InterConf
Scientific Publishing Center

ISSN 2709-4685

THE ISSUE CONTAINS:

Proceedings of the 12th International Scientific and Practical Conference

THEORY AND PRACTICE OF SCIENCE: KEY ASPECTS

Rome, Italy
19-20.11.2025

Scientific Collection
InterConf+

№ 63 (272)
November, 2025

OPEN ACCESS

GENERAL ENGINEERING AND MECHANICS

RADIO ENGINEERING, ELECTRONICS AND ELECTRICAL ENGINEERING

INFORMATION AND WEB TECHNOLOGIES

Proceedings of the 12th International Scientific and Practical Conference «Theory and Practice of Science: Key Aspects»

(November 19-20, 2025). Rome, Italy

No 272

INFORMATION AND WEB TECHNOLOGIES

DOI 10.51582/interconf.19-20.11.2024.032

Застосування методів статичного аналізу коду в JavaScript/TypeScript для покращення якості програмного забезпечення

Голубничий Дмитро Юрійович[1],

Легезін Нікіта Вікторович[2]

[1] кандидат технічних наук, доцент, доцент кафедри Інформаційних систем;
Харківський національний технічний університет радіоелектроніки; Україна
Харківський національний економічний університет імені Семена Кузнеця; Україна

[2] магістр кафедри ЕОМ;
Харківський національний технічний університет радіоелектроніки; Україна

Анотація.
У статті розглянуто застосування методів статичного аналізу коду в JavaScript/TypeScript для покращення якості програмного забезпечення. Статичний аналіз коду є ключовою практикою забезпечення якості програмного забезпечення, особливо в JavaScript та TypeScript. Використання відповідних інструментів дозволяє виявляти помилки на ранніх етапах розробки – під час написання коду або компіляції. Основні завдання статичного аналізу включають виявлення синтаксичних та типових помилок, перевірку відповідності стилю коду, аналіз безпеки, продуктивності та потенційних вразливостей. Інструменти, такі як ESLint, TSLint або SonarQube, допомагають розробникам дотримуватись найкращих практик та уникати дефектів. Об'єктом дослідження є процес статичного аналізу коду. Предметом дослідження є методи статичного аналізу коду. Результати. Проведено аналіз існуючих методів та інструментів статичного аналізу коду. Статичний аналіз дозволяє знаходити неефективні конструкції, можливі вразливості (наприклад, SQL-ін'єкції чи XSS-атаки) та генерувати звіти про стан коду, що сприяє своєчасному усуненню проблем без виконання програми. Помилки, допущені на ранніх етапах розробки, можуть суттєво впливати на якість та безпеку кінцевого продукту. Наукова новизна полягає у комплексному аналізі методів статичного аналізу в контексті JavaScript/TypeScript, що може привести до виявлення нових можливостей для оптимізації процесу.

Ключові слова:
CFA
CFG
DFA
TFA
static code analysis
ESLint
Prettier
SonarQube

Proceedings of the 12th International Scientific and Practical Conference «Theory and Practice of Science: Key Aspects»

(November 19-20, 2025). Rome, Italy

No 272

INFORMATION AND WEB TECHNOLOGIES

Застосування методів статичного аналізу коду в JavaScript/TypeScript дозволяє підвищити якість програмного забезпечення шляхом раннього виявлення помилок, забезпечення відповідності коду стандартам та покращення продуктивності розробників [1]. У порівнянні з динамічним аналізом, статичний аналіз не вимагає виконання коду, що дозволяє швидко та ефективно перевіряти великі проєкти на етапі розробки. Проте складність сучасних програмних систем, а також специфіка JavaScript та TypeScript як мов із динамічною та строгою типізацією відповідно, зумовлюють необхідність використання сучасних інструментів та методів статичного аналізу.

Аналіз літератури показує, що статичний аналіз коду є важливим етапом у забезпеченні якості програмного забезпечення [2]. Існують різноманітні інструменти для JavaScript та TypeScript, такі як ESLint, Prettier, SonarQube, а також статичні аналізатори коду, вбудовані у редактори, наприклад, Visual Studio Code [3]. Ці інструменти дозволяють автоматизувати процес перевірки синтаксису, виявлення потенційних помилок, типізацію та дотримання кодової стилістики. Використання таких засобів значно знижує кількість помилок у коді, підвищує його читабельність та полегшує подальшу підтримку. Крім того, інструменти статичного аналізу можуть виявляти уразливості у безпеці коду, такі як SQL-ін'єкції, XSS-атаки та інші загрози, що є критичними для веб-застосунків.

Розробка сучасних програмних систем передбачає такі етапи:

1) написання коду з використанням типових або кастомізованих правил перевірки;

2) перевірка синтаксису, виявлення вразливостей або потенційних помилок за допомогою статичного аналізу;

3) рефакторинг та оптимізація коду на основі отриманих результатів аналізу;

4) інтеграція статичного аналізу в процес CI/CD для забезпечення безперервного контролю якості;

5) перевірка ефективності аналізу та виявлення проблем на основі метрик якості коду, таких як кількість помилок, покриття тестами, когерентність та цикломатична складність.

У JavaScript статичний аналіз є особливо важливим через його динамічну природу, що дозволяє виконувати код без суворих

INFORMATION AND WEB TECHNOLOGIES

типів, що, у свою чергу, часто призводить до помилок на етапі виконання. TypeScript, завдяки вбудованій статичній типізації, додає додатковий рівень захисту, оскільки дозволяє виявляти помилки на рівні компіляції, ще до запуску програми. Водночас застосування статичного аналізу в TypeScript значно спрощує підтримку великих і складних проєктів, оскільки забезпечує кращу перевірку типів, виявлення дублювання коду та перевірку архітектурних обмежень.

У даній роботі основна увага приділена аналізу методів статичного аналізу коду і їх використання у інструментах для перевірки, визначенню їх переваг та обмежень, а також дослідженню ролі інструментів статичного аналізу у зменшенні технічного боргу. Результати дослідження дозволять сформулювати практичні рекомендації щодо вибору інструментів, поліпшення процесу статичного аналізу та покращення надійності програмних систем на основі JavaScript/TypeScript.

Статичний аналіз коду – це процес перевірки вихідного коду програми без його виконання, спрямований на виявлення помилок, порушень стандартів і вразливостей. Він дозволяє оцінити якість коду, аналізуючи його структуру, стиль, логіку та відповідність правилам програмування. Завдяки цьому методу можна виявити проблеми, як-от неправильна ініціалізація змінних, потенційні витоки даних або застарілі залежності.

Цей аналіз часто автоматизується за допомогою спеціальних інструментів, які перевіряють код за заданими критеріями. Він сприяє покращенню безпеки та надійності програмного забезпечення, дозволяючи розробникам швидко виправляти знайдені недоліки ще до виконання коду. Хоча статичний аналіз не гарантує виявлення всіх помилок, він є важливим етапом у забезпеченні якості програм і інтегрується в процеси розробки для автоматизації перевірок. Існує чи немала кількість методів статичного аналізу коду, кожен з яких відповідає за находження свого типу помилок або вразливостей.

Аналіз потоку керування. Починаючи з аналізу потоку керування (Control Flow Analysis або CFA) який є важливим методом статичного аналізу програмного коду, що зосереджується на дослідженні порядку виконання програмних інструкцій і механізмів, які визначають цей порядок. Основною метою цього методу є ідентифікація логічних помилок, недосяжних ділянок коду та аналіз впливу умов і розгалужень на загальну поведінку програми. Використання CFA дозволяє не

INFORMATION AND WEB TECHNOLOGIES

лише виявляти потенційні уразливості та оптимізувати структуру програми, а й значно покращувати її продуктивність та безпеку. Методологія CFA базується на побудові графа потоку керування (Control Flow Graph або CFG), який наочно представляє логіку виконання програми (рис.1) [4]. У цьому графі вузли відповідають базовим блокам інструкцій, а ребра – можливим переходам між ними. Така модель дозволяє ефективно виявляти помилки, такі як нескінченні цикли, надмірну складність розгалужень або фрагменти коду, що залишаються недосяжними. Аналіз умовних операторів і циклів є ключовим елементом CFA, оскільки саме ці компоненти визначають напрямок і характер виконання програми.

Значну увагу приділяють дослідженню умов, які визначають вихід із циклів, а також переходам між функціями. Цей підхід дозволяє глибоко зрозуміти структуру програми, визначити критичні точки та запропонувати шляхи оптимізації. Зокрема, CFA допомагає виявляти фрагменти, що ніколи не будуть виконані, що спрощує програму, знижує ризик помилок і покращує її читабельність.

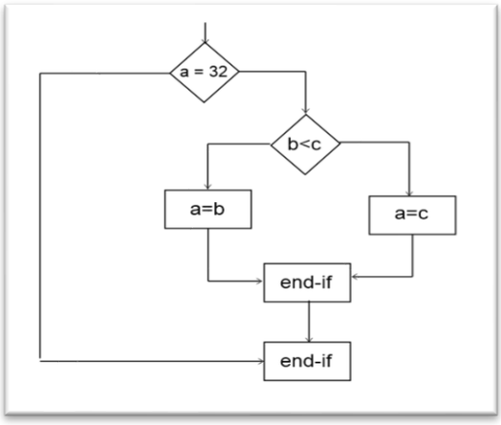

Рисунок 1
Приклад Control Flow Garph

Інтеграція CFA з іншими методами, такими як аналіз потоку даних (Data Flow Analysis, DFA), розширює можливості аналізу, враховуючи вплив змінних на вибір шляхів виконання. Наприклад, CFA використовує дані DFA для моделювання сценаріїв, у яких виконання програми залежить від значень

INFORMATION AND WEB TECHNOLOGIES

змінних. Такий підхід дозволяє враховувати більше контекстуальних аспектів роботи програми. Попри свої переваги, CFA має певні обмеження. Складність аналізу значно зростає зі збільшенням розміру та складності програми, що може призводити до надмірно консервативних результатів. Однак, навіть у таких випадках, CFA є корисним інструментом, оскільки дозволяє виявляти проблеми ще на етапі розробки, що знижує вартість виправлення помилок у майбутньому.

Аналіз потоку даних. Продовжуючи вже вище згаданим аналізом потоку даних (Data Flow Analys або DFA) ще одним із методів статичного аналізу, що дозволяє дослідити, як інформація переміщується через програму, забезпечуючи глибше розуміння її поведінки [4]. Цей підхід є основою для оптимізації, виявлення помилок та забезпечення безпеки коду, особливо в масштабних і складних проєктах. Метод data-flow аналізу складається з трьох основних етапів: спочатку йде парсинг коду, після цього створюється CFG і потім йде його аналіз.

Аналіз потоку даних ґрунтується на представленні програми у вигляді графа, де вузли символізують оператори, а ребра відображають залежності даних між ними [4]. Інформація про потік даних поширюється через цей граф за допомогою правил і математичних рівнянь, які дозволяють обчислювати значення змінних і виразів у кожній точці програми. Під час побудови графа дуже важливим є базова термінологія яка допомогає зрозуміти, як змінюються значення змінних і як ці зміни впливають на виконання програми (рис.2).

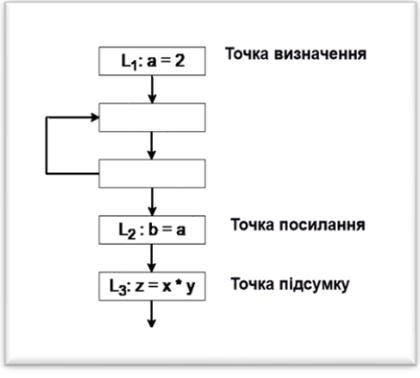

Рисунок 2

CFG для Аналізу потоку даних з представленням базової термінології

INFORMATION AND WEB TECHNOLOGIES

На рис. 2 зображено базовий граф для DFA де кожний термін має власне значення:

– Точка визначення – це місце в програмі, де певна змінна отримує значення. Наприклад, коли в коді є оператори присвоєння або ініціалізації змінної, ці точки є визначеннями.

– Опорна точка – це місце в програмі, де змінна або інший елемент даних використовується (посилається). Наприклад, виклики змінної у виразах чи функціях є опорними точками.

– Точка оцінки – це місце в програмі, де вирази обчислюються. Це може бути результат виконання математичних операцій, логічних умов або викликів функцій.

На рівні окремого блоку аналіз потоку даних описується за допомогою двох основних рівнянь (Формл.1), які моделюють стан змінних на вході та виході блоку.

$$\mathrm{out}_b = \mathrm{trans}_b = (\mathrm{in}_b) \quad (1)$$

$$\mathrm{in}_b = \mathrm{join}_b \in \mathrm{prev}_b = (\mathrm{out}_b) \quad (2)$$

Стан змінних на виході визначається функцією переходу, яка враховує зміни, внесені операторами всередині блоку. Стан змінних на вході обчислюється шляхом об’єднання вихідних станів попередніх блоків. Цей підхід забезпечує ітеративний аналіз, який триває до досягнення точки фіксації, коли стани змінних більше не змінюються.

На більш детальному рівні аналіз кожного оператора в блоці здійснюється з використанням рівняння, яке враховує множини створених даних, знищених даних та початкового стану змінних (Формл.2). Множина створених даних містить нові значення, які генерує оператор, тоді як множина знищених даних включає значення, що стають недійсними через виконання оператора. Початковий стан змінних визначає вхідний контекст для кожного оператора, а вихідний стан враховує вплив як створених, так і знищених даних.

$$\mathrm{Out}[s]=\mathrm{Gen}[s] \cup (\mathrm{In}[s] - \mathrm{Kill}[s]) \quad (3)$$

Цей підхід дозволяє формалізувати аналіз змін станів змінних у програмі та врахувати вплив кожного блоку й

INFORMATION AND WEB TECHNOLOGIES

оператора на їх поведінку. У результаті отримана модель може бути використана для оптимізації коду, наприклад, для усунення мертвого коду, виявлення неініціалізованих змінних або аналізу вразливостей, пов’язаних із неправильним використанням даних. Така модель також є основою для створення безпечного та ефективного програмного забезпечення, особливо у великих і складних проєктах.

Аналіз потоку даних має декілька типів аналізу, кожен з яких має специфічні цілі. Use-Def Analysis виявляє, де змінна використовується і як вона визначається. Це схоже на аналіз змінних у реальному часі, оскільки обидва методи дозволяють знайти "мертві" змінні, які більше не потрібні, або ті, що використовуються до їх ініціалізації, порушуючи правила коректності.

Def-Use Analysis доповнює Use-Def Analysis, відстежуючи, як визначення змінних впливає на подальший код. Цей підхід тісно пов'язаний з аналізом досяжності визначень, оскільки обидва методи досліджують, як визначення змінних поширюються на інші частини програми. Такий аналіз є важливим для виявлення некоректних присвоєнь або помилок у використанні значень. У свою чергу, Аналіз Антонімів (Antonyms Analysis) має більш специфічну мету – ідентифікацію суперечливого використання змінних, наприклад, коли одна змінна може одночасно мати протилежні значення. Це допомагає знаходити логічні помилки у програмному коді.

Аналіз може виконуватися у прямому порядку, коли інформація поширюється від початку програми до кінця, або у зворотному порядку, коли вивчається вплив майбутніх операцій на поточний стан змінних. Це забезпечує гнучкість і точність аналізу для різних типів програм.

Аналіз забруднення даних. Далі буде йти мова про Аналіз забруднення даних (Taint Analysis) – це метод виявлення потенційно небезпечних даних, які надходять із ненадійних джерел, наприклад, користувацького вводу або сторонніх API, і перевірки їхнього впливу на критичні ділянки програми [5]. Основна мета полягає у виявленні шляхів передачі таких даних через систему та перевірці, чи вони проходять належну обробку перед використанням у вразливих функціях, таких як SQL-запити, генерація HTML або виконання системних команд.

Процес починається з побудови CFG, який визначає можливі шляхи виконання. Потім виконується ідентифікація джерел

INFORMATION AND WEB TECHNOLOGIES

забруднених даних, яким присвоюється статус "tainted". Ці дані відслідковуються через усі проміжні елементи програми, поки вони не потраплять до приймачів – функцій або методів, які можуть бути уразливими. У випадку, якщо дані не проходять належну фільтрацію або екранування, вони вважаються небезпечними, і програма повідомляє про можливу загрозу. Очищення забруднених даних передбачає видалення або екранування небезпечних символів. Наприклад, спеціальні символи в HTML можуть бути перетворені на безпечні коди, щоб уникнути XSS-атак. Для SQL-запитів використовуються параметризовані запити, які запобігають ін'єкціям.

Символьне виконання (Symbolic Execution) – це метод статичного аналізу програмного коду, який дозволяє моделювати виконання програми без її фактичного запуску. Замість використання конкретних значень для вхідних даних застосовуються символічні вирази, які представляють множину можливих значень. Цей підхід дозволяє вивчати всі можливі шляхи виконання програми та аналізувати їх вплив на поведінку системи.

Основна ідея методу полягає у заміні реальних вхідних даних на символічні змінні. Умовні оператори в програмі аналізуються на основі цих змінних, що дозволяє визначати всі гілки виконання та умови, за яких вони активуються. Наприклад, для вхідної змінної "x" можна використовувати символічний вираз "α", що представляє всі можливі значення "x". Символьне виконання дозволяє відстежувати, які значення змінних впливають на логіку програми, і виявляти помилки, такі як недосяжний код, дублювання коду, недопустимі операції чи витоки пам’яті.

Метод складається з кількох етапів. Спочатку здійснюється побудова CFG, що описує всі можливі шляхи виконання програми. Потім програма виконується абстрактно, з використанням символічних значень. Для кожного шляху виконання формуються умови (path conditions), які визначають, чи можливо пройти цей шлях. Після завершення аналізу результати можуть бути використані для виявлення реальних помилок шляхом генерування тестових даних, які задовольняють знайдені умови.

Попри свою потужність, символьне виконання має певні обмеження. Зокрема, метод стикається із проблемою експоненційного зростання кількості шляхів виконання у

INFORMATION AND WEB TECHNOLOGIES

великих програмах або при складних умовах. Крім того, символьне виконання може бути обмежене при аналізі програм, які взаємодіють із зовнішніми ресурсами або містять недетерміновані фактори. Для вирішення цих проблем застосовуються різні евристики, такі як обмеження глибини аналізу або абстрактне моделювання зовнішніх бібліотек.

Символьне виконання активно використовується для аналізу безпеки, верифікації програмного забезпечення та виявлення помилок у критичних системах. Його застосування особливо важливе в проєктах, де висока надійність коду є ключовою вимогою, таких як фінансові системи, медичне обладнання чи аерокосмічні додатки. Цей метод допомагає забезпечити високу якість програмного забезпечення, дозволяючи глибоко досліджувати всі аспекти його поведінки.

Дивлячись на більшість з методів статичного аналізу коду, і навіть ті що не були розглянуті в даному досліджені, можна підкреслити те що всі вони пов’язані тим, що перед їх виконанням використовується CFG. Тож після цього зауваження можу запропонувати, створювати досить адаптивний CFG який буде підходити під кожен метод, щоб при аналізі зменшити кількість кроків для кожного методу, бо цей крок буде виконуватись лише 1 раз.

Інструменти статичного аналізу програмного коду є ключовими компонентами забезпечення якості програмного забезпечення. Вони виконують аналіз вихідного коду без його виконання, використовуючи методи статичного аналізу для виявлення потенційних помилок, вразливостей та недоліків у коді [1]. Основна мета таких інструментів полягає у покращенні якості коду, забезпеченні його відповідності стандартам і зниженні ризиків помилок на етапі розробки. Існує чи немала кількість інструментів, найпопулярніші з яких були розглянуті під час цього дослідження (табл.1). Такі інструменти використовуються як частина інтегрованих середовищ розробки (IDE) або як окремі сервіси, чи працюють локально як бібліотеки, або інтегровані в конвеєри CI/CD. Це дозволяє автоматизувати процес перевірки якості коду, зменшуючи ризик помилок і покращуючи ефективність розробки. Завдяки своїй здатності аналізувати великі кодові бази та виявляти складні проблеми, інструменти статичного аналізу стали невіддільною частиною сучасного програмного забезпечення.

INFORMATION AND WEB TECHNOLOGIES

Таблиця 1

Існуючі інструменти статичного аналізу

№	Інструменти статичного аналізу	Виявлені помилки та попередження	Докладність звітів	Автоматизація виправлень
1	StandardJS	Виявляє лише помилки у стилізуванні коду, та каже як треба робити. Немає власних налаштувань бо опирається на власний всезагальний стандарт.	У терміналі пише на якій строчці помилка помилка, але звіт у вигляді символів не досить зручний.	Присутня
2	JSHINT	Має великий спектр налаштувань, але в загалом виловлю лише невеликі помилки які найчастіше з'являються у розробників.	JHINT підкреслює проблемні місця і при наведенні вказує на помилку з рекомендацією як виправити, чи поясненням в чому проблема.	Відсутня
3	ESLINT	Має ще більше налаштувань, але автоматичне налаштування менш чутливіше ніж в jshint.	Так само підкреслює помилки з причиною та поясненням, але має при собі декілька варіантів виправлення.	Часткова
4	LGTM	LGMT створено для підтримки асинхронізації на всьому протязі. Він створений для перевірки об’єкта за раз і розуміє залежності між властивостями. Дуже гнучкий пристрій бо користувач сам може створювати валідатори для себе.	Він залишає рішення про те, як перевіряти, коли перевіряти та як відображати помилки у ваших руках, тобто користувач сам собі господар.	Відсутня
5	Sonar-qube	Найскладніша в використанні, але найкраща у реалізації. Sonar-qube аналізує весь код і детектує майже всі типи помилок, від логічних до вразливостей безпеки.	Так як вмикається локальний сервер, Детальність розпису помилок та зручність використання безумовно на вищому рівні.	Відсутня

INFORMATION AND WEB TECHNOLOGIES

Усі сучасні інструменти статичного аналізу коду базуються на застосуванні більшості відомих методів статичного аналізу, включаючи як описані раніше в цій роботі, так і ті, які залишилися поза її межами. Ці інструменти поєднують різні підходи для забезпечення комплексного аналізу, що дозволяє виявляти потенційні помилки, вразливості та недоліки в коді. Для перевірки ефективності таких інструментів було проведено емпіричне дослідження, спрямоване на оцінку їхньої здатності реагувати на типові помилки та проблеми у підготовленому програмному коді. Дослідження передбачало створення контрольного набору коду, що включав поширені помилки, такі як недосяжний код, неініціалізовані змінні, потенційно небезпечні конструкції, а також порушення стандартів стилю. Аналіз полягав у застосуванні різних інструментів статичного аналізу до цього набору для оцінки їхньої точності та глибини перевірки. Особливу увагу приділяли кількості виявлених помилок, їхньому типу та тому, наскільки детально інструменти описували проблеми. Окрім виявлення помилок, дослідження також аналізувало здатність інструментів надавати рекомендації щодо виправлення проблем. Було розглянуто якість цих рекомендацій, їхню практичну застосовність та відповідність стандартам індустрії. Окремо оцінювалися інструменти, які пропонували автоматичне виправлення певних типів помилок, що значно знижує навантаження на розробників і прискорює процес усунення недоліків. Результати дослідження підкреслили різноманітність підходів до аналізу, які використовуються різними інструментами, а також виявили їх сильні та слабкі сторони. Наприклад, деякі інструменти виявляли ширший спектр помилок, але надавали лише загальні рекомендації, тоді як інші спеціалізувалися на вузькому наборі проблем, пропонуючи детальні поради або автоматичні виправлення. Це свідчить про необхідність ретельного вибору інструментів залежно від потреб конкретного проєкту.

Висновки. Підсумовуючи, в даному досліджені було проаналізовано основні методи статичного аналізу коду, їх принцип роботи шляхи використання та ефективність у тих чи інших ситуаціях. Було запропоновано створення CFG адаптивного типу для зменшення кількості кроків при використанні декількох методів, що може потенційно зменшити час на виконання аналізу коду.

INFORMATION AND WEB TECHNOLOGIES

Окрім того було проведено аналіз існуючих інструментів статичного аналізу, де було розглянуто їх ефективність, вміння аналізувати та детектувати ті чи інші помилки та вміння самостійно їх виправляти.

References:

[1] Honfi D. Static analysis algorithms for JavaScript / D. Honfi, G. Szárnyas // Budapest University of Technology and Economics, 2017. – 114 p. [Електронний ресурс]. – Режим доступу: https://ftsrg.mit.bme.hu/thesis-works/pdfs/lucz-soma-bsc.pdf.

[2] De Silva D. et al. A Comparative Analysis of Static and Dynamic Code Analysis Techniques // Sri Lanka Institute of Information Technology, 2023. [Електронний ресурс]. – Режим доступу: https://d197for5662m48.cloudfront.net/documents/publicationstatus/170004/preprint_pdf/7a61e14555d04faac29aa331e761dca8.pdf.

[3] Brito T. et al. Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages / T. Brito //IEEE Transactions on Reliability. – Vol. № 72. – № 4, 2023. – Pp. 1324-1339. [Електронний ресурс]. – Режим доступу: https://ieeexplore.ieee.org/abstract/document/10168679/authors#authors

[4] Aldrich J. Program analysis / J. Aldrich, C. Le Goues // China Summer International Program, 2019. – 73 p. [Електронний ресурс]. – Режим доступу: https://www.cs.cmu.edu/~aldrich/courses/17-396/program-analysis.pdf.

[5] Marashdih A.W. An enhanced static taint analysis approach to detect input validation vulnerability / A.W. Marashdih, Z.F. Zaaba, K. Suwais // Journal of King Saud University-Computer and Information Sciences, 2023. – Pp. 682 – 701.

SCIENTIFIC EDITION

SCIENTIFIC COLLECTION «INTERCONF+»

№ 63(272) | November 2025

The issue contains:

Proceedings of the 12th International Scientific and Practical Conference

THEORY AND PRACTICE OF SCIENCE: KEY ASPECTS

Rome, Italy
19-20.11.2025

All materials are reviewed.
The editorial office did not always agree with the position of authors.

Journal's frequency: monthly

Sighed for online publication: November 20, 2025.

Printed: December 19, 2025. Circulation: 200 copies. Format 60×84/8.
Batang & Courier New typefaces. Offset paper 100gsm. Digital color printing.

Contacts of the editorial office:

LLC Scientific Publishing Center «InterConf»
info@interconf.center
https://www.interconf.center

Certificate on the entry of publishing business subject in the State Register of Publishers, Manufacturers and Distributors of Publishing Products of Ukraine: ДК № 7882 of 10.07.2023.