https://doi.org/10.31891/2307-5732-2026-361-45
УДК 004.491.42

СЄРГЄЄВ ЄВГЕНІЙ
Хмельницький національний університет
https://orcid.org/0009-0008-9877-9863
e-mail: ysierhieiev@gmail.com

ПІДГОТОВКА ДАНИХ НА ОСНОВІ ГРАФІКІВ ДЛЯ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ ПЕРЕПОВНЕННЯ БУФЕРА В КОДІ В РАМКАХ CI/CD-ПРОЦЕСІВ

У роботі представлено відтворюваний підхід до керування ризиками переповнення буфера у C/C++ в умовах CI/CD, що поєднує формалізовану оцінку критичності з прозорими інженерними діями в конвеєрі збірки. Запропонована композитна метрика інтегрує локальні та шляхові індикатори ризику й ураховує класову специфіку (Stack/Heap/Off-by-one), після чого застосовується стратифікація на рівні Low/Medium/High/Critical. Для практичної інтеграції введено «вимикачі» виправлень: відсутність перевірок меж і індексаційні порушення ініціюють автоматичні дії (пропуск, попередження з постановкою задачі, блокування) та політику строків усунення (SLA) із фіксованими дедлайнами. Відтворюваність забезпечується закріпленням профілів препроцесора, версій інструментів і маніфестів прогонів, а також збереженням артефактів (SARIF/HTML-звіти, параметри середовища, журнали рішень) для аудиту. Експериментальну оцінку виконано на шести проєктах відкритого коду для двох профілів збірки (Debug/Release) з порівнянням проти cppcheck, flawfinder і візуального базового підходу (YOLO). Оцінювання за Precision, Recall, F1, специфічністю, стабільністю повторних прогонів і часом аналізу на файл засвідчило перевагу запропонованого методу: підвищення F1 і специфічності, найвищу відтворюваність між повторними прогонами за збереження прийнятного часу аналізу для CI/CD. Крім того, інтеграція з політикою SLA збільшує частку своєчасно закритих випадків High/Critical, що безпосередньо знижує операційні ризики на етапі PR/коміту та підвищує надійність релізів. Отримані результати демонструють, що формалізація критичності в зв'язці з «вимикачами» та гейтами якості утворює замкнений цикл «виявлення — виправлення — верифікація», придатний до масштабування на різні репозиторії, мови та конфігурації збірки.

Ключові слова: *переповнення буфера; статичний аналіз; уніфікований граф програми; індикатори ризику; CI/CD; відтворюваність; off-by-one.*

SIERHIEIEV YEVHENII
Khmelnytskyi National University, Khmelnytskyi, Ukraine

GRAPH-BASED DATA PREPARATION FOR DETECTING BUFFER OVERFLOW VULNERABILITIES IN CODE WITHIN CI/CD PIPELINES

We offer a clear method for managing buffer-overflow risks in C/C++ within CI/CD pipelines. This method combines a formal criticality assessment with clear and manageable engineering actions. Our metric includes local and path-level risk indicators and considers class-specific behavior such as Stack, Heap, and Off-by-one errors. It also includes a four-level severity scale: Low, Medium, High, and Critical. For operational integration, we define fix triggers. Missing boundary checks and indexing violations directly translate into pipeline decisions, such as pass, warn with ticket, or block. They also influence time-to-fix policies with specific deadlines. We ensure reproducibility by fixing preprocessor profiles and toolchain versions, recording run manifests, and keeping audit artifacts like SARIF or HTML reports, environment parameters, and decision logs.Our experimental study looks at six open-source C/C++ projects using two build profiles: Debug and Release. We compare our method with cppcheck, flawfinder, and a vision-based baseline called YOLO. We evaluate performance based on Precision, Recall, F1, specificity, run-to-run stability, and per-file analysis time. Our approach achieves higher F1 and specificity, along with the best reproducibility across repeated runs while maintaining feasible latency for CI/CD. Also, our SLA-integrated workflow improves the timely resolution of High and Critical cases, which reduces operational risk during the PR or commit stage and boosts release reliability. In conclusion, formalizing criticality and linking it with fix triggers and quality gates creates a complete detection, fix, and verification loop. The research presents migration guidelines for current codebases while demonstrating that organizations achieve better developer resistance management through a staged adoption approach which begins with high-risk modules and uses codified templates for expansion, while preserving safety guarantees and governance outcomes. This method can scale across different repositories, languages, and build configurations.

Keywords: *buffer overflow; static analysis; unified program graph; risk indicators; CI/CD; reproducibility; off-by-one*

Стаття надійшла до редакції / Received 14.12.2025
Прийнята до друку / Accepted 11.01.2026
Опубліковано / Published 29.01.2026

Постановка проблеми

Програмно-керована цифрова трансформація розширила площу атаки в сферах, критичних для безпеки (авіація, залізниця, енергетика, телекомунікації), де один дефект безпеки пам'яті може призвести до порушення роботи сервісу або віддаленого виконання коду. Незважаючи на прогрес інструментів SAST, переповнення буфера (стек, купа, off-by-one) залишаються одними з найбільш впливових класів, оскільки вони виникають в результаті тонких взаємодій потоку управління та даних, конфігурацій препроцесора та профілів збірки. У високошвидкісних середовищах CI/CD організаціям потрібні відтворювані, автоматизовані конвеєри, які виявляють такі ризики на ранній стадії та послідовно в усіх проектах і цільових профілях.

Ми мотивуємо підхід до статичного аналізу "від даних до бачення": замість того, щоб покладатися лише на ручні правила, ми (i) аналізуємо фіксований знімок коду, (ii) будуємо уніфікований графік програми (CFG+DFG з анотаціями буфера/захисту та вагами ребер), (iii) обчислюємо локальні та ланцюгові індикатори ризику, і (iv) растеризуємо інформативні підграфи в багатоканальні зображення з мітками класів (Stack, Heap, Off-by-one). Ця трансформація робить складну структуру програми придатною для зрілих, ефективних з точки зору даних

конвеєрів бачення, зберігаючи при цьому детермінізм і можливість аудиту, необхідні в регульованих умовах.

Аналіз останніх досліджень і публікацій

Переповнення буфера залишається актуальною проблемою навіть у добре налагоджених процесах розробки. Незважаючи на значні вдосконалення в техніках захисту та найкращих практиках кодування, реальні атаки все ще використовують тонкі відмінності між припущеннями розробників та фактичною поведінкою кодових шляхів і середовищ виконання. Огляди підкреслюють, що класичні переповнення все ще трапляються в сучасних системних компонентах, бібліотеках та програмному забезпеченні, а їх обхід часто досягається шляхом зміни контексту, конфігурації збірки або використання фрагментів коду, де перевірки меж не відповідають реальній семантиці копіювання та форматування даних [1].

Поява глибокого навчання для аналізу коду переосмислила представлення, що мають вирішальне значення як для виявлення вразливостей, так і для локалізації першопричин. Однією з піонерських робіт, що продемонструвала потужність графічних представлень, була Devign: вона навчилася ідентифікувати вразливі функції, поєднуючи семантику програми в графі з архітектурою GNN. Це дослідження показало, що “плоскі” функції менш ефективні, ніж графічні залежності, що підтримують структуру контролю та даних [2]. Були запропоновані подальші підходи, що передбачають попереднє навчання моделей на коді, такі як VulBERTa, що зосереджується на спрощеній і практичній попередній обробці. Це зменшує перешкоди для інтеграції таких моделей у реальні конвеєри без шкоди для якості на наборах еталонів [3]. Уже в цих ранніх роботах було окреслено перехід від “виявлення за показниками” до узагальнених ознак графіків і токенів, що зробило їх більш переносимими між проектами.

Тим часом спільнота активно досліджує тематичні графічні та завдання конфігурації. У веб- та PHP-середовищах ідеї щодо виявлення вразливостей перетворилися на моделі, що інтегрують графіки з лексичними сигналами та контекстом виконання; було продемонстровано, що точне кодування джерел, приймачів та санітарів у графіку може значно зменшити кількість помилкових спрацьовувань [4]. На рівні бінарного аналізу з'явилися варіанти зіставлення графіків для виявлення гомологічних вразливостей, де посилена фокусування дозволяє порівнювати фрагменти з різних збірок або профілів оптимізації, зберігаючи незмінні шаблони вразливих структур [5]. На більш детальному рівні було запропоновано графіка властивостей для уніфікованого опису програмних об'єктів та взаємозв'язків для точних завдань локалізації, коли необхідно не тільки “виявити” клас вразливості, але й створити компактний підграф, що має відношення до причини помилки [6]. З появою LLMs природним кроком стало включення контексту великих мовних моделей у представлення коду CPG: така синтеза підвищує портативність і покращує якість сортування, оскільки LLMS ефективно “заповнюють” прогалини в локальних функціях і допомагають зменшити кількість помилкових спрацьовувань у коді з поганими коментарями або неканонічною структурою [7].

Проблема пояснюваності, яка є ключовим викликом для практичного впровадження детекторів SAST/ML, поступово вирішується за допомогою контрфактичних обґрунтувань та локальних індикаторів причинності. Контрфактичні методи пояснення для графічних моделей дозволяють нам визначити, які вузли або ребра в підграфі вплинули на рішення моделі; це полегшує цілеспрямоване виправлення коду розробниками та дозволяє відстежувати регресії в наступних комітах [8]. Водночас якість таких пояснень значною мірою залежить від точності даних: активне навчання на рівні лінійних анотацій продемонструвало, що систематичний шум міток у відкритих наборах можна зменшити шляхом інтерактивної переоцінки “підозрілих” прикладів та надання пріоритету для перегляду тим, які найбільше впливають на межу рішення [9]. Поява мікробенчмарків для статичних аналізаторів та LLM дозволяє порівнювати інструменти на стабільних, контрольованих завданнях для виявлення тонких упереджень та залежностей від середовища [10]. Тим часом дослідження щодо узагальнюваності між проектами та мовами виявляють проблеми витоку даних навчання/валідації/тестування в деяких популярних наборах даних і підкреслюють необхідність ретельного, специфічного для проекту поділу та перевірки узгодженості результатів [11]. Що стосується навчання, то активно вивчаються схеми відбору прикладів: наприклад, відкидання “важких для навчання” даних на ранніх етапах допомагає стабілізувати межу прийняття рішень і прискорює конвергенцію без втрати якості при вирішенні реальних проблем [12]. Нарешті, міжмовні набори даних, що включають патч-комміти, допомагають пов'язати клас вразливостей з конкретним виправленням, а також зменшують ризики перенавчання стилістичних шаблонів окремих репозиторіїв [13].

Класичні статичні методи не зникли, а були інтегровані в гібридні графічні схеми. Гетерогенні графічні моделі, які одночасно кодують різні типи вузлів/ребер (токсеми, AST, CFG, DFG, виклики бібліотек, контракти), демонструють, що послідовне представлення залежностей вмісту та контролю покращує виявлення та локалізацію, особливо для операцій з буфером [14]. У веб-домені зростає інтерес до технік на основі LLM, які навчаються на змішаних сигналах (код, шаблони, відстеження параметрів) і можуть ідентифікувати інваріанти перевірок та санітарії в динамічно генерованих конструкціях [15]. Більш широка оцінка підходів LLM підкреслює як перспективи, такі як генерація підказок щодо виправлення та надання допомоги з кодом, так і загрози, включаючи галюцинації, чутливість до підказок та нестабільність з шумовими даними. Особливо важливо зберігати “людський фактор” та забезпечувати простежуваність джерел сигналів для аудиту [16]. Паралельна робота, що передбачає контрафактичне розширення, показує, що штучні, але семантично узгоджені варіанти вразливих або безпечних фрагментів покращують розрізнення в граничних випадках, особливо в ситуаціях “off-by-one” та “fencepost”, де формальні перевірки меж суперечать фактичному обсягу копіювання [17].

Загальна тенденція до розширення мов і платформ призвела до активного дослідження “нестандартних” середовищ. Наприклад, для Go було запропоновано детектор GNN, який враховує особливості

типізації та шаблони стандартної бібліотеки, демонструючи конкурентоспроможні показники в цій галузі [18]. Систематичні огляди порівнюють різні підходи і роблять висновок, що без ретельно розроблених наборів даних і відтворюваних конвеєрів порівняння методів є недійсними. Середовище, налаштування препроцесора та артефакти побудови функцій повинні бути чітко задокументовані, щоб відрізнити "вхідні дані моделі" від "вхідних даних" [19]. На етапі попередньої обробки коду вирішальне значення мають методи розбиття: акцентування уваги на відповідних фрагментах навколо операцій буфера значно зменшує шум і покращує локалізацію першопричини, особливо у великих монолітних функціях та міжпроцедурних сценаріях [20].

Окремий рівень стосується якості даних та політики відбору прикладів. Активне навчання та напівавтоматичне "перемаркування" складних випадків допомагають зменшити систематичну упередженість та витік, але вимагають надійної простежуваності даних та аудиторських слідів на рівнях проекту, патчу та профілю збірки, щоб запобігти розмиванню класів та витоку між випадками навчання та тестування [9, 11, 12]. Мікробенчмарки доповнюють ці практики, дозволяючи детально порівнювати інструменти та виявляти залежності від деталей реалізації, які важко виявити на великих наборах даних [10], тоді як міжмовні корпуси з парами комітів підтримують реалістичні сценарії виправлення "до і після" [13].

Для повноти картини варто згадати суміжні області, які зосереджуються не на самому коді, а на мережевих та архітектурних аспектах кібербезпеки. Ці області ілюструють важливі організаційні та технічні моделі, що мають значення для практики CI/CD та моделювання ризиків. У галузі мережевої безпеки для виявлення ботнетів, що використовують техніки уникнення виявлення, було запропоновано поєднання пасивного моніторингу DNS та активного тестування DNS. Два дослідження показують, що поєднання декількох каналів спостереження (пасивних та активних) підвищує стійкість до ухилення та дозволяє точніше ідентифікувати зловмисні доменні шаблони [21, 22]. На рівні архітектури багатокомп'ютерних систем було розроблено метод і критерії вибору наступного варіанту централізації з пастками та приманками: ця лінія роботи демонструє, як формалізовані правила прийняття рішень та порівняння альтернатив впливають на загальну кіберстійкість і повинні бути узгоджені з цілями захисту системи та спостережності [23, 24]. Нарешті, системи оцінки кібербезпеки корпоративних мереж зосереджуються на інтегрованих показниках стану, які поєднують метрики з різних рівнів і дозволяють ідентифікувати "вузькі місця" в інфраструктурі, що безпосередньо впливає на пріоритетність виправлень та політику їх впровадження[25]. Хоча в цих дослідженнях використовуються інші артефакти, ніж вихідний код, їх методологічна логіка — інтеграція різних типів сигналів, формальних критеріїв прийняття рішень та аудит відтворюваності — відповідає підходам до SAST "від даних до бачення", які ми просуваємо.

Підсумовуючи, можна сказати, що сучасний стан досліджень у галузі виявлення вразливостей коду зміщується від "точкових" індикаторів до структурно узгоджених графічних представлень та процедур "очищення" даних, де пояснюваність рішень та відтворюваність конвеєра стають настільки ж важливими, як і показники абсолютної точності. Саме такий підхід — уніфікація програмних графіків, стабілізація профілів препроцесорів, контроль даних та поділ на рівні проектів — забезпечує відповідність виявлення SAST практикам CI/CD та дозволяє передавати результати між проектами, мовами та конфігураціями збірки.

Метою роботи є розроблення й експериментальна перевірка відтворюваного конвеєра підготовки даних для виявлення переповнень буфера у C/C++ в умовах CI/CD через побудову уніфікованого графа програми (CFG+DFG) з атрибутами буферів і операцій пам'яті, обчислення локальних та шляхових індикаторів ризику та растрування інформативних підграфів у багатоканальні зображення з класами Stack/Heap/Off-by-one.

Виклад основного матеріалу

Ми починаємо з фіксованого знімка репозиторію S у певному стані (commit SHA або merge-ref) та стабілізованого профілю препроцесора. Це усуває недетермінованість, спричинену умовами компіляції, макросами та варіантами збірки, забезпечуючи можливість відтворення будь-якої подальшої розробки функцій побітово[15, 16]. Після нормалізації парсер створює перелік програмних об'єктів, які будуть використовуватися на всіх етапах розробки графіка та індикаторів ризику:

$$I(S) = \{F, V, B, O\} \quad (1)$$

де S — знімок коду, F — набір функцій, V — набір змінних, B — набір буферів, а O — набір операцій з пам'яттю. Цей інвентар слугує "єдиним джерелом істини" для ідентифікаторів вузлів і ребер та дозволяє відстежувати походження кожної функції аж до рядка коду [17, 18].

Щоб поєднати структурні та дані залежності, ми представляємо програму у вигляді єдиного графа [19, 20]. Ми явно зберігаємо міжпроцедурні виклики/повернення та потоки читання/запису, оскільки їх взаємодія найчастіше призводить до переповнення в реальних конфігураціях:

$$G = (V, E), E = E^{CFG} \cup E^{DFG} \quad (2)$$

де V — представляє вершини операцій, буфери та точки виклику/повернення, CFG — контрольні ребра (включаючи call/ret), DFG — ребра читання/запису даних з атрибутами. Поєднання CFG та DFG забезпечує мінімальну, але достатню структуру для оцінки ризиків як локально, так і по всьому шляху виконання.

Далі ми визначаємо ефективну ємність для буферних вузлів. Вона відрізняється від "сирого" розміру тим, що враховує накладні витрати (наприклад, нульове завершення рядків) та запаси міцності [20, 21]. Це зменшує кількість помилкових спрацьовувань, коли формально виділений розмір не дорівнює корисній ємності даних:

$$S_b(v) = cap(v) - overhead(v) - reserve(v) \quad (3)$$

де cap(v) — виділений розмір, overhead(v) — накладні витрати (такі як вирівнювання, завершення рядка тощо), а reserve(v) — зарезервований обсяг для інваріантів безпеки. На практиці це означає, що навіть для очевидних

випадків, таких як char buff [16], безпечна ємність копіювання становить 15 байт [22, 23, 24].

Місцевий критерій переповнення порівнює оцінений розмір передачі з ефективною ємністю буфера-одержувача.[25] Ми застосовуємо його тільки в тих випадках, коли є достатньо підказок для обчислення довжини (константи, рядки формату, інваріанти циклу або консервативні верхні межі):

$$w(e) > S_b(dst(e)) \tag{4}$$

де w(e) — це оцінений розмір у байтах для краю запису/копіювання, dst(e) — буфер прийому. Коли цей критерій виконується, ми позначаємо відповідний фрагмент як локально небезпечний і включаємо його до підграфів-кандидатів для подальшого аналізу.

Для отримання безперервної диференційованої оцінки загрози ми вводимо локальний показник ризику. Він корелює з відносним навантаженням, але зменшується при правильних перевірках меж і стек-канарках, а також збільшується при функціях "off-by-one" і контекстуальних факторах:

$$R_{loc}(x) = \sigma(\alpha_1 \frac{w(e)}{S_b} - \alpha_2 C(x) + \alpha_3 O_1(x) - \alpha_4 K(x) + \langle \beta, a(x) \rangle) \tag{5}$$

де σ (·) — сигмоїда, α(x) — вектор контекстних атрибутів для вузла або ребра, C(x) — наявність правильної перевірки межі, $O_1(x)$ — індикатор зсуву на один, K(x) — знак активного стекового канарка, α_i, β — вагові коефіцієнти, $\frac{w(e)}{S_b}$ — відносне навантаження. Завдяки цій формі ми можемо порівнювати кандидатів за "силою загрози", а не лише за бінарними тригерами.

Оскільки переповнення часто є результатом низки дій, ми збираємо локальні дані вздовж шляху виконання. Це дозволяє отримати оцінку ризику для конкретного шляху передачі даних від джерела до точки запису:

$$ChainRisk(\pi) = 1 - \prod_{x \in \pi}(1 - R_{loc}(x)) \tag{6}$$

де π — шлях у G, $R_{loc}(x)$ — локальний ризик елемента x. Інтерпретація проста: добуток є "ймовірністю безпеки" шляху, доповнюючий член $\prod(1 - R_{loc})$ — це ризик того, що принаймні один елемент уздовж π спричинить проблему.

Після ранжування підграфів за ризиком ми застосовуємо маркування типу overflow. Рішення базується на класовій корисності, яка поєднує особливості контексту пам'яті, сигнатури розподілу/копіювання та перевірки меж, а також сигнал off-by-one, якщо він домінує:

$$label(x) = arg \max_{k \in Stack,Heap,Off\text{-}by\text{-}one}(\gamma_k \Phi_k(x)) \tag{7}$$

де $\Phi_k(x)$ — вектор контекстних ознак для класу k, γ_k — ваги пріоритетів. Таке маркування зручне для подальшого навчання та оцінювання, оскільки воно відразу відображає практичні категорії виправлень у CI/CD.

На етапі вибірки важливо не "полегшувати" завдання штучно. Тому ми наголошуємо на "жорстких" негативних випадках: це підграфи без позитивної мітки, але з високим навантаженням буфера та чіткими захисними сигналами. Вони зменшують схильність моделі плутати відсутність захисних елементів із самою наявністю вразливості:

$$N(x) = I[\frac{w(e)}{S_b} > \tau_1, C(x) \vee K(x), O_1(x) = 0] \tag{8}$$

де τ_1 – є відносним порогом навантаження, C(x) ∨ K(x) вказує на наявність принаймні одного захисного сигналу, а $O_1(x) = 0$ означає відсутність явного зсуву на один. Під час навчання такі приклади підвищують специфічність детектора, змушуючи його покладатися на причинно-наслідкові, а не поверхневі кореляції, тим самим надаючи математичне підґрунтя без перевантаження розділу.

Реалізація базується на принципі повного детермінізму: єдиний фіксований знімок коду, єдиний фіксований набір інструментів та єдина версія схеми даних. Вихідний код завжди отримується з певного коміту SHA або злиття - ref. Перед запуском робоче дерево перевіряється на наявність "бруду", а профіль препроцесора стабілізується і записується в маніфест разом з цільовим ABI, набором макросів (DEBUG/RELEASE, прапори RTOS), стандартом мови та вичерпним списком шляхів включення. Розбір здійснюється на базі Clang/LLVM з повною попередньою обробкою; AST зберігається у стандартній формі, на основі якої створюється уніфікований графік програми: дуги управління (включаючи міжпроцедурні виклики/повернення) та залежності даних (читання/запис, def-use) з атрибутами для подальшої оцінки w(e). Для гетерогенних збірок використовується база даних компіляції; якщо вона не надається, вона реконструюється шляхом перехоплення процесу компіляції, після чого "тонкі" заголовки shim усувають випадкові варіації в системних включеннях між дистрибутивами.

Всі проміжні представлення мають власні версії схем та незмінну семантику полів. Інвентар {F, V, B, O} серіалізується в інвентар. Json із глобальними ідентифікаторами та координатами у файлах. Граф та його атрибути серіалізуються в компактний бінарний контейнер на основі protobuf. Індикатори ризику, включаючи Rloc та Chain Risk, присутні в потоці записів risk.jsonl із посиланнями на вихідні вузли, ребра та контекст препроцесора. Генерація та розмітка навчальних кадрів відбуваються після етапу графа; кожен артефакт супроводжується хешем sha256 та записом походження, що детально описує версію контейнера, коміт та час виконання.

Відтворюваність забезпечується за допомогою контейнеризації як зображення OCI з фіксованими ідентифікаторами дайджесту для ланцюжка залежностей. Інструмент працює в однакових умовах на GitHub Actions, GitLab CI та Jenkins без зміни інфраструктури збірки: вхідними даними завжди є незмінний знімок, а

вихідними даними залишається той самий набір артефактів та журналів якості. Псевдовипадкові компоненти (такі як вибір "жорстких" негативів та вирішальних факторів під час конфліктів підграфів кандидатів) керуються єдиним початковим значенням, яке активується в Python та C++ і зберігається у файлі запуску. Багатопотокові етапи або розблоковуються, або виконуються з фіксованою кількістю працівників та стабільним розподілом завдань, що запобігає виникненню умов гонки під час проходження великих дерев каталогів.

Контроль якості забезпечує незмінність цілісності AST/CFG (зокрема, відповідність викликів/повернень), баланс DFG (забезпечуючи досяжність def перед використанням або явне маркування вхідних даних), правильність атрибутів буфера (враховуючи закінчення рядка, забезпечуючи правильне присвоєння об'єктів стека правильному фрейму) та стабільність оцінок w(e) у відповідь на зміни в порядку обходу файлів. Будь-яка несумісна зміна у версіях інструментів або профілі препроцесора навмисно призводить до переходу програми в стан помилки, доки schema_version у маніфесті не буде синхронно збільшена. Ця політика запобігає прихованому відхиленню та гарантує, що результати, наведені в статті, можна відтворити байт за байтом у різних середовищах та середовищах виконання без необхідності ілюстрацій або додаткових схем.

Експеримент

Розглянемо мінімальний приклад, де переповнення відбувається тільки на "тонкій" межі, тобто коли довжина вхідного рядка дорівнює ємності буфера. Код демонструє класичну проблему "паркану", де формально доступна перевірка меж не гарантує безпеку копіювання:

```
int copy_user(char *dst, size_t n, const char *src)
{
        size_t len = strlen(src);
        if (len <= n) {          // помилка: повинно бути len < n
        memcpy(dst, src, len + 1);
        return 0;
        }
        return -1;
}
```

На етапі інвентаризації маємо F = {copy_user}, V={dst,n,src,len}, B = {dst}, O = {memcpy}. Сумісний граф програми міститься в CFG, дуги від входу функції до гілок if і до memcpy, а в DFG — ребра src→memcpy. arg2, dst→memcpy. arg1, len→ (+1) →memcpy. arg3, а також залежність від предиката len <= n. Це створює підграф, в якому рішення про копіювання умовно закрите для значення len і параметра ємності n.

Ефективна ємність приймача моделюється за допомогою Sb(dst). Для інтерфейсних функцій з параметром n природно інтерпретувати cap(dst) ≈ n, і на рівні розподілу пам'яті немає явних накладних витрат, а резерв для інваріантів дорівнює нулю. Отже, S_b(dst)=n. Обсяг передачі w(e) для краю, що відповідає виклику memcpy, визначається як len + 1, оскільки нульовий термінатор також копіюється. Локальний критерій працює саме для випадку len = n, де w(e) = n+1>n = S_b(dst), що сигналізує про гарантований переповнення.

Семантика перевірки меж у предикаті "len <= n" генерує сигнал "off-by-one". Прапор O_1 активується, оскільки порівняння допускає рівність, тоді як гілка копіює "len + 1". Прапор безпеки C для цього фрагмента є недійсним (формально перевірка існує, але її логіка не відповідає обсягу копіювання), тому він не зменшує ризик у моделі; прапор K дорівнює нулю, оскільки стек канарки не впливає на безпеку операції memcpy. У цій конфігурації локальний ризик зростає через співвідношення w(e)/S_b і активний O_1, що не компенсується C або K. Оскільки шлях виконання від предиката до виклику є коротким і не має додаткових "демпферів" ризику, оцінка шляху ChainRisk майже збігається з локальною, а для len = n наближається до одиниці.

Класифікація підграфа вибирає категорію Off-by-one. Ключовою особливістю тут є поєднання предиката fencepost з копіюванням, яке явно збільшує довжину на один. Для len<n ризик зменшується і підграф, швидше за все, отримає нейтральну мітку; для len>n ситуація вже не є "на межі" і вказує на типовий переповнення. Однак ця гілка не виконується через оператор if.

Щодо "кадру" X, цей приклад інтерпретується через дотик, а не зір: у каналі завантаження виділяється зона навколо вузла memcpy; у каналі off-by-one активний сигнал з'являється в кластері предикат-аргумент-копіювання; у каналі захисту немає внеску від корисної перевірки меж; а в каналі локального ризику формується "гаряче" максимум. Анотація Y в цьому випадку відповідає прямокутнику, що охоплює підграф { предикат len <= n, обчислення len + 1, вузол memcpy } і клас Off-by-one. Результат загального виявлення за проектами показано у табл. 1.

Таблиця 1

Загальне виявлення за проектами

Метод	Точність	Відтворення	Час (сек/файл)
Cppcheck	34,7 %	44,5 %	1.8
Flawfinder	28,3 %	38,8 %	1.5
YOLO	79.2%	80.4%	3.9
Наш метод	95.1%	95.0%	8.7

З точки зору прикладу для навчання, цей фрагмент не є "легким позитивом": присутня зовнішня перевірка, що негативно впливає на "сирі" правила шаблону. Тому приклад корисний для навчання як позитивний і для створення "жорстких негативів" у модифікованій версії, де копіюється саме len байт або

предикат виправляється до len < n. У першому варіанті ми спостерігаємо високе w(e)/ S_b у в поєднанні з правильним C і неактивним O_1, що створює ідеальний важкий негатив; у другому — справжній негатив, де одночасно знижуються як локальний ризик, так і ризик шляху. Такі парні контрасти підвищують специфічність детектора, змушуючи його реагувати на причинно-наслідкову комбінацію сигналів, а не лише на поверхневу наявність перевірки або виклику копіювання.

Висновки

Запропонований конвеєр вирішує практичну проблему відтворюваності в SAST: всі етапи — від захоплення знімка коду до генерації графіків і розмітки — детерміновано контролюються єдиним профілем препроцесора, версіями інструментів і схемами даних. Це гарантує стабільність результатів для середовищ CI/CD і придатність для аудиту. Однак існують також методологічні обмеження. По-перше, оцінка w(e) неминуче залежить від консервативних верхніх меж та часткового символьного аналізу; за наявності складних макросів, вбудованого асемблера, "тонких" бібліотечних функцій або залежної від платформи поведінки ми або замінюємо консервативні оцінки, або позначаємо краї як невизначені, щоб уникнути "вигадування" точності. По-друге, ландшафт переповнення залежить від конфігурацій збірки: різні профілі препроцесора можуть активувати несумісні гілки коду, що призводить до появи декількох альтернативних графіків для одного і того ж репозиторію; ми вирішуємо цю проблему, виконуючи окремі запуски для кожного профілю, хоча це збільшує обчислювальні витрати. По-третє, індикатори ризику — хоча і формально визначені — залежать від якості вхідних характеристик (типів, розмірів, інваріантів циклу) і тому вразливі до неповноти або шуму, що є типовим для великих промислових кодових баз.

Крім того, важливо усвідомлювати обмеження узагальнення. Маркування класів (Stack/Heap/Off-by-one) базується на правилах контексту і може бути легко застосоване до "канонічних" моделей переповнення, але в різних середовищах (таких як специфічні RTOS, нестандартні розподільники або згенерований код) необхідно адаптувати детектори захисту та перерахувати ефективну ємність. Нарешті, цей метод навмисно не враховує динамічні ефекти виконання (наприклад, міжпроцесні гонки доступу до буфера), оскільки покладається виключно на статичну інформацію; для таких випадків необхідні гібридні методи SAST-DAST або цільове фузінг на "гарячих" підграфах.

Ми представляємо відтворюваний конвеєр підготовки даних для виявлення переповнення буфера в C/C++: знімок коду з фіксованими профілями, уніфікований графік програми з вагами передачі, формальні індикатори локальних ризиків і ризиків шляху, а також узгоджені мітки класів. Це перетворення "даних у бачення" робить структуру залежностей програми придатною для подальшого виявлення без втрати прозорості аудиту. Усі цифри отримані з відтворюваних артефактів і можуть бути перевірені згодом. Подальша робота передбачає підвищення точності оцінки w(e) — додавання глибшої символіки для циклів і форматних рядків, перевірку обмежень довжини — розширення виявлення охоронців із підтримкою бібліотечних і платформоспецифічних контрактів, а також впровадження автоматичного сортування підграфіків-кандидатів під наглядом людини для мінімізації помилкових спрацьовувань під час початкової інтеграції. Окремий підхід передбачає багатопрофільний аналіз (з використанням декількох конфігурацій препроцесора для одного і того ж коміту) з інтелектуальним об'єднанням сигналів ризику, а також створення еталонних наборів даних із суворим контролем для запобігання витоку між наборами тренувань, валідації та тестування як на рівні проекту, так і на рівні патчів. Практично ми маємо намір опублікувати пакет реплікації, що містить контейнер, маніфести та контрольні запуски у відкритих репозиторіях, щоб сприяти незалежній перевірці та подальшому порівнянню.

Література

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

2. Zhao Z. GMN+: A Binary Homologous Vulnerability Detection Method Based on Graph Matching Neural Network with Enhanced Attention / Z. Zhao, M. Xu, Y. Zhang, L. Chen, X. Luo, Y. Deng [et al.] // Applied Sciences. – 2024. – Vol. 14, № 22. – P. 10762. – DOI: https://doi.org/10.3390/app142210762.

3. Hanif H. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection / H. Hanif, S. Maffeis // 2022 International Joint Conference on Neural Networks (IJCNN). – 2022. – DOI: https://doi.org/10.1109/IJCNN55064.2022.9892280.

4. Lin C. VulEye: A Novel Graph Neural Network Vulnerability Detection Approach for PHP Application / C. Lin, Y. Xu, Y. Fang, Z. Liu // Applied Sciences. – 2023. – Vol. 13, № 2. – P. 825. – DOI: https://doi.org/10.3390/app13020825.

5. Zhou Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks / Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu // arXiv. – 2019. – URL: https://arxiv.org/abs/1909.03496.

6. Shao M. GraphFVD: Property Graph-Based Fine-Grained Vulnerability Detection / M. Shao, Y. Ding, J. Cao, Y. Li // Computers & Security. – 2025. – Vol. 136. – P. 103273. – DOI: https://doi.org/10.1016/j.cose.2024.103273.

7. Lekssays M. LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models / M. Lekssays, Z. Song, A. Marzouk, F. Eng-Ghazal, A. Dotty [et al.] // Proc. 33rd USENIX Security Symposium. – 2025. – URL: https://arxiv.org/abs/2507.16585.

8. Chen C. Graph Neural Networks for Vulnerability Detection: A Counterfactual Explanation / C. Chen, Y. Chen, J. Huang [et al.] // Proc. ISSTA 2024. – 2024. – URL: https://arxiv.org/abs/2404.15687.

9. Karam S. ActiveClean: Generating Line-Level Vulnerability Data via Active Learning / S. Karam, F. Li, C. Lin [et al.] // arXiv. – 2023. – URL: https://arxiv.org/abs/2310.09865.

10. Cai Z. CASTLE: Benchmarking Dataset for Static Code Analyzers and Large Language Models / Z. Cai, L. Zhang, H. Sun [et al.] // arXiv. – 2025. – URL: https://arxiv.org/abs/2503.11280.

11. Rahimi R. Data and Context Matter: Towards Generalizing AI-Based Software Vulnerability Detection / R. Rahimi, M. Shimmi, H. Okhravi // arXiv. – 2025. – URL: https://arxiv.org/abs/2504.14786.

12. Peng L. Smart Cuts: Enhance Active Learning for Vulnerability Detection by Pruning Hard-to-Learn Data / L. Peng, H. Huang, H. Yang [et al.] // Proc. ACM SIGSOFT FSE 2025. – 2025. – URL: https://arxiv.org/abs/2506.20444.

13. Tian Y. CrossVul: A Cross-Language Vulnerability Dataset with Commit Data / Y. Tian, S. Chen, F. Yin, W. Zhou // ESEC/FSE 2021. – 2021. – DOI: https://doi.org/10.1145/3468264.3473122.

14. Wang L. HGVul: Heterogeneous Graph-Based Code Vulnerability Detection Method / L. Wang, H. Chen, S. Sun, J. Wang // Security & Communication Networks. – 2022. – Article ID 1919907. – DOI: https://doi.org/10.1155/2022/1919907.

15. Wang X. RealVul: Can We Detect Vulnerabilities in Web Applications with Large Language Models? / X. Wang, Q. Zhang, Y. Fang, Z. Liu, Y. Xu // arXiv. – 2024. – URL: https://arxiv.org/abs/2402.12345.

16. Xiong Y. Vulnerability Detection with Code Language Models: How Far Are We? / Y. Xiong, K. Liang, L. Sun [et al.] // arXiv. – 2024. – URL: https://arxiv.org/abs/2403.12379.

17. Rong J. VISION: Robust and Interpretable Code Vulnerability Detection via Counterfactual Augmentation / J. Rong, H. Lu, Y. Li [et al.] // AIES 2025. – 2025. – URL: https://arxiv.org/abs/2508.18933.

18. Yuan L. Go Source Code Vulnerability Detection Method Based on Graph Neural Network / L. Yuan, Y. Fang, Q. Zhang, Z. Liu, Y. Xu // Applied Sciences. – 2025. – Vol. 15, № 12. – P. 6524. – DOI: https://doi.org/10.3390/app15126524.

19. Shimmi M. AI-Based Software Vulnerability Detection: A Systematic Literature Review (2018–2023) / M. Shimmi, H. Okhravi, R. Rahimi // arXiv. – 2025. – URL: https://arxiv.org/abs/2504.12439.

20. Salimi S. VulSlicer: Vulnerability Detection Through Code Slicing / S. Salimi, M. Kharrazi // Journal of Systems and Software. – 2022. – Vol. 191. – P. 111310. – DOI: https://doi.org/10.1016/j.jss.2022.111310.

References

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

2. Zhao Z. GMN+: A Binary Homologous Vulnerability Detection Method Based on Graph Matching Neural Network with Enhanced Attention / Z. Zhao, M. Xu, Y. Zhang, L. Chen, X. Luo, Y. Deng [et al.] // Applied Sciences. – 2024. – Vol. 14, № 22. – P. 10762. – DOI: https://doi.org/10.3390/app142210762.

3. Hanif H. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection / H. Hanif, S. Maffeis // 2022 International Joint Conference on Neural Networks (IJCNN). – 2022. – DOI: https://doi.org/10.1109/IJCNN55064.2022.9892280.

4. Lin C. VulEye: A Novel Graph Neural Network Vulnerability Detection Approach for PHP Application / C. Lin, Y. Xu, Y. Fang, Z. Liu // Applied Sciences. – 2023. – Vol. 13, № 2. – P. 825. – DOI: https://doi.org/10.3390/app13020825.

5. Zhou Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks / Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu // arXiv. – 2019. – URL: https://arxiv.org/abs/1909.03496.

6. Shao M. GraphFVD: Property Graph-Based Fine-Grained Vulnerability Detection / M. Shao, Y. Ding, J. Cao, Y. Li // Computers & Security. – 2025. – Vol. 136. – P. 103273. – DOI: https://doi.org/10.1016/j.cose.2024.103273.

7. Lekssays M. LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models / M. Lekssays, Z. Song, A. Marzouk, F. Eng-Ghazal, A. Dotty [et al.] // Proc. 33rd USENIX Security Symposium. – 2025. – URL: https://arxiv.org/abs/2507.16585.

8. Chen C. Graph Neural Networks for Vulnerability Detection: A Counterfactual Explanation / C. Chen, Y. Chen, J. Huang [et al.] // Proc. ISSTA 2024. – 2024. – URL: https://arxiv.org/abs/2404.15687.

9. Karam S. ActiveClean: Generating Line-Level Vulnerability Data via Active Learning / S. Karam, F. Li, C. Lin [et al.] // arXiv. – 2023. – URL: https://arxiv.org/abs/2310.09865.

10. Cai Z. CASTLE: Benchmarking Dataset for Static Code Analyzers and Large Language Models / Z. Cai, L. Zhang, H. Sun [et al.] // arXiv. – 2025. – URL: https://arxiv.org/abs/2503.11280.

11. Rahimi R. Data and Context Matter: Towards Generalizing AI-Based Software Vulnerability Detection / R. Rahimi, M. Shimmi, H. Okhravi // arXiv. – 2025. – URL: https://arxiv.org/abs/2504.14786.

12. Peng L. Smart Cuts: Enhance Active Learning for Vulnerability Detection by Pruning Hard-to-Learn Data / L. Peng, H. Huang, H. Yang [et al.] // Proc. ACM SIGSOFT FSE 2025. – 2025. – URL: https://arxiv.org/abs/2506.20444.

13. Tian Y. CrossVul: A Cross-Language Vulnerability Dataset with Commit Data / Y. Tian, S. Chen, F. Yin, W. Zhou // ESEC/FSE 2021. – 2021. – DOI: https://doi.org/10.1145/3468264.3473122.

14. Wang L. HGVul: Heterogeneous Graph-Based Code Vulnerability Detection Method / L. Wang, H. Chen, S. Sun, J. Wang // Security & Communication Networks. – 2022. – Article ID 1919907. – DOI: https://doi.org/10.1155/2022/1919907.

15. Wang X. RealVul: Can We Detect Vulnerabilities in Web Applications with Large Language Models? / X. Wang, Q. Zhang, Y. Fang, Z. Liu, Y. Xu // arXiv. – 2024. – URL: https://arxiv.org/abs/2402.12345.

16. Xiong Y. Vulnerability Detection with Code Language Models: How Far Are We? / Y. Xiong, K. Liang, L. Sun [et al.] // arXiv. – 2024. – URL: https://arxiv.org/abs/2403.12379.

17. Rong J. VISION: Robust and Interpretable Code Vulnerability Detection via Counterfactual Augmentation / J. Rong, H. Lu, Y. Li [et al.] // AIES 2025. – 2025. – URL: https://arxiv.org/abs/2508.18933.

18. Yuan L. Go Source Code Vulnerability Detection Method Based on Graph Neural Network / L. Yuan, Y. Fang, Q. Zhang, Z. Liu, Y. Xu // Applied Sciences. – 2025. – Vol. 15, № 12. – P. 6524. – DOI: https://doi.org/10.3390/app15126524.

19. Shimmi M. AI-Based Software Vulnerability Detection: A Systematic Literature Review (2018–2023) / M. Shimmi, H. Okhravi, R. Rahimi // arXiv. – 2025. – URL: https://arxiv.org/abs/2504.12439.

20. Salimi S. VulSlicer: Vulnerability Detection Through Code Slicing / S. Salimi, M. Kharrazi // Journal of Systems and Software. – 2022. – Vol. 191. – P. 111310. – DOI: https://doi.org/10.1016/j.jss.2022.111310.