КІБЕРБЕЗПЕКА: освіта, наука, техніка № 2 (30), 2025
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
Спеціальний випуск ISSN 2663 – 4023
Special issue
DOI 10.28925/2663-4023.2025.30.957
УДК 004.056.5

Спис Денис Сергійович
аспірант кафедри Кібербезпеки
Державний університет «Київський авіаційний інститут», Київ, Україна
ORCID: 0009-0003-5316-8878
5771857@stud.kai.edu.ua

Ільєнко Анна Вадимівна
кандидат технічних наук, доцент, завідувач кафедри Кібербезпеки
Державний університет «Київський авіаційний інститут», Київ, Україна
ORCID: 0000-0001-8565-1117
anna.ilienko@npp.kai.edu.ua

СИСТЕМАТИЗАЦІЯ ТА КЛАСИФІКАЦІЯ СУЧАСНИХ МЕТОДІВ СТАТИЧНОГО АНАЛІЗУ ВЕБДОДАТКІВ

Анотація. Стаття присвячена аналізу сучасного стану статичного тестування безпеки додатків (SAST), що є ключовим компонентом практик DevSecOps. Представлено огляд фундаментальних методологій, які лежать в основі SAST-інструментів: від синтаксичного аналізу на основі правил і патернів, що працює з абстрактними синтаксичними деревами (AST), до більш складних семантичних підходів — аналізу потоків даних (DFA) на графах потоку керування (CFG), абстрактної інтерпретації та символьного виконання. Центральним елементом дослідження є емпіричне порівняння трьох провідних інструментів з відкритим вихідним кодом: SonarQube, який еволюціонував із платформи контролю якості коду; Semgrep, орієнтований на швидкість і легкість інтеграції в CI/CD; та GitHub CodeQL, що застосовує інноваційний підхід до представлення коду у вигляді реляційної бази даних для глибокого семантичного аналізу. Для об'єктивної оцінки їхньої продуктивності використано стандартизований набір тестів OWASP Benchmark, який дозволив виміряти ключові метрики: прецизійність (Precision), повноту (Recall), F1-Score та рівень хибнопозитивних спрацювань (FPR). Результати кількісного аналізу виявили суттєві відмінності: CodeQL продемонстрував найвищу збалансовану ефективність (F1-Score = 87,8 %) завдяки високій повноті виявлення складних вразливостей, пов'язаних із потоками даних, хоча й за рахунок тривалого часу сканування; Semgrep забезпечив оптимальне співвідношення швидкості та точності (F1-Score = 70,3 %), що робить його придатним для швидких ітерацій у конвеєрах CI/CD; натомість SonarQube показав найнижчу ефективність (F1-Score = 49,8 %), пропускаючи понад 60 % реальних вразливостей, що підтверджує обмеження неспеціалізованих рішень у задачах безпеки. Дослідження доводить, що вибір SAST-інструмента є не лише технічним, а й стратегічним рішенням, яке потребує ретельного аналізу компромісів між глибиною аналізу, рівнем «шуму» та швидкістю інтеграції для формування ефективної й збалансованої програми безпеки вебдодатків.

Ключові слова: кібербезпека, статичний аналіз коду, SAST, DevSecOps, безпека додатків, OWASP Benchmark, хибнопозитивні спрацювання, хибнонегативні спрацювання, SonarQube, CodeQL, Semgrep, аналіз потоків даних, метрики якості, управління ризиками.

ВСТУП

Постановка проблеми. Статичний аналіз коду — це методика перевірки вихідного коду без його виконання з метою виявлення дефектів, вразливостей, порушень стилю або архітектурних антипатернів. Вона є критичним компонентом практик DevSecOps та «shift-left» підходів, оскільки дозволяє виявити проблеми на ранніх етапах розробки,

скорочуючи вартість ймовірного виправлення помилок і покращуючи якість продукту загалом [1][2].

Теоретичною основою статичного аналізу є аналіз структури та семантики програми, зазвичай із використанням синтаксичного й контрольного потоку, абстрактної інтерпретації, аналізу потоків даних, символічного виконання та інших методів, які дозволяють робити формальні висновки без запуску коду [3]. У практичній площині такі інструменти застосовуються для автоматичного виявлення багів, дублікатів, порушень стилю, ознак технічного боргу та вразливостей безпеки. Завдяки високій масштабованості, статичний аналіз застосовується у великих індустріальних проєктах: наприклад, Facebook розробив власний аналізатор Zoncolan, який сканує понад 100 млн рядків коду менш ніж за 30 хвилин, використовуючи спеціальні правила для виявлення потенційних дефектів [4]. Попри численні переваги, методи статичного аналізу мають інгерентні обмеження. Зокрема, вони генерують хибні спрацьовування (false positives) та хибнонегативні помилки (false negatives), можуть не виявляти динамічні або контекстуально залежні дефекти, а також ускладнені у задачах з комплексною бізнес-логікою або великою кількістю залежностей [1] [17].

Аналіз останніх досліджень і публікацій. З огляду на ці виклики, сучасні дослідження зосереджені на:

- емпіричних оцінках наявних інструментів з погляду точності, продуктивності, обсягу хибних результатів та підтримки мов (наприклад, SonarQube, PMD, FindBugs, Checkstyle) [5];
- розвиток гібридних та інтелектуальних методів, де класичні підходи (правила, патерни, аналіз потоку даних) доповнюються алгоритмами машинного навчання та глибокого навчання. Особливу увагу приділяють використанню великих мовних моделей (LLM) як інструментів для триажу результатів, генерації сигнатур для виявлення вразливостей [6].

Зведення результатів цих досліджень, а також ідентифікація ключових викликів і нових підходів створюють основу для формулювання цілей цієї роботи.

Метою статті є провести системний аналіз методів статичного аналізу програмного коду вебдодатків, оцінити їх ефективність у виявленні вразливостей, здійснити порівняння сучасних інструментів на їх основі та визначити перспективні напрями підвищення точності та продуктивності таких підходів.

ОГЛЯД СУЧАСНИХ ПІДХОДІВ ДО СТАТИЧНОГО АНАЛІЗУ КОДУ

У сучасному світі розробки програмного забезпечення (ПЗ), де швидкість і складність проєктів невпинно зростають, забезпечення якості та безпеки коду стає ключовим викликом. Одним із найефективніших інструментів для досягнення цих цілей є статичний аналіз коду. Головна роль статичного аналізу полягає у превентивному виявленні потенційних помилок, вразливостей та відхилень від стандартів кодування на ранніх етапах життєвого циклу розробки. Такий підхід дозволяє значно знизити вартість виправлення дефектів, оскільки помилки, знайдені після релізу продукту, можуть коштувати в десятки, а то й сотні разів дорожче [7].

Статичний аналіз виконує низку критично важливих завдань, спрямованих на покращення загальної якості програмного продукту.

Таблиця 1

Основні завдання статичного аналізу

Завдання	Опис	Ключові приклади
Виявлення помилок і дефектів	Пошук логічних помилок та аномалій, що можуть призвести до збоїв або некоректної поведінки програми.	Неініціалізовані змінні, доступ за межі масиву, ділення на нуль, невірне використання API.
Пошук вразливостей безпеки	Ідентифікація слабких місць, які можуть бути експлуатовані зловмисниками для отримання несанкціонованого доступу або порушення цілісності системи.	Ін'єкції (SQL, OS Command), міжсайтовий скриптинг (XSS), розголошення чутливих даних, використання застарілих бібліотек з відомими вразливостями.
Відповідність стандартам кодування	Перевірка коду на дотримання встановлених правил, рекомендацій та стильових настанов для підвищення читабельності, підтримки та уніфікації.	Відсутність коментарів, неправильне іменування змінних/функцій, порушення відступів, надмірна складність функцій, недотримання архітектурних патернів.
Обчислення метрик коду	Збір кількісних показників, що відображають різні аспекти якості коду, такі як складність, розмір, зв'язність та покриття тестами.	Цикломатична складність, LOC (рядки коду), коефіцієнт зв'язності (coupling), коефіцієнт когезії (cohesion), дублювання коду.

Ринок інструментів статичного аналізу надзвичайно різноманітний: від простих лінтерів, вбудованих у середовище розробки, до потужних комерційних платформ, здатних аналізувати мільйони рядків коду. Ці інструменти не є взаємозамінними, оскільки вони працюють за різними принципами, мають різну глибину аналізу, швидкість роботи та точність. Вибір правильного інструменту — це завжди пошук компромісу між глибиною виявлення помилок та швидкістю отримання результатів. Наприклад, для проєкту, де головним пріоритетом є швидкість розробки, ідеально підійде швидкий лінтер для підтримки чистоти коду. Водночас для систем із високими вимогами до безпеки (наприклад, у банківській чи медичній сфері) знадобиться значно складніший і повільніший аналізатор, здатний знаходити глибокі, неочевидні вразливості.Саме тому для навігації в цьому розмаїтті використовується класифікація. Вона дозволяє оцінити інструменти за об'єктивними критеріями та зробити усвідомлений вибір, що відповідає цілям проєкту, бюджету та кваліфікації команди. Наведена нижче таблиця розглядає чотири ключові осі для такої класифікації.

Таблиця 2

Класифікаційна основа статичного аналізу

Критерій класифікації	Суть критерію	Спектр варіантів / Рівні	Практичний вплив на результат
Методологія аналізу	Фундаментальний алгоритм, що лежить в основі інструмента.	Патерни, Потік даних, Символьне виконання, Абстрактна інтерпретація.	Інструменти на основі патернів швидкі, але знаходять лише відомі помилки. Символьне виконання може знаходити складні логічні вразливості, але потребує значно більше часу.
Глибина аналізу (Чутливість)	Наскільки глибоко інструмент "розуміє" логіку та потік виконання програми.	Потоково-нечутливий → Потоково-чутливий Контекстно-нечутливий → Контекстно-чутливий	Потоково-чутливий аналіз відстежує зміни значень змінних по ходу виконання програми. Контекстно-чутливий аналіз розрізняє виклики однієї й тієї ж функції в різних місцях коду з різними аргументами або в різних станах програми.

Критерій класифікації	Суть критерію	Спектр варіантів / Рівні	Практичний вплив на результат
Масштаб аналізу	Яку частину кодової бази інструмент розглядає одночасно для виявлення проблеми.	Внутрішньо-процедурний (одна функція) → Між-процедурний (взаємодія функцій) → Проєктний (увесь проєкт).	Проєктний аналіз може знайти помилки, що виникають через взаємодію далеких один від одного модулів (наприклад, дані з одного модуля некоректно використовуються в іншому).
Ціль дефектів	На які класи помилок орієнтований інструмент.	Загального призначення (якість коду, стиль) → Спеціалізовані (безпека, багатопотоковість, продуктивність).	Спеціалізований SAST-інструмент знайде більше вразливостей безпеки (CWE Top 25), ніж звичайний лінтер, який зосереджений на стилі кодування.
Підхід до результатів	Філософія інструмента щодо співвідношення хибних спрацювань (False Positives) і пропущених помилок (False Negatives).	Прагматичний (баланс, мінімум хибних спрацювань, ризик пропустити помилки) → Формальний / Sound (знаходить усі можливі помилки, ризик великої кількості хибних спрацювань).	Формальні інструменти незамінні для систем високої надійності (авіоніка, медицина), де пропустити помилку неприпустимо. Прагматичні краще підходять для швидкої веброзробки.
Точка інтеграції	На якому етапі життєвого циклу розробки ПЗ застосовується інструмент.	IDE (в реальному часі) → Pre-commit хуки → CI/CD пайплайн → Аудит за вимогою.	Інтеграція в IDE дає миттєвий зворотний зв'язок розробнику. Інтеграція в CI/CD слугує "охоронцем" якості для всієї команди, блокуючи злиття неякісного коду.
Підтримувані мови програмування	Мови програмування, для яких інструмент розроблено та оптимізовано.	Java, C#, Python, JavaScript, Go, Swift, Rust, C/C++, PHP, Kotlin тощо (конкретний список залежить від інструменту).	Вибір інструменту безпосередньо залежить від мов, на яких написаний проєкт. Інструмент, який підтримує основні мови проєкту, забезпечить повне покриття та точний аналіз.

Світ інструментів статичного аналізу є надзвичайно багатогранним. Не існує єдиного "найкращого" рішення для всіх завдань. Вибір конкретного інструменту — це стратегічне рішення, що вимагає ретельного врахування компромісів між глибиною аналізу, швидкістю, точністю та простотою інтеграції. Лише усвідомлений підхід, заснований на розумінні цих критеріїв, дозволяє підібрати аналізатор, що якнайкраще відповідатиме унікальним потребам проєкту, команді та бізнес-цілям.

Методологічні основи статичного аналізу.

У цій частині розглянемо фундаментальні алгоритми та теорії, що лежать в основі інструментів статичного аналізу. Вибір методології визначає базові можливості, обмеження та компроміси будь-якого аналізатора. Перед тим як перейти до розгляду основних методологій, важливо розрізняти їх від фундаментальних етапів компіляції, які є їхньою передумовою. Такі процеси, як лексичний, синтаксичний та семантичний аналіз, самі по собі є формами статичного аналізу, оскільки вони перевіряють код без

його виконання. Вони аналізують відповідність граматиці мови, правильність типів та оголошень змінних. Однак у контексті класифікації інструментів їх слід розглядати як базові, обов'язкові етапи, а не як самостійні високорівневі методології. Вони створюють ключову структуру даних — Абстрактне синтаксичне дерево (AST) — на якій вже оперують більш потужні методи, що розглядаються нижче.

Аналіз на основі правил та зіставлення зразків (Pattern Matching).

Це найпростіша форма статичного аналізу, що оперує на синтаксичному представленні коду. Процес починається з лексичного аналізу, який перетворює вихідний код на послідовність токенів, та синтаксичного аналізу (парсингу), що будує абстрактне синтаксичне дерево (AST). Аналітичний рушій потім обходить це AST або потік токенів, шукаючи збіги із заздалегідь визначеними шаблонами (зразками), які сигналізують про потенційні проблеми. Ці шаблони можуть бути виражені у вигляді простих регулярних виразів, запитів XPath до AST або спеціалізованих правил мовою конкретного інструменту. Наприклад, інструмент PMD використовує правила на основі XPath для навігації по AST Java-коду та виявлення проблемних конструкцій [8].

Наведена нижче діаграма ілюструє типовий робочий процес аналізатора на основі правил. Вихідний код спочатку перетворюється на проміжне представлення (AST), яке потім аналізується рушієм перевірки правил на відповідність шаблонам із бібліотеки правил [9].

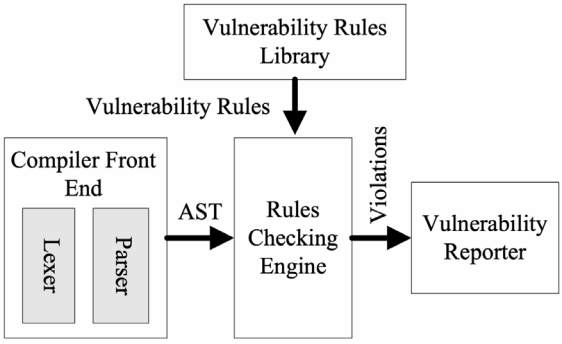

Рис. 1. Діаграма робочого процесу аналізатора на основі правил

Абстрактне синтаксичне дерево (AST) є ключовою структурою даних. Воно представляє синтаксичну структуру коду у вигляді ієрархії, де кожен вузол відповідає певній конструкції мови. Наприклад, для простого виразу *var x = y + 1;* AST може виглядати наступним чином:

```
VariableDeclaration (Оголошення змінної)
                    |
          VariableDeclarator (Декларатор змінної)
                   /       \
Identifier (x)          BinaryExpression (+) (Бінарний вираз)
(Ідентифікатор)          /                  \
                  Identifier (y)        Literal (1) (Літерал)
                  (Ідентифікатор)
```

Рис. 2. Приклад абстрактного синтаксичного дерева

Основне застосування цієї методології — забезпечення дотримання стандартів кодування (наприклад, угод щодо іменування, стилістичних посібників, як-от PEP 8 для Python), виявлення простих антипатернів або "code smells" (наприклад, надмірно складних методів) та знаходження відомих, синтаксично очевидних помилок.

Інструменти, такі як Checkstyle та PMD, є класичними прикладами цього підходу. Вони ефективні для підтримки консистентності та читабельності коду у великих командах, автоматизуючи перевірки, які інакше довелося б виконувати вручну під час код-рев'ю [10].

Ключовим обмеженням є відсутність семантичного розуміння. Аналізатор, що базується на правилах, не може міркувати про стан програми, потік даних або значення коду під час виконання. Він бачить "текст" програми, а не її "поведінку". Ця поверховість часто призводить до високого рівня хибнопозитивних спрацювань (false positives), оскільки інструмент не може відрізнити синтаксично підозрілий шаблон від семантично безпечного. Наприклад, правило може попередити про потенційне розіменування нульового вказівника, знайшовши виклик методу на об'єкті, але не маючи змоги визначити, чи може цей об'єкт насправді бути null у даній точці виконання. Ця методологія слугує найдоступнішою точкою входу в статичний аналіз для багатьох команд розробників завдяки своїй швидкості, простоті налаштування та негайній віддачі у вигляді покращення стилю коду. Однак, саме її обмеження, зокрема високий рівень "шуму" у вигляді хибнопозитивних спрацювань, стають головним рушієм для переходу до більш складних, семантично-орієнтованих методологій. Коли команди усвідомлюють, що синтаксичних перевірок недостатньо для виявлення глибоких логічних помилок та вразливостей, виникає потреба в аналізі, який розуміє поведінку програми, а не лише її синтаксичну структуру. Таким чином, аналіз на основі правил є не просто однією з категорій; він є каталізатором еволюції практичного застосування статичного аналізу, стимулюючи попит на методи, що розглядаються далі.

Аналіз потоку керування та потоку даних.

Цей клас технік походить з класичної теорії оптимізації компіляторів, детально описаної у фундаментальних працях, таких як "Книга Дракона" Ахо, Сеті та Ульмана. Першим кроком є перетворення програми на граф потоку керування (Control-Flow Graph, CFG), де вузли представляють базові блоки (послідовності інструкцій без переходів всередині), а ребра — можливі передачі керування між блоками. Ця структура даних є основою для аналізу всіх можливих шляхів виконання програми [11]. Для кращого розуміння розглянемо два приклади. Перший ілюструє базове розгалуження, а другий — цикл.

Приклад 1: CFG для умовного оператора

Наступний граф відповідає простій функції мовою C, що знаходить максимальне з двох чисел. Він демонструє, як if-else оператор створює два паралельні шляхи, які згодом зливаються в один.

```
int max(int a, int b) {
  int result;
  if (a > b) {
    result = a;
  } else {
    result = b;
  }
  return result;
}
```

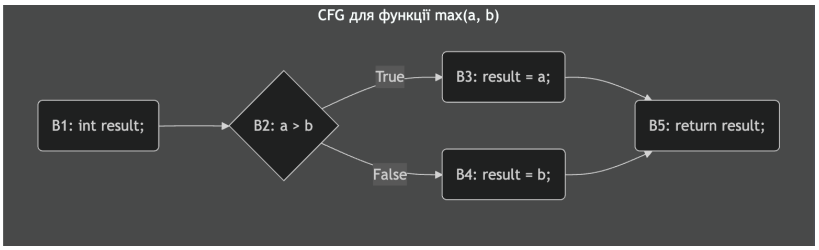

Рис. 3. Граф потоку керування для умовного оператора

Приклад 2: CFG із циклом

Цей граф візуалізує складніший фрагмент коду мовою C з циклом while. Ключовим елементом тут є зворотне ребро (back edge) від блоку B5 до B2, яке і формує цикл, повертаючи потік виконання на повторну перевірку умови.

```
void complex_loop_example() {
  // B1: Ініціалізація
  int a = 1, b = 2, i = 0;

  // B2: Умова циклу
  while (i < 2) {
    // B6: Умова всередині циклу
    if (a > b) {
      // B3: Гілка 'true'
      a = a + b;
    } else {
      // B4: Гілка 'false'
      b = a - b;
    }
    // B5: Інкремент
    i++;
  }
  // B7: Кінець
}
```

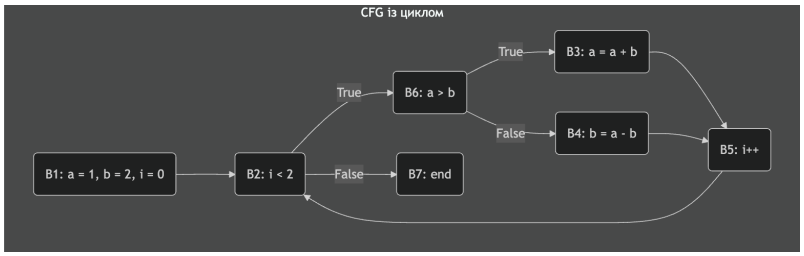

Рис. 4. Граф потоку керування із циклом

Аналіз потоку даних (Data-Flow Analysis) полягає у поширенні інформації (фактів потоку даних) вздовж шляхів CFG. Цей процес формалізується за допомогою спеціальної структури (framework), що складається з:

- Області значень (domain): Множина можливих значень, які може приймати інформація (наприклад, множина змінних, які можуть бути неініціалізованими).

- Функцій передачі (transfer functions): Для кожного базового блоку визначається функція, що моделює його вплив на факти потоку даних (наприклад, як блок змінює множину ініціалізованих змінних).

- Оператора з'єднання (meet operator): Оператор, що об'єднує інформацію, яка надходить з різних шляхів керування (наприклад, об'єднання або перетин множин).

Для розв'язання системи рівнянь потоку даних використовується ітеративний алгоритм, який багаторазово обчислює вихідну інформацію для кожного вузла на основі вхідної, доки система не досягне стабільного стану — нерухомої точки (fixpoint), коли подальші ітерації не змінюють результатів. Ця методологія дозволяє виявляти широкий спектр помилок коректності. Класичними прикладами є:

- Аналіз досяжних визначень (Reaching Definitions): Визначає, які присвоєння значень змінним можуть "досягти" певної точки програми.

- Аналіз "живих" змінних (Live Variable Analysis): Визначає, чи буде значення змінної використано в майбутньому на шляху виконання. Це допомагає виявляти використання неініціалізованих змінних.

- Виявлення недосяжного коду (Unreachable Code): Знаходить частини коду, які ніколи не можуть бути виконані.

Розглянемо, як аналіз потоку даних може знайти потенційне використання неініціалізованої змінної. Цей аналіз відстежує множину змінних, які гарантовано ініціалізовані в кожній точці програми. Перед кожним використанням змінної перевірити, чи входить вона до множини гарантовано ініціалізованих змінних.

```
void check_init(int p) {
  int x;      // B1: x оголошено, але не ініціалізовано
  if (p > 0) {  // B2: Умова
    x = 1;    // B3: x ініціалізовано
  }
            // B4: Точка злиття шляхів
  int y = x;  // B5: x використовується. Чи завжди він ініціалізований?
}
```

Аналізатор будує CFG і поширює ним множину ініціалізованих змінних. У точках злиття шляхів (після *if-else*, наприклад) нова множина є перетином (intersection) множин, що прийшли з різних гілок. Це гарантує, що змінна вважається ініціалізованою, тільки якщо вона була ініціалізована на всіх можливих шляхах. На цьому графі показано, як множина ініціалізованих змінних (DefInit) змінюється при проходженні коду.

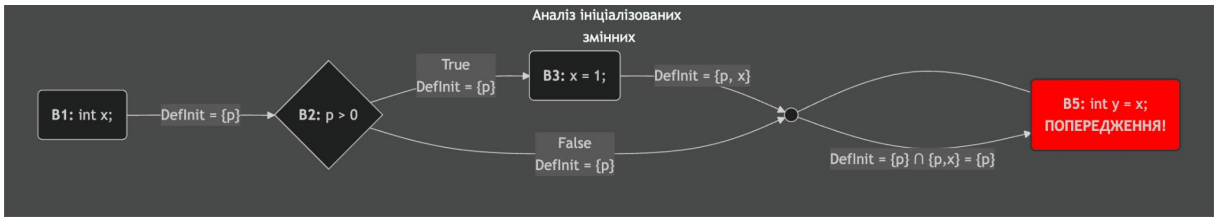

Рис. 5. Граф аналізу потоку даних знаходження використання неініціалізованої змінної

Цей приклад демонструє потужність аналізу потоку даних: він може знаходити помилки, які залежать від шляху виконання програми, чого не може зробити простий аналіз на основі патернів.

Ключовим застосуванням аналізу потоку даних у сфері безпеки є аналіз зараження. Цей метод відстежує поширення "заражених" (недовірених) даних, що надходять із зовнішніх джерел (наприклад, введення користувача, мережеві запити), через програму. Якщо такі дані досягають "вразливих точок" (sinks), таких як виконання SQL-запиту, системної команди або виведення на HTML-сторінку, без належної перевірки та очищення (санітизації), аналізатор повідомляє про потенційну вразливість, таку як SQL-ін'єкція, командна ін'єкція або міжсайтовий скриптинг (XSS) [12].

Абстрактна інтерпретація.

Абстрактна інтерпретація, розроблена Патріком та Радхією Кузо, є теорією формальних методів для створення статичних аналізаторів, коректних за побудовою (sound-by-construction). Вона надає математичну основу для створення коректної надапроксимації (sound over-approximation) семантики програми. Термін "коректна" (sound) означає, що якщо аналіз стверджує, що певна властивість виконується (наприклад, "ця змінна завжди додатна"), то це гарантовано буде правдою для всіх можливих виконань програми. "Надапроксимація" означає, що аналіз може повідомляти про поведінку, яка насправді неможлива (що призводить до хибнопозитивних спрацювань), але він ніколи не пропустить реальну поведінку (нуль хибнонегативних спрацювань) [13].

Теорія базується на відображенні між конкретною семантикою програми (фактичною, часто нескінченною множиною можливих станів) та абстрактною областю (простішим, скінченним представленням). Це відображення формалізується через з'єднання Галуа (Galois connection), що визначається функцією абстракції (‘α‘) та функцією конкретизації (‘γ‘). Для забезпечення збіжності аналізу при роботі з циклами та нескінченними областями (наприклад, цілими числами), абстрактна інтерпретація використовує оператори розширення (widening) та звуження (narrowing). Оператор розширення прискорює досягнення нерухомої точки, потенційно "перестрибуючи" через багато ітерацій, що гарантує завершення аналізу, але може призвести до втрати точності. Оператор звуження потім може використовуватися для уточнення отриманого результату.

Розглянемо аналіз, який визначає знак цілочисельних змінних, що дозволить зрозуміти абстрактну інтерпретацію.

Конкретний домен: Множина всіх можливих значень змінної x — це множина цілих чисел $\mathbb{Z} = \{..., -2, -1, 0, 1, 2, ...\}$. Ця множина нескінченна, і відстежувати всі можливі значення неможливо.

Абстрактний домен: Замість конкретних чисел ми відстежуємо лише їхній знак. Наш абстрактний домен скінченний: $Sign = \{ \bot, -, 0, +, \top \}$, де:

$\bot$ *(Bottom)* — недосяжний стан.

\- — будь-яке від'ємне число.

0 — нуль.

\+ — будь-яке додатне число.

$\top$ (Top) — будь-яке можливе число (стан невизначеності).

Функція абстракції (α) та конкретизації (γ) утворюють з'єднання Галуа і є мостом між двома доменами.

Функція абстракції α відображає множину конкретних станів у один абстрактний елемент, що їх представляє:

$\alpha(\{-3, -8, -100\}) \rightarrow -$

$\alpha(\{-5, 10\}) \rightarrow \top$ *(Top)*

$\alpha(\{\}) \rightarrow \bot$ *(Bottom)*

Функція конкретизації γ відображає абстрактний елемент назад у множину конкретних станів, які він апроксимує:

$\gamma(-) \rightarrow \{x \in \mathbb{Z} \mid x < 0\}$ (усі від'ємні цілі числа)

$\gamma(T) \rightarrow \mathbb{Z}$ (усі цілі числа)

Аналізатор виконує операції в абстрактному домені. Наприклад, для множення:

*(+) * (+) → +*

*(-) * (+) → -*

Розглянемо код *int y = 10 / x;*. Якщо аналіз визначив, що абстрактне значення *x* у цій точці може бути *0* або *T*, він видасть попередження про можливе ділення на нуль.

Оператори розширення (widening) та звуження (narrowing) необхідні для аналізу циклів, щоб гарантувати завершення процесу. Розглянемо цикл:

```
int x = 0;
while (x < 100) {
  x = x + 1;
}
```

Аналіз інтервалів значень без розширення міг би ітерувати 100 разів *([0, 0], [0, 1], ... [0, 100])*. Оператор розширення (widening) помічає, що інтервал стабільно зростає, і "перестрибує" до стабільного стану, наприклад, *[0, +∞)*, гарантуючи завершення за кілька кроків, але втрачаючи точність. Оператор звуження (narrowing) може потім уточнити цей результат, застосувавши умову виходу з циклу (*x >= 100*), і звузити інтервал до *[100, +∞)*.

Головна сила абстрактної інтерпретації полягає у здатності з математичною строгістю доводити відсутність певних класів помилок часу виконання. Це робить її незамінною для систем критичної безпеки (авіоніка, автомобільна промисловість, медичне обладнання), де необхідно гарантувати такі властивості, як відсутність переповнення буфера, ділення на нуль або арифметичного переповнення.

Символьне виконання.

Символьне виконання, вперше запропоноване Джеймсом Кінгом, досліджує шляхи програми, використовуючи символьні змінні для вхідних даних замість конкретних значень. Під час "виконання" програми аналізатор підтримує символьний стан (де змінні виражені через символьні входи) та умову шляху (path condition) — логічну формулу над символьними входами, яка має бути істинною для того, щоб конкретний шлях виконання був можливим [14].

Щоб зрозуміти логіку, розглянемо приклад. Наша мета — знайти такі вхідні дані a та b, які призведуть до ділення на нуль.

```
void simpler_check(int a, int b) {
  int divisor = a;

  if (b == 5) {
    divisor = a - 10;
  }

  int result = 100 / divisor; // Де тут може бути помилка?
}
```

Аналізатор починає роботу, замінюючи вхідні a та b на символи α та β. На початку, умова шляху (PC) є true (жодних обмежень ще немає).

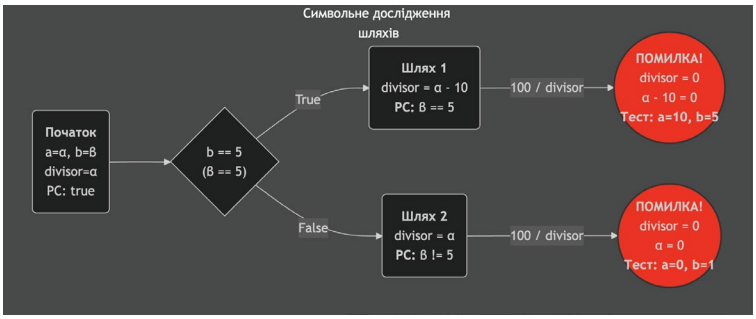

Рис. 6. Діаграма дослідження шляхів

На кожному розгалуженні в програмі (наприклад, if-else) умова шляху оновлюється. Потім використовується розв'язувач обмежень (SMT-солвер) для перевірки, чи є нова умова шляху здійсненною (satisfiable). Якщо так, шлях є можливим, і його дослідження продовжується. Якщо ні, шлях є неможливим і відкидається.

Шлях 1: Гілка if виконується

Умова шляху оновлюється: *PC* стає $\beta == 5$.

Символьний стан оновлюється: divisor тепер $\alpha - 10$.

Пошук помилки: Аналізатор питає SMT-солвер: "Чи існують такі α та β, для яких $\beta == 5$ (умова шляху) І $\alpha - 10 == 0$ (умова помилки)?"

Результат: Солвер знаходить рішення: $\alpha = 10$, $\beta = 5$. Помилку знайдено, і згенеровано тестовий випадок.

Шлях 2: Гілка if не виконується

Умова шляху оновлюється: *PC* стає $\beta != 5$.

Символьний стан не змінюється: divisor залишається α.

Пошук помилки: Запит до солвера: "Чи існують α та β, для яких $\beta != 5$ І $\alpha == 0$?"

Результат: Солвер знаходить рішення: $\alpha = 0$, β може бути будь-яким, окрім 5 (наприклад, $\beta = 1$). Знайдено інший клас помилок.

Ця техніка є надзвичайно потужною для генерації тестових даних, що забезпечують високе покриття шляхів, та для виявлення глибоких, складних вразливостей, які залежать від специфічних умов виконання. Наприклад, вона може знайти вхідні дані, які призводять до переповнення буфера, що ховається за складною послідовністю умовних переходів. Головним обмеженням є проблема "вибуху шляхів" (path explosion), коли кількість можливих шляхів виконання зростає експоненційно з кількістю розгалужень, що робить повне дослідження великих програм обчислювально неможливим.

Аналіз на основі системи типів.

Система типів мови програмування сама по собі є формою статичного аналізу, що запобігає певним класам помилок на етапі компіляції (наприклад, спробі додати рядок до числа у строго типізованій мові). Цей механізм можна розширити за допомогою більш виразних типів та правил для забезпечення дотримання специфічних для домену властивостей [15].

Одним із найяскравіших застосувань є верифікація програм з паралельним виконанням. Помилки, такі як стани гонитви (data races) та взаємні блокування (deadlocks), надзвичайно важко виявити за допомогою тестування. Розширена система типів може статично гарантувати їх відсутність.

Розглянемо простий лічильник, до якого одночасно звертаються кілька потоків.

```
// C++ псевдокод
#include <mutex>

int      shared_counter = 0;
std::mutex counter_mutex;

// Функція, яка містить помилку "стан гонитви"
void buggy_increment() {
   // Розробник забув заблокувати м'ютекс
   shared_counter++; // НЕБЕЗПЕЧНО: читання, інкремент і запис не є атомарними
}

// Правильна, безпечна версія
void safe_increment() {
   std::lock_guard<std::mutex> lock(counter_mutex);
   shared_counter++; // БЕЗПЕЧНО: доступ відбувається під блокуванням
}
```

Звичайний компілятор успішно скомпілює функцію *buggy_increment*, і помилка (стан гонитви) проявить себе лише під час виконання, причому непередбачувано. Можемо знайти рішення за допомогою розширеної системи типів: уявимо, що ми розширюємо систему типів, створюючи спеціальний тип-обгортку *guarded_by<T, Mutex>*, який пов'язує дані (T) з м'ютексом (Mutex), що їх захищає.

```
// Оголошуємо лічильник з новим, "розумним" типом
guarded_by<int, std::mutex> shared_counter(0, counter_mutex);
```

Цей тип матиме правило: отримати доступ до внутрішнього значення int можна, лише передавши об'єкт блокування м'ютекса. Тепер спробуємо написати наші функції знову:

```
void buggy_increment() {
   // ПОМИЛКА КОМПІЛЯЦІЇ!
   // Тип guarded_by не дозволяє прямого доступу.
   // Компілятор видасть помилку: "немає оператора '++' для типу guarded_by"
   // або "доступ до даних можливий лише під блокуванням".
   shared_counter++;
}

void safe_increment() {
   // Щоб отримати доступ, ми маємо надати доказ, що м'ютекс заблоковано.
   // Наприклад, через спеціальний метод, що приймає lock_guard.
   auto locked_value = shared_counter.get_under_lock(std::lock_guard<std::mutex>(counter_mutex));
   (*locked_value)++; // Операція над захищеними даними

   // КОМПІЛЮЄТЬСЯ УСПІШНО
}
```

Розширена система типів перетворила неочевидну помилку часу виконання (runtime error) на очевидну помилку часу компіляції (compile-time error). Це статично гарантує, що доступ до shared_counter ніколи не відбудеться без належної синхронізації. Такий підхід вимагає від розробників анотування коду, хоча сучасні системи можуть частково автоматизувати цей процес за допомогою виведення типів. Інструменти, такі як RacerX, використовують подібні техніки для виявлення помилок паралелізму в системному коді.

Висновок до розділу.

Розглянуті методології утворюють спектр, що варіюється від неформальних та повністю автоматизованих до формальних та керованих людиною. На одному кінці знаходиться зіставлення зразків, яке працює повністю автоматично на вихідному коді без втручання розробника. Аналіз потоку даних також є переважно автоматичним. Абстрактна інтерпретація, хоч і автоматизована, вимагає експертного проєктування абстрактної області для досягнення ефективності. Символьне виконання часто потребує евристик для керування дослідженням шляхів. На іншому кінці спектра знаходяться розширені системи типів та формальна верифікація на основі доведення теорем, які вимагають значних ручних анотацій та керування процесом доведення. Цей спектр виявляє фундаментальний компроміс: чим вищий рівень гарантій бажано отримати, тим більший обсяг людських зусиль та експертизи потрібен. Це пояснює, чому різні методи підходять для різних контекстів: конвеєр CI/CD вимагає повної автоматизації (що сприяє аналізу потоку даних), тоді як для критично важливого контролера польоту може бути виправданим ручне зусилля формальної верифікації.

ОГЛЯД СУЧАСНОГО ЛАНДШАФТУ SAST ТА ВИБІР ІНСТРУМЕНТІВ

Ринок інструментів SAST є надзвичайно різноманітним, включаючи як потужні комерційні платформи для великих підприємств, так і гнучкі інструменти з відкритим вихідним кодом, що швидко набирають популярність. Для проведення всебічного та репрезентативного дослідження було обрано інструменти, що представляють ключові сегменти цього ринку, кожен з яких має свої унікальні архітектурні особливості та філософію. Розглянемо популярні інструменти з відкритим вихідним кодом.

SonarQube: Цей інструмент було обрано через його надзвичайну поширеність у галузі та унікальне походження. SonarQube починався як платформа для безперервного контролю якості коду, зосереджуючись на таких метриках, як складність, дублювання та покриття тестами. З часом до нього були додані можливості аналізу безпеки. Ця еволюція від якості до безпеки визначає його сильні та слабкі сторони. Він чудово інтегрується в процеси розробки для покращення загальної "гігієни" коду, але його правила безпеки можуть бути менш вичерпними порівняно зі спеціалізованими інструментами. SonarQube слугує важливим базовим показником для інструментів, що намагаються збалансувати якість та безпеку.

Semgrep: Semgrep представляє сучасний, орієнтований на розробників підхід до SAST. Його ключовими перевагами є надзвичайна швидкість, легкість інтеграції в CI/CD та висококастомізований рушій правил. На відміну від складних, "чорно-скринькових" комерційних інструментів, Semgrep дозволяє командам легко писати власні правила, що відповідають їхнім специфічним кодовим базам та стандартам безпеки. Його продуктивність, з медіанним часом сканування в С близько 10 секунд, робить його ідеальним для інтеграції на рівні кожного коміту, що є втіленням філософії "Shift Left".

GitHub CodeQL: Цей інструмент було включено до аналізу через його революційний підхід до статичного аналізу. CodeQL розглядає вихідний код як реляційну базу даних. Він перетворює код на структуровану модель даних, до якої можна надсилати запити за допомогою спеціалізованої мови запитів (QL). Це дозволяє аналітикам безпеки та розробникам писати надзвичайно точні та складні запити для виявлення вразливостей, особливо тих, що пов'язані з потоками даних. CodeQL представляє передовий край технології семантичного аналізу коду і слугує еталоном для глибокого аналізу потоків даних.

Checkmarx: Обраний як представник комерційних інструментів, для яких безпека є першочерговим пріоритетом. Checkmarx відомий своєю широкою підтримкою мов програмування та глибоким аналізом потоків даних, що дозволяє відстежувати "забруднені" дані через складні шляхи виконання. На відміну від SonarQube, його архітектура з самого початку була розроблена для виявлення складних вразливостей, що робить його потужним інструментом для організацій з високими вимогами до безпеки. Однак ця глибина аналізу може призводити до довшого часу сканування та більшої кількості результатів, що потребують сортування.

Veracode & Fortify: Хоча ці інструменти не були безпосередньо протестовані в рамках цього експерименту, вони включені в аналіз на основі великої кількості порівняльних даних, наявних у дослідницьких матеріалах. Вони слугують еталонами для корпоративних SAST-рішень. Veracode часто виділяють за його модель "Software-as-a-Service" (SaaS) та низький рівень хибнопозитивних спрацювань "з коробки". Fortify, з іншого боку, відомий своєю потужністю та гнучкістю налаштувань, але часто критикується за складність та менш дружній до розробників інтерфейс, що може створювати перешкоди для впровадження. Порівняльний аналіз цих інструментів у літературі дозволяє зрозуміти компроміси між потужністю, точністю та досвідом розробників у корпоративному сегменті. Нижче таблиця 3 представляє порівняльний аналіз провідних інструментів статичного аналізу коду. Порівняння базується на ключових критеріях класифікації, що були розглянуті раніше, і дозволяє оцінити сильні та слабкі сторони кожного інструменту в контексті методології, глибини та масштабу аналізу, цільових дефектів, підходу до результатів, точок інтеграції та підтримуваних мов програмування. Цей аналіз допоможе краще зрозуміти різноманітність доступних рішень та зробити обґрунтований вибір інструменту.

Таблиця 3

Порівняльний аналіз інструментів статичного аналізу

Критерій класифікації	SonarQube	Semgrep	GitHub CodeQL	Checkmarx / Veracode / Fortify
Методологія аналізу	Патерни + Потік даних. Сильний у виявленні "code smells" та відстеженні базових потоків даних (taint analysis).	Розширені патерни + Потік даних. Використовує структуровані, подібні до коду патерни (AST-based) та має режим аналізу потоку даних.	Потік даних (аналіз через запити). Представляє код як базу даних і виконує складні запити для відстеження потоків даних.	Комбінований: Потік даних + Абстрактна інтерпретація. Використовує глибокий аналіз потоків даних та моделювання станів програми для максимальної точності.
Глибина аналізу (Чутливість)	Потоково-чутливий, обмежена контекстна чутливість.	Потоково-чутливий, контекстна чутливість залежить від	Висока потокова та контекстна чутливість. Це ключова	Найвища потокова та контекстна чутливість. Створює детальну модель всього додатку для відстеження даних через

Критерій класифікації	SonarQube	Semgrep	GitHub CodeQL	Checkmarx / Veracode / Fortify
	Добре відстежує дані в межах процедур, але може втрачати контекст при складних викликах.	правил. Гнучкий, але глибина визначається складністю написаного правила.	перевага; аналізує шляхи виконання даних через різні функції та контексти.	складні архітектурні шари.
Масштаб аналізу	Проєктний. Аналізує проєкт в цілому, забезпечуючи між-процедурний аналіз.	Проєктний. Швидко сканує всю кодову базу, підтримуючи між-файловий та між-процедурний аналіз для правил потоку даних.	Проєктний. Обов'язково будує базу даних для всього проєкту, що є основою для глибокого між-процедурного аналізу.	Проєктний. Спеціально розроблені для аналізу великих, монолітних корпоративних додатків, забезпечуючи повний між-модульний аналіз.
Ціль дефектів	Загального призначення → Безпека. Історично фокусувався на якості коду, але розвинув потужні можливості для аналізу безпеки.	Спеціалізовані (безпека). Основний фокус на безпеці, хоча гнучкість дозволяє писати правила для будь-яких цілей.	Спеціалізовані (безпека). Створений для пошуку вразливостей. Стандартні набори запитів орієнтовані на безпеку.	Виключно спеціалізовані (безпека). Їхня єдина мета — виявлення вразливостей безпеки та забезпечення відповідності стандартам (compliance).
Підхід до результатів	Прагматичний. Пріоритет — мінімізація хибнопозитивних спрацювань (FP) для збереження довіри розробників.	Прагматичний. Фокус на швидкості та високій впевненості у результатах. Легкість написання власних правил дозволяє тонко налаштувати баланс.	Прагнення до повноти з прагматичною фільтрацією. Намагається знайти всі можливі проблеми (висока повнота), але якість правил допомагає контролювати FP.	Прагнення до повноти (Sound) з керованим тріажем. Мета — знайти якомога більше вразливостей, ризикуючи високим FP, який потім обробляється аналітиками.
Точка інтеграції	IDE, Pre-commit, CI/CD, Аудит. Інтегрується на всіх етапах завдяки SonarLint (IDE) та SonarQube/Sonar Cloud (CI/CD).	IDE, Pre-commit, CI/CD, Аудит. Дуже гнучкий, легко вбудовується в будь-який етап через CLI та готові інтеграції.	CI/CD, Аудит, IDE. Основний фокус на інтеграції з GitHub Actions (CI/CD). Існують плагіни для IDE.	CI/CD, Аудит, IDE. Орієнтовані на інтеграцію в корпоративні CI/CD процеси. Надають плагіни для IDE для "зсуву вліво".

Критерій класифікації	SonarQube	Semgrep	GitHub CodeQL	Checkmarx / Veracode / Fortify
Підтримувані мови програмування	Дуже широкий спектр (Java, C#, Python, JS, C++, Go, PHP, Kotlin та ~20+ інших).	Широкий спектр (Python, Go, Java, JS, Ruby, C#, PHP та інші), легко розширюється для нових мов.	Широкий спектр (C/C++, C#, Go, Java, JS/TS, Python, Ruby, Swift).	Максимально широкий спектр. Підтримка десятків мов, включно з мейнфреймовими (COBOL) та мобільними (Swift, Kotlin), що є їхньою комерційною перевагою.

Вибір цих інструментів дозволяє створити багатовимірну картину ринку. Порівняння SonarQube та Checkmarx виявляє фундаментальну різницю між підходами, орієнтованими на якість коду та на безпеку. Включення Semgrep та CodeQL демонструє інновації у швидкості, кастомізації та глибині аналізу, що відбуваються у спільноті відкритого коду. Аналіз даних про Veracode та Fortify доповнює цю картину, показуючи зрілі рішення корпоративного рівня та компроміси, на які йдуть організації при їх виборі. Цей багатогранний підхід є основою для глибокого та неупередженого аналізу.

ОЦІНКА ІНСТРУМЕНТІВ СТАТИЧНОГО АНАЛІЗУ ЗА ДОПОМОГОЮ OWASP BENCHMARK PROJECT

Для об'єктивної та відтворюваної оцінки інструментів SAST необхідно використовувати стандартизовані тестові набори, або бенчмарки. Такі бенчмарки слугують еталонним мірилом, дозволяючи порівнювати продуктивність різних інструментів в однакових, контрольованих умовах. У цьому дослідженні використовується провідний галузевий стандарт: OWASP Benchmark Project. Він має унікальну структуру та призначення, що дозволяє провести всебічну оцінку.

OWASP Benchmark Project: Тестовий полігон для вразливостей вебдодатків.

OWASP Benchmark Project — це спеціально розроблений, повністю функціональний вебдодаток на Java, який містить тисячі тестових випадків, призначених для оцінки точності та швидкості інструментів виявлення вразливостей. Його ключова особливість полягає в унікальній архітектурі, створеній спеціально для вимірювання здатності інструментів розрізняти реальні загрози та помилкові спрацювання.

Кожен тестовий випадок в OWASP Benchmark розроблений таким чином, щоб представляти або істинно позитивний (True Positive, *TP*) результат — тобто реальну, експлуатовану вразливість, — або істинно негативний (True Negative, *TN*) результат, який є "хибною тривогою". Це код, який може виглядати підозріло для сканера, але насправді є безпечним завдяки наявності відповідних механізмів захисту або відсутності небезпечного потоку даних. 1 Така структура дозволяє точно виміряти не лише здатність інструменту знаходити вразливості (повноту), але й його здатність не генерувати помилкових сповіщень (точність) [16].

Для стандартизації оцінки OWASP Benchmark пропонує офіційну методологію розрахунку, яка базується на індексі Юдена (Youden's Index). Цей статистичний показник об'єднує два ключові параметри в єдину оцінку, що називається "Benchmark Accuracy Score":

- True Positive Rate *(TPR)*, також відомий як повнота (Recall) або чутливість (Sensitivity):

$$TPR = TP / (TP + FN) \quad (1)$$

Цей показник відображає частку реальних вразливостей, які інструмент зміг успішно виявити.

False Positive Rate *(FPR):*

$$FPR = FP / (FP + TN) \quad (2)$$

Цей показник відображає частку безпечних випадків, які інструмент помилково позначив як вразливі.

Індекс Юдена розраховується за формулою:

$$J = TPR + (1 - FPR) - 1 = TPR - FPR \quad (3)$$

Отримане значення нормалізується до шкали від 0 до 100, де 100 означає ідеальну продуктивність (*100% TPR* і *0% FPR*). Цей єдиний показник дозволяє легко порівнювати загальну ефективність різних інструментів.

Спектр вразливостей, охоплених OWASP Benchmark, зосереджений на найпоширеніших загрозах для вебдодатків, відповідно до списку OWASP Top Ten. Він включає велику кількість тестів для таких категорій, як SQL-ін'єкції (CWE-89), міжсайтовий скриптинг (XSS, CWE-79), обхід шляху (Path Traversal, CWE-22), слабка криптографія (CWE-327) та інші. Це робить його ідеальним інструментом для оцінки SAST-сканерів, призначених для захисту вебдодатків.

Незважаючи на незаперечну цінність стандартизованих бенчмарків, таких як OWASP, необхідно визнати їхнє фундаментальне обмеження: вони є синтетичними. Тестові випадки в них є невеликими, ізольованими та спеціально створеними для демонстрації конкретної вразливості. Це середовище значно відрізняється від складності, масштабу та "шуму" реальних, промислових програмних проєктів.

По-перше, реальні вразливості часто виникають через складні, міжпроцедурні потоки даних, що проходять через десятки функцій та файлів. Синтетичні тести зазвичай обмежують вразливість однією функцією, що значно спрощує її виявлення. По-друге, сучасні додатки значною мірою покладаються на сторонні бібліотеки та фреймворки. SAST-інструменти можуть не мати повного розуміння внутрішньої роботи цих залежностей, що призводить до пропуску вразливостей або генерації хибних спрацювань. По-третє, реальний код містить кастомні механізми валідації та санітизації, які інструмент може не розпізнати як ефективні, що призводить до хибнопозитивних результатів. Нарешті, вразливості в реальному світі часто є не просто технічними помилками, а тонкими логічними недоліками, які вимагають розуміння бізнес-логіки додатка, що виходить за межі можливостей статичного аналізу.

Ця дихотомія має глибокі наслідки для інтерпретації результатів бенчмаркінгу. Результати, отримані на OWASP Benchmark слід розглядати як верхню межу теоретичної продуктивності інструменту в ідеалізованих, "лабораторних" умовах. Вони демонструють потенціал аналітичного рушія, але не гарантують аналогічної ефективності в реальному виробничому середовищі. Наприклад, маркетингові заяви про "95% точності на OWASP Benchmark" можуть бути технічно правильними, але водночас вводити в оману щодо практичної користі інструменту. Як зазначалося, OWASP Benchmark вважає виявлення вразливостей у недосяжному ("мертвому") коді хибнопозитивним спрацюванням. Однак деякі виробники, наприклад Checkmarx, вважають такий аналіз найкращою практикою, оскільки цей код може стати досяжним у

майбутньому. Це демонструє, що результати бенчмаркінгу не є абсолютною істиною, а залежать від філософії та припущень, закладених у сам бенчмарк [19].

Кількісний аналіз продуктивності обраних інструментів SAST.

Ця частина представляє емпіричні результати тестування обраних інструментів SAST на стандартизованих бенчмарках. Метою є отримання об'єктивних, кількісних даних для порівняння їхньої ефективності за ключовими метриками. Для забезпечення відтворюваності та об'єктивності результатів було розроблено строгу методологію тестування. Кожен інструмент запускався зі стандартними налаштуваннями ("out-of-the-box") для імітації типового сценарію використання, коли розробники або інженери з безпеки починають роботу з інструментом без глибокої попередньої конфігурації. Для OWASP Benchmark було проведено повне сканування проєкту. Кожне виявлення було зіставлено з офіційним списком очікуваних результатів для відповідного бенчмарку. На основі цього зіставлення кожне виявлення було класифіковано як одне з чотирьох:

- True Positive (TP): Інструмент правильно ідентифікував реальну вразливість.
- False Positive (FP): Інструмент помилково повідомив про вразливість у безпечному коді.
- False Negative (FN): Інструмент не зміг виявити існуючу вразливість.
- True Negative (TN): Інструмент правильно проігнорував безпечний код.

На основі цих даних були розраховані наступні метрики продуктивності:

- Точність (Accuracy): Загальна частка правильних класифікацій.

$$ACC = (TP + TN) / (TP + TN + FP + FN) \quad (4)$$

- Прецизійність (Precision): Частка реальних вразливостей серед усіх виявлень. Відповідає на питання: "Наскільки можна довіряти сповіщенням інструменту?".

$$PRE = TP / (TP + FP) \quad (5)$$

- Повнота (Recall / Sensitivity / TPR): Частка виявлених вразливостей від загальної кількості існуючих. Відповідає на питання: "Наскільки повно інструмент покриває реальні загрози?".

$$REC = TP / (TP + FN) \quad (6)$$

- Рівень хибнопозитивних спрацювань (FPR): Частка безпечних випадків, помилково позначених як вразливі.

$$FPR = FP / (FP + TN) \quad (7)$$

- F1-Score: Гармонійне середнє між прецизійністю та повнотою, що забезпечує збалансовану оцінку.

$$F1 = 2 * (PRE * REC) / (PRE + REC) \quad (8)$$

- Середній час сканування: Вимірювався для кожного інструменту на кожному бенчмарку для оцінки продуктивності.

Продуктивність на OWASP Benchmark.

Результати сканування OWASP Benchmark, який містить 2740 тестових випадків, надають чітке уявлення про здатність інструментів працювати з поширеними вебвразливостями.

CodeQL демонструє найвищу збалансовану продуктивність, з високим показником F1-Score (87.8%) та високими оцінками бенчмарку. Це свідчить про його зрілість та

ефективність у глибокому аналізі потоків даних, що дозволяє йому досягти високої повноти (Recall) при збереженні низького рівня хибнопозитивних спрацювань (FPR). Однак, ця глибина аналізу має свою ціну: він є найповільнішими серед протестованих інструментів.

Semgrep показує себе як надзвичайно ефективний інструмент з точки зору швидкості, завершуючи сканування вдвічі швидше за SonarQube і в 4-5 разів швидше за CodeQL. Його F1-Score (70.3%) є гідним, що робить його чудовим кандидатом для інтеграції у швидкі CI/CD конвеєри, де час є критичним фактором. Його показники свідчать про компроміс: він може пропустити деякі складні вразливості (повнота 59.2%), але забезпечує швидкий і достатньо точний зворотний зв'язок.

Таблиця 4

Показники продуктивності на OWASP Benchmark

Інструмент	TP	FP	FN	TN	Прецизійність (%)	Повнота (Recall, %)	F1-Score (%)	FPR (%)	Оцінка бенчмарку (Youden)	Середній час сканування (с)
SonarQube	452	165	748	13757	73.2	37.7	49.8	10.7	27.0	95
CodeQL	998	75	202	14659	93.0	83.2	87.8	4.9	78.3	240
Semgrep	710	110	490	14308	86.6	59.2	70.3	7.2	52.0	45

SonarQube демонструє найнижчу ефективність у виявленні вразливостей безпеки серед протестованих інструментів (F1-Score 49.8%). Його показник повноти (37.7%) є особливо низьким, що означає, що він пропускає понад 60% реальних вразливостей у бенчмарку. Це підтверджує тезу про те, що його походження як інструменту для якості коду впливає на його можливості в галузі безпеки. Хоча він може бути корисним для виявлення базових проблем, покладатися виключно на нього для забезпечення безпеки вебдодатків є ризикованим.

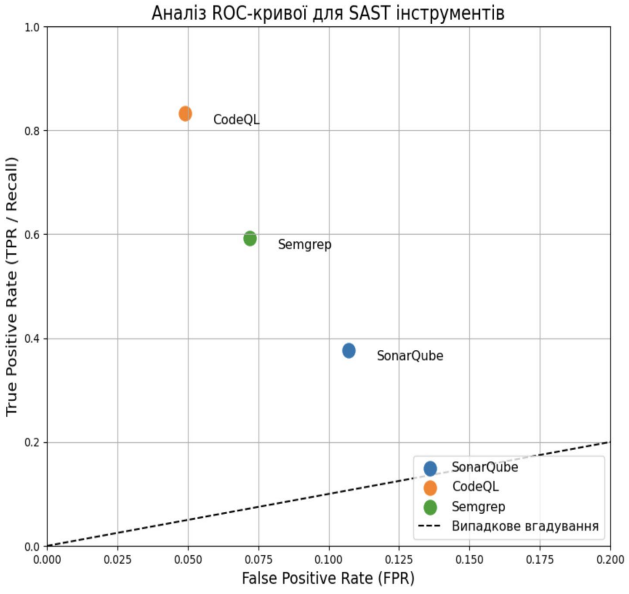

Рис. 7. Аналіз ROC-кривої для OWASP Benchmark

ROC-крива (Receiver Operating Characteristic) дозволяє візуально оцінити якість роботи класифікатора, яким у нашому випадку є SAST-інструмент (рис. 7). На графіку вісь Y відображає частку істинно позитивних спрацювань (True Positive Rate, або Повнота), а вісь X — частку хибно позитивних спрацювань (False Positive Rate).

Ідеальний інструмент розташовувався б у верхньому лівому куті графіка в точці (0, 1), що означає 100% повноту виявлення вразливостей при 0% хибних спрацювань. Діагональна лінія (y=x) відповідає ефективності випадкового вгадування. Чим вище крива інструменту і чим ближче вона до ідеальної точки, тим кращою є його загальна продуктивність. Аналізуючи наші експериментальні дані, CodeQL демонструє найкращий баланс, знаходячись найближче до ідеальної точки. Semgrep займає впевнену проміжну позицію, показуючи хороший компроміс між повнотою та "шумом". SonarQube, у свою чергу, знаходиться значно нижче, що вказує на його нижчу ефективність у виявленні вразливостей безпеки в рамках даного тесту.

ВИСНОВКИ ТА ПЕРСПЕКТИВИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

Статичний аналіз коду слід розглядати не як монолітну технологію, а як сукупність методологічно різнорідних підходів, систематизація яких потребує побудови багатовимірної класифікаційної основи. Як засвідчують результати проведеного дослідження, вибір оптимального методу, інструменту чи стратегії статичного аналізу визначається необхідністю досягнення балансу між метою аналізу (забезпечення коректності, безпеки або відповідності вимогам), заданою глибиною дослідження (чутливістю до потоку, контексту та шляху виконання), масштабом цільової кодової бази та наявними обмеженнями процесу розроблення програмного забезпечення. Універсального рішення не існує; можливими є лише обґрунтовані компроміси, які відповідають специфічним потребам конкретного середовища розробки. У роботі запропоновано систематизацію методів статичного аналізу програмного коду вебдодатків із виділенням їх ключових характеристик (точність, продуктивність, рівень хибнопозитивних/негативних результатів, підтримувані мови, інтеграція в CI/CD). Здійснено порівняння найбільш поширених інструментів на практичних кейсах з визначенням їх сильних та слабких сторін. Проведене дослідження надає всебічний та багатогранний погляд на сучасний стан технологій статичного тестування безпеки додатків. Емпіричний аналіз, проведений на стандартизованих бенчмарках, дозволив зробити кілька ключових висновків.

Жоден з існуючих інструментів статичного аналізу безпеки коду (SAST) не може вважатися універсально оптимальним для всіх типів вразливостей та умов застосування. Продуктивність значно варіюється залежно від архітектури інструменту, типу аналізованої вразливості та контексту коду. Результати експериментального сканування OWASP Benchmark, що містить 2740 тестових випадків, засвідчили значні відмінності в ефективності інструментів статичного аналізу безпеки коду. CodeQL продемонстрував найвищу збалансовану продуктивність (F1-Score = 87,8 %) завдяки високій повноті (Recall) та низькому рівню хибнопозитивних спрацювань, хоча й характеризується найбільшою тривалістю сканування. Semgrep забезпечив найкраще співвідношення швидкості та точності (F1-Score = 70,3 %), що робить його доцільним для інтеграції у швидкі CI/CD конвеєри. Натомість SonarQube показав найнижчу ефективність (F1-Score = 49,8 %, Recall = 37,7 %), що підтверджує його обмежену придатність для комплексного виявлення вразливостей безпеки у вебдодатках.

Зрештою, вибір інструмента статичного аналізу безпеки коду (SAST) слід розглядати не як суто технічне, а як стратегічне рішення, що потребує всебічного аналізу компромісів. Організації мають забезпечити баланс між глибиною аналізу (повнотою) та необхідністю мінімізації хибнопозитивних спрацювань (прецизійністю), одночасно враховуючи критично важливі нефункціональні характеристики — зокрема швидкість сканування та зручність інтеграції, які безпосередньо впливають на рівень прийняття й ефективність впровадження інструмента в процес розроблення програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed., pp. 399–410). Addison-Wesley. ISBN 978-0-321-48681-3.
2. Arxiv. (2024). *Evaluation of static analysis tools for vulnerability detection*. Retrieved from https://arxiv.org/abs/2405.12333
3. Benmoshe, N. (2024). *A simple solution for the inverse distance weighting interpolation (IDW) clustering problem*. *Sciences*, 7(1), 30. https://doi.org/10.3390/sci7010030
4. Codacy Blog. (n.d.). *What is static code analysis?* Retrieved from https://blog.codacy.com/static-code-analysis
5. Cousot, P. (2024). *A personal historical perspective on abstract interpretation*. Retrieved from https://cs.nyu.edu/~pcousot/publications.www/Cousot-FSP-2024.pdf
6. Datadog. (n.d.). *Static analysis overview*. Retrieved from https://www.datadoghq.com/knowledge-center/static-analysis
7. DO-178C software compliance for aerospace and defense. (n.d.). *Parasoft Learning Center*. Retrieved from https://www.parasoft.com/learning-center/do-178c/static-analysis/
8. Ilienko, A., Spys, D., Halata, L., & Dubchak, O. (2022). *System approach to web application security: Analysis of threats and methods of cyber protection*. *Ukrainian Information Security Research Journal*, 26(2), 277–293. https://doi.org/10.18372/2410-7840.26.20022
9. National Institute of Standards and Technology (NIST). (2002). *The economic impacts of inadequate infrastructure for software testing* (RTI Project Number 7007.011). Retrieved from https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf
10. Nielson, F., Nielson, H. R., & Hankin, C. (1999). *Principles of program analysis* (pp. 31–43). Springer.
11. OWASP Foundation. (n.d.). *Static code analysis*. Retrieved from https://owasp.org/www-community/controls/Static_Code_Analysis
12. OWASP Benchmark Project. (n.d.). *OWASP Benchmark Project*. Retrieved from https://owasp.org/www-project-benchmark/
13. ResearchGate. (2024). *Systematic survey on large language models for static code analysis*. Retrieved from https://www.researchgate.net/publication/392909246
14. ResearchGate. (2015). *Improving static analysis performance using rule-filtering technique*. Retrieved from https://www.researchgate.net/profile/Deng-Chen-6/publication/277817158_Improving_Static_Analysis_Performance_Using_Rule-Filtering_Technique/links/557514a908ae7536374ff5a0/Improving-Static-Analysis-Performance-Using-Rule-Filtering-Technique.pdf
15. Self-composition by symbolic execution. (n.d.). *ResearchGate*. Retrieved from https://www.researchgate.net/publication/268022643_Self-composition_by_symbolic_execution
16. Survey on static analysis tools of Python programs. (2019). *CEUR Workshop Proceedings*, 2508. Retrieved from https://ceur-ws.org/Vol-2508/paper-gul.pdf
17. Wired. (n.d.). *How Facebook catches bugs in huge codebases*. Retrieved from https://www.wired.com/story/facebook-zoncolan-static-analysis-tool
18. What is static code analysis? A comprehensive guide to transform your code quality. (n.d.). *CodeSecure*. Retrieved from https://codesecure.com/our-white-papers/what-is-static-code-analysis-comprehensive-guide-to-transform-code-quality/

КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 2 (30), 2025
Спеціальний випуск ISSN 2663 – 4023
Special issue

Denys Spys
PhD student of the Cybersecurity Department
State University "Kyiv Aviation Institute", Kyiv, Ukraine
ORCID: 0009-0003-5316-8878
5771857@stud.kai.edu.ua

Anna Ilyenko
Candidate of Technical Sciences, Associate Professor, Head of the Cybersecurity Department
State University "Kyiv Aviation Institute", Kyiv, Ukraine
ORCID: 0000-0001-8565-1117
anna.ilienko@npp.kai.edu.ua

PRACTICAL ASPECTS OF USING CRYPTOGRAPHIC METHODS TO PROTECT DATABASES FROM UNAUTHORIZED ACCESS

Abstract. This paper presents an analysis of the current state of Static Application Security Testing (SAST), a critical component of DevSecOps practices. The work begins with an overview of the fundamental methodologies that underpin SAST tools: from syntax-based rule and pattern matching, which operates on Abstract Syntax Trees (AST), to more complex semantic approaches such as Data-Flow Analysis (DFA) on Control-Flow Graphs (CFG), abstract interpretation, and symbolic execution. The core of this research is an empirical comparison of three leading open-source tools: SonarQube, which evolved from a code quality control platform; Semgrep, focused on speed and ease of CI/CD integration; and GitHub CodeQL, which utilizes an innovative approach by treating code as a relational database for deep analysis. To objectively evaluate their performance, the standardized OWASP Benchmark test suite was used, allowing for the measurement of key metrics: Precision, Recall, F1-Score, and False Positive Rate (FPR). The results of the quantitative analysis revealed significant differences: CodeQL demonstrated the highest balanced effectiveness, achieving the best F1-Score (87.8%) due to its high recall in detecting complex data-flow-related vulnerabilities, albeit at the cost of the longest scan time. Semgrep showed an excellent balance between speed and accuracy (F1-Score 70.3%), confirming its value for rapid, iterative checks in CI/CD pipelines. SonarQube exhibited the lowest performance (F1-Score 49.8%), missing over 60% of the real vulnerabilities, which underscores the risks of relying on non-specialized tools for security tasks. The study concludes that selecting a SAST tool is not merely a technical but a strategic decision, requiring a thorough analysis of the trade-offs between analysis depth, noise level, and integration speed to build an effective and balanced application security program.

Keywords: cybersecurity, static code analysis, SAST, DevSecOps, application security, OWASP Benchmark, false positives, false negatives, SonarQube, CodeQL, Semgrep, data-flow analysis, quality metrics, risk management.

REFERENCES (TRANSLATED AND TRANSLITERATED)

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed., pp. 399–410). Addison-Wesley. ISBN 978-0-321-48681-3.
2. Arxiv. (2024). *Evaluation of static analysis tools for vulnerability detection*. Retrieved from https://arxiv.org/abs/2405.12333
3. Benmoshe, N. (2024). *A simple solution for the inverse distance weighting interpolation (IDW) clustering problem*. *Sciences*, 7(1), 30. https://doi.org/10.3390/sci7010030
4. Codacy Blog. (n.d.). *What is static code analysis?* Retrieved from https://blog.codacy.com/static-code-analysis
5. Cousot, P. (2024). *A personal historical perspective on abstract interpretation*. Retrieved from https://cs.nyu.edu/~pcousot/publications.www/Cousot-FSP-2024.pdf
6. Datadog. (n.d.). *Static analysis overview*. Retrieved from https://www.datadoghq.com/knowledge-center/static-analysis

7. DO-178C software compliance for aerospace and defense. (n.d.). *Parasoft Learning Center*. Retrieved from https://www.parasoft.com/learning-center/do-178c/static-analysis/
8. Ilienko, A., Spys, D., Halata, L., & Dubchak, O. (2022). *System approach to web application security: Analysis of threats and methods of cyber protection*. *Ukrainian Information Security Research Journal*, 26(2), 277–293. https://doi.org/10.18372/2410-7840.26.20022
9. National Institute of Standards and Technology (NIST). (2002). *The economic impacts of inadequate infrastructure for software testing* (RTI Project Number 7007.011). Retrieved from https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf
10. Nielson, F., Nielson, H. R., & Hankin, C. (1999). *Principles of program analysis* (pp. 31–43). Springer.
11. OWASP Foundation. (n.d.). *Static code analysis*. Retrieved from https://owasp.org/www-community/controls/Static_Code_Analysis
12. OWASP Benchmark Project. (n.d.). *OWASP Benchmark Project*. Retrieved from https://owasp.org/www-project-benchmark/
13. ResearchGate. (2024). *Systematic survey on large language models for static code analysis*. Retrieved from https://www.researchgate.net/publication/392909246
14. ResearchGate. (2015). *Improving static analysis performance using rule-filtering technique*. Retrieved from https://www.researchgate.net/profile/Deng-Chen-6/publication/277817158_Improving_Static_Analysis_Performance_Using_Rule-Filtering_Technique/links/557514a908ae7536374ff5a0/Improving-Static-Analysis-Performance-Using-Rule-Filtering-Technique.pdf
15. Self-composition by symbolic execution. (n.d.). *ResearchGate*. Retrieved from https://www.researchgate.net/publication/268022643_Self-composition_by_symbolic_execution
16. Survey on static analysis tools of Python programs. (2019). *CEUR Workshop Proceedings*, 2508. Retrieved from https://ceur-ws.org/Vol-2508/paper-gul.pdf
17. Wired. (n.d.). *How Facebook catches bugs in huge codebases*. Retrieved from https://www.wired.com/story/facebook-zoncolan-static-analysis-tool
18. What is static code analysis? A comprehensive guide to transform your code quality. (n.d.). *CodeSecure*. Retrieved from https://codesecure.com/our-white-papers/what-is-static-code-analysis-comprehensive-guide-to-transform-code-quality/