УДК 004.9 **doi: 10.26906/SUNZ.2025.1.156-161**

А. А. Коваленко, Т. Г. Куценко, А. С. Шаповал, О. В. Ситник, Я. С. Ні

Харківський національний університет радіоелектроніки, Харків, Україна

ОГЛЯД МЕТОДІВ АНАЛІЗУ БЕЗПЕЧНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Анотація. У статті проведений детальний огляд методів аналізу безпечного програмного забезпечення. **Метою дослідження** є детальний огляд існуючих методів статичного і динамічного аналізу програм, які є базою для забезпечення безпеки. Для цього проводиться порівняльний аналіз функціональних можливостей різних методів і інструментів та виявляються основні недоліки. **Результати дослідження.** Представлений детальний огляд і порівняльний аналіз існуючих методів зіставлення вихідних і бінарних файлів, аналізу змін між версіями програмного забезпечення, пошуку витоків динамічної пам'яті і помилок використання звільненої пам'яті. Доведено, що незважаючи на велику кількість доступних методів, існують серйозні обмеження на клас завдань, для яких вони можуть бути ефективно застосовані. **Висновок.** Необхідно прибирати існуючі обмеження, розвиваючи запропоновані методи. Доцільно розглядати методи не окремо, а в сукупності. При цьому треба враховувати вклад, який по окремості привносять методи, в загальну картину забезпечення безпеки програм. Такий спільний розвиток та сумісне використання розглянутих у статті методів дозволить отримати більш якісний аналіз.

Ключові слова: безпечне програмне забезпечення, вразливість програми, витік інформації, зміни версії, бінарний код.

Вступ

Актуальність дослідження. Програмне забезпечення (ПЗ) швидко інтегрується в усі сфери життя. Воно використовується в персональних пристроях і критичних системах. З розвитком ПЗ зростають загрози безпеки. Обсяг і складність систем постійно збільшуються. Хакери використовують складні методи атак. Вони експлуатують вразливості програмних систем. Атаки включають фішинг і вразливості «нульового дня».

Згідно зі звітами NIST, кількість вразливостей зросла. За десять років вона зросла з 8 000 до 28 000 [1]. Це відбувається попри розвиток інструментів аналізу. Кількість і якість таких інструментів постійно зростають. Виявлення та усунення загроз стає пріоритетним завданням. Тому необхідні нові методи захисту програмного забезпечення. Для цього необхідно провести аналіз існуючих методів та виявити їх недоліки. Це і зумовлює актуальність даного дослідження.

Аналіз джерел. Великі компанії розуміють важливість розробки безпечного програмного забезпечення, тому активно створюють спеціалізовані рішення та платформи для аналізу коду відкритого доступу. Одним із найвідоміших прикладів є CodeQL від компанії GitHub, який дозволяє виконувати запити до коду та виявляти потенційні вразливості [2]. CodeQL обробляє програмний код як базу даних, що дає можливість моделювати вразливості та помилки у вигляді запитів. Цей підхід забезпечує використання як стандартних запитів, розроблених дослідниками GitHub і спільнотою, так і створення власних запитів для вузькоспеціалізованого аналізу. Бази даних CodeQL містять абстрактне синтаксичне дерево, графи потоку даних і управління, що надає всебічне представлення коду та є основою для глибокого аналізу його безпеки.

Google також активно розвиває напрям безпеки програмного забезпечення, створивши проєкт OSS-Fuzz, який дозволяє виконувати фазинг відкритого коду на кластерних потужностях [3]. Цей проєкт було запущено після виявлення критичної вразливості Heartbleed в бібліотеці OpenSSL – одному з найпопулярніших інструментів для шифрування веб-трафіку [4]. Вразливість могла зачепити мільйони користувачів Інтернету та була викликана помилкою переповнення буфера пам'яті, яку можна було б виявити за допомогою фазингу. Однак на той момент цей метод тестування не був достатньо автоматизований і не отримав широкого розповсюдження серед розробників. Щоб змінити ситуацію, Google створив OSS-Fuzz, який автоматизував процес фазингу та зробив його доступним для відкритого програмного забезпечення.

OSS-Fuzz працює, запускаючи фазери для проєктів із відкритим вихідним кодом та сповіщаючи розробників про виявлені помилки, що дозволяє оперативно усувати уразливості.

З моменту свого запуску OSS-Fuzz став одним із найважливіших інструментів для спільноти розробників відкритого коду. Спочатку його було розроблено для мов C та C++, але згодом підтримка розширилася на інші безпечні для пам'яті мови, такі як Go, Rust і Python. Завдяки цьому проєкту вдалося виявити понад 22 000 критичних помилок у програмному забезпеченні, що свідчить про його ефективність та значення для забезпечення безпеки цифрових систем.

Окрім широкого спектру інструментів, необхідно також впроваджувати правильну методологію забезпечення безпеки програмного забезпечення, починаючи ще з етапу проєктування та розробки. Для цього застосовуються методи безпечного циклу розробки ПЗ, які дозволяють виявляти та усувати потенційні вразливості ще до того, як програмний продукт буде впроваджений у реальне використання. Одним із ключових елементів забезпечення комплексної безпеки є використання якісних інструментів аналізу коду, які допомагають розробникам знаходити та виправляти критичні помилки.

Однак, незважаючи на значну кількість доступних інструментів, що охоплюють різні аспекти

аналізу ПЗ, існує безліч невирішених питань, які суттєво впливають на рівень безпеки.

Методи статичного аналізу, що використовуються для пошуку вразливостей у програмному забезпеченні, не завжди враховують наявність відомих уразливостей у сторонніх бібліотеках та відкритих проєктах. Це створює ситуації, коли вразливий код може бути включений у проєкт без відома розробників, що піддає систему потенційним загрозам. Кожного року виявляються десятки критичних вразливостей «нульового дня», які стають причиною мільйонів атак зловмисників на користувачів та організації. Згідно зі звітом Google Project Zero за 2023 рік, було знайдено понад 50 таких уразливостей у різних операційних системах і браузерах, що потенційно зачепили мільйони користувачів [5].

Різні інструменти аналізу коду широко застосовуються у багатьох сферах, включаючи виявлення вірусів, пошук застарілих бібліотек, ідентифікацію клонів коду з помилками та інші важливі завдання. Однак методи фазингу, які використовуються для тестування безпеки, мають серйозні обмеження, зокрема при генерації структурованих даних та побудові ланцюжків викликів системних або бібліотечних функцій. Це призводить до недостатнього покриття коду під час тестування. За результатами аналізу ефективності проєкту OSS-Fuzz, в середньому він охоплює менше 40% коду, що вказує на необхідність удосконалення підходів до фазингу. Крім того, наразі не існує централізованої платформи, яка б дозволяла об'єднувати кілька інструментів аналізу в єдину систему, а також забезпечувала ефективне збирання, збереження та зручне використання великих обсягів артефактів відкритого програмного забезпечення.

Мета та задачі дослідження. Метою цього дослідження є детальний огляд існуючих методів статичного і динамічного аналізу програм, які є базою для забезпечення безпеки. Для цього проводиться порівняльний аналіз функціональних можливостей різних методів і інструментів та виявляються основні недоліки.

Результати досліджень

1. Огляд методів аналізу виправлених помилок та вразливостей. У роботі [6] автори визначають, яке конкретне змінення призвело до виникнення помилки в програмі. Для цього вони аналізують систему управління помилками, скануючи зареєстровані проблеми. Далі для кожної помилки вони намагаються встановити, яке саме внесене зміна є її виправленням. Для цього аналізується історія змін у системі керування версіями програмного забезпечення, а також проводиться детальний розгляд логів комітів. У статті описуються два основні методи аналізу – синтаксичний та семантичний, які застосовуються у певній послідовності. Синтаксичний аналіз виконується першим і здійснює пошук шаблонів у вигляді номерів помилок із системи управління помилками, використовуючи регулярні вирази. Семантичний аналіз, у свою чергу, шукає додаткову інформацію про виправлення помилки в логах комітів на основі таких факторів:

• наявність номера помилки у логах комітів;

• опис помилки міститься безпосередньо у повідомленні коміту;

• автор коміту також є відповідальним за її виправлення;

• файли, прикріплені до завдання, також присутні у відповідному коміті.

Після виявлення коміту, що містить виправлення, дослідники намагаються визначити, яке саме попереднє зміна спричинило появу цієї помилки в проєкті. Головним недоліком даного методу є залежність від якості описів змін, які залишають розробники у коментарях до комітів. Це не завжди є достовірним джерелом інформації, оскільки розробники можуть не включати детальні відомості про помилку у логи комітів або навіть не реєструвати її у системі управління помилками. У підсумку, це призводить до численних пропущених зв’язків між помилками та відповідними змінами. У середньому, 54% виправлених помилок не можуть бути однозначно пов’язані з відповідними виправленнями у коді. Подібні проблеми спостерігаються і в інших дослідженнях. Наприклад, інструмент BagCache [7], що використовує аналогічні метрики, також стикається з подібними труднощами.

Окремий напрям досліджень присвячений пошуку відомих вразливостей у вихідному коді. У роботі [8] автори використовують базу відомих вразливостей CVE для пошуку схожих помилок у заданому програмному проєкті. Для цього застосовуються три підходи: текстове порівняння, аналіз лексем та порівняння синтаксичних абстрактних дерев. У статті [9] представлений метод пошуку вразливостей із використанням технологій глибокого навчання. У цьому підході пошук вразливостей розглядається як задача класифікації тексту, а для навчання використовується тестовий набір SARD, що містить великий обсяг даних про помилки. Дані витягуються з цього набору, після чого застосовується глибоке навчання для автоматичного розпізнавання потенційних вразливостей у коді.

У роботі [10] описано інструмент VCCFinder, який знаходить вразливості, використовуючи класифікатори SVM. Навчання цього інструменту здійснюється на основі аналізу змін у репозиторіях понад 60 проєктів. Для виявлення змін, які могли усунути вразливість, реалізовано два підходи. Перший алгоритм здійснює пошук у базах даних вразливостей CVE та перевіряє, чи містяться у них посилання на відповідні коміти виправлення. Другий алгоритм аналізує історію комітів у різних проєктах та шукає вказівки на ідентифікатори CVE. На основі отриманих комітів з виправленнями реалізовано алгоритм для автоматичного пошуку подібних помилок у коді.

2. Методи відстеження змін між версіями програмного забезпечення. Відстеження змін у програмному забезпеченні має широкий спектр практичних застосувань. Аналізуючи внесені зміни, можна визначити, чи не містять вони нові помилки або потенційні вразливості. У випадку програм із відкритим вихідним кодом отримати інформацію про зміни можна за допомогою простого синтаксичного

порівняння двох версій файлів. Одним із найбільш відомих інструментів для такого порівняння є утиліта diff, що використовується в операційних системах Linux. Аналогічний підхід застосовується у всіх сучасних системах керування версіями програмного забезпечення. Проте синтаксичне порівняння файлів надає лише інформацію про додані чи видалені рядки у файлах і не враховує структурні (семантичні) зміни, що можуть впливати на функціональність програми.

Існує багато досліджень, присвячених удосконаленню методів аналізу змін між версіями програмного забезпечення.

Наприклад, інструмент change distler [11] використовує підхід, заснований на порівнянні синтаксичного абстрактного дерева програмного коду. Ця технологія базується на попередніх дослідженнях, у яких зміни у програмному коді представляються у вигляді модифікацій в абстрактному синтаксичному дереві. Алгоритм, який застосовується в цьому інструменті, дозволяє знаходити різні типи змін, зокрема додавання, видалення, переміщення або оновлення окремих фрагментів коду.

Ще одним ефективним інструментом для аналізу змін у програмному забезпеченні є LSDIFF, що був представлений у дослідженні [12]. Основною перевагою цього інструменту є його здатність виявляти систематичні зміни у коді та групувати їх у відповідні категорії.

Такий підхід значно спрощує перегляд та аналіз змін, оскільки базується на дослідженнях, які доводять, що сегменти коду зі схожими структурними характеристиками, як правило, зазнають аналогічних модифікацій. Подібний підхід застосовується в інструменті ClDiff [13], який, як і LSDIFF, спрямований на згруповане представлення змін у програмному коді для полегшення їхнього подальшого аналізу.

Аналіз існуючих методів показує, що більшість рішень для пошуку змін між версіями програмного забезпечення засновані на аналізі логів комітів та систем управління помилками. Однак такий підхід має значні обмеження, оскільки він не дозволяє виявити всі можливі виправлення вразливостей у коді. Це створює потенційні ризики, оскільки вразливі компоненти можуть залишитися непоміченими та продовжувати використовуватися в майбутніх версіях програмного забезпечення.

3. Методи зіставлення вихідного і бінарного коду. Інструмент Pigaios [14] призначений для зіставлення символів вихідного коду із символами відповідного бінарного коду. Важливо відзначити, що цей інструмент не вимагає, щоб вихідний код попередньо компілювався або компонувався, що значно спрощує процес аналізу. Для виконання зіставлення Pigaios проходить через кілька основних етапів:

– розбір вихідного і бінарного коду;

– вилучення артефактів для кожної функції вихідного та бінарного коду;

– зіставлення функцій вихідного коду з функціями бінарного коду на основі отриманих артефактів і визначення коефіцієнта схожості для кожного зіставлення.

Для розбору вихідного коду використовується компілятор Clang. Артефакти, які витягуються із коду, включають:

• імена функцій;

• константні рядки;

• кількість циклів, умов, викликів функцій, зовнішніх і глобальних змінних;

• кількість операторів "switch" та кількість випадків для них;

• наявність рекурсивних викликів у функціях;

• список викликаних функцій.

Після попереднього зіставлення функцій вихідного і бінарного коду на основі артефактів застосовуються додаткові евристики у разі, якщо не всі функції були зіставлені:

– зіставлення викликаних та викликаючих функцій: якщо дві функції були зіставлені, порівнюються також функції, які вони викликають.

– зіставлення найближчих функцій: якщо функції F1, F2 і F3 знаходяться в одному файлі вихідного коду, і функції F1 і F3 вже були зіставлені, то F2 порівнюється з функціями, адреси яких знаходяться між цими функціями у бінарному коді.

– зіставлення на основі рідко зустрічаючихся констант: для рядкових констант, що зустрічаються менше трьох разів, знаходяться функції у вихідному та бінарному коді, що їх використовують, після чого виконується їх порівняння та зіставлення.

Визначення коефіцієнта схожості для кожного зіставлення ґрунтується на технологіях машинного навчання. База для навчання була сформована вручну на основі аналізу спрацьовувань інструмента.

Узагальнюючи результати, можна сказати, що середній відсоток помилок першого роду становить 17,9%, тоді як помилки другого роду досягають 72,8%.

Додатково варто зазначити, що коли у бінарному файлі доступні імена функцій (тобто не було виконано команду "strip"), відсоток правильно зіставлених функцій значно зростає. Якщо ж інформація про імена функцій у бінарному файлі відсутня, середній відсоток помилок першого роду збільшується до 32%, а другого роду – до 80,2%.

Інший інструмент, RE-Source [158], також використовується для зіставлення бінарного коду з вихідним.

На вході він отримує лише бінарний файл, виконує пошук у спеціалізованих онлайн-сховищах (список наведено далі) і повертає інформацію про знайдені зіставлення.

Основні етапи онлайн-аналізу включають:

– вилучення характерних особливостей бінарного файлу, таких як константні значення операндів, імпортовані бібліотеки та виклики функцій, значення константних рядків;

– створення запиту на основі отриманих особливостей та його відправка у пошукові системи для аналізу вихідного коду.

Ще одним підходом є використання інструмента CodeBin [15], що застосовує новий метод ідентифікації бінарних функцій шляхом порівняння їх із базою попередньо обробленого вихідного коду.

Головна ідея цього інструмента полягає у вилученні характеристик вихідного коду, які зберігаються навіть після компіляції та складання, і, як правило, не залежать від конкретної платформи, компілятора або рівня оптимізації.

Характеристики вилучаються як із вихідного коду, так і з бінарних файлів, після чого вони використовуються для зіставлення відповідних функцій.

Порівняння функцій здійснюється шляхом генерації унікальних відбитків, що базуються на наступних характеристиках:

– виклики функцій у вихідному коді;
– виклики API-функцій та стандартних бібліотек;
– кількість аргументів у функціях;
– складність потоку управління, що визначається за цикломатичною складністю;
– цілочисельні та рядкові константи.

Для зіставлення бінарного коду з вихідним використовується спеціалізований анотований граф викликів функцій.

Кожна вершина цього графа містить додаткову інформацію про виклики функцій, використання змінних та інші важливі особливості коду. Процес зіставлення проходить через наступні кроки:

– витягуються характеристики вихідного коду;
– витягуються характеристики бінарного коду;
– виконується пошук бінарного анотованого графа викликів функцій у базі вихідних графів для знаходження відповідностей.

Результати тестування CodeBin показують, що середній відсоток помилок першого роду становить 4,2%, а другого роду – 79,3%.

Водночас аналіз проведених досліджень виявив кілька недоліків цього підходу:

– підтримуються лише певні бібліотеки з фіксованими версіями;
– для підтримки нових бібліотек чи конкретних версій необхідно вручну додавати унікальні ідентифікатори;
– визначення версії здійснюється на основі спеціального рядка, у якому вказана версія, але такий рядок може бути відсутній у деяких випадках;
– метод не враховує потік даних, що може призвести до неточностей у зіставленні;
– метод має труднощі з правильною обробкою випадків, коли функції були вбудовані в інший код.

У ході дослідження було проведено порівняльний аналіз інструментів для зіставлення вихідного і бінарного коду, серед яких Pigaios, RESource, CodeBin і Karta.

Початковий код CodeBin є закритим, що обмежує можливість його модифікації та інтеграції у власні розробки. Інструмент RESource має відкритий вихідний код, проте наразі не підтримується і потребує підключення до мережі для виконання онлайн-аналізу.

Важливо відзначити, що деякі кодові бази, з якими працював RESource, більше не доступні, що суттєво знижує його ефективність.

Інструмент Karta дозволяє зіставляти бінарний код із бібліотеками, при цьому його вихідний код є відкритим, що дає можливість розширення функціоналу. Також Karta може відновлювати версії бібліотек, які були статично злинковані у бінарному файлі.

Проте середній відсоток помилок першого роду в Karta становить 4,2%, а помилок другого роду – 79,3%, що свідчить про його обмеження у точності зіставлення.

Інструмент Pigaios приймає як вихідний, так і бінарний код, після чого здійснює зіставлення функцій. Його початковий код є відкритим, що дозволяє його вдосконалення.

Водночас середній відсоток помилок першого роду у Pigaios складає 17,9%, а другого роду – 72,8%.

Основними недоліками інструменту є неможливість коректного аналізу вбудованих функцій та відсутність підтримки аналізу потоку даних. Попри це, Pigaios продовжує активно розвиватися та залишається одним із найкращих методів зіставлення серед доступних рішень.

Аналіз отриманих результатів показує, що завдання якісного зіставлення вихідного та бінарного коду залишається актуальним і потребує подальших досліджень для розробки більш точних та ефективних методів.

4. Методи пошуку витоків динамічної пам'яті. Інструмент SMOKE [16] використовує два основних етапи для досягнення високої точності та масштабованості.

На першому етапі застосовується легкий, але менш точний аналіз для ідентифікації всіх можливих шляхів витоків пам'яті, при цьому відсіюються шляхи, які не можуть призвести до реального витоку. Для цього будується спеціальне програмне подання, яке називається графом потоку використання динамічної пам’яті (use-flow graph). Цей граф містить всю необхідну інформацію про потік управління для об'єктів купи, включаючи послідовність їх використання. Кожне ребро графа анотоване умовами, що визначають можливість переходу. SMOKE також моделює життєвий цикл покажчиків динамічної пам’яті, створюючи спеціальний вузол "out-of-scope", який позначає момент, коли об'єкт купи більше не використовується.

На другому етапі застосовується більш точний аналіз. Спочатку інструмент ідентифікує всі шляхи, які ведуть від точки виділення пам'яті до вузла "поза областю видимості" без виконання операції її звільнення.

Потім для кожного з виявлених шляхів застосовується вирішувач обмежень Z3 [17], щоб перевірити їхню здійсненність та відфільтрувати хибні спрацьовування. SMOKE забезпечує чутливість до потоку, шляхів і контексту програми, але має кілька недоліків:

– відсутня чутливість до полів об’єктів;
– аналіз покажчиків у графі потоку використання пам’яті є неточним;
– не враховує арифметичні операції над покажчиками;
– не підтримує аналіз бібліотечних функцій.

Інструмент PCA [18] базується на платформі LLVM [19] і виконує аналіз у кілька етапів. Спочатку

вихідні файли компілюються у проміжне подання LLVM. Потім, використовуючи компонувальник LLVM Gold, усі проміжні файли об'єднуються в єдине подання, після чого виконується аналіз покажчиків за алгоритмом Андерсена. Отримані дані використовуються для побудови графа викликів, який включає як прямі, так і непрямі виклики функцій. На завершальному етапі створюється міжпроцедурний граф залежностей даних. Головний недолік PCA – він забезпечує лише часткову чутливість до потоку управління.

Інструмент SVF [20], також побудований на базі LLVM, виконує подібні кроки: компіляція у проміжне подання, об'єднання проміжних файлів за допомогою компонувальника LLVM Gold і запуск модуля аналізу покажчиків (підтримуються різні алгоритми, зокрема аналіз Андерсена). SVF дозволяє розбивати пам'ять на окремі області, що значно покращує масштабованість і дає змогу аналізувати великі програми, а також застосовувати спеціалізовані детектори на основі графів потоку значень (ГПЗ).

Незважаючи на переваги, цей інструмент має такі обмеження:

– відсутня чутливість до полів та шляхів виконання програми;

– погано масштабується для аналізу великих програм.

Інструмент Fastcheck [21] застосовує міжпроцедурний алгоритм для виявлення витоків пам'яті в програмах мовою Сі. Він аналізує потік значень від точок виділення динамічної пам'яті до точок їхнього звільнення, використовуючи розріджене представлення програми у вигляді графа потоку значень. Ребра графа анотовані умовами розгалуження, що визначають можливість присвоєння значень.

Інформація про виклики та повернення функцій додається до ребер, що забезпечує контекстно-чутливий аналіз.

Пошук витоків пам'яті зводиться до задачі досяжності у графі потоку значень.

Інструмент Clang Static Analyzer (CSA) [183] використовує метод символьного виконання для пошуку різноманітних помилок, включаючи витоки пам'яті. Він забезпечує міжпроцедурний аналіз і чутливість до шляхів виконання. В основі аналізу лежить розібраний граф (РГ), який містить інформацію про можливі варіанти виконання програми. Для пошуку витоків пам'яті використовуються спеціалізовані детектори, які перевіряють, чи існують шляхи виконання, на яких не виконується операція звільнення пам'яті. Якщо такі шляхи виявлені, детектор формує відповідне повідомлення про помилку. Однак CSA має два критичні обмеження:

– цикли виконуються обмежену кількість разів (за замовчуванням 4), після чого РГ не розширюється, і подальші інструкції не аналізуються;

– кількість шляхів зростає експоненційно залежно від кількості інструкцій розгалужень, що робить аналіз великих програм малоефективним.

Інструмент Infer [22] використовується для аналізу програм, написаних мовами Java і Сі/Сі++, і дозволяє виявляти витоки пам'яті та використання нульових покажчиків. Його особливістю є висока чутливість до потоку, шляхів виконання, полів та контексту програми, що робить його ефективним у виявленні помилок.

Інструмент PML Checker [23] працює шляхом лексичного та синтаксичного аналізу вихідного коду програми, після чого будується абстрактне синтаксичне дерево (АСД). Для остаточного аналізу використовується символьний вирішувач, що дозволяє виявляти потенційні витоки пам'яті. Цей інструмент забезпечує чутливість до потоку, шляхів виконання та полів програми.

У табл. 1 наведено результати тестування інструментів.

Таблиця 1– **Результати порівняння чутливості та здатності до виявлення витоків інструментів**

Інструмент	Чутливість до потоку	Чутливість до шляхів	Чутливість до контексту	Чутливість до полів	Витік-1	Витік-2
CSA	Так	Так	Так	частково	Ні	Так
Infer	Так	Так	Так	частково	Ні	Так
SMOKE	Так	Так	Так	Ні	Ні	частково
PCA	Ні	Ні	Ні	Ні	Ні	Ні
Fastcheck	Так	Так	частково	Ні	частково	Так
SVF	Так	-	Так	Ні	Ні	Ні
PML Checker	Так	Так	Так	Так	Ні	частково

Аналіз даних, отриманих в результаті тестування, показує, що жоден із доступних інструментів не здатен виявляти витоки пам'яті з достатньо високою точністю та не володіє максимально можливою чутливістю.

Це підкреслює актуальність подальшої розробки вдосконалених методів пошуку витоків пам'яті.

Висновки

У статті представлений детальний огляд і порівняльний аналіз існуючих методів зіставлення вихідних і бінарних файлів, аналізу змін між версіями програмного забезпечення, пошуку витоків динамічної пам'яті і помилок використання звільненої пам'яті.

Незважаючи на велику кількість доступних методів, існують серйозні обмеження на клас завдань, для яких вони можуть бути ефективно застосовані.

Необхідно прибирати ці обмеження, розвиваючи запропоновані методи та доцільно розглядати їх не окремо, а в сукупності, враховуючи вклад, який по окремості привносять методи, в загальну картину забезпечення безпеки програм. Такий спільний розвиток та сумісне використання розглянутих у статті методів дозволить отримати нову якість аналізу.

Список літератури

1. NIST CVE report. URL: https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&search_type=all&isCpeNameSearch=false
2. Github. URL: https://github.com
3. Oss-fuzz. URL: https://github.com/google/oss-fuzz
4. Openssl. URL: https://www.openssl.org
5. Google-zero. URL: https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html
6. Sliwerski J., Zimmermann T., Zeller A. When Do Changes Induce Fixes, *ACM SIGSOFT Software Engineering Notes*, vol. 3, no. 4. 2005. DOI: https://doi.org/10.1145/1082983.1083147
7. F. Rahman, D. Posnett, A. Hindle, E. Barr, P. Devanbu, "BugCache for Inspections: Hit or Miss?", *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011. DOI: https://doi.org/10.1145/2025113.2025157
8. J. Yang, X. Song, Y. Xiong, Y. Meng, "An Open Source Software Defect Detection Technique Based on Homology Detection and Pre-identification Vulnerabilitys", *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2018. DOI: https://doi.org/10.1007/978-3-319-93554-6_94
9. A. Xu, T. Dai, H. Chen, Z. Ming, W. Li, "Vulnerability Detection for Source Code Using Contextual LSTM", *5th International Conference on Systems and Informatics*, 2018. DOI: https://doi.org/10.1109/ICSAI.2018.8599360
10. H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, Y. Acar, "VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits", *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications*, 2015. DOI: https://doi.org/10.1145/2810103.2813604
11. Change Distiller. URL: https://bitbucket.org/sealuzh/tools-changedistiller/src/master
12. M. Kim, D. Notkin, "Discovering and representing systematic code changes", *International Conference on Software Engineering (ICSE)*, pp. 309-319, 2009. URL: https://web.cs.ucla.edu/~miryung/Publications/icse09-lsdiff.pdf
13. Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, Wenyun Zhao, "ClDiff: Generating Concise Linked Code Differences", *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018. DOI: https://doi.org/10.1145/3238147.3238219
14. Pigaios. URL: https://github.com/joxeankoret/pigaios
15. ON MATCHING BINARY TO SOURCE CODE. https://users.encs.concordia.ca/~mmannan/student-resources/Thesis-MASc-Shahkar-2016.pdf
16. G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, C. Zhang, "SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code", *International Conference on Software Engineering (ICSE)*, 2019. URL: https://gangfan.github.io/assets/papers/gang_smoke_icse2019_preprint.pdf
17. Z3. URL: https://github.com/Z3Prover/z3
18. W. Li, H. Cai, Y. Sui, D. Manz, "PCA: memory leak detection using partial call-path analysis", *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020. DOI: https://doi.org/10.1145/3368089.3417923
19. Andersen pointer analysis. URL: https://github.com/grievejia/andersen
20. SVF. URL: https://github.com/SVF-tools/SVF
21. S. Cherem, L. Princehouse, R. Rugina, "Practical memory leak detection using guarded value-flow analysis", *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007. DOI: https://doi.org/10.1145/1250734.1250789
22. Fbinfer. URL: Available: https://fbinfer.com
23. X. Sun, S. Xu, C. Guo, J. Xu, N. Dong, X Ji, S. Zhang, "A Projection-Based Approach for Memory Leak Detection", *IEEE 42nd Annual Computer Software and Applications Conference*, 2018. https://doi.org/10.1109/COMPSAC.2018.10271

Received (Надійшла) 12.01.2025
Accepted for publication (Прийнята до друку) 05.03.2025

Overview of secure software analysis methods

A. Kovalenko, T. Kutsenko, A. Shapoval, O. Sytnyk, Ya. Ni

Abstract. The article provides a detailed review of methods for analyzing secure software. The purpose of the study is to provide a detailed review of existing methods for static and dynamic program analysis, which are the basis for ensuring security. For this purpose, a comparative analysis of the functional capabilities of various methods and tools is carried out and the main shortcomings are identified. Research results. A detailed review and comparative analysis of existing methods for comparing source and binary files, analyzing changes between software versions, searching for dynamic memory leaks and errors in using freed memory are presented. It is proved that despite the large number of available methods, there are serious limitations on the class of tasks for which they can be effectively applied. Conclusion. It is necessary to remove existing limitations by developing the proposed methods. It is advisable to consider the methods not separately, but in combination. In this case, it is necessary to take into account the contribution that the methods individually bring to the overall picture of ensuring program security. Such joint development and joint use of the methods considered in the article will allow for a higher-quality analysis.

Keywords: secure software, program vulnerability, information leak, version changes, binary code.