decision logs) all help to guarantee reproducibility. We benchmark the approach against cppcheck, flawfinder, and a vision-based baseline (YOLO) in our experimental study, which covers six open-source C/C++ projects under two build profiles (Debug/Release). According to evaluations of precision, recall, F1, specificity, run-to-run stability, and per-file analysis time, the suggested method maintains CI/CD-feasible latency while achieving higher F1 and specificity as well as the best reproducibility across multiple runs. Additionally, the SLA-integrated workflow improves release reliability and lowers operational risk at the PR/commit stage by increasing the percentage of High/Critical cases that are resolved on time. All things considered, formalizing criticality and combining it with quality gates and fix triggers results in a closed "detection–fix–verification" loop that works with different build configurations, languages, and repositories.

Keywords: buffer overflow; static analysis; criticality; composite risk; CI/CD gates; SLA.

Стаття надійшла до редакції / Received 27.09.2025
Прийнята до друку / Accepted 30.10.2025

ПОСТАНОВКА ПРОБЛЕМИ У ЗАГАЛЬНОМУ ВИГЛЯДІ ТА ЇЇ ЗВ'ЯЗОК ІЗ ВАЖЛИВИМИ НАУКОВИМИ ЧИ ПРАКТИЧНИМИ ЗАВДАННЯМИ

Безпека пам'яті в програмах мов C/C++ залишається одним із ключових чинників надійності сучасних комп'ютерних систем. Серед відомих помилок саме переповнення буфера — як стекове, так і купне, включно з варіацією off-by-one — формує критичний клас уразливостей, що проявляється в системному ПЗ, драйверах і RTOS. Практика експлуатації таких помилок доводить, що навіть за наявності класичних захистів (canaries, ASLR, DEP тощо) ризики не усуваються повністю, а варіативність збірки, умов компіляції та

конфігурацій препроцесора призводить до різних "ландшафтів" уразливостей для однієї й тієї ж кодової бази. У контексті CI/CD це створює окремий виклик відтворюваності: незначні зміни профілю, версій інструментів чи залежностей можуть змінювати результати статичного аналізу, ускладнюючи аудит і контроль якості.

Проблема, яку ми розв'язуємо, полягає у відсутності цілісного, детермінованого конвеєра "аналіз → рішення → контроль", який перетворює результати виявлення потенційних переповнень на послідовні інженерні дії з прозорими правилами реагування. Такий конвеєр має забезпечувати стабільність висновків між запусками (за фіксованих профілів препроцесора та версій інструментів), знижувати ймовірність дефектів на ранніх етапах життєвого циклу і обмежувати наслідки можливої експлуатації вразливостей.

Метою статті є перетворення отриманих у попередньому аналізі формальних індикаторів ризику на практичний метод захисту коду з явними порогами критичності та інтеграцією в типові CI/CD-процеси. Наукова новизна полягає у:

1. Уніфікації правил критичності для різних підкласів переповнення (Stack/Heap/Off-by-one) на основі формалізованих ознак та вагових оцінок потоків даних/керування;
2. Побудові відтворюваного циклу "виявлення — виправлення — верифікація", де рішення приймаються за єдиними політиками (гейтами) і супроводжуються артефактами для аудиту.

Об'єктом дослідження є процес автоматизованого забезпечення безпеки коду щодо переповнення буфера в умовах промислового CI/CD. Предметом — методи формалізації ризиків та правила їхнього інженерного застосування (пороги, політики, шаблони виправлень). Запропонований підхід ґрунтується на уніфікованому графовому поданні коду (поєднання CFG/DFG) і системі ризикових індикаторів, що дозволяє зводити знайдені детекції до чітких правил безпеки, пов'язаних із шаблонами патчів та автоматизованими рішеннями "пропустити/блокувати/вимагати виправлення" у пайплайнах збірки. Така конструкція робить результати стабільними для аудиту, придатними до масштабування на великі репозиторії та сумісними з вимогами індустріальної відтворюваності..

АНАЛІЗ ОСТАННІХ ДОСЛІДЖЕНЬ І ПУБЛІКАЦІЙ

Сучасні механізми пом'якшення помилок пам'яті — адресна рандомізація, запобігання виконанню даних, стек-канарейки та підсилені бібліотечні перевірки — докладно узагальнені в міждисциплінарних оглядах і демонструють значиме, але неповне зниження ризику експлуатації переповнень буфера. Зокрема, аналіз показує, що комбінації витоків адрес, JIT-спреїнгу та ROP-ланцюжків створюють дієві траєкторії обходу навіть за наявності кількох одночасних бар'єрів, що підкреслює потребу усунення першопричини на рівні вихідного коду та процесу збірки [1].

На рівні аналізу вихідних текстів коду помітний прогрес пов'язаний із представленням програм як послідовностей і графів. Підходи типу VulBERTa, що спираються на попередньо навчені трансформери, зменшують залежність від ручної інженерії ознак і водночас відтворюють контекстні зв'язки, релевантні для переповнень (зокрема залежності між операторами запису та граничними перевірками) [2]. Розвитком цієї лінії є гетерогенні графи вразливостей, де поєднання AST/CFG/DFG/CPG надає моделі структурно-семантичну інформацію; показано, що таке подання покращує виявлення контекстно зумовлених дефектів пам'яті, у т.ч. стекових і купних переповнень [3]. Окремий напрям фокусується на локалізації: застосування програмного slicing дає змогу звузити підграф до релевантних гілок потоку керування/даних, зменшуючи хибні спрацьовування та полегшуючи прив'язку детекції до конкретного рядка/виклику функції [4].

Паралельно розвиваються підходи, орієнтовані на бінарні артефакти: графовий матчинг дозволяє зіставляти гомологічні уразливості в виконуваних файлах і бібліотеках, коли вихідний код недоступний або розсинхронізований зі збіркою. Такі моделі ефективні для повторного виявлення відомих патернів на рівні машинного коду та підтримують ретроактивний аналіз пост-компіляційних збірок, але обмежено придатні для раннього інженерного виправлення в репозиторії, що вимагає SAST-фокусу та інтеграції з CI/CD [5].

Проблематика даних і розмітки безпосередньо визначає узагальнюваність моделей. У підходах на основі commit-пар (наприклад, CrossVul) показано, що формування пар "вразливий→виправлений" з репозиторіїв відкритого коду дає корисні сигнали для навчання, однак водночас створює канали витоку ознак (leakage) через шаблонні патерни редагувань, дублікати та перетин авторів/проєктів між train/test [6]. Систематичні огляди 2018–2023 рр. наголошують, що якість джерел (баланс класів, репрезентативність стеків і доменів, стабільність розмітки), а також розведення за проєктами та версіями є ключовими передумовами валідного порівняння й реальної переносимості результатів [7].

Окремим напрямом є поєднання контекстних графів коду (CPG) з великими мовними моделями, де застосовуються як інструкційно-керовані підказки, так і контрфактичні пояснення для графових рішень (пояснюваність через мінімальні зміни у вузлах/ребрах, що впливають на клас) [8]. Такі підходи покращують інтерпретованість і локалізацію, але залишаються вразливими до галюцинацій, чутливості до формулювання підказок і проблем відтворюваності між запуском моделей і середовищами (версії токенізаторів, сід, дрібні різниці у препроцесингу) [8, 9]. Для промислового CI/CD це накладає вимоги до фіксації профілів препроцесора, контрольованих версій моделей та журналювання артефактів.

Для емпіричних порівнянь у межах статті доцільно використовувати класичні статичні аналізатори як доступні бейзлайни — зокрема cppcheck і flawfinder — які забезпечують відтворювані запуски, прийнятну швидкість аналізу та зрозумілу семантику попереджень [10, 11]. Ці інструменти не конкурують із сучасними ML/графовими методами за максимальною якістю детекції, але надають корисні опорні точки для оцінювання точності/специфічності та часових витрат, а також добре інтегруються в пайплайни збірки для контролю регресій.

Узагальнюючи огляд, простежується зсув від ізольованих ознак та евристик до структурних подань коду (узгоджених графів керування й даних) у поєднанні з дисципліною підготовки датасетів (контроль витоків, балансування класів, проєктне розведення, стабілізація профілів препроцесора). За таких умов пояснюваність рішень і відтворюваність усього ланцюга аналізу набувають не меншої ваги, ніж максимізація точності на окремих вибірках, що узгоджує SAST-підходи з вимогами CI/CD та полегшує перенесення результатів між репозиторіями, мовами і конфігураціями збірки.

ФОРМУЛЮВАННЯ ЦІЛЕЙ СТАТТІ

Метою роботи є розроблення й експериментальна перевірка відтворюваного конвеєра виявлення переповнень буфера у C/C++ в умовах CI/CD; конвеєр ґрунтується на уніфікованому графі програми (CFG+DFG) з атрибутами буферів і операцій пам’яті, обчисленні локальних і шляхових індикаторів ризику та растуванні інформативних підграфів у багатоканальні зображення з класами Stack/Heap/Off-by-one для подальшого прийняття інженерних рішень (правила критичності, гейти, SLA).

ВИКЛАД ОСНОВНОГО МАТЕРІАЛУ

Вхідним матеріалом для методу є впорядкована множина локалізованих детекцій переповнення буфера з класами {Stack, Heap, Off-by-one}, кожна з яких має координати у вихідному коді (файл, рядок), відображення на елементи графів керування та даних (CFG/DFG) — буфер, операцію запису/копіювання, індекс або зсув — а також службові метадані для відтворюваності (ідентифікатор коміту, профіль препроцесора, версії інструментів побудови AST/CFG/DFG). Для подальшого прийняття рішень така детекція супроводжується сукупністю формальних індикаторів ризику, що задаються незалежно від конкретної реалізації базового детектора.

Базову перевірку перевищення обсягу вводу над місткістю цільового буфера задаємо відносним показником:

$$r_{cap} = \max\left(0, \frac{L - C}{\max(C, \varepsilon)}\right), \varepsilon > 0 \tag{1}$$

де L — оцінена довжина вводу (у байтах/елементах), C — оцінена місткість буфера (в тих самих одиницях), ε — малий додатний параметр для уникнення ділення на нуль.

Для стекових сценаріїв фіксується внесок відсутності контролю цілісності кадру виклику:

$$r_{stack} = 1 - c_{can} \tag{2}$$

де $c_{can} \in \{0, 1\}$ позначає наявність механізму типу stack-canary (1 — наявний, 0 — відсутній), а r_{stack} інтерпретується як додаткова складова ризику для стеку.

Граничні помилки індексації (off-by-one) моделюються гладким порогом:

$$r_{obo} = \sigma\big(i - (n - 1)\big), \qquad \sigma(x) = \frac{1}{1 + e^{-kx}}, k > 0 \tag{3}$$

де i — індекс доступу, n — кількість елементів у буфері, σ(·) — сигмоїдна функція, k — параметр крутості переходу; отримуємо зростання ризику при виході за верхню межу.

Накопичення небезпеки вздовж програмного шляху описується композицією локальних внесків:

$$R(p) = 1 - \prod_{e \in p}(1 - r_e), \qquad R_{path} = \max_{p \in P} R(p) \tag{4}$$

де P — множина релевантних шляхів у CFG/DFG від джерела даних до операції запису, p — конкретний шлях, e — ребро/операція на шляху, $r_e \in [0,1]$ — локальний ризик (наприклад, копіювання без перевірки меж чи небезпечна індексація), R(p) — кумулятивний ризик шляху, R_{path} — найгірший із таких ризиків для даної детекції.

Для узагальнення на рівні компонента використовується зважене середнє:

$$R_{mod} = \frac{\sum_{d \in D} w_d R_{path}(d)}{\sum_{d \in D} w_d} \quad (5)$$

де D — множина детекцій у модулі, $w_d \geq 0$ — ваговий коефіцієнт детекції d (з урахуванням критичності чи обсягу коду), $R_{path}(d)$ — шляховий ризик для d.

Підсумкова метрика критичності для одиничної детекції визначається як клас-обізнана лінійна комбінація внесків:

$$S = w_k\left(\alpha r_{cap} + \beta r_{obo} + \gamma R_{path}\right) + \delta r_{stack} \quad (6)$$

де k∈{Stack,Heap,Off} — клас детекції, $w_k > 0$ — ваговий коефіцієнт класу, α, β, γ, δ≥0 — коефіцієнти змішування відповідних індикаторів; значення S слугує основою для подальших правил критичності, вибору шаблонів виправлення та інтеграції в політики якості збірки в CI/CD.

Для клас-обізнаної пріоритизації фіксуємо вектор ваг:

$$w = w_{Stack}, w_{Heap}, w_{Off} \quad (7)$$

де $w_{Stack}, w_{Heap}, w_{Off} > 0$ — вагові коефіцієнти для стекових, купних та off-by-one детекцій відповідно, w задає відносну небезпеку класів з урахуванням домену застосування.

Тоді стратифікацію ризику для прийняття інженерних рішень задаємо пороговою шкалою:

$$sev(S) = \begin{cases} Low, & 0 \leq S \leq \tau_1 \\ Medium, & \tau_1 \leq S < \tau_2 \\ High, & \tau_2 \leq S < \tau_3 \end{cases}, 0 < \tau_1 < \tau_2 < \tau_3 \quad (8)$$

де τ_1, τ_2, τ_3 — пороги критичності, що узгоджують безперервну метрику S з дискретними діями в CI/CD (допуск/попередження/блокування).

Ініціювання виправлень та автоматичних блокувань спирається на формалізовані тригери. Для перевірки наявності межового гварда вводимо:

$$T_{guard} = 1 - b_{guard} \quad (9)$$

де $b_{guard} \in \{0,1\}$ — індикатор наявності явної перевірки меж або місткості буфера перед записом (1 — є, 0 — немає), $T_{guard} = 1$ сигналізує потребу у виправленні та/або блокуванні залежно від sev(S).

Для практичного застосування параметри інтерпретуємо так: S акумулює локальний надлишок вводу над місткістю, ризики індексації та шляховий внесок поширення небезпеки в графі програми, w підсилює класи, що мають гірші наслідки в даному домені, пороги τ_1, τ_2, τ_3 встановлюють політику якості.

Шаблони виправлень відповідають типу дефекту та контексту. Для рядкових операцій у стеку (strcpy/конкатенації) застосовується заміна на обмежене копіювання з явним контролем місткості, нормалізацією довжини вводу та коректною обробкою термінатора; для байтових копіювань (memcpy у локальний буфер) — подвійний гвард на розмір вхідного й вихідного буферів та, за потреби, перехід на memmove; для купних сценаріїв (malloc/new + копіювання) — перевірка на цілочисельне переповнення при обчисленні місткості (аналог reallocarray), перевірка результату виділення та безпечна ініціалізація. Усі дії формально прив'язуються до спрацювань і до рівня sev(S), що забезпечує відтворювану поведінку в конвеєрах збірки.

Інтеграція методу в CI/CD передбачає однозначне відображення оціненої критичності на рішення конвеєра та артефакти, що супроводжують кожне спрацювання. Для кожної детекції, яка потрапляє у збірку, визначається рівень небезпеки (Low/Medium/High/Critical) і на його основі приймається одна з трьох дій: пропустити, пропустити з попередженням і постановкою задачі, або заблокувати збірку до виправлення. Рішення доповнюється згенерованим звітом у форматі SARIF (для машинної інтеграції з репозиторіями та трекерами) і коротким HTML-звітом для людей: у ньому наводяться координати у коді, клас детекції (Stack/Heap/Off-by-one), релевантний контекст (фрагмент коду, частини CFG/DFG у текстовому описі), пояснення причин спрацювання, а також рекомендований шаблон виправлення. На етапах "попередження" і "блокування" система автоматично створює тікет у трекері завдань із підстановкою усіх атрибутів (ідентифікатор коміту, гілка, сервісна інформація про профіль препроцесора та версії інструментів), щоб забезпечити прозору трасованість.

Політика строків усунення (SLA) фіксується на рівні проекту: для Critical — найкоротший дедлайн і автоматичне блокування конвеєра; для High — жорсткий термін із можливістю тимчасового винятку лише за погодженням відповідального; для Medium — постановка задачі з розумним дедлайном без блокування

збірки; для Low — інформаційне попередження й моніторинг на предмет регресій. Таке стратифіковане керування зменшує операційні втрати (збірки не "валяться" через низьку критичність), але гарантує, що найнебезпечніші випадки не пройдуть у реліз без виправлення.

Післявиправлювальний контроль виконується у два кроки. По-перше, кожна детекція має парні стани "до" і "після" патча; система фіксує, чи усунено причину (наприклад, додано перевірку меж, змінено небезпечний виклик, переглянуто розмір буфера або логіку індексації), та оновлює статус тікета. По-друге, на рівні модуля/репозиторію розраховуються процесні показники: частка небезпек високої критичності, закритих у межах SLA; середній час усунення для різних рівнів; частка повторних спрацювань у тих самих ділянках коду (індикатор латентного боргу); стабільність детекцій між повторними прогонами за сталих профілів збірки (метрика відтворюваності). Узагальнені підсумки виносяться у щотижневий/щомісячний звіт і використовуються для коригування ваг класів і порогів критичності з урахуванням реальних ризиків конкретного домену.

Для забезпечення аудиту зберігаються всі артефакти прогонів: SARIF і HTML-звіти, журнал рішень конвеєра з точними мітками часу, трейс "детекція → патч → повторна верифікація", параметри середовища (версії компілятора, інструментів побудови графів, профілі препроцесора, хеш коміту), а також витяги контексту коду. Це дозволяє відтворити будь-яке рішення, порівняти збірки між гілками чи платформами, довести виконання політик на зовнішніх перевірках і — найважливіше — замкнути цикл "виявлення — виправлення — верифікація" у режимі щоденних розробницьких практик.

ЕКСПЕРИМЕНТ

Оцінювання проведено на шести відкритих проєктах C/C++ з двома профілями збірки (Debug/Release). Для кожного репозиторію зафіксовано "коміт-знімок", а версії інструментів, ключі збірки та профіль препроцесора закріплено у маніфесті прогону; це забезпечує відтворюваність, тобто повторні запуски на тих самих "знімках" дають ідентичні результати. Для порівняння взято два класичні статичні аналізатори (cppcheck, flawfinder), візуальний базовий підхід на основі детекції у кадрах інформативних підграфів (YOLO), а також запропонований метод, який поєднує правила критичності й конвеєрні рішення допуску у CI/CD. Оцінювання здійснювалося за показниками Precision, Recall, F1, специфічності, стабільності результатів у повторних прогонах, середнього часу аналізу на файл та частки випадків High/Critical, закритих у межах визначених SLA-термінів. Усі інструменти запускалися на однаковій апаратній конфігурації; у таблиці наведено усереднені (по проєктах і обох профілях) значення з медіанним часом.

Таблиця 1

Загальне виявлення за проектами

Метод	*Точність*	*Відтворення*	*F1-міра*	*Час (сек/файл)*
Cppcheck	61,7 %	48,5 %	54.4 %	2.1
Flawfinder	55,3 %	42,8 %	48.9 %	1.5
YOLO	68.2 %	64.4 %	66.1 %	7.8
Наш метод	76.1 %	71.0 %	73.3 %	9.2

Отримані результати свідчать, що запропонований підхід забезпечує вищі F1 і специфічність порівняно з бейзлайнами, водночас демонструє найкращу стабільність між повторними прогонами завдяки фіксації профілів і параметрів збірки. Порівняно з класичними статичними аналізаторами час обробки є більшим, однак залишається прийнятним для практичних конвеєрів; натомість інтегрована стратифікація критичності та SLA підвищують частку своєчасно закритих High/Critical випадків. Візуальний базовий підхід (YOLO) виявляє більше кандидатів, але гірше утримує відтворюваність при зміні умов препроцесора; це узгоджується з тим, що наш метод опирається на детерміноване графове подання і чіткі правила прийняття рішень у CI/CD.

ВИСНОВКИ З ДАНОГО ДОСЛІДЖЕННЯ І ПЕРСПЕКТИВИ ПОДАЛЬШИХ РОЗВІДОК У ДАНОМУ НАПРЯМІ

У роботі сформовано відтворюваний підхід до керування ризиками переповнення буфера в C/C++. Запроваджено композитну метрику критичності та її стратифікацію, визначено формальні "вимикачі" виправлень для контекстів без гвардів і для індексаційних порушень, інституціоналізовано рішення конвеєра у вигляді CI/CD-гейтів і політики строків усунення, а також закріплено післявиправлювальний контроль через показники зниження ризику та KPI виконання SLA. Сукупно це утворює замкнений життєвий цикл "виявлення — виправлення — верифікація", який зберігає детермінованість, трасованість і придатність до аудиту.

Практичний ефект полягає у переносимості рішень у DevOps-процес. Ризики стратифікуються ще на етапі коміту/PR, гейти перетворюють оцінку критичності на однозначні дії, а артефакти прогонів (звітність, журнал рішень, параметри середовища) забезпечують прозору відповідність політикам якості. Експериментальна частина продемонструвала підвищення якості виявлення та стабільності результатів за

прийнятного часу аналізу, а також зростання частки вчасно закритих випадків високої критичності.

Водночас слід враховувати обмеження: точність оцінювання локальних і шляхових внесків (зокрема ваг потоків w(e)) залишається чутливою до особливостей кодової бази; платформена та RTOS-специфіка може вимагати окремих профілів і налаштувань; перспективним є поєднання запропонованого SAST-підходу з DAST/fuzzing для верифікації граничних випадків і зменшення як хибнопозитивних, так і хибнонегативних спрацювань. Напрями подальших робіт включають автоматизовану калібровку порогів під домен, навчання класових ваг на історичних даних, інтеграцію зі створенням патч-шаблонів і розширення підхідності до змішаних конфігурацій зборок і платформ.

References

1. Butt, M.A., Ajmal, Z., Khan, Z.I., Idrees, M., & Javed, Y. 2022. An In-Depth Survey of Bypassing Buffer Overflow Mitigation Techniques. Applied Sciences. 12(13):6702. https://doi.org/10.3390/app12136702
2. Zhao, Z., Xu, M., Zhang, Y., Chen, L., Luo, X., Deng, Y., et al. 2024. GMN+: A Binary Homologous Vulnerability Detection Method Based on Graph Matching Neural Network with Enhanced Attention. Applied Sciences. 14(22):10762. https://doi.org/10.3390/app142210762
3. Hanif, H., & Maffeis, S. 2022. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. In: 2022 International Joint Conference on Neural Networks (IJCNN). https://doi.org/10.1109/IJCNN55064.2022.9892280
4. Salimi, S., & Kharrazi, M. 2022. VulSlicer: Vulnerability Detection Through Code Slicing. Journal of Systems and Software. 191:111310. https://doi.org/10.1016/j.jss.2022.111310
5. Tian, Y., Chen, S., Yin, F., & Zhou, W. 2021. CrossVul: A Cross-Language Vulnerability Dataset with Commit Data. In: ESEC/FSE 2021. https://doi.org/10.1145/3468264.3473122
6. Wang, L., Chen, H., Sun, S., & Wang, J. 2022. HGVul: Heterogeneous Graph-Based Code Vulnerability Detection Method. Security & Communication Networks. Article ID 1919907. https://doi.org/10.1155/2022/1919907
7. Guo, Y., Yi, Y., Zhang, B., et al. 2024. Precise Compositional Buffer Overflow Detection via Heap Disjointness. In: ISSTA 2024. https://doi.org/10.1145/3650212.3652110
8. Xu, L., Yang, H., & Xu, M. 2020. ELAID: Integer Overflow to Buffer Overflow Detection Using Static Analysis. Cybersecurity. 3(1):13. https://doi.org/10.1186/s42400-020-00058-2
9. Özger, R., & ÖztekİN, B. 2023. Detection of Buffer Overflow Attacks with Memoization-Based Rule Set. Journal of Computer Science Research. 5(4):60–66. https://doi.org/10.30564/jcsr.v5i4.6044
10. Dahl, A., Erdodi, L., & Zennaro, M. 2020. Detecting Stack Buffer Overflow in Assembly with Recurrent Neural Networks. arXiv. https://arxiv.org/abs/2012.15116
11. Zhou, H., Sun, L., Zhang, X., et al. 2024. A Quantum Convolutional Neural Network-Based Approach for Software Vulnerability Detection. Scientific Reports. 14:56871. https://doi.org/10.1038/s41598-024-56871-z
12. Zhang, Z., Li, Z., Chen, Y., et al. 2024. CMFVD: Context-Aware Multi-Feature Vulnerability Detection with GCN and Bi-GRU. Sensors. 24(5):1351. https://doi.org/10.3390/s24051351
13. Bennouki, N., Hadi, M., & Kobbane, A. 2024. Vulnerability Detection Methods: A Systematic Review. Journal of Cybersecurity & Privacy. 4(4):40. https://doi.org/10.3390/jcp4040040
14. Patil, S., Kim, D., Meng, S., Kong, M., & Gupta, R. 2019. CSOD: Context-Sensitive Overflow Detection. In: 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). https://doi.org/10.1109/CGO.2019.8661198
15. Padhi, S., Ranganathan, N., & De, S. 2019. Buffer Overflow Vulnerabilities in FreeRTOS and Ubuntu: An Experimental Study. In: 2019 Global IoT Summit (GIoTS). https://doi.org/10.1109/GIOTS.2019.8766434
16. Li, Z., Yang, Z., Xu, M., et al. 2023. Large Language Models for Software Vulnerability Detection: A Survey. arXiv. https://arxiv.org/abs/2311.12420
17. Xu, H., Liu, Y., Wang, H., et al. 2023. BOP: A Buffer Overflow Protection for RISC-V. Cybersecurity. 6(1):26. https://doi.org/10.1186/s42400-023-00164-x
18. Başar, A., Kurnaz, S., & Bostancı, E. 2025. Deep Learning Architectures for Detecting SQL Injection in Python Code. Electronics. 14(17):3436. https://doi.org/10.3390/electronics14173436
19. Weyss, T., Schabauer, L., & Schrittwieser, S. 2025. Rust-IR-BERT: Vulnerability Detection for Rust Using LLVM IR and GraphCodeBERT. Machine Learning & Knowledge Extraction. 7(3):79. https://doi.org/10.3390/make7030079
20. Yuan, L., Fang, Y., Zhang, Q., Liu, Z., & Xu, Y. 2025. Go Source Code Vulnerability Detection Method Based on Graph Neural Network. Applied Sciences. 15(12):6524. https://doi.org/10.3390/app15126524