МАГІСТЕРСЬКА РОБОТА

МР.КІ-04.00.00.000 ПЗ

Група КІм-24-1

Галій Іван

2025

Міністерство освіти і науки України

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра комп'ютерних систем і мереж

Галій Іван Андрійович
(прізвище, ім'я, по батькові)

УДК 004.82
(індекс)

МАГІСТЕРСЬКА РОБОТА

Тема: *Удосконалення алгоритму виявлення шкідливих сценаріїв JavaScript на основі дерева рішень*
(назва роботи)

Комп'ютерна інженерія
(назва освітньої програми)

123 – комп'ютерна інженерія
(шифр і назва спеціальності)

Галій І. А.
(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник *Заячук Ярослав Іванович, к.т.н., доцент*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту
Завідувач кафедри

проф. *С. І. Мельничук*
(посада) (підпис) (дата) (ініціали та прізвище)

Рецензент

_____
(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Інститут *Інформаційних технологій*

Кафедра *Комп'ютерних систем та мереж*

Спеціальність *123 – Комп'ютерна інженерія*

Освітній рівень *магістр*

ЗАТВЕРДЖУЮ:

Зав. кафедрою КСМ

проф. С. І. Мельничук

“ ____ ” _____ 2025 р.

ЗАВДАННЯ
НА ВИКОНАННЯ МАГІСТЕРСЬКОЇ РОБОТИ СТУДЕНТОВІ

Галію Івану Андрійовичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи ***“Удосконалення алгоритму виявлення шкідливих сценаріїв JavaScript на основі дерева рішень”***

Керівник проекту *Заячук Ярослав Іванович*

затверджена наказом по університету від №

2. Термін здачі студентом закінченої роботи *10.12.25*

3. Вихідні дані до проекту (роботи) *Матеріали та результати, отримані під час проходження переддипломної практики, методичні вказівки, технічна література*

4. Зміст розрахунково - пояснювальної записки (перелік питань, що їх належить розробити)

Аналіз проблем безпеки та вразливостей javascript

Порівняння алгоритмів захисту від шкідливих javascript сценаріїв

Реалізація алгоритму виявлення шкідливих javascript сценаріїв

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

6. Консультанти по дипломній роботі, із зазначенням розділів роботи, що стосуються їх

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
нормоконтроль	***О. В. Мойсеєнко***		

7. Дата видачі завдання _*12.03.2025*_____

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів бакалаврської роботи	Термін виконання етапів роботи	Примітка
1	*Аналіз проблем безпеки та вразливостей javascript*	*12.03.2025-31.05.2025*	*Виконано*
2	*Порівняння алгоритмів захисту від шкідливих javascript сценаріїв*	*01.06.2025-31.07.2025*	*Виконано*
3	*Реалізація та тестування удосконаленого алгоритму*	*01.08.2025-30.09.2025*	*Виконано*
4	*Визначення точності модифікованого алгоритму*	*01.10.2025-15.11.2025*	*Виконано*
5	*Оформлення пояснювальної записки*	*16.11.2025-10.12.2025*	*Виконано*

Студент-магістр _____ ***Галій І. А.***
(підпис)

Керівник роботи _____ ***Заячук Я. І.***
(підпис)

АНОТАЦІЯ

У роботі проаналізовано підходи до виявлення кіберзагроз у режимі реального часу, а також методи моніторингу та реагування на них за допомогою JavaScript і вбудованих засобів захисту. Дослідження демонструє способи підвищення рівня безпеки вебзастосунків через використання можливостей цієї мови програмування та сучасних інструментів фронтенд- і бекенд-захисту. Окрему увагу приділено технікам оперативного визначення підозрілої активності та їх практичному використанню в системах, створених на основі JavaScript-технологій.

Показано приклади безпечних принципів розробки, методів аналізу поведінки користувачів, перехоплення аномальних подій і механізмів автоматичного реагування на потенційні атаки.

ВИЯВЛЕННЯ, ВЕБДОДАТКИ, МОНІТОРИНГ, JAVASCRIPT

SUMMARY

The paper analyzes approaches to real-time cyber threat detection, as well as methods of monitoring and responding to such threats using JavaScript and built-in security mechanisms. The study demonstrates ways to enhance the security level of web applications through the use of this programming language's capabilities and modern frontend and backend protection tools. Special attention is given to techniques for promptly identifying suspicious activity and their practical application in systems built on JavaScript technologies.

Examples of secure development principles, methods for analyzing user behavior, intercepting anomalous events, and mechanisms for automatic response to potential attacks are presented.

DETECTION, WEB APPLICATIONS, MONITORING, JAVASCRIPT

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AWS – Amazon Web Services

CDN – Content Delivery Network

CORS – Cross-Origin Resource Sharing

CSP – Content Security Policy

CSRF – Cross-Site Request Forgery

CSS – Cascading Style Sheets

DOM – Document Object Model

ES – ECMAScript

HTML – HyperText Markup Language

HTTP – Hypertext Transfer Protocol

HTTPS – Hypertext Transfer Protocol Secure

JIT – Just-in-time compilation

JS – JavaScript

JWT – JSON Web Token

NPM – Node Package Manager

OWASP – Open Worldwide Application Security Project

SOP – Standard Operating Procedure

SRI – Subresource Integrity

SVM – Support Vector Machine

RBF – Radial Basis Function

XSS – Cross-Site Scripting

ОС – операційна система

ПЗ – програмне забезпечення

ВСТУП

Актуальність теми дослідження. JavaScript – це гнучка та динамічна клієнтська мова програмування, яку використовують для формування інтерактивного контенту в Інтернеті разом із HTML та CSS. Вона присутня на переважній частині сучасних веб-ресурсів і сумісна з практично всіма актуальними браузерами. Ця технологія активно застосовується під час створення інтерактивних сторінок, однак останнім часом вона також перетворилася на один із найпоширеніших інструментів для проведення кібератак. Шкідливі фрагменти JavaScript можуть бути інтегровані у сторінки та запускатися під час завантаження, інколи обходячи захисні системи, серед яких антивіруси чи мережеві фільтри. Зловмисники часто змінюють код на різних платформах, змушуючи його виконувати небезпечні інструкції. Через високу динамічність і гнучкість мови некоректні реалізації чи помилки розробників створюють додаткові можливості для атак. Коли користувач відкриває веб-ресурс, JavaScript-файли завантажуються автоматично, що додає ще один рівень ризику.

Нові загрози виникають постійно, тому компанії, розробники та звичайні користувачі повинні посилювати підходи до кіберзахисту.

Метою роботи є створення вдосконаленого методу виявлення небезпечних JavaScript-сценаріїв.

Об’єктом дослідження є процес захищення користувачів від шкідливих сценаріїв на JavaScript.

Предметом дослідження є покращений алгоритм, спрямований на підвищення ефективності виявлення шкідливих сценаріїв на JavaScript.

Методи дослідження. У процесі виконання роботи застосовувалися аналітичні та математичні методи досліджень.

Наукова новизна: отримав подальший розвиток алгоритм виявлення загроз на основі дерева рішень за рахунок розроблення спеціального набору функцій і

введення системи оцінювання шкідливості коду, що забезпечує значно точнішу класифікацію підозрілих сценаріїв.

Практичне значення: можливість подальшої інтеграції запропонованого рішення в системи аналізу трафіку, інструменти статичного аналізу та модулі автоматизованого моніторингу безпеки

Структура і обсяг роботи. Магістерська робота складається зі вступу, 3 розділів, висновків і списку літератури, що включає 33 найменування. Основна частина роботи викладена на 57 сторінках машинописного тексту. Робота містить 12 рисунків, 8 таблиць та 2 додаток на 6 сторінках.

1 АНАЛІЗ ПРОБЛЕМ БЕЗПЕКИ ТА ВРАЗЛИВОСТЕЙ JAVASCRIPT

1.1 Загальні відомості про JavaScript

JavaScript (JS) – це інтерпретована, подієва та прототипно-орієнтована мова сценаріїв, яку найчастіше застосовують у веббраузерах для створення інтерактивних інтерфейсів. Разом із HTML та CSS вона формує основу сучасної фронтенд-розробки та залишається однією з найвпливовіших технологій мережі. Спершу мову називали LiveScript, що відображало її первинне призначення: «оживляти» вебсторінки, робити їх динамічними та здатними реагувати на дії користувача [1].

За останні десятиліття JavaScript перетворився із простої клієнтської мови на універсальний інструмент, який використовується і у фронтенді, і в бекенді, і в мобільній розробці, і навіть у створенні десктопних програм. У 2025 році екосистема JS включає тисячі бібліотек, десятки фреймворків та потужні платформи для виконання коду, такі як Node.js, Deno та Bun, які значно розширили сферу застосування мови.

Опитування українських IT-фахівців, проведене DOU у 2025 році, показало, що JavaScript продовжує утримувати провідні позиції, хоча TypeScript поступово виходить на перше місце завдяки вбудованій підтримці типів [2,3,4]. Популярність JS в Україні залишається стабільною, адже велика частина вебпроєктів та стартапів використовують саме стек JavaScript/TypeScript у поєднанні з React, Vue або Angular (рис. 1.1).

У світовому масштабі JavaScript стабільно входить до топу найпоширеніших мов за даними професійних опитувань [5]. Під час глобального дослідження Stack Overflow 2024–2025 років JS знову очолив рейтинги найбільш уживаних мов, підтвердивши свою універсальність у веброзробці, автоматизації, створенні інтерфейсів, роботизації та інтерактивних вебдодатків (рис.1.2).

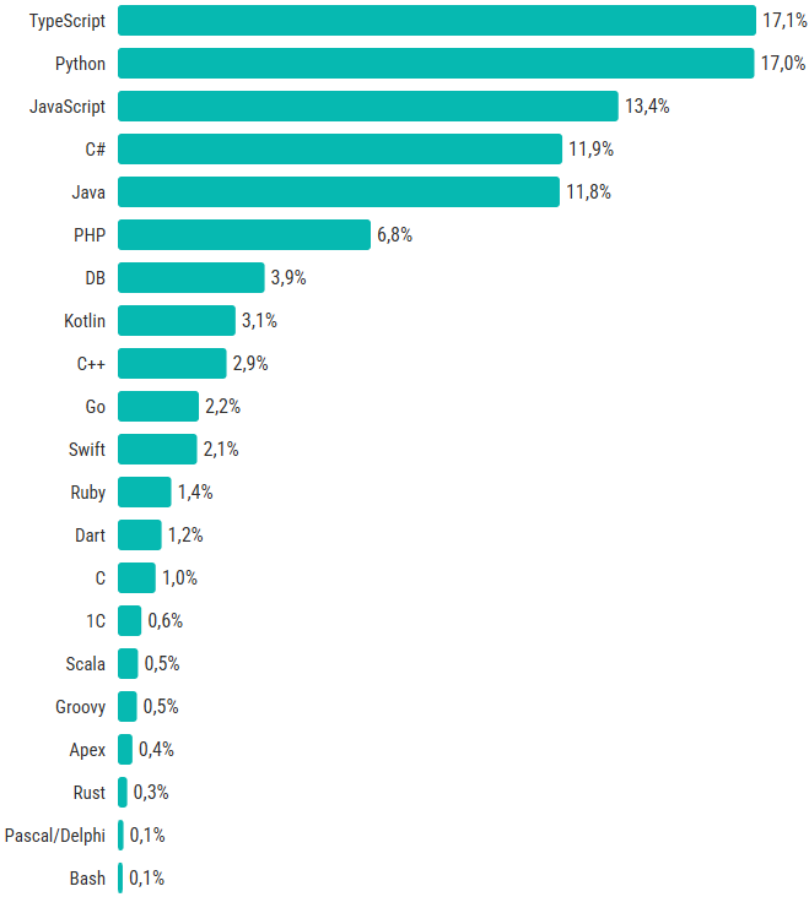

Рисунок 1.1 – Рейтинг мов програмування 2025 в Україні

Завдяки JS вебсторінки можуть включати такі функції, як анімації, інтерактивні меню, перевірку коректності даних у формах, відкриття додаткових вікон, виявлення характеристик браузера, адаптацію інтерфейсу під пристрій та багато інших можливостей. Усі ці дії виконуються без перезавантаження сторінки, що робить взаємодію з вебом значно зручнішою.

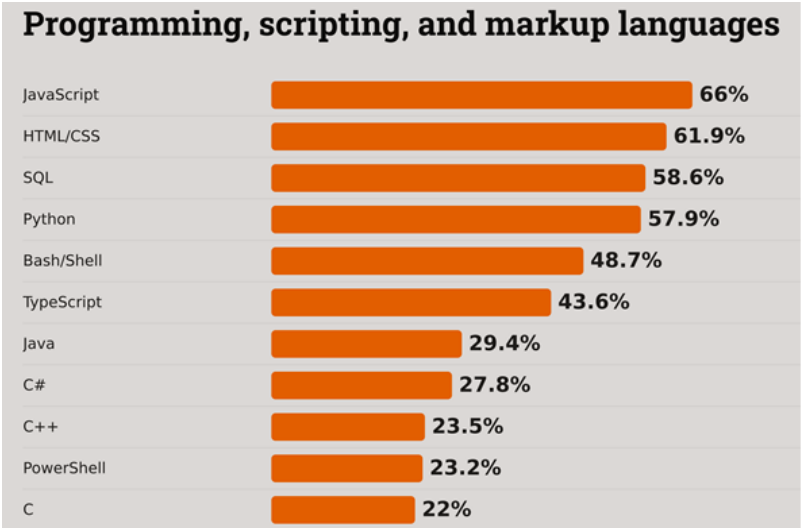

Рисунок 1.2 – Рейтинг мов програмування 2025 на Stackoverflow

Важливо зазначити, що JavaScript і Java, попри схожу назву, не мають спільної архітектури. Java є компільованою об’єктно-орієнтованою мовою, тоді як JS – сценарна та динамічна. Синтаксична схожість між ними виникла лише через маркетингові рішення, а не через спільне походження.

Для точнішого розуміння принципів роботи JS варто виділити кілька чинників [1]:

- інтерпретація – браузер виконує JS-код у момент його завантаження, без попередньої компіляції. Це дає змогу швидко тестувати та змінювати програму, хоча й покладає відповідальність за оптимізацію на рушій браузера;

- взаємодія з DOM – вбудована клієнтська версія JS дозволяє керувати структурою HTML-документа, реагувати на події, змінювати стильове оформлення та створювати складні механізми взаємодії з користувачем;

- подієво-орієнтована модель – код запускається лише після виникнення конкретної події: кліку, наведення, надсилання форми, прокручування сторінки, завантаження ресурсу або зміни стану елемента;

- об’єктність - мова підтримує прототипне наслідування, а з появою класів (починаючи з ES6) стала ще зручнішою для розробників, які звикли до класичної об’єктно-орієнтованої моделі.

Інтерпретаторами JavaScript є рушії браузерів. Серед найпоширеніших JS-движків 2025 року [6]:

- V8 – використовується в Google Chrome, Microsoft Edge, Node.js, Deno;

- SpiderMonkey – рушій Firefox;

- JavaScriptCore (Nitro) – рушій WebKit/Safari;

- Bun Engine – високопродуктивний рушій нового покоління, інтегрований у платформу Bun.

Ці рушії не просто виконують код, а проводять складні оптимізації, збирають сміття, будують абстрактні синтаксичні дерева та застосовують JIT-компіляцію для пришвидшення роботи.

JavaScript відповідає стандарту ECMAScript (ES). Документ ECMA-262 визначає синтаксис, можливості мови, логічні структури, роботу з об’єктами та

механізми виконання. Кожна нова версія специфікації додає покращення: синтаксичні скорочення, нові методи, конструкції для роботи з асинхронністю, оптимізацію класів та інші можливості [7].

Варто пам’ятати, що оновлення стандарту не означає миттєвої підтримки в усіх браузерах. Рушії впроваджують нові можливості поступово, тому перед використанням окремих функцій важливо перевіряти їхню сумісність через такі інструменти, як caniuse.com або MDN Compatibility Data.

Починаючи з 2015 року ECMAScript публікує нові версії щороку [8]. На 2025 рік актуальними є функції, додані в ES2022–ES2024, зокрема:

- покращена робота з асинхронністю (async iterators, top-level await);
- приватні поля та методи класів;
- оператори ??, ?. та інші зручні конструкції;
- оптимізовані структури даних;
- розширена підтримка модулів у різних середовищах.

Версія ES2025 перебуває у процесі затвердження, і деякі її пропозиції вже частково реалізовані в рушіях найновіших браузерів [9].

Браузери оновлюються не одночасно, а користувачі нерідко залишаються на старих версіях, тому впровадження нових можливостей може затягуватися. Саме тому популярними залишаються інструменти на кшталт Babel або TypeScript, які компілюють сучасний код у версії, сумісні з ширшим спектром пристроїв.

Середовище виконання JavaScript – це об’єкт або система, яка забезпечує виконання сценарної мови. Код проходить через JS-рушій, де об’єкт або система аналізує його, розбирає логіку та виконує інтерпретовані команди. Середовище надає засоби взаємодії з іншими об’єктами, дозволяючи сценаріям працювати не тільки з внутрішнім кодом, але й з зовнішніми додатками або ресурсами [1].

На стороні клієнта середовище JS – це веб-браузер. Він відкриває доступ до багатьох хост-об’єктів для маніпуляцій, таких як вікна, HTML-документи, CSSOM, Canvas, WebGL, WebGPU, Web Audio API, WebRTC, IndexedDB та інші веб-API. Браузер виступає як хост-середовище, надаючи JavaScript можливості управління сторінкою, комунікації з сервером і створення інтерактивних додатків.

На сервері JavaScript виконується через Node.js, платформу на основі V8. Node.js перетворює JS з вузькоспеціалізованої мови в універсальну платформу для серверної розробки. Він надає API для роботи з файлами, мережевими запитами, потоками, базами даних, криптографією та пристроями введення-виведення. Node.js підтримує підключення сторонніх бібліотек через npm, а також дозволяє будувати десктопні додатки за допомогою Electron, Tauri або NW.js на Windows, macOS та Linux. Сценарії Node.js можуть керувати мікроконтролерами та IoT-пристроями, такими як Espruino, Tessel або сучасні Arduino- та Raspberry Pi-платформи.

Різні середовища можуть використовувати один і той же JavaScript-рушій. Наприклад, V8 застосовується у Chrome, Edge, Node.js і деяких серверних платформах, забезпечуючи високу продуктивність і стабільність. Це дозволяє створювати код, який працює в браузері і на сервері без серйозних змін.

Бібліотека JavaScript – це набір повторно використовуваних фрагментів коду, що включає функції, об’єкти та класи для швидкого використання в додатках. Вона абстрагує складні операції і дозволяє викликати функцію з параметрами та отримувати результат, не заглиблюючись у реалізацію [10].

Сучасні дослідження 2025 року показують широке використання JS-бібліотек на веб-сайтах. Було проаналізовано топ-100 тисяч сайтів і ресурсів. Виявлено, що близько 70% сайтів застосовують хоча б одну відому бібліотеку [11]. Як і раніше продовжує домінувати jQuery [12], також популярні сучасні бібліотеки та фреймворки: Bootstrap, Underscore, React, Modernizr та нові модульні інструменти для фронтенду (рис. 1.3).

JavaScript має багато переваг, зокрема завдяки виконанню на клієнтській стороні, але сучасні підходи дозволяють реалізовувати повноцінні серверні рішення. Основні переваги JS [14]:

- швидкість – код виконується в браузері або через Node.js без попередньої компіляції. Just-In-Time-компіляція та оптимізації рушіїв прискорюють виконання навіть складних алгоритмів;

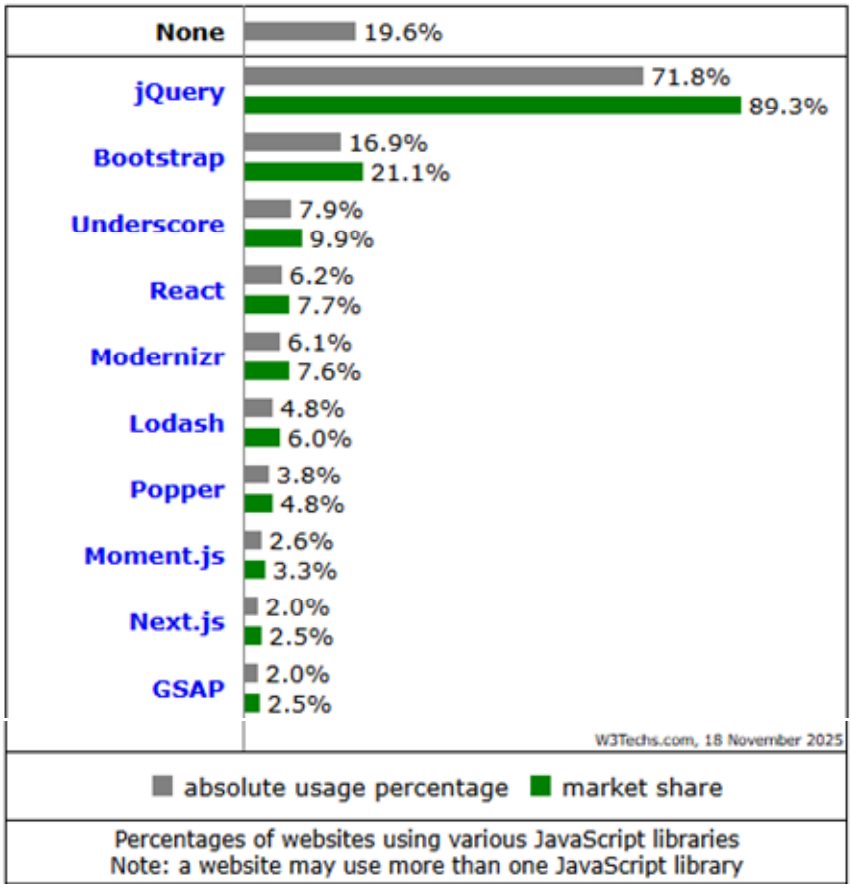

Рисунок 1.3 – Статистика використання JavaScript бібліотек сайтами

- простота – синтаксис JavaScript легкий для освоєння, нагадує Java, і має просту систему типів у поєднанні з новітніми стандартами TypeScript, що забезпечує безпечне масштабування проектів;

- популярність – JS застосовується у вебі, мобільних та серверних додатках. Спільноти на GitHub і StackOverflow постійно розширюються, а кількість проектів і бібліотек зростає щороку;

- взаємодія – мову можна легко інтегрувати з іншими технологіями: HTML, CSS, PHP, Python, WebAssembly. Вона сумісна з більшістю веб-платформ і сторонніх сервісів;

- зменшене навантаження на сервер – виконання логіки на клієнті знижує потребу у потужному серверному обладнанні.

- розширюваність – можливе створення власних модулів, надбудов та пакетів для будь-яких задач: від графіки до роботи з базами даних і потоковою обробкою;

- універсальність – Node.js дозволяє будувати повноцінні додатки від фронтенду до бекенду, використовуючи лише JS у поєднанні з Express, NestJS, MongoDB, PostgreSQL або Redis;

- регулярні оновлення – ECMA International підтримує стандарт щороку, останній – ES2025, який додає нові можливості, оптимізації та покращену безпеку.

Сучасні веб-API 2025 року, такі як WebGPU, WebAuthn, WebXR та офлайн-сховища (IndexedDB, Cache API), розширюють можливості JS для створення ігор, VR/AR-додатків, безпечної автентифікації та офлайн-працездатних веб-додатків. Node.js і npm-бібліотеки дозволяють інтегрувати ML-моделі, робити обробку потоків даних, керувати IoT-пристроями та створювати високонавантажені серверні системи.

Завдяки цьому JavaScript залишається однією з найгнучкіших, швидких і універсальних мов для веб- та серверної розробки в 2025 році.

Проте варто відмітити також і недоліки.

Хоча сучасні рушії стандартизовані, деякі функції можуть по-різному працювати у старих версіях браузерів. Завдяки регулярним оновленням і сучасним движкам відмінності мінімальні.

Код виконується локально, на стороні клієнта, тому помилки чи уразливості можуть використовуватися зловмисниками. Це головна причина, чому деякі організації обмежують або відключають JS.

Сьогодні JavaScript є однією з ключових технологій цифрового світу, а розвиток вебу, AI-інтерфейсів, мікросервісної архітектури та хмарних платформ лише збільшує його значення. JS продовжує еволюціонувати, формуючи основу інтерактивного та високопродуктивного Інтернету.

1.2 Вразливості і безпека JavaScript

JavaScript-бібліотеки можуть стати об’єктом злочинної модифікації, і таке втручання створює умови для виконання шкідливих сценаріїв, здатних фіксувати дії користувачів у мережі та порушувати їхню приватність. Навіть веб-ресурс, який здається захищеним, може бути скомпрометований через вразливість у сторонній JS-бібліотеці, інтегрованій розробником у сайт. Нижче описано три типові способи зловмисного використання JavaScript.

1. Перехоплення введення та дій користувача. Зловмисні скрипти здатні стежити за текстовими інпутами, у які користувачі вводять повідомлення, коментарі або інші дані, використовуючи невеликий фрагмент вбудованого JS-коду. У 2012 році двоє дослідників зібрали статистику від близько п’яти мільйонів користувачів Facebook у Великій Британії та США, фіксуючи лише факт наявності введення, а не сам вміст чи натискання клавіш [14,15]. Проте технічно вони могли відстежувати й реальний текст, але відмовилися через законодавчі обмеження – на відміну від кіберзлочинців, які цим не переймаються. Невеликий JS-код дозволяє зберігати будь-які події на сторінці незалежно від того, чи надсилає користувач дані: рух курсора, прокрутку, натискання клавіш – і все це може відбуватися без згоди та знання відвідувача. У 2023–2025 роках подібні техніки стали ще небезпечнішими через поширення віддалених модулів-кейлогерів, які підвантажуються через CDN-ланцюжки або компрометовані npm-пакети.

2. Спостереження за історією переглядів і поведінкою. Можливості JS виходять далеко за межі обробки натискання клавіш: невеликий код може взаємодіяти з браузерними cookie, LocalStorage та SessionStorage. Завдяки таким механізмам веб-ресурси отримують інформацію про налаштування браузера, геолокацію, мовні преференції та інші дані. Переважна більшість сайтів і у 2025 році застосовують cookie для аналітики та адаптації роботи сторінки, однак це також дозволяє створювати майже повну картину поведінки користувача в Інтернеті [16]. Браузери поступово посилюють захист приватності – наприклад,

Chrome, Firefox і Safari у 2024–2025 роках впроваджують заборону third-party cookies за замовчуванням, але відстеження все ще можливе через відбитки пристроїв (fingerprinting), моделі поведінки та маніпуляції з пам’яттю браузера.

3. Впровадження шкідливого JavaScript-коду. Одним із найпоширеніших методів атак залишається міжсайтовий скриптинг (XSS). Це вразливість, що дозволяє впровадити шкідливий код JS у легітимну веб-сторінку, а браузер користувача запускатиме цей код так, ніби він є частиною сайту. Якщо сайт працює з чутливими даними – наприклад, особистою інформацією, банківськими транзакціями чи медичними записами – шкідливий скрипт може перехоплювати й пересилати ці дані зловмиснику. XSS також може бути використаний для поширення інших видів зловмисного ПЗ. Один із відомих прикладів – черв’як StalkDaily, що заражав Twitter [1,17]. У 2023–2025 роках подібні інциденти траплялися в екосистемі npm, де популярні пакети підмінювали після компрометації облікових записів авторів.

JavaScript не має доступу до файлової системи чи мережевих API безпосередньо, але браузерні розширення, застарілі механізми на кшталт ActiveX (який майже повністю виведено з використання Microsoft у 2022–2023 роках), а також інтеграція з WebAssembly розширюють потенційні вектори атак. Тому навіть обмежений набір функцій JS не гарантує абсолютної безпеки.

Крім того, браузер автоматично розкриває інформацію про себе – тип, версію, постачальника, IP-адресу тощо. Однак інші дані, як-от електронна пошта, історія переглядів чи закладки, мають залишатися приватними. Якщо б браузер або скрипт дали доступ до цих даних, вони одразу потрапили б до маркетологів або шахраїв, які активно купують подібну інформацію [18].

Поширені вразливості, які повинен знати сучасний веб-розробник.

1. Міжсайтовий скриптинг. XSS дозволяє виконати сторонній JavaScript у межах веб-додатка. Навіть базова атака реалізується через вразливе поле введення даних: хакер вставляє код у посилання або форму, і після переходу користувача цей код виконується в його браузері. XSS дає можливість отримати cookie,

LocalStorage, SessionStorage – тому зберігати конфіденційну інформацію в цих сховищах категорично не радять.

У 2025 популярними захисними механізмами стали Content Security Policy рівня 3, Trusted Types, а також ізоляція API за принципом permission-based access.

2. Міжсайтова підробка запитів (CSRF). CSRF використовує автоматичну відправку cookie разом із HTTP-запитами. Якщо сайт не перевіряє CSRF-токени, зловмисник може виконувати дії від імені користувача: змінювати пошту, скидати пароль, створювати транзакції тощо.

У 2024–2025 більшість фреймворків (Next.js, NestJS, Laravel, Django) ввели механізми CSRF-захисту за замовчуванням, але неправильна конфігурація все ще часто призводить до вразливостей.

3. Ін’єкція серверного JavaScript. Проблема виникає, коли сервер виконує довірений JS-код, а розробник використовує небезпечні функції, такі як eval, Function, небезпечний vm або динамічну конкатенацію.

Якщо вхідні дані не проходять валідацію, зловмисник може виконати будь-який скрипт на сервері Node.js. У 2023–2025 роках атаки на серверний JS особливо зросли через експлойти в неправильній конфігурації server-side rendering (SSR) у Next.js і Nuxt.

4. Вибір між JWT і сеансовою аутентифікацією. JWT (JSON Web Token) – спосіб передавання інформації між сторонами у форматі JSON. Він містить ім’я користувача, email, час створення (iat) та закінчення дії (exp). Токени підписують секретним ключем, але вміст токена не шифрується, тому конфіденційні дані передавати не можна [1,19].

У 2025 році все більше компаній переходять на сеанси з HttpOnly-cookie, бо:

- cookie із прапорами HttpOnly, Secure, SameSite=Strict значно безпечніші;
- JWT у LocalStorage часто викрадають через XSS;
- сеанси легше відкликати та оновлювати.

JWT залишаються корисними для короткотривалих операцій – наприклад, OAuth-автентифікації, одноразових токенів або роботи між мікросервісами.

5. Захист і безпечне збереження паролів. Паролі не можна зберігати у відкритому вигляді – необхідно використовувати гешування разом із сіллю. У 2025 році найбільш рекомендованими алгоритмами стали:

- Argon2id (стандарт OWASP);
- bcrypt з високою складністю.

Секрети (паролі БД, токени API) не можна зберігати у репозиторії. Для їх захисту застосовують:

- Hashicorp Vault;
- 1Password Secrets Automation;
- AWS Secrets Manager;
- git-crypt або git-secret для шифрування в репозиторії.

Відкритість Інтернету робить питання кіберзахисту одним з ключових. Це стало ще важливішим після того, як динамічні мови, зокрема JavaScript, отримали здатність вбудовувати логіку виконання у звичайні веб-сторінки. Оскільки завантаження сторінки часто веде до автоматичного виконання сценаріїв на пристрої користувача, потрібні потужні захисні механізми, що можуть запобігти пошкодженню даних, викраденню конфіденційної інформації або іншим небажаним наслідкам [1].

Першим рубежем безпеки є те, що сам JavaScript у браузері навмисно позбавлений доступу до операційної системи користувача. Наприклад, сучасні клієнтські скрипти не володіють можливістю створювати чи видаляти файли, змінювати файлову структуру або працювати з локальною файловою системою без прямої взаємодії користувача. Саме через це сценарій не може таємно пошкодити дані або встановити шкідливе ПЗ. Крім того, JavaScript не має низькорівневих мережевих інструментів: дозволяється лише робота з HTTP(S)-запитами чи передаванням форм, але неможливо встановлювати довільні з’єднання з іншими пристроями у мережі. Таким чином, браузерний JS не здатен використати комп’ютер користувача як пункт для запуску зовнішніх атак [1].

Другим рівнем оборони виступають обмеження на використання певних функцій. Наприклад, хоча об’єкт Window підтримує метод close(), більшість

браузерів дозволяють закривати лише ті вкладки, які були відкриті самим скриптом. Спроба закрити вікно, яке відкрив користувач вручну, викликає обов'язкове підтвердження. Подібні обмеження існують і в сучасних браузерах – Chrome, Firefox та Edge – які реалізують політику «sandbox», суворе розмежування дозволених дій і окремі модулі захисту (наприклад, Site Isolation).

Серед відомих обмежень також варто відмітити:

- об'єкт History давно не надає доступ до історії переглядів. Замість цього залишено лише можливість переходу назад, уперед або за конкретним зміщенням. Це унеможливлює витік приватних URL-адрес;

- поле value у FileInput неможливо встановити за допомогою JS. Це зупиняє зловмисника від примусового завантаження конфіденційного файлу, наприклад, файлу паролів;

- відправка форм на протоколи mailto або інші не-HTTP схеми вимагає згоди користувача, оскільки містить персональні дані (зокрема email);

- закриття вкладок без дозволу користувача заборонене. Це блокує спроби приховати шкідливу активність;

- маніпуляції з розмірами й позиціями вікон суттєво обмежені. Неможливо створити мініатюрне або невидиме вікно, імітувати системні діалоги чи маскувати фішингові елементи;

- адреси виду about: (наприклад, інформація про кеш) недоступні для сценаріїв з міркувань конфіденційності;

- події неможливо підробити або змінювати їхні властивості, і сценарії не можуть перехоплювати події з інших доменів – це захищає від крадіжки натискань клавіш чи інших дій користувача.

Ці обмеження доповнюються такими механізмами, як Content Security Policy (CSP), Trusted Types, заборона небезпечних API (наприклад, document.write у строгих політиках), а також вимога до сайтів застосовувати HTTPS.

Найважливішим фундаментальним правилом браузерної безпеки залишається політика спільного походження (Same-Origin Policy) [1,20]. Згідно з нею, код може отримувати доступ до даних лише тоді, коли документ і скрипт

завантажені з однієї схеми (https/https), того самого домену та порту. Хоча вона не поширюється абсолютно на всі властивості всіх об’єктів, це правило фактично блокує доступ до вмісту документа з іншого джерела, тим самим запобігаючи крадіжці приватної інформації.

Інколи така суворість створює проблеми для великих веб-проєктів, що складаються з піддоменів. Саме це призвело до появи механізму document.domain, але на сьогодні він офіційно визнаний застарілим і небезпечним, тому сучасні браузери припинили його підтримку. Натомість застосовуються безпечні технології:

- Cross-Origin Resource Sharing (CORS);
- PostMessage API для обміну даними між фреймами;
- Iframe sandboxing;
- Site Isolation / Origin Isolation.

Ці засоби забезпечують безпечну взаємодію сайтів без ослаблення SOP, що сьогодні вважається критично важливим елементом захисту.

Раніше різні браузери пропонували власні способи дозволу «надійним» сценаріям отримувати більше прав, ніж звичайним. Існували «зони безпеки» (у старих версіях Internet Explorer) і «підписані сценарії» (в екосистемі Netscape). Однак ці підходи не витримали випробування часом [21]. Тепер браузери підтримують інші механізми:

- Subresource Integrity (SRI) – перевірка цілісності підключених скриптів за хешами;
- CSP та Trusted Types – обмежують виконання довіреного та недовіреного коду;
- Permission API, що чітко регулює доступ до геолокації, камери тощо;
- модульні політики доступу в WebAssembly;
- політики ізоляції процесів (Chrome Site Isolation, Firefox Fission).

Ці інструменти не дозволяють скрипту отримати розширені привілеї без явної згоди користувача та без криптографічної перевірки джерел.

Процеси створення "підписаних сценаріїв" в їхньому старому вигляді більше не використовуються, а їх функції виконують SRI, цифрові сертифікати сервера та контроль ланцюжка довіри через HTTPS.

Попри всі засоби захисту, веб-загрози продовжують розвиватися. Хакери застосовують техніки обфускації, DOM-based XSS, атаки через залежності в npm-пакетах, прихований майнінг, виконання стороннього коду через скомпрометовані CDN та інші підходи. Вразливості часто виникають не лише через технології, а й через помилки розробників – саме людський фактор і сьогодні залишається ключовою причиною інцидентів.

Тому питання аналізу та виявлення шкідливого JavaScript на веб-сторінках залишається надзвичайно актуальним у 2025 році. Для цього застосовують:

- статичний аналіз JS-коду;
- динамічне тестування поведінки у "пісочниці";
- поведінкову аналітику браузерних API;
- машинне навчання для класифікації підозрілих скриптів;
- автоматичну перевірку залежностей і контроль ланцюжків постачання.

Розробники JavaScript продовжують обмежувати можливості мови для зниження ризиків, а браузери щороку посилюють моделі безпеки. Проте нові методи атак, компрометація ланцюгів постачання (supply chain attacks) та вразливості npm-пакетів у 2023–2025 роках показують, що проблема залишається актуальною.

1.3 Постановка завдання

До сильних сторін JavaScript нині зараховують високу продуктивність, легкість опанування, широку поширеність серед розробників, ефективну взаємодію з інтерфейсом користувача, можливість зменшення навантаження на сервер, а також розширені можливості екосистеми, що охоплює фреймворки, бібліотеки та інструменти. JavaScript і надалі залишається універсальною мовою, яка працює як у браузері, так і на сервері (наприклад, у середовищі Node.js), а

також постійно оновлюється завдяки щорічним стандартам ECMAScript. Особливо помітний розвиток напрямів, пов’язаних з WebAssembly, покращеним рушієм, оптимізацією виконання та підвищенням енергоефективності для мобільних пристроїв.

Серед недоліків традиційно виокремлюють різну реалізацію можливостей у браузерах і ризики безпеки на стороні клієнта, оскільки вразливості або недбалий код можуть бути використані зловмисниками. Це може призвести до викрадення даних, стеження за діями користувача, маніпулювання введенням інформації чи впровадження шкідливих сценаріїв.

До найвідоміших загроз безпеці належать: міжсайтове скриптування (XSS), підміна міжсайтових запитів (CSRF), ін’єкція серверних JS-скриптів (наприклад, у середовищах SSR або у функціях serverless), проблеми з паролями та іншими чутливими даними. Враховуючи такі виклики, спільнота JavaScript сформувала низку обмежень, які стримують потенційно небезпечні дії. Розробники браузерів також постійно додають додаткові механізми захисту у відповідь на нові методи атак.

Однак навіть розвинуті засоби захисту не гарантують абсолютної безпеки. Якщо зловмисник знаходить вразливість та використовує її у власних цілях, ключовим способом убезпечити користувача стає своєчасне виявлення шкідливих JavaScript-сценаріїв. Для цього застосовують статичний та динамічний аналіз коду, поведінкові системи моніторингу, сучасні антивірусні механізми в браузерах і спеціалізовані засоби, що аналізують аномалії у виконанні JS у реальному часі.

У зв’язку з цим виникає потреба створення вдосконаленого методу виявлення небезпечних JavaScript-сценаріїв. Для досягнення поставленої мети слід виконати такі завдання:

- вивчити існуючі способи виявлення небезпечних скриптів;

- проєктування поліпшеного підходу, що підвищує точність і швидкість розпізнавання шкідливого коду;

- експериментально оцінити результативність запропонованого рішення.

2 ПОРІВНЯННЯ АЛГОРИТМІВ ЗАХИСТУ ВІД ШКІДЛИВИХ JAVASCRIPT СЦЕНАРІЇВ

2.1 Алгоритми класифікації виявлення шкідливих javascript

За характером виконання потенційно небезпечного коду методи виявлення поділяють на статичний та динамічний аналіз. Перший ґрунтується на вивченні фіксованих властивостей файлів чи веб-ресурсів – їх структури, набору інструкцій та характеристик. Другий підхід досліджує поведінку об’єкта під час виконання, спостерігаючи за тим, як сценарій взаємодіє з системою та середовищем [1].

Актуальні у 2025 році механізми виявлення шкідливих веб-сторінок, як і шкідливого ПЗ загалом, усе ще спираються на пошук ознак відомих атак. Антивірусні системи регулярно оновлюють свої бази сигнатур, доповнюючи їх новими характеристиками загроз, приклади яких подані у таблиці 2.1.

Однак суто сигнатурний підхід має суттєві обмеження:

- легкість обходу сигнатур за допомогою мінімальних модифікацій коду;
- потреба у частому оновленні баз даних, інколи – декілька разів на добу;
- низька результативність щодо нових, невідомих атак, зокрема zero-day.

Для зменшення цих ризиків сучасні антивірусні рішення застосовують евристичні підходи, здатні розпізнавати підозрілі патерни. Хоча такі методи допомагають виявляти нові варіанти шкідливих дій, вони призводять до підвищеного рівня хибних спрацювань, що потребує додаткової фільтрації.

Значного поширення набули технології динамічного аналізу в ізольованому середовищі, зокрема пісочниці (sandbox). Для аналізу drive-by-download атак пісочницю зазвичай реалізують у вигляді віртуальної машини з операційною системою, наприклад Windows або Ubuntu, у якій запускають браузер. Веб-сторінка відкривається у контрольованому середовищі, після чого фіксуються зміни у файловій системі, процесах, реєстрі чи інших компонентах ОС. Наявність небажаних модифікацій може свідчити про здійснення атаки.

Таблиця 2.1 – Функції відомих загроз (набір 1)

JS функція	Опис
eval()	Кількість функцій eval()
setTimeout()	Кількість функцій setTimeout()
iframe	Кількість рядків, які містять iframe
unescape()	Кількість функцій unescape()
escape()	Кількість функцій escape()
classid	Кількість класів
parseInt()	Кількість функцій parseInt()
fromCharCode()	Кількість функцій fromCharCode()
ActiveXObject()	Кількість функцій ActiveXObject()
No. of string direct assignments	Кількість прямих назначених рядків
concat()	Кількість функцій concat()
indexOf()	Кількість функцій indexOf()
substring()	Кількість функцій substring()
replace()	Кількість функцій replace()
document.addEventListener()	Кількість функцій document.addEventListener()
attachEvent()	Кількість функцій attachEvent()
createElement()	Кількість функцій createElement()
getElementById()	Кількість функцій getElementById()
document.write()	Кількість функцій document.write()
JavaScript word count	Кількість слів
JavaScript Keywords	Кількість ключових слів
No. of characters in JavaScript	Кількість символів
The ratio between keywords and words	Співвідношення між ключовими словами і словами
Entropy of JavaScript	Ентропія
Length of Longest JavaScript Word	Довжина найдовшого слова
The No. of Long Strings >200	Кількість довгих строк більше 200
Length of shortest JavaScript Word	Довжина найкоротшого слова
Entropy of the Longest JavaScript Word	Ентропія найдовшого слова
No. of Blank Spaces	Кількість пробілів
Average Length of Words	Середня довжина слів
No. Hex Values	Кількість шістнадцяткових значень
Share of space characters	Частка пробілів

Попри ефективність, підхід із пісочницями має свої недоліки:

- перевірка великої кількості сайтів вимагає значних ресурсів та часу;

- деякі шкідливі програми працюють лише у пам’яті та не залишають артефактів, знищуючи себе після виконання;

- частина загроз орієнтована на конкретні версії браузерів або плагінів, тому для комплексного аналізу потрібні багаторазові перевірки у різних конфігураціях.

У зв’язку з цими труднощами у сфері кіберзахисту активно досліджують застосування машинного навчання та глибоких нейронних мереж, які дозволяють моделювати поведінку шкідливих об’єктів, аналізувати великі масиви даних та виявляти складні небезпечні закономірності.

Наївний баєсів класифікатор базується на теоремі Баєса та припущенні про незалежність ознак:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}, \tag{2.1}$$

де B – доказ; A – гіпотеза.

Його ключова перевага – простота реалізації й висока швидкість навчання, що робить метод доречним для аналізу великих наборів даних. Модель можна оновлювати поступово, додаючи нові дані без повного перенавчання.

Ідея моделі полягає в обчисленні ймовірності події A за умови, що сталася подія B [22]. Незалежність ознак спрощує обчислення, що й робить метод «наївним».

Основні різновиди наївного баєсового класифікатора:

- багаточленний (Multinomial) – широко застосовується у класифікації текстів, де ознаками виступають частоти слів або токенів;

- Бернуллі (Bernoulli) – працює з булевими ознаками, наприклад, чи зустрічається певне слово у документі;

- Гаусів (Gaussian) – використовується тоді, коли ознаки мають безперервні значення, що можуть бути наближені гаусівським розподілом.

У випадку неперервних характеристик умовна ймовірність обчислюється з урахуванням параметрів розподілу:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\pi\sigma_y^2}\right), \tag{2.2}$$

де y – змінна класу;

σ – середньоквадратичне відхилення;

μ – математичне очікування.

Сьогодні наївні баєсові методи активно використовують у задачах:

- фільтрації електронного спаму;
- класифікації текстового контенту;
- простих систем рекомендацій;
- попереднього аналізу мережевого трафіку.

Головним недоліком залишається припущення про незалежність ознак, що рідко цілком відповідає реальним даним.

Алгоритм J48 є імплементацією дерева рішень C4.5, яке формує двійкову структуру (рис. 2.1). Метод широко застосовується для класифікаційних завдань, оскільки дозволяє наочно моделювати процес прийняття рішень. Після побудови дерево можна використати для класифікації кожного екземпляра навчальної множини або нових даних [1,23].

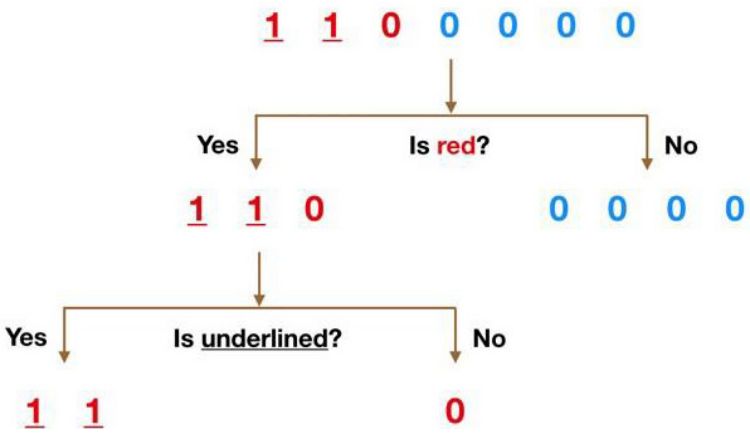

Рисунок 2.1 – Дерево рішень

Для навчання потрібно, щоб кількість прикладів суттєво перевищувала число класів, і кожен об’єкт мав вказану приналежність до відповідної категорії. Цей алгоритм належить до методів із учителем.

Процес побудови дерева виглядає так: є навчальна множина T та набір класів C з k елементів. Для кожного екземпляра відомо, до якого класу він належить. Спочатку існує корінь дерева, пов'язаний із множиною T. Потрібно обрати атрибут A, який має n можливих значень. Це дозволяє розбити T на n підмножин та сформувати n дочірніх вузлів. Далі процедура повторюється рекурсивно для кожної підмножини і завершується у випадках:

- в усіх прикладах підмножини присутній лише один клас (вузол стає листом);

- підмножина порожня (лист отримує найчастіший клас серед предків).

Ключовим викликом є вибір атрибута на кожному рівні. Для цього застосовують два показники:

- приріст інформації (Information Gain) – базується на зміні ентропії після розбиття множини:

$$Gain(S,A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{S} Entropy(S_v), \tag{2.3}$$

де S – набір екземплярів;

A – атрибут;

S_v – підмножина S з $A = v$;

$Values(A)$ – набір можливих значень A;

- індекс Джині (Gini Index) – вимірює ймовірність помилкової класифікації випадково обраного елемента:

$$GiniIndex = 1 - \sum_j p_j^2 . \tag{2.4}$$

Ентропія оцінює ступінь невпорядкованості або «нечистоти» даних: чим вона вища, тим більше інформації містить вибірка. Натомість нижчий індекс Джині означає, що атрибут краще підходить для поділу.

У 2025 році методи на основі дерев рішень і далі широко використовують, зокрема в таких модифікаціях, як Random Forest, XGBoost та LightGBM, які значно підвищують точність за рахунок ансамблю моделей.

Випадковий ліс – це ансамблевий підхід, у якому формується велика множина дерев рішень, а підсумковий прогноз визначається шляхом голосування цих моделей [1,24]. Кожне дерево формує власне припущення про належність об’єкта до певного класу, а сукупний вибір, що отримав найбільшу підтримку, стає результатом моделі (рис. 2.2).

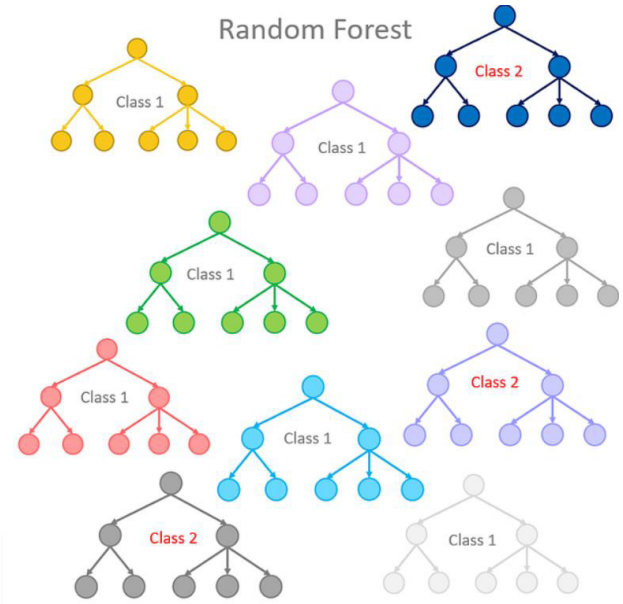

Рисунок 2.2 – Архітектура Random Forest

Головна ідея методу ґрунтується на концепції колективного інтелекту, добре знайомій дослідникам даних: велика група моделей, що майже не корелюють між собою, працює точніше, ніж будь-яка з них окремо. Це пояснює, чому випадкові ліси і на сьогодні залишаються базовими, надійними й водночас конкурентними моделями у задачах класифікації та регресії.

Ключову роль відіграє низький рівень взаємозалежності між деревами. Аналогія з фінансовими портфелями тут цілком доречна: як інвестиції з невисокою кореляцією дозволяють зменшувати ризики, так і незалежні моделі дають можливість отримати більш стабільні та точні передбачення. Дерева рішень «підстраховують» одне одного: якщо частина з них схиляється до помилки, інші дерева компенсують їхні неточності, не дозволяючи всій системі «збитися з курсу». Винятком є лише ситуації, коли всі дерева мають однакову систематичну помилку.

Щоб ансамблевий метод функціонував ефективно, необхідні такі умови [1,25]:

- у наборі ознак має існувати достатньо інформативний сигнал, інакше моделі не досягнуть успішності, вищої за випадкове вгадування;

- сформовані дерева повинні демонструвати мінімальну кореляцію між власними помилками.

Модель SVM – один із найпоширеніших класичних алгоритмів навчання з учителем, орієнтований на завдання класифікації та, за потреби, регресії. Після того як алгоритм отримує розмічені приклади для кожного класу, він визначає оптимальну гіперплощину у просторі ознак, що найкраще розділяє ці класи.

Існує багато можливих способів провести гіперплощину у N-вимірному просторі (N – кількість ознак), але завдання алгоритму – знайти ту, що забезпечує найширший можливий зазор (margin) між вибірками різних класів (рис. 2.3). Чим більша ця відстань, тим стійкіше працює модель щодо нових прикладів.

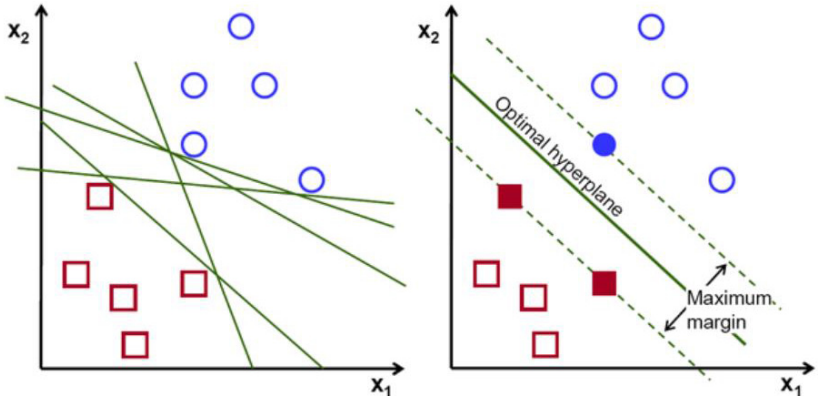

Рисунок 2.3 – Гіперплощини

Гіперплощина виступає межею прийняття рішень. Для двох ознак вона представляє собою пряму, для трьох – площину, а для більшої кількості характеристик – багатовимірну поверхню.

Опорні вектори – це ті точки навчальних даних, що лежать найближче до межі класифікації. Саме вони визначають положення та форму гіперплощини. Якщо їх вилучити, кордон між класами зміститься, що підтверджує їхню критичну роль у моделі.

Варто зазначити відмінність від логістичної регресії. У логістичній моделі вихід лінійної функції перетворюється сигмоїдою у значення в межах [0;1], а рішення приймається за порогом 0,5. Натомість SVM оцінює знак та величину лінійної функції: значення понад 1 відносять об’єкт до одного класу, а нижче -1 – до іншого. Отже, інтервал [-1;1] формує область, що відділяє класи.

На сьогодні SVM залишається актуальним, а широке використання ядерних функцій (RBF, поліноміального, сигмоїдного ядра) дозволяє застосовувати його для складних нелінійних розділень.

AdaBoost (Adaptive Boosting) – перший практичний алгоритм підсилення, запропонований Йоавом Фройндом та Робертом Шапіре у 1996 році. Мета методу – об'єднати велику кількість слабких моделей у потужний ансамбль. Остаточний прогноз визначається формулою:

$$F(x) = sign\left(\sum_{m=1}^{M} \Theta_m f_m(x)\right), \tag{2.5}$$

де f_m – m-й класифікатор;

а Θ_m – відповідна вага.

Нехай є n об'єктів даних: $x_i \in R^d$, $y_i \in \{-1;1\}$, де 1 – позитивний клас, -1 – негативний. Процес навчання включає:

- ініціалізацію ваг кожного прикладу:

$$w(x_i, y_i) = \frac{1}{m}, m = 1,...,M; \tag{2.6}$$

- послідовне навчання слабких класифікаторів і вибір того, хто демонструє найменшу зважену помилку:

$$e_m = E_{w_m}\left[1_{y \neq f(x)}\right]; \tag{2.7}$$

- обчислення ваги слабкої моделі:

$$\Theta_m = \frac{1}{2}\ln\left(\frac{1-e_m}{e_m}\right). \tag{2.8}$$

Якщо її точність перевищує 50%, коефіцієнт позитивний і підсилює її внесок, якщо точність нижча – вага стає негативною, а прогноз інвертується, що перетворює слабку модель на корисну.

Оновлення ваг прикладів:

$$w_{m+1}(x_i, y_i) = \frac{w_m(x_i, y_i) \cdot e^{-\Theta_m y_i f_m(x_i)}}{Z_m}, \tag{2.9}$$

де Z_m – коефіцієнт нормалізації, нормалізує суму ваг до 1.

Ті об’єкти, які були класифіковані неправильно, отримують більшу вагу, оскільки експоненційний множник у формулі збільшується.

Після M ітерацій сумується зважений внесок усіх класифікаторів, формуючи прогноз ансамблю.

AdaBoost усе ще застосовується в класичних задачах, хоча все частіше його замінюють більш стійкі до шуму варіанти – XGBoost, CatBoost та LightGBM.

Алгоритм REPTree (Reduced Error Pruning Tree) — спрощений та дуже швидкий метод побудови дерев рішень, популярний завдяки реалізації у Weka. Алгоритм використовує обрізання дерева, спрямоване на зменшення помилок, та перебудову піддерев для отримання найкомпактнішої структури, що зберігає високу точність [1,26].

REPTree формує бінарні дерева для класифікаційних або регресійних завдань. У ролі критеріїв розгалуження застосовуються ентропійні та статистичні показники. Обробка пропущених значень і числових атрибутів схожа до методів C4.5, хоча Weka не надає детальних технічних аспектів реалізації.

Користувач може налаштовувати:

- мінімальну кількість прикладів у листі;
- максимальну глибину;
- мінімальну дисперсію (для регресії).

Хоча REPTree не є сучасним стандартом, він і досі використовується в навчальних цілях та у швидких експериментах.

ADTree (Alternating Decision Tree) – модель, що поєднує дерева рішень та бустинг. Під час навчання дані неодноразово поділяються відповідно до обраних ознак. Мета – розбити множину прикладів на підгрупи до моменту, коли вони стануть максимально однорідними за цільовою змінною. Цей процес відомий як індукція дерева рішень [26].

ADTree складається з:

- вузлів рішень, що містять логічні умови;
- вузлів прогнозування, які містять числові значення.

Кореневий і кінцеві вузли завжди є вузлами прогнозу. На відміну від класичних дерев (CART, C4.5), де кожен об’єкт проходить лише один шлях, ADTree дозволяє проходити всі гілки, умови яких виконуються. Остаточне рішення формується шляхом підсумовування чисел із вузлів прогнозування, через які пройшов об’єкт.

Хоча ADTree не належить до найпопулярніших методів, він залишається цікавим прикладом гібридного підходу, що поєднує прозорість дерев та підсилення.

2.2 Проблеми алгоритмів класифікація виявлення загроз

Навіть при використанні сучасних алгоритмів для виявлення шкідливих сценаріїв неможливо гарантувати абсолютну безпеку веб-ресурсів. Кожен метод може генерувати як хибнопозитивні, так і хибнонегативні результати, що обумовлено слабкими місцями конкретного класифікатора.

Наївний баєсів класифікатор залишається швидким та простим інструментом, але має певні обмеження, що можуть впливати на ефективність:

- припущення незалежності атрибутів. Ця проблема вирішується шляхом попереднього статистичного аналізу, щоб оцінити кореляцію між ознаками та відібрати найбільш некорельовані;

- рівне ставлення до всіх ознак. Можна додавати ваги важливим атрибутам для підвищення їхнього впливу на фінальне рішення;

- нульова ймовірність. Її усувають або додаванням одиниці до частоти атрибута, або застосуванням гаусового розподілу;

- проблема постійних значень. Перетворення безперервних величин у дискретні дозволяє коректно обробляти такі атрибути;

Дерево рішень – один із найбільш поширених методів машинного навчання, проте теж має недоліки:

- обчислювальні операції можуть бути складнішими, ніж у багатьох інших алгоритмів;

- тренування моделі займає більше часу;

- процес навчання є ресурсомістким через велику складність обчислень;

- дерева рішень не завжди ефективні для регресійних завдань або прогнозування безперервних змінних.

Випадковий ліс покращує точність порівняно з окремими деревами, але теж має недоліки [27]:

- для складних завдань його точність часто нижча, ніж у моделей із градієнтним бустингом;

- інтерпретація моделі складніша, ніж у одиночного дерева;

- пам’ять, необхідна для зберігання сотень дерев, може бути значною.

Метод опорних векторів також має специфічні обмеження [28]:

- не підходить для великих обсягів даних;

- чутливий до шуму та перекриття класів;

- при великій кількості ознак порівняно з кількістю вибірок ефективність падає;

- алгоритм не забезпечує ймовірнісне пояснення класифікації.

AdaBoost має такі негативні особливості [29]:

- поступове навчання потребує якісних даних для стабільної роботи;

- чутливий до зашумних даних, що потребує їх попереднього очищення;

- швидкість роботи поступається іншим алгоритмам, особливо при великих обсягах інформації [1,30].

2.3 Порівняння ефективності алгоритмів класифікації загроз

У проведеному дослідженні для реалізації семи різних класифікаційних методів було застосовано Weka API, який забезпечує зручний інтерфейс для навчання й тестування моделей машинного навчання. Для визначення алгоритму з найкращими показниками була використана матриця неточностей (табл. 2.2), де відображено співвідношення між прогнозованими значеннями та фактичними результатами, отриманими під час виконання процедур класифікації.

Таблиця 2.2 – Матриця неточностей

		Прогнозоване значення	
		Шкідливий	**Не шкідливий**
Фактичне значення	**Шкідливий**	*a*	*b*
	Не шкідливий	*c*	*d*

Щоби об’єктивно зіставити роботу кожного алгоритму, їхню здатність виявляти шкідливі сценарії оцінювали за чотирма ключовими характеристиками: точністю, часткою хибно позитивних спрацьовувань, часткою хибно негативних випадків, а також за кривою помилок (error curve).

Точність відображає відсоток коректних передбачень відносно всіх зроблених прогнозів та визначається відповідною формулою

$$A = \frac{a+d}{a+b+c+d}. \tag{2.10}$$

Хибно позитивний показник описує частку безпечних JavaScript-фрагментів, які алгоритм помилково відніс до шкідливих FPR:

$$FPR = \frac{c}{c+d}. \tag{2.11}$$

Хибно негативний результат (FNR) показує кількість шкідливих кодів, хибно класифікованих як безпечні:

$$FNR = \frac{b}{a+b}. \tag{2.12}$$

Крива помилок (часто її називають кривою помилок класифікації або ROC-кривою (Receiver Operating Characteristic)) – графічне подання якості бінарного класифікатора, яке демонструє залежність справжніх позитивів від хибних позитивів. По ось X відкладається частка хибнопозитивних класифікацій FPR (2.11), а по осі Y – частка істинно позитивних класифікацій (True Positive Rate):

$$TPR = \frac{a}{a+b}. \tag{2.13}$$

Вона дозволяє зрозуміти, як модель поводиться при різних порогах прийняття рішень.

Нижче наведено детальний аналіз ефективності моделей, виконаний на основі даних із табл. 2.1. Згідно з підсумками, представленими в Таблиці 2.3, найкращий результат демонструє ADTree, який досягнув точності 99,91%, а також найменших хибно позитивних (0,001) і хибно негативних (0,000) значень.

Таблиця 2.3 – Показники ефективності алгоритмів для набору 1

Класифікатор	**Точність, %**	*FPR*	*FNR*	**ROC**
Наївний Баєсів	95,48	0,063	0,009	0,995
Дерево рішень	99,67	0,005	0,000	0,998
Випадковий ліс	99,76	0,004	0,000	1,000
Метод опорних векторів	99,70	0,003	0,004	0,997
AdaBoost	99,70	0,004	0,000	1,000
REPTree	99,67	0,005	0,000	0,998
ADTree	99,91	0,001	0,000	1,000

Найнижчу продуктивність отримано у наївного баєсівського класифікатора, чия точність становила 95,48%, причому саме він мав найбільші показники неправильних класифікацій (0,063 - *FPR* та 0,009 - *FNR*). Такий розрив добре ілюструє різницю між простими й ансамблевими моделями, що активно використовують деревоподібні структури для значно кращої адаптації до даних.

Щоб оцінити, як змінюється поведінка статичного класифікатора при використанні іншого набору ознак, було взято новий перелік характеристик (табл. А.1, додаток А), а одержані результати наведено в табл. 2.4.

Таблиця 2.4 – Показники ефективності алгоритмів для набору 2

Класифікатор	**Точність, %**
Наївний Баєсів	79,45
Дерево рішень	96,82
Випадковий ліс	95,51
Метод опорних векторів	93,37
AdaBoost	99,79
REPTree	96,04
ADTree	95,66

З представлених даних видно, що лідером виявився AdaBoost, який досяг точності на рівні 99,79%. Однак наївний баєсівський метод знову показав слабкі результати – точність лише 79,45%, що підтверджує його залежність від

специфіки вибірки та припущення про незалежність ознак, яке рідко виконується в реальних наборах JavaScript-даних.

Для подальшого підвищення якості класифікації було використано об’єднаний набір ознак, що містить усі функції із таблиці 2.1 та таблиці А.1.

Результати відображає таблиця 2.5

Таблиця 2.5 – Показники ефективності алгоритмів

Класифікатор	**Точність, %**	*FPR*	*FNR*	**ROC**
Наївний Баєсів	97,56	0,003	0,0139	0,994
Дерево рішень	99,82	0,003	0,000	0,998
Випадковий ліс	99,85	0,002	0,000	1,000
Метод опорних векторів	99,91	0,001	0,001	1,000
AdaBoost	99,79	0,003	0,000	1,000
REPTree	99,67	0,005	0,000	0,998
ADTree	99,79	0,003	0,000	1,000

Застосування розширеного набору суттєво покращило роботу всіх розглянутих алгоритмів: кожен класифікатор забезпечив дуже високу точність виявлення шкідливих JS-фрагментів. Попри це, наївний баєсівський метод знову продемонстрував найскромніші результати – точність 97,56% і найбільшу кількість хибних визначень. Це підкреслює перевагу більш складних моделей, здатних працювати з нелінійними взаємозв’язками та взаємозалежними ознаками, що характерно для аналізу шкідливих скриптів.

2.4 Висновок до розділу

Сучасні системи для виявлення шкідливих сторінок та програм здебільшого базуються на аналізі ознак відомих загроз. Найпоширеніші алгоритми включають Наївний Баєсів, Дерево Рішень, Випадковий Ліс, Метод Опорних Векторів, AdaBoost, REPTree, ADTree та сучасні модифікації градієнтного бустингу. Основні проблеми продуктивних методів – високі витрати пам’яті та обчислювальні ресурси.

Для вибору ефективного класифікатора застосовують матрицю неточностей, що містить фактичні й прогнозовані результати алгоритму. На її основі оцінюють

точність, хибнопозитивні та хибнонегативні спрацьовування, а також форму кривої помилок.

Результати порівняння показали, що алгоритми демонструють приблизно однакову точність, проте їхні показники залежать від набору функцій шкідливих сценаріїв. У першому випадку лідером став ADTree, а в другому – метод опорних векторів. Найменш точним залишався Наївний Баєсів, відстаючи на 2–3%.

Для покращення роботи було обрано дерево рішень. У всіх експериментах воно показувало високі результати за ключовими метриками. Перевага цього методу полягає в тому, що не потрібно масштабувати чи нормалізувати дані, а попередня обробка вимагає менше коду, часу та аналітики. Крім того, концепція дерева рішень зрозуміла більшості програмістів і легше сприймається порівняно з іншими алгоритмами.

3 РЕАЛІЗАЦІЯ АЛГОРИТМУ ВИЯВЛЕННЯ ШКІДЛИВИХ JAVASCRIPT СЦЕНАРІЇВ

3.1 Реалізація та тестування удосконаленого алгоритму

Дерево рішень – це структура, що використовується для представлення послідовності логічних правил у формі розгалуженої схеми. Вона складається з двох основних типів елементів: вузлів, у яких містяться перевірки умов, та листових вузлів, де розташовуються підсумкові рішення. У внутрішніх вузлах розміщують правила, які визначають, як саме слід аналізувати атрибути навчального набору даних, і відповідно до цих правил здійснюється перевірка прикладів [1].

У найпростішому випадку після виконання перевірки всі об’єкти, що потрапили до вузла, поділяються на дві підгрупи:

- ті, що відповідають умові;

- ті, що не відповідають.

Після цього кожну підмножину розглядають окремо, знову застосовуючи певне правило. Процес продовжується до моменту, коли спрацьовує критерій завершення алгоритму. Цей критерій може визначати, що поділ більше не потрібний, наприклад, коли всі приклади в підмножині належать до одного класу або коли подальший поділ не покращує точність. У такому разі вузол перетворюється на лист – елемент, у якому вже не відбувається розбиття даних. Лист задає остаточне рішення для всіх об’єктів, які до нього потрапили. Для дерева класифікації таким рішенням є відповідний клас, що приписується елементам цієї групи.

Отже, на відміну від внутрішніх вузлів, листові вузли не містять правил, а лише набір прикладів, що відповідають усім умовам на шляху від кореня дерева до конкретного листа. Оскільки шлях від кореня до кожного листа є єдиним, кожен об’єкт може опинитися тільки в одному листі. Це забезпечує однозначність

результату та відсутність суперечливих рішень для одного й того самого прикладу.

Модернізація алгоритму класифікації шкідливих JavaScript сценаріїв на основі дерева рішень полягає в наступному.

1 Реалізована спеціальна підбірка ознак (feature engineering) – сформовано власний набір ознак для опису JavaScript-коду (символьні, лексичні й структурні метрики), тобто не покладалися лише на «класичні» функції (eval, iframe тощо), а додали нові характеристики, які краще відрізняють шкідливі скрипти від звичайних.

2 Визначається рейтинг шкідливості коду (maliciousness score) – удосконалений алгоритм не просто класифікує «шкідливий/нешкідливий», а обчислює рейтинг шкідливості для фрагментів коду – це дає змогу пріоритизувати підозрілі місця й виділяти саме уразливі фрагменти для подальшого розслідування.

3 Модифіковано алгоритм «Дерево рішень» - удосконалено під задачі аналізу JS, а саме: впроваджено обробку специфічних ознак і механізмів врахування ваг/рейтингів ознак. Переваги такого підходу – інтерпретованість, менші вимоги до предобробки (немає потреби в жорсткій нормалізації) та зрозумілі правила прийняття рішення.

4 Підвищення точності - після введення нових ознак і рейтингу шкідливості всі класифікатори показали значне зростання точності, а дерево рішень у випробуваннях продемонструвало високі метрики, що свідчить про ефективність запропонованих змін.

5 Виділення вразливих фрагментів – окрім загальної оцінки скрипта, алгоритм вміє виводити які саме фрагменти коду (рядки/функції) найімовірніше є підозрілими – це прискорює аналіз і зменшує кількість хибних спрацьовувань при розслідуванні.

Блок-схема запропонованого алгоритму перевірки шкідливих JavaScript сценаріїв на основі дерева рішень представлена на рисунку 3.1.

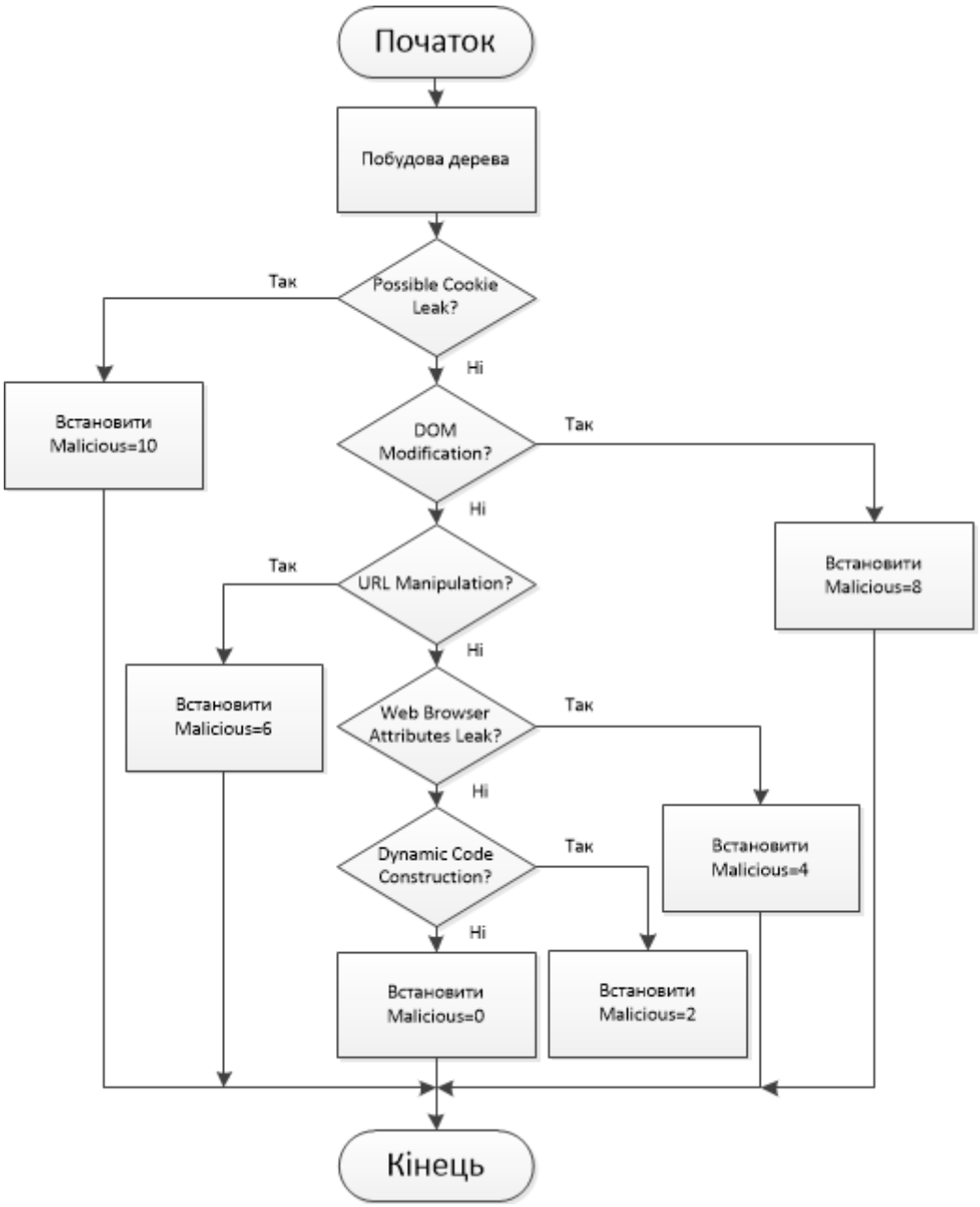

Рисунок 3.1 – Блок-схема алгоритму дерева рішень

Реалізацію запропонованого удосконалення алгоритму виявлення шкідливих JavaScript сценаріїв на основі дерева рішень було здійснено засобами мови програмування C #.

Блок-схема алгоритму роботи реалізованого програмного забезпечення представлено на рисунку 3.2

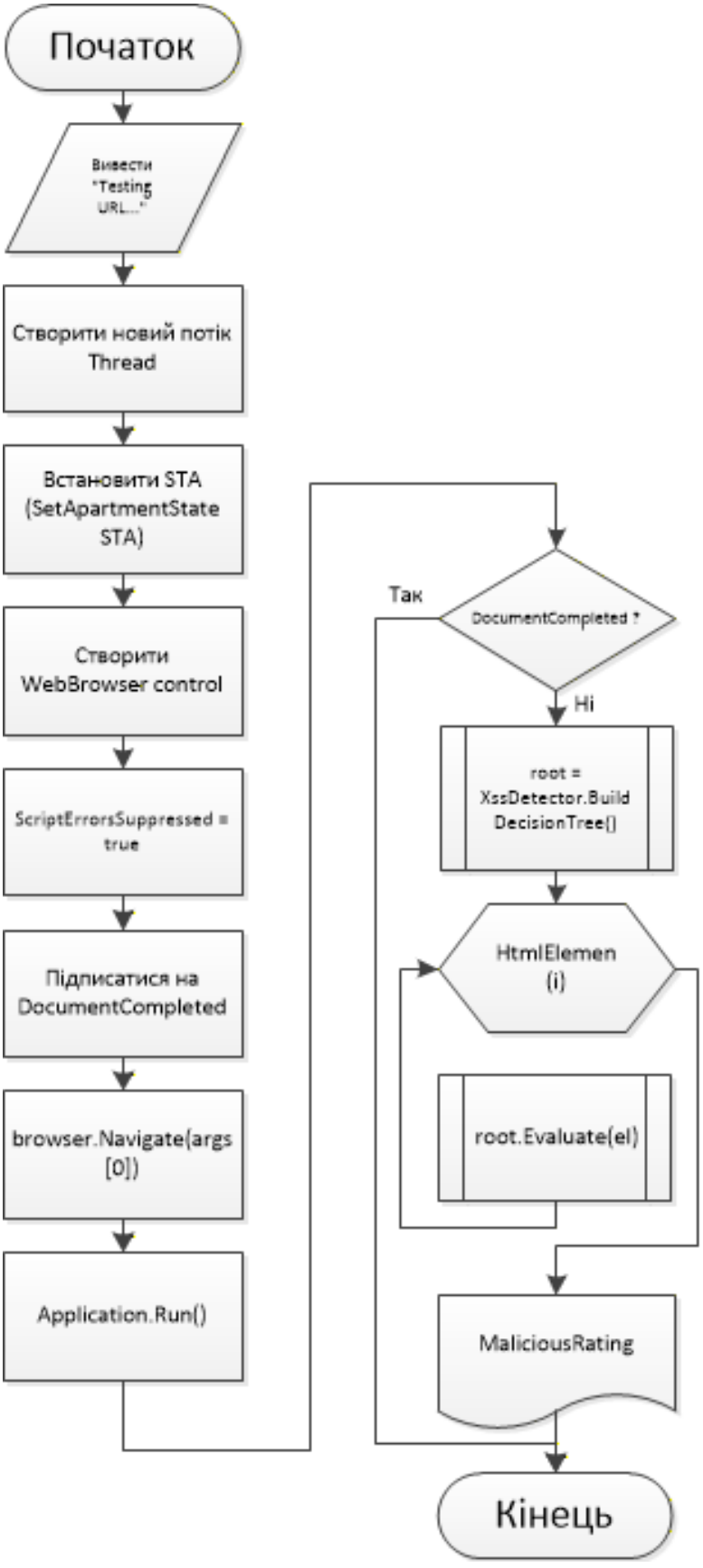

Рисунок 3.2 – Блок-схема алгоритму пошуку шкідливих JavaScript сценаріїв

У роботі формується перелік JavaScript-функцій, які можуть бути ознаками шкідливих дій у коді:

```
public static class XssDetector
{
    private static readonly string[] cookiesAccess =
    {
        "document.cookie",
        "window.localStorage",
        "document.URL",
        "getCookies",
        "gotCookie"
    };

    private static readonly string[] dynamicDOM =
    {
        "document.write",
        "document.writeln",
        "insertAdjacentHTML"
    };

    private static readonly string[] urlManipulations =
    {
        "document.URI",
        "location.assign",
        "location.replace"
    };

    private static readonly string[] webBrowserAttributes =
    {
        "getAppName",
        "getUserAgent"
    };

    private static readonly string[] dynamicCode =
    {
        "setInterval",
        "setTimeout",
        "eval"
      };
```

Ці ознаки використовуються як правила розділення в дереві.

Опишемо поведінку окремих JS-елементів.

`Document.cookie` – це властивість браузера, що дає доступ до cookie-файлів. Cookie – це невеликі текстові дані, які зберігаються на стороні клієнта відповідно до специфікації RFC 6265. Найпоширеніші сценарії використання – зберігання інформації для автентифікації, підтримки сесії або персоналізації.

`String`, який повертається з `document.cookie`, формується як перелік пар ключ=значення, розділених символом «;». Для пошуку потрібного cookie можна розбити цей рядок на масив або застосувати регулярні вирази. Наприклад, для швидкого отримання значення потрібного параметра зручно використовувати функцію на кшталт `getCookie(name)`, а для створення чи оновлення cookie використовується `setCookie(name, value, options)`.

`localStorage` забезпечує доступ до об'єкта `Storage` і дозволяє зберігати дані без обмеження часу. На відміну від `sessionStorage`, значення не видаляються після закриття браузера. Обидва механізми мають спільну рису: ключі й значення зберігаються як текстові рядки. Дані є специфічними для протоколу сторінки, тому одна й та сама сторінка, але з іншим протоколом (HTTP/HTTPS), матиме окремі сховища.

`Document.URL` повертає повний шлях до HTML-документа, включно зі схемою, доменом і параметрами рядка запиту.

Найшвидший спосіб доступу до cookie – регулярні вирази. `getCookie(name)` витягує cookie за заданим ім'ям, а `setCookie` дозволяє встановити значення з необхідними атрибутами – `path`, `expires`, `secure`, `sameSite` тощо.

Метод `document.write` вносить текст у документ під час його завантаження до побудови DOM-структури. Якщо застосувати його після завершення завантаження, він перезапише всю сторінку.

Метод `writeln()` працює подібно до `write()`, але додає наприкінці рядка символ нового рядка. Для вставки тексту через ці методи іноді необхідно відкрити потік документа `document.open()`, а після завершення виводу закрити його `document.close()`.

Властивість innerHTML дає змогу отримувати або змінювати HTML-вміст елемента у вигляді тексту. Це часто використовується для оновлення інтерфейсу веб-сторінки. Проте такий спосіб не завжди ефективний, бо призводить до повного переформування DOM-фрагмента. На відміну від цього, insertAdjacentHTML() дозволяє вставити HTML-рядок у потрібне місце без руйнування існуючої структури, що позитивно впливає на продуктивність.

Метод Location.assign() перенаправляє користувача на новий URL, а Location.replace() виконує таке саме перенаправлення, але без додавання попередньої сторінки в історію браузера. Тому повернутися назад за допомогою кнопки "Назад" у цьому випадку неможливо.

Метод getUserAgent надає значення заголовка user-agent, а appName повертає назву браузера.

У JavaScript є можливість запускати функцію з затримкою або з певною періодичністю. Для одноразового виклику існує метод setTimeout, а для циклічного – setInterval. Крім того, функція eval дозволяє виконувати рядок як код, що становить потенційну загрозу безпеці, оскільки зловмисники можуть динамічно створювати та виконувати JS-фрагменти.

Елемент HTML <input> є ключовим інтерфейсним компонентом вебформи. Він може мати десятки різних режимів роботи – наприклад, text, email, date, password тощо. Завдяки великій кількості атрибутів (<required>, <pattern>, <maxlength>) цей елемент залишається одним із найскладніших і найуніверсальніших засобів введення даних. Проте саме через таку універсальність він може бути використаний сторонніми особами для модифікації форм і викрадення даних.

У структурі дерева рішень <input> є кореневим вузлом, який виконує первинний поділ даних:

```
var isButton = new DecisionQuery
{
     Title="Is button type element",
     Test (elem) => string.Equals (elem. TagName, "button",
StringComparison.OrdinalIgnoreCase),
     Positive = cookieAccessBranch,
     Negative = new DecisionResult {maliciousRating = 0}
};
var isScript = new DecisionQuery
{
     Title= "Is script type element",
     Test (elem) => string.Equals (elem. TagName, "script",
StringComparison.OrdinalIgnoreCase),
     Positive = cookieAccessBranch,
     Negative = isButton
};

var isInput = new DecisionQuery
{
    Title = "Is input type element:",
    Test (elem) => string.Equals(elem.TagName, "input",
  StringComparison.OrdinalIgnoreCase),

    Positive = cookieAccessBranch,
    Negative = isScript
};

return isInput;
```

Якщо ознака свідчить про доступ до cookie, виконується перехід до гілки «cookieAccessBranch». Якщо ж умова не підтверджується, перевіряється наявність тегу <script>.

Тег <script> призначений для підключення та опису скриптів. Коли код розташовано у зовнішньому файлі, сторінка завантажується швидше завдяки кешуванню. Однак зловмисники використовують можливість підключення файлу, щоб завантажити шкідливі інструкції. З цієї причини <script> входить до гілок дерева, які аналізують потенційні небезпечні дії JavaScript.

Елемент button реалізує кнопки, що активують різноманітні функції. Сам по собі він не має поведінки за замовчуванням, але може запускати JavaScript-код через обробники подій. Тому він також розглядається як елемент, що може бути використаний недобросовісним користувачем для виконання небажаних скриптів.

Кожна потенційна загроза, наприклад «Possible cookie leak», «Dynamic Code Construction», «URL manipulation», «Web browser attributes leak» та інші, отримує числову оцінку небезпечності в діапазоні від 2 до 10:

```
var urlManipulationBranch = new DecisionQuery
      {
          Title = "URL Manipulation",
          Test = elem => urlManipulations.Any(c =>
elem.OuterHtml.Contains(c)),
            Positive = new DecisionResult { MaliciousRating = 6 },
            Negative = webBrowserBranch
        };

        var dynamicDOMBranch = new DecisionQuery
        {
            Title = "Dynamic DOM Modification",
            Test = elem => dynamicDOM.Any(c =>
elem.OuterHtml.Contains(c)),
            Positive = new DecisionResult { MaliciousRating = 8 },
            Negative = urlManipulationBranch
        };

        var cookieAccessBranch = new DecisionQuery
        {
            Title = "Possible Cookie Leak",
            Test = elem => cookiesAccess.Any(c =>
elem.OuterHtml.Contains(c)),
            Positive = new DecisionResult { MaliciousRating = 10 },
            Negative = dynamicDOMBranch
        };
```

Після завершення аналізу всі присвоєні значення сумуються. Цей підсумковий рейтинг дозволяє оцінити загальний рівень ризику, який несе досліджуваний фрагмент JavaScript-коду.

Останніми елементами дерева рішень виступають вузли «Web browser attributes leak» та «Dynamic Code Construction», яким присвоєно рівні небезпеки 4 та 2 відповідно:

```
public static DecisionQuery BuildDecisionTree()
    {
        var dynamicCodeBranch = new DecisionQuery
        {
       Title = "Dynamic Code Construction",
       Test = elem => dynamicCode.Any(c =>
elem.OuterHtml.Contains(c)),
       Positive = new DecisionResult { MaliciousRating = 2 },
       Negative = new DecisionResult { MaliciousRating = 0 }
        };

        var webBrowserBranch = new DecisionQuery
        {
       Title = "Web Browser Attributes Leak",
       Test = elem => webBrowserAttributes.Any(c =>
elem.OuterHtml.Contains(c)),
       Positive = new DecisionResult { MaliciousRating = 4 },
       Negative = dynamicCodeBranch
        };
```

Продемонструємо формування окремого вузла:

```
public class DecisionQuery : Decision
{
    public string Title { get; set; }
    public Func<HtmlElement, bool> Test { get; set; }
    public Decision Positive { get; set; }
    public Decision Negative { get; set; }

    public override Decision Evaluate(HtmlElement elem)
    {
        bool result = Test(elem);
        Console.WriteLine($"\n\t{Title}?  {(result ? "YES" :
"NO")}");
        return result ? Positive.Evaluate(elem) :
Negative.Evaluate(elem);
    }
}
```

Цей фрагмент програмної логіки виконує тестування, виводить отриманий результат і визначає подальшу гілку аналізу.

Якщо підсумований показник шкідливості (`maliciousRating`), що вираховується наприкінці перевірки, відрізняється від нуля, система робить висновок про наявність небезпечних JavaScript-конструкцій:

```
public class DecisionResult : Decision
{
    public int MaliciousRating { get; set; }
    public bool Result => MaliciousRating > 0;

    public override Decision Evaluate(HtmlElement elem)
    {
        Console.WriteLine($"\nMalicious: {(Result ? "YES" : "NO")},
Rating = {MaliciousRating}");
        return this;
    }
}
```

Частина програми, що відповідає за виявлення та інтерпретацію JS-фрагментів веб-сторінки і здійснює перевірку на можливі загрози:

```
private static void PageHtml_DocumentCompleted (object sender,
WebBrowserDocumentCompletedEventArgs e)
{
   var browser sender as WebBrowser;
   var tree MainDecisionTree();
   int maliciousRating = 0;
   int maxRating = 0;
   foreach (HtmlElement element in browser.Document.All)
   {
      maxRating += string.Equals (element.TagName, "button",
   StringComparison.OrdinalIgnoreCase) ||
                  string.Equals (element.TagName, "script",
                  StringComparison. OrdinalIgnoreCase) ||
                  string.Equals(element.TagName, "input",
                  StringComparison. OrdinalIgnoreCase)
                  ? 10: 0;
     if (tree.Evaluate (element) is DecisionResult result)
     {
     maliciousRating += result.maliciousRating;
     if (result.Result)
     {
        Console.WriteLine (element. OuterHtml);
        Console.WriteLine (Decision. Warning);
     }
     Decision. Warning string.Empty;
     }
  }
```

Головний модуль, який аналізує вхідні дані, запускає процедуру перевірки та відображає її результати:

```
class Program
{
    private static void Main(string[] args)
    {
        Console.WriteLine($"Testing {args[0]} for XSS...");

        Thread t = new Thread(() =>
        {
            var browser = new WebBrowser();
            browser.DocumentCompleted += (s, e) =>
            {
                var root = XssDetector.BuildDecisionTree();
                foreach (HtmlElement el in browser.Document.All)
                {
                    root.Evaluate(el);
                }
            };

            browser.ScriptErrorsSuppressed = true;
            browser.Navigate(args[0]);
            Application.Run();
        });

        t.SetApartmentState(ApartmentState.STA);
        t.Start();
        t.Join();
    }
}
```

Для демонстрації роботи алгоритму на безпечному прикладі протестовано сайт Івано-Франківського національного університету нафти і газу (https://nung.edu.ua/) (рис. 3.3). Аналіз знайшов 11 небезпечних ділянок коду у сценаріїв типу script категорій Dynamic Code Construction. Ресурс віднесено до ризикових, тому рівень небезпеки становить 22 із 2162 перевірених компонентів.

Наступним прикладом ресурсу було використано сайт lostfilm.tv. Алгоритм знайшов небезпечні ділянки коду – виклик document.write у сценаріїв типу script потрапив до категорій Possible cookie leak та Dynamic Code Construction. Ресурс віднесено до ризикових із рейтингом 414 зі 2190 протестованих елементів (рис. 3.4).

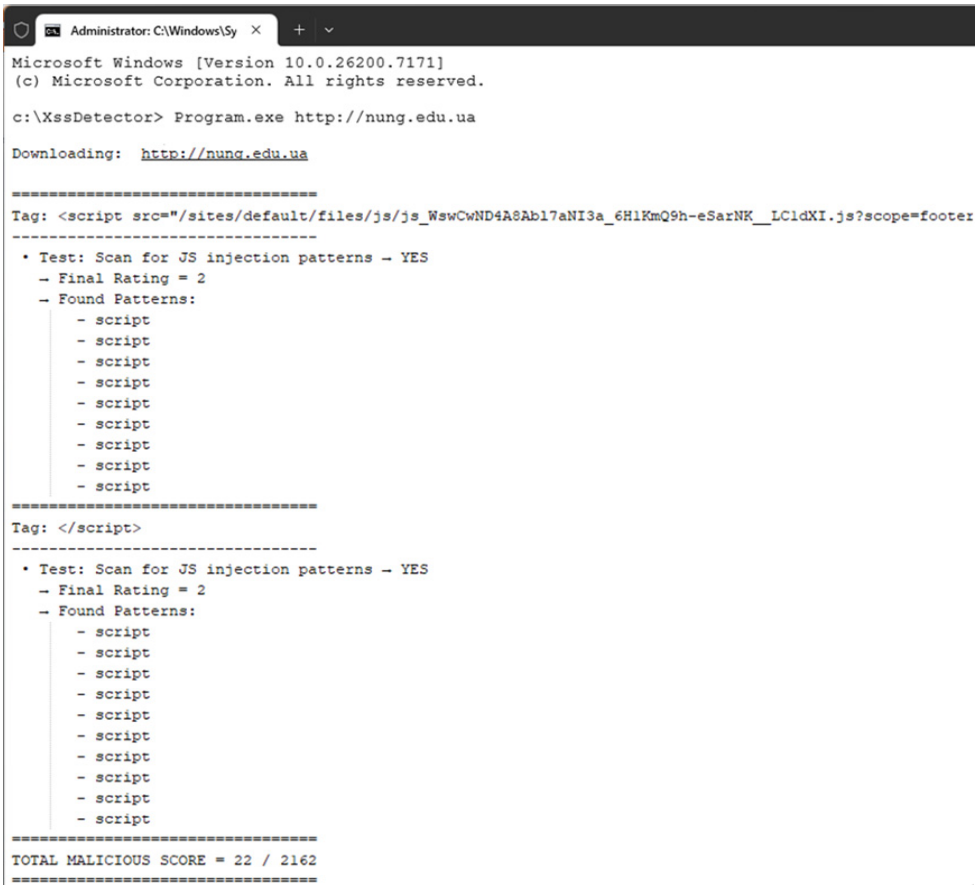

Рисунок 3.3 – Приклад аналізу сайту nung.edu.ua

Рисунок 3.4 – Тестування сайту lostfilm.tv

Наступним прикладом став сайт speedtest.org.ua/en. Знайдено небезпечні частину коду, де атрибут `script` відповідає класу `Dynamic Code Construction`. Сторінку визнано потенційно небезпечною з показником 54 із 270 протестованих компонентів (рис. 3.5).

Рисунок 3.5 – Тестування сайту securityweek.com

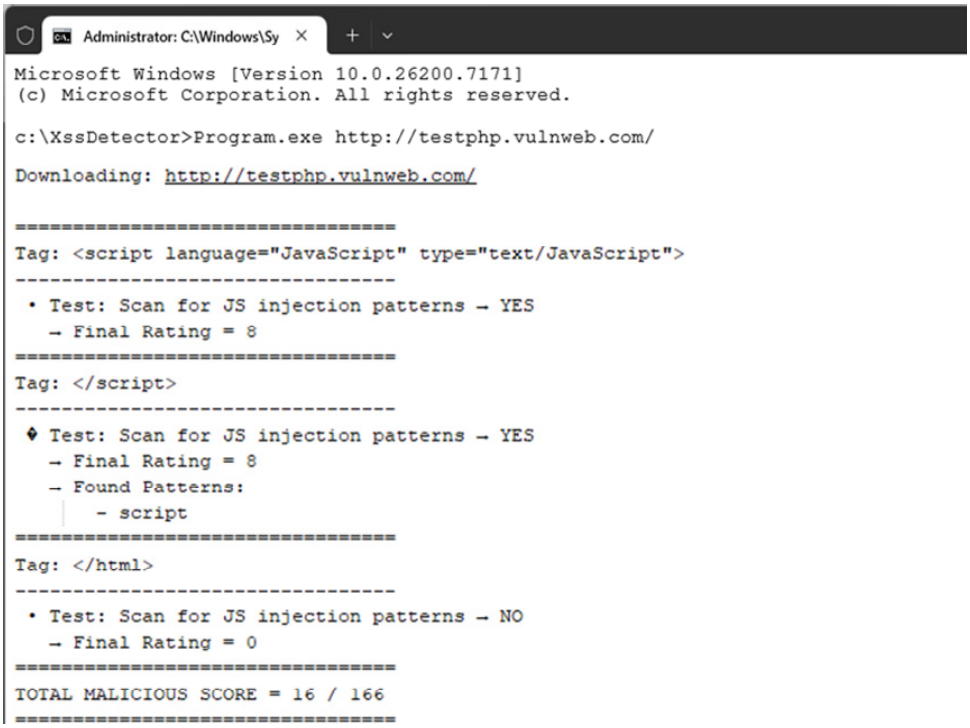

Рисунок 3.6 – Тестування сайту testphp.vulnweb.com

Тестування сайту testphp.vulnweb.com виявлено підозрілу конструкцію, цього разу атрибут script відповідає класу `Dynamic DOM Modification`. Рейтинг становив 16 із 166 перевірених елементів (рис. 3.6).

3.2 Визначення точності модифікованого алгоритму

Для створення збалансованої вибірки URL-адрес було зібрано по 100 прикладів безпечних і шкідливих ресурсів. Безпечні посилання взято з рейтингу top 50 most visited websites for October 2025 від Similarweb, тоді як шкідливі – з актуалізованої бази фішингових сторінок CyberHost Malware and Phishing Blocklist [32] та перевірених GitHub-репозиторіїв дослідницьких груп з кібербезпеки, зокрема PeterDaveHello/threat-hostlist [33].

Перевірка веб-сайтів проводилась на основі переліку небезпечних JS-функцій, поданих у таблиці 3.1.

Таблиця 3.1 – Перелік функцій загроз для тестування

JS функція	Опис
document.cookie	Кількість функцій document.cookie ()
window.localStorage	Кількість функцій window.localStorage ()
document.URL	Кількість функцій document.URL
getCookie	Кількість функцій getCookie
setCookie	Кількість функцій setCookie
document.writeln	Кількість функцій document.writeln
.innerHTML	Кількість подій .innerHTML
.outerHTML	Кількість подій .outerHTML
.insertAdjacentHTML	Кількість подій .insertAdjacentHTML
document.URL	Кількість функцій document.URL
location.assign	Кількість функцій location.assign
location.replace	Кількість функцій location.replace
getAppName	Кількість функцій getAppName
getUserAgent	Кількість функцій getUserAgent
setInterval	Кількість функцій setInterval
setTimeout	Кількість функцій setTimeout

У таблиці 3.2 наведено підсумки роботи алгоритму на цьому наборі: точність становить 96,86%, що на 0,04% краще, ніж показники, наведені в табл. 2.4.

Таблиця 3.2 – Показники ефективності алгоритму при тестуванні

Класифікатор	**Точність, %**	*FPR*	*FNR*	**ROC**
Дерево рішень	96,86	0,003	0,000	0,998

У таблиці 3.3 виконано зіставлення результативності алгоритмів на всіх використовуваних вибірках.

Таблиця 3.3 – Показники ефективності алгоритму на різних вибірках

Класифікатор, вибірка	**Точність, %**	*FPR*	*FNR*	**ROC**
Дерево рішень, табл. 2.1	99,67	0,005	0,000	0,998
Дерево рішень, табл. 2.1+ табл. А.1	99,82	0,0035	0,000	0,998
Дерево рішень, табл. 2.1+ табл. А.1+ табл. 3.1	99,91	0,0034	0,000	0,998

Варто відмітити, що в майбутніх дослідженнях варто врахувати також небезпечні JS-патерни, які реально зустрічаються: `new Function();` `setInterval();` `document.location=;` `atob()` тощо. А також зважати на небезпеку в атрибутах: `onload=;` `onerror=;` `onclick=;` `onfocus=;` `onmouseover=;` `onmouseenter=;` `onmouseleave=`; і навіть CSS, що викликає JS: `expression( background: url("javascript:...")).`

Додавання 16 авторських функцій до наборів 1 і 2 підвищило точність на 0,09%, при цьому частка хибно-позитивних та хибно-негативних спрацювань, а також форма кривої помилок залишилися стабільними.

3.3 Висновок до розділу

Дерево рішень – це математична модель, яка описує процес вибору оптимального варіанту дій, враховуючи можливі події до та після кожного рішення, а також наслідки різних варіантів виконання. Воно включає вузли рішень, вузли подій та кінцеві листки. У даній роботі реалізовано покращений варіант алгоритму дерева рішень, який протестовано на авторському наборі з 16 функцій. Він генерує підсумковий показник небезпечності та виділяє підозрілі частини коду. Рівень точності склав 96,86%, що на 0,04% перевищує результат тестів, проведених на Наборі 2.

Після об’єднання всіх функцій із різних наборів точність збільшується на 0,09%, а кількість хибнопозитивних і хибнонегативних спрацювань не змінюється. Додаткові тести підтвердили стабільність алгоритму на оновлених джерелах даних та сучасних веб-технологіях, включаючи динамічні SPA-інтерфейси та асинхронне завантаження скриптів.

ВИСНОВКИ

Створення JavaScript суттєво розширило інтерактивні можливості на стороні клієнта. Проте зловмисники активно зловживають динамічними особливостями цієї мови, впроваджуючи шкідливий код у веб-сторінки задля завантаження небажаних скриптів, перенаправлення на сторонні ресурси та прихованого відстежування дій користувача.

Провівши аналіз сучасної ефективності популярних алгоритмів виявлення, таких як наївний баєс, дерево рішень, випадковий ліс, метод опорних векторів, AdaBoost, REPTree та ADTree, можна констатувати, що вони демонструють приблизно однакову точність. Найнижчі результати показав наївний баєс, поступаючись конкурентам приблизно на 2-3 % у точності для обох конфігурацій.

Серед цих методів дерево рішень постійно демонструвало одні з найкращих значень за всіма метриками. Воно не потребує нормалізації або масштабування даних, а підготовка вхідних даних для нього вимагає значно менше коду, часу й зусиль у порівнянні з іншими алгоритмами. Концепція дерева рішень більш інтуїтивна та знайома багатьом розробникам, що полегшує розуміння його роботи й інтерпретацію результатів.

Завершальним етапом дослідження стало розроблення й удосконалення алгоритму на базі дерева рішень, протестованого на власному наборі з 16 ознак, характерних для JS-атак. Розроблений підхід генерує рейтинг шкідливості JavaScript-фрагментів, ідентифікує найбільш вразливі сегменти коду та досягає максимальної точності виявлення – до 99,91 %, що на 0,09 % перевищує показники попередніх моделей. При цьому рівні хибнопозитивних і хибнонегативних спрацьовувань, а також форма кривої помилок залишилися практично незмінними.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1 Самойлова Т.В. Удосконалений алгоритм захисту від JavaScript сценаріїв : кваліфікаційна робота за освітнім рівнем «магістр» : 123 Комп'ютерна інженерія / Самойлова Т.В. – Київ, 2020. – 64 с.

2 Рейтинг мов програмування 2025. TypeScript і Python — найпопулярніші, частки C# та Java зменшуються. URL : (дата звернення 30.09.2025).

3 Найпопулярніші мови програмування в українському IT: TypeScript обійшла JavaScript, а частки C# та Java продовжують падати. URL: https://itc.ua/ua/novini/najpopulyarnishi-movy-programuvannya-v-ukrayinskomu-it-typescript-obijshla-javascript-a-chastky-c-ta-java-prodovzhuyut-padaty (дата звернення: 30.09.2025).

4 Рейтинг мов програмування 2025: TypeScript обійшов JavaScript. URL: https://scroll.media/2025/02/10/rejtyng-mov-programuvannya-2025-typescript-obijshov-javascript/?utm_source=chatgpt.com (дата звернення: 30.09.2025).

5 Рейтинг мов програмування 2025 на Stackoverflow. URL: https://notatka.com.ua/praktyka/reiting-mov-programuvannia-2025-na-stackoverflow?utm_source=chatgpt.com (дата звернення: 30.09.2025).

6 Exploring JavaScript (ES2025 Edition). URL : https://exploringjs.com/js/downloads/exploring-js-book-preview.pdf (дата звернення: 30.09.2025)

7 What is the Deal with Javascript Versions? URL: https://levelup.gitconnected.com/what-is-the-deal-with-javascript-versions-88334491906c (дата звернення: 30.09.2025).

8 JavaScript Versions. URL: https://www.w3schools.com/js/js_versions.asp (дата звернення: 30.09.2025).

9 ECMAScript Daily News. URL: https://ecmascript-daily.github.io/archive (дата звернення: 30.09.2025).

10 The 38 Best JavaScript Libraries and Frameworks. URL : https://kinsta.com/blog/javascript-libraries/ (дата звернення: 30.09.2025).

11 JavaScript Library Web Usage Distribution in the Top 100k sites. URL: https://trends.builtwith.com/javascript/javascript-library/traffic/Top-100k (дата звернення: 18.11.2025).

12 Usage statistics and market shares of JavaScript libraries. URL: https://w3techs.com/technologies/overview/javascript_library (дата звернення: 18.11.2025).

13 Advantages of Using JavaScript in Web Technologies. URL: https://dev.to/javablog/advantages-of-using-javascript-in-web-technologies-1c4g (дата звернення: 10.10.2025).

14 Rodriguez S. Start-up says 80% of its Facebook ad clicks came from bots. Los Angeles Times. July 31, 2012.

15 Sengupta S. Bots Raise Their Heads Again on Facebook. New York Times. July 31, 2012.

16 3 Ways JavaScript Can Breach Your Privacy & Security. URL : https://www.makeuseof.com/tag/3-ways-javascript-can-used-breach-privacy-security/ (дата звернення: 10.10.2025).

17 A simple overview of the Twitter StalkDaily virus. URL : https://www.net-workworld.com/article/2235261/a-simple-overview-of-the-twitter-stalkdaily-vi-rus.html (дата звернення: 28.10.2025).

18 The Advantages and Disadvantages of JavaScript. URL : https://www.freecodecamp.org/news/the-advantages-and-disadvantages-of-javascript/ (дата звернення: 28.10.2025).

19 Internet Engineering Task Force. JSON Web Token (JWT). URL : https://tools.ietf.org/html/rfc7519 (дата звернення: 29.10.2025).

20 Same Origin Policy. URL : https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy (дата звернення: 29.10.2025).

21 Security Zones and Signed Scripts. URL : https://doc-store.mik.ua/orelly/webprog/jscript/ch21_04.htm (дата звернення: 29.10.2025).

22 NaiveBayesClassifier. URL : https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c (дата звернення: 29.10.2025).

23 J48 decision Tree. URL : http://data-mining.business-intelligence.uoc.edu/home/j48-decision-tree. (дата звернення: 30.10.2025).

24 Understanding Random Forest. URL : https://towardsdatascience.com/understanding-random-forest-58381e0602d2 (дата звернення: 30.10.2025).

25 The alternating decision tree learning algorithm. URL : https://cseweb.ucsd.edu/~yfreund/papers/atrees.pdf (дата звернення: 30.10.2025).

26 Naive Bayes Explained : Function, Advantages. URL : https://www.upgrad.com/blog/naive-bayes-explained/ (дата звернення: 30.10.2025).

27 Random Forest Algorithm Advantages and Disadvantages. URL : https://dhirajkumarblog.medium.com/random-forest-algorithm-advantages-and-dis-advantages-1ed22650c84f (дата звернення: 10.11.2025).

28 SVMasa technique. URL: https://core.ac.uk/down-load/pdf/188978526.pdf (дата звернення: 10.11.2025).

29 Aguide to AdaBoost : Boosting To Save The Day. URL : https://blog.paperspace.com/adaboost-optimizer/ (дата звернення: 10.11.2025).

30 Class REPTree. URL : https://weka.sourceforge.io/doc.dev/weka/classifiers/trees/REPTree.html (дата звернення: 20.11.2025).

31 The top 50 most visited websites for October 2025. URL : https://www.similarweb.com/ru/top-websites/ (дата звернення: 25.11.2025).

32 CyberHost Malware and Phishing Blocklist. URL : https://cyberhost.uk/malware-blocklist/ (дата звернення: 25.11.2025).

33 PeterDaveHello / threat-hostlist. URL : https://github.com/PeterDaveHello/threat-hostlist/ (дата звернення: 25.11.2025).

ДОДАТКИ

Додаток А

Таблиця А.1 – Функції відомих загроз (набір 2)

JS функція	Опис
search()	Кількість функцій search ()
split()	Кількість функцій split ()
onbeforeunload	Кількість подій onbeforeunload
onload	Кількість подій onload
onerror()	Кількість функцій onerror ()
onunload	Кількість подій onunload
onbeforeload	Кількість подій onbeforeload
onmouseover	Кількість подій onmouseover
dispatchEvent	Кількість подій dispatchEvent
fireEvent	Кількість подій fireEvent
setAttribute()	Кількість функцій setAttribute ()
window.location()	Кількість функцій window.location ()
charAt()	Кількість функцій charAt ()
console.log()	Кількість функцій console.log ()
.js	Кількість зовнішніх файлів JS
.php	Кількість файлів .php
var	Кількість ключових слів var, використовуваних в JS
function	Кількість ключових слів функцій, використовуваних в JS
Math.random()	Кількість функцій Math.random ()
charCodeAt()	Кількість функцій charCodeAt ()
WScript	Кількість WScript, використовуваних в JS
decode()	Кількість функцій decode ()
toString()	Кількість функцій toString ()
No. of Digits	Кількість цифр, що використовуються в JS
No. of Encoded Characters	Кількість закодованих символів, використовуваних в JS
No. of backslash Characters	Кількість символів /, використовуваних в JS
No. of Pipe Characters	Кількість символів вертикальної риси (|), використовуваних в JS
No. of % Characters	Кількість символів %, використовуваних в JS
No. of ‘(‘ Characters	Кількість символів '(', використовуваних в JS
No. of ‘)’ Characters	Кількість символів ')', які використовуються в JS

кінець табл. А.1

JS функція	**Опис**
No. of ‘,’ Characters	Кількість символів ',', використовуваних в JS
No. of ‘#’ Characters	Кількість символів '#', використовуваних в JS
No. of ‘+’ Characters	Кількість символів '+', використовуваних в JS
No. of ‘.’ Characters	Кількість '.' символів, які використовуються в JS
No. of ‘ Characters	Кількість символів ', використовуваних в JS
No. of ‘[‘ Characters	Кількість символів '[', використовуваних в JS
No. of ‘]’ Characters	Кількість символів ']', використовуваних в JS
No. of ‘{‘ Characters	Кількість символів '{', використовуваних в JS
No. of ‘}’ Characters	Кількість '}' символи, використовувані в JS
Share of Encoded characters	Частка закодованих символів в JS
Share of Digits characters	Частка десяткових знаків
Share of Hex/Octal characters	Частка шістнадцятирічних / вісімкових знаків
Share of Backslash characters	Частка «/» знаків
Share of Pipe (|) characters	Частка «|» знаків
Share of % characters	Частка «%» знаків

Додаток Б

XssDetector. Код програми

```
using System;
using System.Linq;
using System.Threading;
using System.Windows.Forms;

public abstract class Decision
{
    public abstract Decision Evaluate(HtmlElement elem);
}

public class DecisionResult : Decision
{
    public int MaliciousRating { get; set; }
    public bool Result => MaliciousRating > 0;

    public override Decision Evaluate(HtmlElement elem)
    {
        Console.WriteLine($"\nMalicious: {(Result ? "YES" :
"NO")}, Rating = {MaliciousRating}");
        return this;
    }
}

public class DecisionQuery : Decision
{
    public string Title { get; set; }
    public Func<HtmlElement, bool> Test { get; set; }
    public Decision Positive { get; set; }
    public Decision Negative { get; set; }

    public override Decision Evaluate(HtmlElement elem)
    {
        bool result = Test(elem);
        Console.WriteLine($"\n\t{Title}?  {(result ? "YES" :
"NO")}");
        return result ? Positive.Evaluate(elem) :
Negative.Evaluate(elem);
    }
}

public static class XssDetector
{
    private static readonly string[] cookiesAccess =
```

```
 {
        "document.cookie",
        "window.localStorage",
        "document.URL",
        "getCookies",
        "gotCookie"
    };

    private static readonly string[] dynamicDOM =
    {
        "document.write",
        "document.writeln",
        "insertAdjacentHTML"
    };

    private static readonly string[] urlManipulations =
    {
        "document.URI",
        "location.assign",
        "location.replace"
    };

    private static readonly string[] webBrowserAttributes =
    {
        "getAppName",
        "getUserAgent"
    };

    private static readonly string[] dynamicCode =
    {
        "setInterval",
        "setTimeout",
        "eval"
    };

    public static DecisionQuery BuildDecisionTree()
    {
        var dynamicCodeBranch = new DecisionQuery
        {
            Title = "Dynamic Code Construction",
            Test = elem => dynamicCode.Any(c =>
elem.OuterHtml.Contains(c)),
            Positive = new DecisionResult { MaliciousRating = 2
},
            Negative = new DecisionResult { MaliciousRating = 0
}
```

```
    };

        var webBrowserBranch = new DecisionQuery
        {
            Title = "Web Browser Attributes Leak",
            Test = elem => webBrowserAttributes.Any(c =>
elem.OuterHtml.Contains(c)),
            Positive = new DecisionResult { MaliciousRating = 4
},
            Negative = dynamicCodeBranch
        };

        var urlManipulationBranch = new DecisionQuery
        {
            Title = "URL Manipulation",
            Test = elem => urlManipulations.Any(c =>
elem.OuterHtml.Contains(c)),
            Positive = new DecisionResult { MaliciousRating = 6
},
            Negative = webBrowserBranch
        };

        var dynamicDOMBranch = new DecisionQuery
        {
            Title = "Dynamic DOM Modification",
            Test = elem => dynamicDOM.Any(c =>
elem.OuterHtml.Contains(c)),
            Positive = new DecisionResult { MaliciousRating = 8
},
            Negative = urlManipulationBranch
        };

        var cookieAccessBranch = new DecisionQuery
        {
            Title = "Possible Cookie Leak",
            Test = elem => cookiesAccess.Any(c =>
elem.OuterHtml.Contains(c)),
            Positive = new DecisionResult { MaliciousRating = 10
},
            Negative = dynamicDOMBranch
        };

        return cookieAccessBranch;
    }
}
```

```
class Program
{
    [STAThread]
    static void Main(string[] args)
    {
        if (args.Length == 0)
        {
            Console.WriteLine("Usage: xssdetector.exe <url>");
            return;
        }

        Console.WriteLine($"Testing {args[0]} for XSS...");

        Thread t = new Thread(() =>
        {
            var browser = new WebBrowser();
            browser.DocumentCompleted += (s, e) =>
            {
                var root = XssDetector.BuildDecisionTree();
                foreach (HtmlElement el in browser.Document.All)
                {
                    root.Evaluate(el);
                }
            };

            browser.ScriptErrorsSuppressed = true;
            browser.Navigate(args[0]);
            Application.Run();
        });

        t.SetApartmentState(ApartmentState.STA);
        t.Start();
        t.Join();
    }
}
```

БІБЛІОГРАФІЧНА ДОВІДКА

Тема дипломної роботи: "Удосконалення алгоритму виявлення шкідливих сценаріїв JavaScript на основі дерева рішень"

Обсяг пояснювальної записки 57 аркушів.

12 рисунки;

8 таблиці;

2 додатки.

Дата завершення роботи: *10 грудня 2025 р.*

Підпис студента-дипломника _____ *Галій І. А.*