УДК 004.415.2
DOI https://doi.org/10.32782/2663-5941/2025.6.2/43

Сєргєєв Є.В.
Хмельницький національний університет

Савенко О.С.
Хмельницький національний університет

ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ ПЕРЕПОВНЕННЯ БУФЕРА В СИСТЕМНОМУ ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННІ НА ОСНОВІ ГРАФА ТА МОДЕЛІ ТРАНСФОРМАТОРА

Виявлення вразливостей, пов'язаних із пам'яттю, лишається ключовим викликом для системного та вбудованого програмного забезпечення. Помилки переповнення буфера часто стають причиною інцидентів безпеки в середовищах із обмеженими ресурсами, де слабка ізоляція та специфіка апаратних платформ ускладнюють гарантування коректної роботи. У роботі запропоновано підхід, що поєднує формальні умови ризику з графовими поданнями коду та нейромережею детектора, орієнтованою на виявлення реальних патернів у системних проєктах. Вихідний код перетворюється на узгоджені структури абстрактного синтаксичного дерева, графа керування та графа потоків даних, де вузли та ребра анотовано інформацією про буфери, межі, гварди й шляхи поширення значень. На цій основі формуються інформативні підграфи та їхні багатоканальні візуалізації, що кодують взаємодію керування й даних, а також відповідність типізованим шаблонам загальновідомих баз знань про вразливості. Нейромережевий детектор працює з цими поданнями як із навчальними прикладами, здійснює локалізацію підозрілих фрагментів і повертає пояснювані спрацювання у вигляді класу, ділянки коду та короткої причини, узгодженої з формальними індикаторами ризику. Підхід підтримує інкрементальне оновлення артефактів аналізу, працює в конвеєрах безперервної інтеграції та постачання, зберігає профілі препроцесора та версії інструментів, що забезпечує відтворюваність і зручність для аудиту. Наукова новизна полягає у поєднанні формальної оцінки ефективної ємності буфера, ваг потоків даних і графових візуалізацій як навчального простору для детектора, що дає змогу зменшити хибні спрацювання та підвищити переносимість між кодовими базами. Обмеження стосуються вартості побудови графів на великих проєктах і чутливості до конфігурацій збірки, проте вони пом'якшуються профілюванням, кешуванням і політиками запуску. Отримані результати свідчать про практичну придатність запропонованого рішення для системного програмного забезпечення та інтеграції в інженерні процеси.

Ключові слова: *переповнення буфера, системне програмне забезпечення, вразливості пам'яті, графові нейронні мережі, YOLO, трансформер.*

Постановка проблеми. Проблеми безпеки пам'яті в системному та вбудованому програмному забезпеченні залишаються критичними [1]: дефекти переповнення буфера все ще є поширеною причиною інцидентів в RTOS [2], драйверах та програмному забезпеченні мікроконтролерів [3], де ресурси обмежені, а часові обмеження жорсткі. У таких умовах традиційні інструменти статичного забезпечення якості (SAST) часто або генерують надмірну кількість помилкових спрацьовувань, або пропускають нетипові шаблони, особливо у великих кодових базах з великою кількістю залежностей [4]. Паралельно активно розвиваються підходи ML/AI: від послідовних і мовних моделей до графічних нейронних мереж, які краще фіксують структурні залежності в програмному коді [5], а також напрямок роботи, зосереджений на пояснюваності та надійності моделей [6]. Однак без чіткої формальної основи та узгоджених шаблонів ризиків важко створити масштабований і водночас інтерпретований детектор. У програмному забезпеченні прикладних систем (RTOS, драйвери, прошивка MCU) переповнення буфера виявляє складну поведінку: обмеження стека, невідповідності контракту API, відсутність ізоляції апаратного забезпечення та обмежений «проміжок» часу для перевірок сприяють тому, що навіть одна помилка в індексації або форматі

виводу може спричинити збій системи або приховане підвищення привілеїв. Стандартні засоби захисту (ASLR, DEP, stack canaries) зменшують площу атаки, але не усувають основну проблему – семантичні помилки в обробці пам'яті. Це підкріплює використання поєднання формальних умов ризику з машинним навчанням: перше забезпечує суворі обмеження правильності, а друге дозволяє масштабувати локалізацію та пріоритезацію підозрілих ділянок коду.

У цій статті поєднуються формальні умови ризику та шаблони з компактним детектором YOLO. Ми представляємо код у вигляді графічних зображень, що відповідають бібліотеці шаблонів (CVE/NVD/OWASP), а рівняння визначають еталонні критерії ризику для локалізації та пояснюваності. Кінцеве рішення інтегровано в CI/CD, що дозволяє швидко перевіряти зміни у великих репозиторіях.

У цій статті під «буфером» ми розуміємо статичний масив (стек), динамічно виділений блок (купу) або поліморфну структуру, що інкапсулює пам'ять, за умови наявності явного оператора запису. Під «перевіркою меж» ми розуміємо або перевірку предиката (умова/асерт з правильними межами), або безпечну версію API з правильно вказаним розміром. При розрахунку ризику ми припускаємо консервативну недооцінку ємності (ефективної ємності), що гарантує відсутність помилкових негативних результатів на етапі попереднього відбору. Стаття в журналі з розбивкою на сторінки [7].

Метою цієї роботи є розробка універсального методу виявлення вразливостей переповнення буфера, що охоплює помилки стека, купи та off-by-one.

Основні аспекти статті:

1. Порівняльний огляд сучасних підходів ML/DL до виявлення BOF в контексті системного/вбудованого програмного забезпечення;

2. Формальна основа ризику у вигляді чотирьох умов з розділу 3 та бібліотека шаблонів, зібрана з CVE/NVD/OWASP;

3. Трикласний детектор YOLO (Stack/Heap/Off-by-One) на графічному коді з поясненням результату (клас, оцінка, діапазон коду, відповідний шаблон, пояснення);

4. Протокол оцінки та інтеграція в CI/CD, де показано підвищення якості порівняно з базовими інструментами SAST та поточними базовими показниками ML.

Аналіз останніх досліджень і публікацій. Методи виявлення переповнення буфера зазвичай поділяють на дві великі групи: класичні статичні/динамічні аналізатори та підходи на основі машинного навчання. Перші базуються на правилах, евристиці та аналізі потоку даних/контролю: від простих сигнатур небезпечних викликів (strcpy, sprintf, неконтрольовані копії) до складних міжпроцедурних перевірок із використанням дерев розбору, графіків контролю потоку (CFG) та аналізу забруднення. Символьне виконання та абстрактна інтерпретація дають змогу довести доступність помилок, але страждають від «експлозії шляхів» і вимагають складних інваріантних специфікацій. Динамічні підходи (фузінг, санітайзери) забезпечують високу точність відтворюваних траєкторій виконання, але мають значні витрати ресурсів і покриття. Такі інструменти, як AddressSanitizer/UBSan, виявляють дефекти під час виконання, але не гарантують покриття «холодних» гілок коду. Крім того, фузінг важко налаштувати для сценаріїв, що залежать від стану або багатопотокових[8]. Додаткові ускладнення вносить сучасні механізми захисту (ASLR, DEP), які змінюють прояви експлойтів (ланцюжки ROP, розпилення купи), не усуваючи першопричину, а саме порушення граничних умов і неправильну роботу з довжинами.

Підходи машинного навчання спрямовані на навчання моделей розпізнавати вразливі шаблони в різних представленнях коду: текстові маркери, проміжне представлення/байт-код, абстрактні та конкретні графіки (AST/CFG/DFG/PDG). Ранні рішення використовували n-грамні представлення з класичними класифікаторами (SVM, Random Forest), які добре працювали для повторюваних шаблонів, але часто «ламалися» на неочевидних варіаціях стилю коду та нових API. Подальший прогрес пов'язаний з векторизацією коду (вбудовуванням коду) та глибоким навчанням: рекурентні мережі та трансформатори ефективно моделюють послідовності токенів та інструкцій, а графічні нейронні мережі (GNN) додають структурний контекст – залежність даних та контролю, взаємодію між буферами та довжинами. Саме графічні підходи продемонстрували помітне підвищення якості на великих репозиторіях, але зіткнулися з іншими проблемами: дисбаланс класів (рідкісні off-by-ones), «хибні» кореляції (модель прив'язана до неінформативних ознак, таких як імена змінних), чутливість до зміщення домену (переносимість між проектами та стилями коду) та відсутність інтерпретованості результатів для розробників[9].

У відповідь на це з'явився третій, гібридний напрямок роботи: поєднання формальних/

основаних на правилах перевірок із машинним навчанням (ML). Ідея полягає в тому, щоб використовувати «жорсткі» умови безпеки (перевірки меж, контракти параметрів, інваріанти довжини) як фільтри/підказки на етапі побудови функцій, тоді як модель навчається на структурованих представленнях коду, надаючи пріоритет підграфам із підвищеним ризиком. Така синтеза має дві ключові переваги: вона зменшує кількість помилкових спрацьовувань у великих репозиторіях завдяки явній логіці «ємності» та повертає інтерпретованість – кожне виявлення може бути пов'язане з конкретним підграфом, перевіркою меж або порушеною умовою, що полегшує сортування та виправлення. Водночас залишається кілька відкритих питань: як стабілізувати узагальнюваність на «атипових» стилях коду та нових бібліотеках; як правильно агрегувати локальні сигнали (потоки байтів, перевірки охорони) до рівня модуля/проекту; як інтегрувати оцінку ризиків машинного навчання в процес CI/CD без надмірних накладних витрат[10]. Саме на це спрямований наш підхід: ми поєднуємо формальні умови ризику з критерієм графа та спеціалізованим трикласовим детектором, щоб одночасно підтримувати строгість перевірок і досягати масштабованості на промислових кодових базах.

Статичні аналізатори, такі як Cppcheck і Flawfinder, історично були першою лінією захисту від переповнення буфера. Вони виявляють підозрілі виклики («небезпечні» API-інтерфейси рядків, неконтрольовані копії), відсутність перевірок меж і типові шаблони з баз даних CWE[11]. Сучасні реалізації виходять за межі простого пошуку шаблонів. Вони використовують абстрактні синтаксичні дерева (AST), міжпроцедурний аналіз потоку даних (taint) та аналіз потоку управління (CFG), спрощене символьне виконання та абстрактну інтерпретацію для створення доказів досяжності помилок. Однак контекстна чутливість залишається ключовою проблемою. Макроси та умовна компіляція змінюють фактичний код для різних конфігурацій; шаблони C++, вбудований асемблер та оптимізації компілятора приховують зв'язки між джерелами даних та приймачами; міжмодульні залежності вимагають «загальнопрограмного» аналізу, який не масштабується належним чином. У великих репозиторіях це призводить до «шуму» від численних помилкових спрацьовувань та пропущених нетривіальних випадків – наприклад, коли безпечні обгортки над strncpy комбінуються так, що порушення відбувається лише в певному порядку викликів. Цей компроміс між надійністю та повнотою є неминучим: агресивне зменшення помилкових спрацьовувань часто призводить до «засліплення» на складних ланцюжках даних[12].

Такі механізми запобігання, як ASLR, DEP/NX, stack canaries, FORTIFY_SOURCE та часткові форми CFI, значно ускладнили використання класичних переповнень, але не усунули їхні основні причини – порушення граничних умов та неправильне маніпулювання довжиною. Техніки атак еволюціонували: програмування, орієнтоване на повернення/перехід (ROP/JOP), heap-spraying та "heap feng shui" дозволяють поєднувати невеликі витоки інформації та незначні помилки в перевірці меж для обходу захисту[13]. Ці сценарії є особливо критичними для системного та вбудованого програмного забезпечення: тонкі стеки, відсутність повної MMU, жорсткі терміни в режимі реального часу та вимоги до розміру бінарних файлів залишають мало місця для потужного захисту та діагностики.

На практиці інженерні команди поєднують різні підходи: швидкий SAST запускається при кожному коміті, динамічні інструменти використовуються під час нічних збірок і перед релізами, а для «гарячих» модулів додається ручний перегляд коду. Однак навіть таке поєднання дає нерівномірний результат: чим більший репозиторій, тим вищими стають витрати на сортування попереджень і тим складніше гарантувати стабільне динамічне покриття[14]. Це створює нішу для гібридних методів, де формальні умови безпеки та структурні критерії слугують «якорями», а машинне навчання допомагає масштабувати локалізацію вразливих підграфів та зменшувати «шум» на рівні великих кодових баз. Саме це поєднання буде розглянуто нижче.

Сьогодні штучний інтелект демонструє прогрес від простих послідовних моделей до графічних та мовних архітектур. Ранні підходи застосовували рекурентні мережі до послідовностей інструкцій і досягли значних успіхів у виявленні переповнення стека [15]. Паралельна лінія досліджувала сигнатури мережевого трафіку та використовувала класичні ансамблі (випадкові ліси) для діагностики віддалених атак BOF, демонструючи високу точність виявлення інцидентів [16]. Загальний огляд (2025) систематизував методи ML/DL для BOF та висвітлив відкриті проблеми, такі як дисбаланс даних, неякісні анотації, помилкові кореляції та погана портативність між проектами [17].

Підходи на основі графів стали домінуючими завдяки їхній здатності враховувати як

синтаксис, так і семантику. Модель MSVAGraph обробляє різні типи бінарних форматів об'єктів (BOF), поєднуючи топології програмних графів з функціями виклику, що призводить до постійного підвищення точності [18]. SySeVR представляє фреймворк, який об'єднує синтаксичні та семантичні представлення вразливих фрагментів, покращуючи можливості виявлення. MVD використовує чутливі до потоку даних графічні нейронні мережі (GNN) для поліпшення локалізації вразливих вузлів навіть у складних сценаріях передачі значень. Крім того, двонаправлена GNN в BGNN4VD враховує двонаправлені залежності для кращого вибору контексту, що є особливо корисним у довгих шляхах передачі даних [19].

В даний час розробляються гібридні архітектури, що інтегрують послідовні мовні моделі з механізмами уваги та специфічними для домену функціями. Дослідження, що включають BiLSTM з увагою, особливо зі спеціалізованими компонентами KAN, показали конкурентоспроможні результати при застосуванні до «лінійних» представлень коду. Крім того, поєднання трансформаторів з агрегаторами графіків, такими як GraphSAGE, підкреслює переваги включення глобального контексту, особливо в додатках, пов'язаних з мовним кодом на рівні системи та мовою програмування Go [20].

Серед робіт останніх років також варто згадати «Виявлення аномалій у мережі на основі машинного навчання: проектування, впровадження та оцінка» (2024), де автори розробили комплексний підхід до пошуку аномалій у мережевому трафіку з використанням функцій сеансу та класифікаторів SVM і XGBoost. Стаття «Вплив машинного навчання на системи виявлення вторгнень для захисту критичної інфраструктури (2021) систематизує досвід використання ML для захисту промислових систем та окреслює напрямки вдосконалення IDS[21]. У статті «Підходи глибокого навчання для систем виявлення вторгнень: огляд» (2020) надається огляд моделей CNN, RNN та автокодерів, які демонструють конкурентні результати на наборах даних KDDCup та NSL-KDD. У статті «Підхід на основі ансамблю глибокого навчання для виявлення кібератак у хмарних обчисленнях» (2019) пропонується ансамбль CNN+LSTM для виявлення атак у хмарних середовищах, який демонструє високу точність завдяки поєднанню просторових і часових характеристик. У статті "Виявлення шкідливого програмного забезпечення за допомогою конволюційних нейронних мереж і трансферного навчання" (2020) демонструється, як використання попередньо навчених архітектур CNN покращує класифікацію шкідливого програмного забезпечення. Гібридна система штучного інтелекту для виявлення DDoS-атак (2022) представляє поєднання нейронних мереж і стохастичних методів для раннього виявлення DDoS-потоків. Нарешті, огляд технологій машинного навчання та глибокого навчання для кібербезпеки пропонує широку систематизацію застосувань ML/DL для виявлення вторгнень, шкідливого програмного забезпечення та інших кіберзагроз[22].

Незважаючи на значний прогрес, більшість підходів зосереджуються або на одному підтипі переповнення (найчастіше переповнення стека), або на конкретній мові/парадигмі і рідко враховують обмеження системного/вбудованого програмного забезпечення (RTOS, драйвери, середовище MCU)[23]. Це проявляється в нестабільності метрик в «нетипових» випадках, труднощах з переносимістю між проектами та відсутності чітких пояснень для розробників. Ось чому далі в цій статті ми покладаємося на формальні умови ризику та шаблони з CVE/NVD/OWASP як інтерпретовану основу[24]. Ми застосовуємо трикласний детектор YOLO (Stack/Heap/Off-by-One) до графічних відображень коду, щоб поєднати масштабованість, узагальнюваність та пояснюваність у контексті системного програмного забезпечення[25].

Нещодавно було описано застосування штучного інтелекту до вразливостей переповнення буфера в контексті інтегрованих систем, що послужило основою для вдосконалення запропонованого методу.

Постановка завдання. Метою статті є розробка та експериментальна перевірка методу виявлення вразливостей переповнення буфера в системному програмному забезпеченні на основі графових представлень коду та моделі трансформатора. Метод поєднує формальні умови ризику, шаблони CVE/NVD/OWASP та компактну нейронну архітектуру для виявлення типів Stack, Heap та Off-by-One переповнень.

Виклад основного матеріалу. Щоб отримати універсальний метод виявлення переповнення буфера, необхідно формалізувати поведінку запису в пам'ять на двох взаємодоповнюючих рівнях: локальному, де фіксуються необхідні нерівності для окремої операції або індексації, та глобальному, де враховується склад декількох індивідуально безпечних дій у підграфі програми. У нашій формулюванні локальні умови визна-

чають основні ризикові ситуації: перевищення ємності буфера запису, часовий аспект для динамічної пам'яті (купа) та помилка виходу за межі (off-by-one) для масивів. Глобальний критерій (4) узагальнює ці випадки для підграфів, де сукупний ефект декількох копій або форматних виводів призводить до перевищення ефективної ємності, навіть якщо окремі кроки здаються «законними».

Такий розклад має практичну та методологічну цінність. По-перше, необхідні локальні умови легко перевірити, і вони добре узгоджуються з відомими класами помилок у системному/вбудованому програмному забезпеченні (C/C++ на RTOS, драйвери, MCU), де типовими є шаблони sprintf/strcat/memcpy та помилки межового циклу. По-друге, глобальний графічний критерій додає контекст: ланцюжки операцій на одному буфері, поєднання декількох шляхів передачі даних в одному трасі або накопичення довжини через допоміжні буфери. Саме тут виникають складні сценарії: поєднання «безпечних» API, недооцінка нульового термінатора, зміна розміру блоку після перерозподілу та об'єднання гілок з різними гарантіями перевірки. На закінчення, (4) пояснює реалістичні інциденти, коли переповнення відбувається не миттєво, а в результаті комбінації дій.

Формальні моделі також відіграють подвійну роль у нашій системі: (i) вони слугують еталонними критеріями безпеки (які можна інтерпретувати в оглядах та політиках CI/CD), (ii) вони стають визначеннями для машинного навчання – через ефективну буферну ємність, вагу потоків даних та узгодженість прогнозів детектора з оцінками ризиків. Практичним наслідком є пояснюваність: кожен тригер не тільки локалізований у проміжку коду, але й має посилання на конкретну умову (1)–(4) та відповідний шаблон (Stack/Heap/Off-by-One), що значно спрощує сортування та визначення пріоритетності виправлень.

Случаї стекування виникають, коли обсяг запису в локальний буфер перевищує його ємність відповідно до формули (1). Типовими наслідками є перезапис сусідніх змінних і даних обслуговування кадру виклику, таких як адреси повернення. У практичних атаках це відкриває можливість зміни потоку виконання; сучасні техніки (зокрема ROP) поєднують невеликі переліки та вразливі фрагменти коду, але в основі цього завжди лежить порушення.

$$|x|>|b|, \tag{1}$$

де x – операція запису (копіювання, форматований вивід, конкатенація), |x| – обсяг записаних даних у байтах (для API-інтерфейсів рядків це включає нульовий термінатор, для sprintf – довжина сформованого рядка), b – цільовий буфер (масив або виділений блок пам'яті), |b| – ємність буфера в байтах (за типом, місцем розміщення або контрактом параметра).

У подальшому аналізі клас стека розглядається як окремий ярлик для локалізацій, де буфер розміщується в стеку, а запис виконується без правильної перевірки меж. Типовим зразком переповнення стека є sprintf(buf, «%s.%s», a, b) при фіксованому buf і неперевірених довжинах a, b: номінально код є «законним», але він порушує співвідношення |x|>|b| через поєднання формату виводу та конкатенації.

У динамічній пам'яті розмір буфера змінюється під час виконання програми. Тому ризик визначається динамічною умовою. Перезапис за межами меж блоку призводить до пошкодження сусідніх виділень або метаданих видільника (наприклад, полів керування списком вільних блоків), що, в свою чергу, сприяє використанню після звільнення, подвійному звільненню та обхідним технікам, таким як розпилення купи.

$$|x_t|>|b_t| \tag{2}$$

де x_t – розмір запису в момент часу t, $|x_t|$ – довжина цих даних у байтах, b_t – цільовий буфер у момент часу t, $|b_t|$ ємність буфера в момент часу t , з урахуванням розподільника, вирівнювання, можливого перерозподілу та фрагментації, t – це дискретна точка виконання між операціями розподілу/звільнення.

Типовою ситуацією для переповнення купи є повторні виклики strcat після realloc у програмному забезпеченні комп'ютерної системи: ємність блоку пам'яті могла зменшитися через фрагментацію, і тому $|x_t|>|b_t|$ стає істинним тільки «на певному кроці».

У наступних розділах часовий аспект та залежності між операціями malloc/realloc/free будуть розглянуті окремо для випадків купи.

Це клас граничних помилок індексації, при яких допускається доступ з індексом, що виходить за межі дозволеного діапазону. Формально такий випадок визначається умовою (3): існує індекс i, для якого виконується доступ, Access(A,i)=1 і одночасно $i < 0$ або $i \geq n$.

$$\exists i: Access(A,i)=1 \wedge (i<0 \vee i \geq i) \tag{3}$$

де A – масив довжиною n, n – кількість елементів у масиві, i – індекс доступу, Access(A,i) = 1 – факт читання/запису елемента A[i], $i < 0$ або $i \geq n$ – вихід за межі допустимого діапазону $[0, n-1]$.

До поширених причин належать такі умови, як i ≤ n замість i < n, змішування типів зі знаком і без знака, а також недооцінка довжини під час форматованого виводу. Результати можуть нагадувати проблеми з підрахунком стека або купи, такі як пошкодження сусідніх елементів або метаданих. Проте ми розглядаємо помилки «off-by-one» як окрему категорію, оскільки їхні індикатори та шаблони перевірки відрізняються.

Найпоширенішими причинами є i ≤ n у циклі або змішування типів із знаком і без знака в перевірках контейнерів даних.

Щоб узгодити локальні випадки з глобальною структурою програми, ми використовуємо критерій графа. Нехай $G=(V,E)$, кожній ребері $e\in E$ присвоєно вагу $w(e)$ (верхня межа байтового потоку/ймовірність «небезпечних» даних). Ефективна пропускна здатність буферного вузла позначається

$$\sum_{e\to V_b} w(e) > cap_{eff}(V_b) \tag{4}$$

де $e\to V_b$ – набір ребер, що входять у буферний вузол V_b, $w(e)$ – вага ребра e (верхня межа байтового потоку або ймовірність “небезпечних” / заражених даних), $cap_{eff}(V_b)$ – ефективна (консервативна) ємність буфера.

Запропонований метод поєднує формальні умови ризику (1)–(4), бібліотеку шаблонів (CVE/NVD/OWASP), створення графічних представлень програм (AST/CFG/DFG) та компактний детектор YOLO з трьома класами (Stack, Heap, Off-by-One). Формальні умови слугують критеріями відліку: вони допомагають нам вибирати та анотувати відповідні підграфи, а також пов’язувати майбутні виявлення з конкретними причинами (наприклад, який шаблон і яка умова порушені). YOLO пропонує масштабовану локалізацію фрагментів ризику та ранжує їх за рівнем надійності, працюючи не з «сирим» текстом, а з багатоканальними графічними зображеннями, де кодуються структури управління, потоки даних/забруднення, буфери, довжини та наявність або відсутність перевірок меж. Загальна схема методу показана на рис. 1.

Вихідний код системного програмного забезпечення (C/C++ та мови для MCU) перетворюється на графічні представлення: абстрактне синтаксичне дерево (AST), графік управління (CFG) та графік потоку даних (DFG). Вузли відповідають функціям/блокам/змінним і буферам, ребра – викликом і передачам значень; ребрам присвоюються ваги w(e), які інтерпретуються як верхні оцінки байтового потоку або ймовірність отримання «небезпечних» (заражених) даних. Для номінальної ємності вузла буфера, $cap(V_b)$ ми вводимо консервативну ефективну ємність.

$$cap_{eff}(V_b)=\gamma\cdot cap(V_b), де 0<\gamma\le 1 \tag{5}$$

де V_b – буферний вузол у графі G, $cap(V_b)$ – номінальна ємність буфера, а $\gamma\in(0,1]$ – коефіцієнт консерватизму (компенсує неоднозначність типів, заповнення, фрагментацію, надлишкові символи тощо).

На основі ефективної ємності буферного вузла ми визначаємо ризик вузла як відповідний критерію графа «розрив» з нормалізацією та з урахуванням перевірок меж і сигналів аномалій.

$$R(V_b)=min\left(1,\frac{\sum_{e\to V_b} w(e)}{cap_{eff}(V_b)}\right)\cdot\left(1\text{-Check}(V_b)\right)\cdot P\left(Anomaly(V_b)\right) \tag{6}$$

де $\sum_{e\to V_b} w(e)$ – є загальним вхідним потоком у буфер V_b, $cap_{eff}(V_b)$ – є ефективною ємністю, $\text{Check}(V_b)\in\{0,1\}$ є оцінкою «аномалії» за правилами/шаблонами (наприклад, небезпечні API, підозрілі рядки формату, підписані/непідписані тощо). Значення $R(V_b)$ використовується як вага/пріоритет у виборі підграфа та як ціль для регуляризації детектора.

Після абстракції ми генеруємо багатоканальні рендери підграфів: канали кодують структури управління, потоки даних/забруднення, індикатори буфера/довжини та збіги шаблонів, за необхідності додається канал $R(V_b)$.

Для локалізації та класифікації ризикованих фрагментів ми використовуємо компактний детектор YOLO з трьома класами (Stack/Heap/Off-by-One). Основна головка YOLO генерує кадри-кандидати з оцінками об’єктивності та значеннями класів. Для підвищення чутливості до класових патернів додаються класові головки уточнення (легкі «експерти»), які отримують ознаки від базової «шиї» і навчаються на відповідному підкласі (stack/heap/off-by-one). Вибір головки уточнення здійснюється за допомогою arg max базових класових логітів YOLO (без окремого маршрутизатора або важких трансформаторів).

Функція навчання поєднує стандартні компоненти YOLO та регуляризацію ризик-сумісності.

$$L=\lambda_{obj}L_{obj}+\lambda_{cls}L_{cls}+\lambda_{loc}L_{loc}+\lambda_{cons}L_{cons}, \tag{7}$$

де L_{obj} – втрата об’єктивності, L_{cls} – класифікація (3 класи), L_{loc} – локалізація (регресія кадру), L_{cons} – регуляризатор узгодженості, що вирівнює оцінку виходу детектора зі скалярним $R(V_b)$ (наприклад, за допомогою BCE/Huber між нормалізованим ризиком і балом). Коефіцієнти λ вибираються для валідації з урахуванням дисбалансу класів.

Для оцінки ефективності ми використовували відкриті репозиторії CVE/NVD та власні зразки

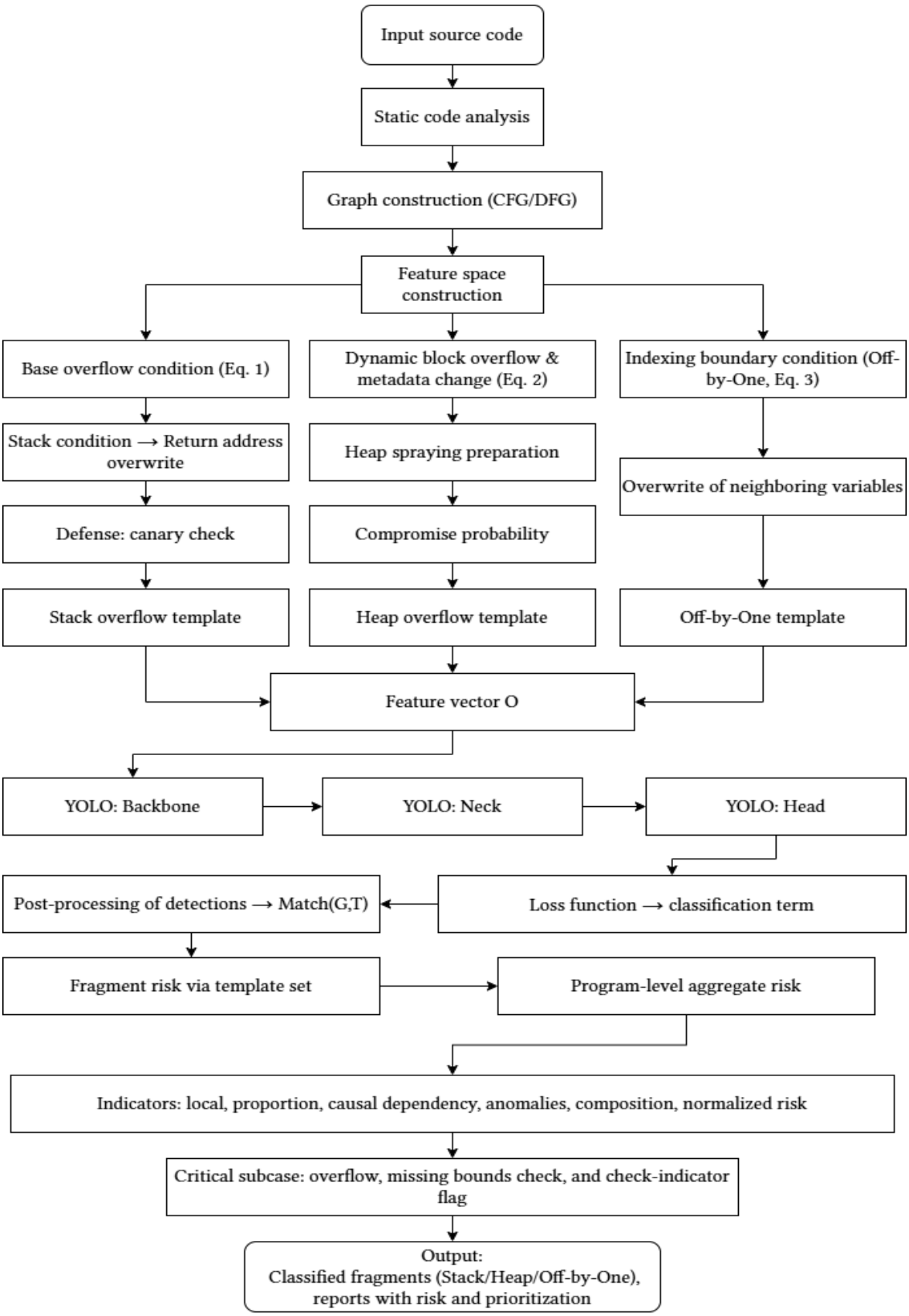

Рис. 1. Архітектура запропонованої моделі: вихідний код перетворюється на анотований AST/CFG/DFG (буфери, довжини, перевірки, забруднення), порівнюється з шаблонами і обчислює вагу потоків, потім перетворюється на багатоканальний рендерер зображень і проходить YOLO (3 класи) з класовими головками уточнення, після чого генерується звіт (клас, оцінка, код-проміжок, відповідний шаблон, пояснення)

системного коду. Загалом було згенеровано понад 12 000 фрагментів: 5100 із переповненням стека, 3200 із переповненням купи, 2700 із помилками «off-by-one»; решта – безпечні приклади для балансування. Щоб уникнути витоку між наборами, дані розділені у співвідношенні 70:15:15 без перетину проектів (розділені за репозиторіями, а не файлами). Для кожного фрагмента було побудовано AST/CFG/DFG; були анотовані буфери, довжини, перевірки меж та джерела забруднення. На підграфах відбувається автоматичне маркування відповідності шаблонів (CVE/NVD/OWASP), визначається ефективна ємність cap_{eff} та місцеві індикатори ризику $R(V_b)$. Під час відбору позитивних/негативних результатів були видалені дублікати комітів та тривіальні тестові артефакти.

В якості основних інструментів порівняння були обрані Cppcheck та Flawfinder (поточні стабільні версії з профілями «безпечного рядкового API»). Для нейронної бази був обраний детектор YOLO без графічних каналів та без регуляризації узгодженості з $R(V_b)$ використовувався – тільки «плоскі» рендери без особливих функцій.

Контроль якості розмітки проводився у два етапи: автоматичне заповнення шаблонів (покриття шаблонів CVE) та вибіркова ручна перевірка «складних» випадків (цикли, макроси, умовна компіляція). Для помилок типу «off-by-one» використовувалися стратифікований відбір та синтетичні варіації граничних умов (тільки під час навчання) для покращення представлення рідкісних шаблонів.

Детектор побудовано з використанням YOLO. Розмір партії – 32, оптимізатор Adam із початковою швидкістю навчання 0,0005. Навчання тривало 80 епох. Рендери вхідних підграфів масштабуються до 640×640 із застосуванням стандартних доповнень (перевертання/масштабування, коливання світлого кольору). Базова головка YOLO була доповнена головками уточнення для конкретних класів (для Stack/Heap/Off-by-One), які навчаються на відповідних підкласах. Функція втрат включає $\lambda_{obj}\lambda_{cls}\lambda_{loc}$ відібрані шляхом валідації, тоді як λ_{cons} активує регуляризацію узгодженості за допомогою $R(V_b)$.

У таблиці 1 наведено порівняння з базовим YOLO та традиційними інструментами. Було оцінено точність, відтворюваність та показник F1.

Запропонований метод перевершує базовий YOLO за всіма показниками. Покращення є результатом поєднання багатоканального рендерингу графіків, формальних умов, регуляризатора узгодженості з $R(V_b)$ та класових головок уточнення. Середній час обробки одного файлу становить 8,7 секунди, що є прийнятним для інтеграції в CI/CD.

Таблиця 1

Інструменти порівняння

Tool	Precision	Recall	F1-measure
YOLO (ours)	95,7 %	93,5 %	94,6 %
YOLO (base)	94,3 %	91,8 %	93,0 %
Cppcheck	76.2%	70.4%	73.2%
Flawfinder	72.5%	68.9%	70.6%

Висока точність (95,7%) і повнота (93,5%) свідчать про те, що модель виявляє як типові, так і «нетипові» шаблони BOF. Впровадження cap_{eff} і ризик $R(V_b)$ покращує локалізацію підозрілих областей і пріоритезацію повідомлень, а візуалізація графіків зменшує кількість помилкових збігів, спричинених структурним контекстом. Обмеження залишається в побудові графіків для дуже великих кодових баз (ресурсоємних), але це компенсується пояснюваністю та стабільністю метрик. Крім того, запропонований підхід демонструє вищі метрики, ніж системи з CNN+LSTM або XGBoost, згадані в, головним чином завдяки використанню формальних ознак та структурного аналізу коду.

Висновки. Запропоновано метод виявлення переповнення буфера в системному програмному забезпеченні, що поєднує формальні умови для визначення ефективної ємності cap_{eff}, індикатор ризику $R(V_b)$ та компактний детектор YOLO з класовими головами уточнення (Stack/Heap/Off-by-One). Формальні умови гарантують інтерпретованість та контрольованість порогів, бібліотека шаблонів (CVE/NVD/OWASP) підтримує узгодженість із визнаними шаблонами, а багатоканальні графічні рендери (AST/CFG/DFG) пропонують структурний контекст для точної локалізації підграфів. На реальних даних з CVE/NVD ми досягли точності 95,7%, показника F1 94,6% і середнього часу 8,7 секунди на файл, що перевершує базовий YOLO без графічних каналів і регуляризації, а також класичні інструменти SAST (Cppcheck, Flawfinder). Цей підхід сумісний з технологією CI/CD: він підтримує інкрементний аналіз, кешування артефактів, звітування SARIF та політики блокування випусків.

Практична цінність методу полягає у зменшенні «шуму» у великих репозиторіях та оптимізації сортування: кожне виявлення супрово-

джується класом, оцінкою, кодом, відповідним шаблоном та коротким поясненням, що дозволяє швидко призначити дефект відповідній команді та визначити терміновість виправлення. Збіг оцінок детектора з $R(V_b)$ покращує пріоритезацію сповіщень і зменшує кількість помилкових збігів у модулях з великою кількістю правильних перевірок меж. Разом із можливістю перенавчання на внутрішніх даних, це робить метод придатним для драйверів, компонентів RTOS і прошивки MCU, де обмеження часу та ресурсів є критичними.

Однак все ще існують обмеження в побудові графіків, що може бути ресурсоємним для дуже великих кодових баз; перенесення на інші мови (Rust/Go) вимагає адаптації парсерів і шаблонів. Загальна якість залежить від повноти розмітки та охоплення конфігурацій проекту (макроси, прапори збірки). Ці ризики пом'якшуються інкрементним режимом, кешуванням та політикою «швидкого профілювання» в CI.

Список літератури:

1. Dahl W.A., Erdodi L., Zennaro F.M. Stack-based buffer overflow detection using recurrent neural networks. arXiv, 2020. DOI: 10.48550/arXiv.2012.15116.

2. Li S., Zheng R., Zhou A., Liu L. A machine learning-based method for detecting buffer overflow attack with high accuracy. In: Proceedings of the 2020 International Conference on Computer, Network, Communication and Information Systems (CNCI 2020). 2020. DOI: 10.23977/CNCI2020090.

3. Kanaan E., Alam S.S., Akter M.S. Survey of machine learning techniques for detecting buffer overflow vulnerabilities. In: Proceedings of the 8th International Conference on Engineering Research, Innovation and Education (SUST). 2025. DOI: 10.13140/RG.2.2.22978.08640.

4. Zheng Z., Liu Y., Zhang B., Liu X., He H., Gong X. MSVAGraph: A multitype software buffer overflow vulnerability prediction method based on self attentive graph neural network. *Information and Software Technology*. 2023. DOI: 10.1016/j.infsof.2023.107246.

5. Zhai J., Qi Z., Yang H. Stack overflow vulnerability detection based on BiLSTM attention KAN deep learning model. *The Journal of Supercomputing*. 2025. DOI: 10.1007/s11227-025-07605-z.

6. Luo P., Zou D., Du Y., Jin H., Liu C., Shen J. Static detection of real world buffer overflow induced by loop. *Computers & Security*. 2019. DOI: 10.1016/j.cose.2019.101616.

7. Zou D., Wang S., Xu S., Li Z., Jin H. μVulDeePecker: A deep learning based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing.* 2019. DOI: 10.1109/TDSC.2019.2942930.

8. 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.* 2021. DOI: 10.1109/TDSC.2021.3051525.

9. Cao S., Sun X., Bo L., Wu R., Li B., Tao C. MVD: Memory related vulnerability detection based on flow sensitive graph neural networks. *In: Proceedings of the International Conference on Software Engineering.* 2022. DOI: 10.1145/3510003.3510219.

10. Cao S., Sun X., Bo L., Wei Y., Li B. BGNN4VD: Constructing bidirectional graph neural network for vulnerability detection. *Information and Software Technology.* 2021. DOI: 10.1016/j.infsof.2021.106576.

11. Yuan L., Fang Y., Zhang Q., Liu Z., Xu Y. Go source code vulnerability detection method based on graph neural network. *Applied Sciences*. 2025. DOI: 10.3390/app15126524.

12. Wang H., Qu Z., Sun L. EGVD: Efficient software vulnerability detection techniques based on graph neural network. *EAI Endorsed Transactions on Scalable Information Systems*. 2024. DOI: 10.4108/eetsis.5056.

13. Chen J., Yin Y., Cai S., Wang W., Wang S., Chen J. iGnnVD: A novel software vulnerability detection model based on integrated graph neural networks. *Science of Computer Programming.* 2024. DOI: 10.1016/j.scico.2024.103156.

14. Yang J., Ruan O., Zhang J. TensorGNN: Tensor based gated graph neural network for automatic vulnerability detection. *Software Testing, Verification and Reliability*. 2023. DOI: 10.1002/stvr.1867.

15. Hin D., Kan A., Chen H., Babar M.A. LineVD: Statement level vulnerability detection using graph neural networks. In: Proceedings of the 19th International Conference on Mining Software Repositories (MSR). 2022. DOI: 10.1145/3524842.3527949.

16. Yang X., Wang S., Li Y., Wang S. Does data sampling improve deep learning based vulnerability detection? Yeas! and Nays! In: Proceedings of the 45th International Conference on Software Engineering (ICSE). 2023. DOI: 10.1109/ICSE48619.2023.00192.

17. Das S., Fabiha S.T., Shafiq S., Medvidovic N. Are we learning the right features? A framework for evaluating DL based software vulnerability detection solutions. *arXiv*. 2025. DOI: 10.48550/arXiv.2501.13291.

18. Rahman M.M., Ceka I., Mao C., Chakraborty S., Ray B., Le W. Towards causal deep learning for vulnerability detection. *arXiv*. 2024. DOI: 10.48550/arXiv.2310.07958.

19. Yang X., Wang S., Zhou J., Zhu W. One for all does not work! Enhancing vulnerability detection by Mixture of Experts (MoE). *arXiv*. 2025. DOI: 10.48550/arXiv.2501.16454.

20. Schaad A., Binder D. Deep learning based vulnerability detection in binary executables. *arXiv*. 2022. DOI: 10.48550/arXiv.2212.01254.

21. Zhou X., Zhang T., Lo D. Large language model for vulnerability detection: Emerging results and future directions. *arXiv*. 2024. DOI: 10.48550/arXiv.2401.15468.

22. Nguyen V.-A., Nguyen D.Q., Nguyen V., Le T., Tran Q.H., Phung D. ReGVD: Revisiting graph neural networks for vulnerability detection. *arXiv*. 2021. DOI: 10.48550/arXiv.2110.07317.

23. Zeng Q., Xiong D., Wu Z., Qian K., Wang Y., Su Y. TACSan: Enhancing vulnerability detection with graph neural network. *Electronics*. 2024. Vol. 13, № 19, article 3813. DOI: 10.3390/electronics13193813.

24. Chu Z., Wan Y., Li Q., Wu Y., Zhang H., Sui Y., Xu G., Jin H. Graph neural networks for vulnerability detection: A counterfactual explanation. *arXiv*. 2024. DOI: 10.48550/arXiv.2404.15687.

25. Cao S., Sun X., Wu X., Lo D., Bo L., Li B., Liu W. Coca: Improving and explaining graph neural network based vulnerability detection systems. *arXiv*. 2024. DOI: 10.48550/arXiv.2401.14886.

Sierhieiev Y.V., Savenko O.S. DETECTION OF BUFFER OVERFLOW VULNERABILITIES IN SYSTEM SOFTWARE BASED ON GRAPH AND TRANSFORMER MODEL

Memory safety remains a central challenge for system and embedded software, where limited resources and weak isolation make correctness and robustness hard to guarantee. This paper introduces a method that couples formal risk conditions with graph-based code representations and a neural detector tailored to real-world patterns in system projects. Source code is transformed into consistent structures of abstract syntax, control flow, and data flow graphs, whose nodes and edges are annotated with buffer semantics, boundary checks, guards, and value propagation. From these structures, informative subgraphs are selected and rendered into multi-channel visualizations encoding the interaction between control and data, together with matches to curated vulnerability templates from widely used knowledge bases. The neural detector consumes these renderings as training and inference inputs, localizes suspicious fragments, and returns explainable alerts that include the predicted class, the code span, and a concise rationale aligned with formal risk indicators. The approach supports incremental analysis, artifact caching, and seamless operation within continuous integration and delivery pipelines, preserving preprocessor profiles and tool versions to ensure reproducibility and auditability. The contribution lies in unifying a formal notion of effective buffer capacity and flow weights with graph-centric visual learning, which reduces false alarms and improves portability across diverse codebases. Known limitations involve the cost of graph construction for very large projects and sensitivity to build configurations; these issues are mitigated through profiling, caching, and carefully designed pipeline policies. The reported outcomes indicate practical suitability for system software and straightforward integration into engineering workflows.

Key words: *buffer overflow, system software, abstract syntax tree, control flow graph, data flow graph, vulnerability knowledge bases, detector, interpretability, continuous integration and delivery.*

Дата надходження статті: 04.11.2025
Дата прийняття статті: 21.11.2025
Опубліковано: 30.12.2025