# БАКАЛАВРСЬКА РОБОТА

## ДРБ. ІП – 09.00.00.000 ПЗ

**Група ІП-21-1**

**Грицило Максим**

**2025**

**Івано-Франківський національний технічний університет нафти і газу**

**Інститут інформаційних технологій**

**Кафедра інженерії програмного забезпечення**

**Грицило Максим Ігорович**

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

# БАКАЛАВРСЬКА РОБОТА

**Аналіз безпеки програмного забезпечення з точки зору мови реалізації**
(назва роботи)

Інженерія програмного забезпечення

( назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

**Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:**

Здобувач освітнього ступеня _____Грицило М.І._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____Пасєка Надія Мирославівна, к.т.н., доцент_____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

**Допущено до захисту**
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

**Івано-Франківськ – 2025**

**Івано-Франківський національний технічний університет нафти і газу**

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

**ЗАТВЕРДЖУЮ:**
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2025 р.

# ЗАВДАННЯ
## НА ДИПЛОМНУ РОБОТУ БАКАЛАВРА СТУДЕНТОВІ

Грицило Максим Ігорович

(прізвище, ім’я, по-батькові)

**1. Тема проекту (роботи) "Аналіз безпеки програмного забезпечення з точки зору мови реалізації"**

керівник проекту (роботи) к.т.н., доцент Пасєка Надія Мирославівна

затверджені наказом вищого навчального закладу від “ 28 ” квітня 2025 р. № 264/7

**2. Строк подання студентом проекту (роботи)** 10 червня 2025 р.

**3. Вихідні дані до проекту (роботи)** Результати і матеріали отримані під час проходження переддипломної практики

**4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)**

1 Теоретичні засади використання великих мовних моделей для виявлення уразливостей у програмному забезпеченні

2 Організація даних і підхід до впровадження моделі в реальній середовищі

3 Оцінка ефективності моделі, її впровадження та практичне застосування

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

1 Інструменти статичного/динамічного автоматичного/ручного аналізу програм (рис. 1.1, ст.19)

2. Фазинг-тестування (рис. 1.2, ст.20)

3. Кількість фрагментів коду на мову програмування (рис. 2.2, ст.25)

4. Матриця загальної плутанини (рис. 3.1, ст.38)

5. Діаграма потоку даних конвеєра CI/CD (рис. 3.2, ст.40)

**6. Консультанти розділів проекту (роботи)**

| Розділ | Консультант | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| | | | |
| | | | |

**2. Дата видачі завдання** **2025 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

# КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів дипломного проекту (роботи) | Строк виконання етапів проекту | Примітка |
|---|---|---|---|
| 1 | Визначення та обґрунтування теми роботи | 15.02.2025 | виконано |
| 2 | Огляд існуючих концепцій, рішень та сервісів в даній області | 25.02.2025 | виконано |
| 3 | Побудова моделі або алгоритму власного рішення | 15.03.2025 | виконано |
| 4 | Документування реалізації власного оригінального рішення вибраними засобами | 25.04.2025 | виконано |
| 5 | Оформлення пояснювальної записки бакалаврської роботи | 10.06.2025 | виконано |

Студент _____
(підпис)

Керівник роботи _____
(підпис)

# АНОТАЦІЯ

Бакалаврська робота містить 50 сторінок, 8 рисунків, 2 таблиці, список використаних джерел із 24 найменування,

**Метою роботи є** проаналізувати рівень безпеки програмного забезпечення залежно від мови реалізації, виявити типові вразливості, притаманні певним мовам, та оцінити ефективність підходів до автоматизованого виявлення та усунення цих уразливостей.

**Об'єкт дослідження:** Програмне забезпечення як цілісна система, що підлягає аналізу на наявність уразливостей.

**Предмет дослідження:** Вплив мови програмування на безпеку програмного забезпечення, зокрема з точки зору ймовірності появи уразливостей та ефективності виявлення загроз.

**Результати дослідження:** Досліджено питання гетерогенних обчислень та їх інтеграції в процес автоматизованого спільного проектування апаратного та програмного забезпечення.

**В першому розділі** представлено ефективні практики побудови датасетів та розгортання моделей у реальних середовищах, в тому числі в CI/CD-конвеєрах.

**В другому розділі** здійснено детальний процес збору, аналізу та попередньої обробки даних, що дозволило сформувати якісний та збалансований набір для подальшого навчання моделей виявлення уразливостей.

**В третьому розділі** проведено комплексний аналіз продуктивності моделі виявлення уразливостей у коді з використанням ключових метрик: точності, точності передбачень, повноти, F1-оцінки та матриці плутанини.

**Висновок:** Дослідження доводить, що великі мовні моделі, мають значний потенціал для виявлення уразливостей у високорівневих мовах програмування. Їх здатність до аналізу контексту, гнучкість у налагодженні та інтеграції в роблять їх потужним інструментом у сфері безпеки програмного забезпечення.

**КЛЮЧОВІ СЛОВА: УРАЗЛИВОСТІ, МАШИНЕ НАВЧАННЯ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ІНТЕРФЕЙС, ВЕБ-САЙТ**

# ANNOTATION

The bachelor's thesis contains 50 pages, 8 figures, 2 tables, a list of used sources with 24 names,

**The purpose of the work** is to analyze the level of software security depending on the implementation language, identify typical vulnerabilities inherent in certain languages, and evaluate the effectiveness of approaches to automated detection and elimination of these vulnerabilities.

**Object of research:** Software as a holistic system subject to analysis for the presence of vulnerabilities.

**Subject of research**: The impact of programming language on software security, in particular from the point of view of the probability of vulnerabilities and the effectiveness of threat detection.

**Research results:** The issue of heterogeneous computing and its integration into the process of automated joint design of hardware and software is investigated.

**The first section presents** effective practices for building datasets and deploying models in real environments, including in CI/CD pipelines.

**The second section provides** a detailed process of data collection, analysis, and preprocessing, which allowed us to form a high-quality and balanced set for further training of vulnerability detection models.

**The third section provides** a comprehensive analysis of the performance of the vulnerability detection model in the code using key metrics: accuracy, prediction accuracy, completeness, F1-score, and confusion matrix.

**Conclusion:** The study proves that large language models have significant potential for detecting vulnerabilities in high-level programming languages. Their ability to analyze context, flexibility in debugging, and integration make them a powerful tool in the field of software security.

**KEYWORDS: VULNERABILITIES, SOFTWARE, MACHINE LEARNING, INTERFACE, WEBSITE,**

**ЗМІСТ**

## ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

**NLP** - обробки природної мови (

**EDA** - пошуковий аналіз даних

**ML** - Машинне навчання

**XSS** - міжсайтовий сценарій (

**LLM** - моделей великого масштабу

**CI** - безперервної інтеграції

**CD** - безперервного розгортання

**DAST** - динамічне тестування безпеки додатків

**AST** - абстрактного синтаксичного дерева

**CFG** - графіка потоку керування

**NVD** - Національна база даних про вразливості

**OWASP** - Open Web Application Security Project

# ВСТУП

Актуальність теми зумовлена, тим що у сучасному світі все працює на програмному забезпеченні - наші телефони, автомобілі та навіть наші холодильники. Оскільки суспільства стають все більш залежними від цих цифрових систем, ставки стають вищими, коли щось йде не так. Одна вразливість у частині програмного забезпечення може дозволити зловмисникам використати всю систему, потенційно призводячи до вторгнень, відмови в обслуговуванні або навіть масового витоку даних. Наприклад, у 2014 році була виявлена вразливість, пізніше названа *Heartbleed* [1] , яка дозволяла зловмисникам отримати доступ до секретів приблизно 24-55% популярних веб-сайтів, захищених HTTPS [21]. Це викликає дедалі більше занепокоєння, і, на жаль, стає все більш поширеним.

Оскільки це занепокоєння зростало, дослідники почали досліджувати способи запобігання вразливості. Одна з перших ідентифікаційних схем, починаючи з 1976 року, полягала в тому, щоб розробники вручну перевіряли код, намагаючись знайти помилки чи інші недоліки [22]. Потім, у 1990 році, було розроблено динамічну роботу програми та подачу певним методам випадкових даних як засіб тестування [13], пізніше відомий як *Fuzzing.* Ще зовсім недавно, у 1995 році, інструменти для перевірки коду без його запуску для пошуку помилок і слабких місць, відомі як статичний аналіз, почали набирати обертів [28].

Традиційно люди вручну переглядають код або використовують ці інструменти, щоб спробувати виявити ці вразливості. Моделі AI, особливо великі мовні моделі (LLM), які можуть читати, контекстуалізувати та розуміти код, мають потенціал для автоматизації процесу виявлення цих слабких місць до того, як зловмисники зможуть до них дістатися.

Більшість досліджень наразі були зосереджені на C і C++, мовах, які сумно відомі тим, що їх важко захистити. А як щодо інших мов, на яких працюють сучасні веб-програми, проекти машинного навчання та мобільні програми, наприклад Python, JavaScript і Java? Незважаючи на ймовірність того,

що ці мови містять настільки ж ризиковані проблеми безпеки, існує значна прогалина в дослідженнях, спрямованих на виявлення слабких місць у цих мовах високого рівня.

Ця дипломна робота спрямована на розробку інструментів ШІ для виявлення вразливостей у вихідному коді. Незважаючи на попередні дослідження, існуючим моделям було важко точно передбачити вразливості коду.

**Метою роботи** є проаналізувати рівень безпеки програмного забезпечення залежно від мови реалізації, виявити типові вразливості, притаманні певним мовам, та оцінити ефективність підходів до автоматизованого виявлення та усунення цих уразливостей.

**Завданнями дослідження** є проаналізувати сучасні підходи до виявлення уразливостей у програмному запбезпеченні, дослідити можливості застосування великих мовних моделей, здійснити збір, підготовку та анотацію якісного набору даних, реалізувати тонке налаштування обраної моделі LLM для завдання виявлення вразливостей із фокусом на високорівневі мові програмування. Оцінити продуктивність налаштованої моделі за допомогою ключових метрик, порівняти результати моделі з традиційними підходами та попередніми дослідженнями, розгорнути модель в реальній середовищі CI/CD з інтеграцією в API та веб-інтерфейс, проаналізувати обмеження, етичні ризики та виклики.

**Результати дослідження:** Досліджено питання гетерогенних обчислень та їх інтеграції в процес автоматизованого спільного проектування апаратного та програмного забезпечення.

**Об'єкт дослідження:** програмне забезпечення як цілісна система, що підлягає аналізу на наявність уразливостей

**Предметом** дослідженн: вплив мови програмування на безпеку програмного забезпечення, зокрема з точки зору ймовірності появи уразливостей та ефективності виявлення загроз.

**Методи дослідження**: аналіз літератури та існуючих рішень, було здійснено збір і очищення, використано експериментальний підхід, із застосуванням методів регуляризації, стратифікованої валідації та оптимізації гіперпараметрів., експериментальне тестування продуктивності, інтеграційне моделювання, аналіз обмежень і етичних аспектів.

Досліджено питання гетерогенних обчислень та їх інтеграції в процес автоматизованого спільного проектування апаратного та програмного забезпечення.

Бакалаврська робота містить 50 сторінок, 8 рисунків, 2 таблиці, 3 розділи, список використаних джерел із 24 найменуванням.

# РОЗДІЛ 1. КОНТЕКСТ ТА МОТИВАЦІЯ ДОСЛІДЖЕННЯ

## 1.1 Мотивація та актуальність використання LLM для виявлення уразливостей у програмному забезпеченні»

Із зростаючою складністю систем програмного забезпечення та цифрової інфраструктури, проблеми безпеки набувають дедалі більшої значущості. Уразливості, що виникають у вихідному коді, стали критичними проблемами, які можуть мати серйозні наслідки для функціонування компаній, організацій і навіть цілих державних структур. Вектори атак не стоять на місці: кіберзлочинці постійно розвивають нові методи, що дозволяють використовувати навіть незначні недоліки в коді для проникнення у систему. Такі вразливості можуть призвести до серйозних порушень безпеки, загрожуючи конфіденційності даних користувачів, а також фінансовій стабільності організацій.

В умовах швидкого розвитку цифрових технологій, де нові сервіси, додатки та інфраструктурні рішення з’являються практично щодня, обсяг і складність коду збільшуються в геометричній прогресії. В результаті, традиційні методи виявлення вразливостей, що ґрунтуються на ручній перевірці коду, стають неефективними і нездатними вчасно виявляти загрози. Збільшення обсягів коду та числа його компонентів робить процеси вручну малоефективними, що створює необхідність у нових, більш масштабованих рішеннях для автоматизації пошуку вразливостей.

Інструменти статичного та динамічного аналізу, такі як Checkmarx і Acunetix, продемонстрували свою ефективність у виявленні уразливостей в коді, проте, незважаючи на значний прогрес, вони стикаються з певними обмеженнями, такими як проблеми з масштабованістю, тривалими часом виконання і великими витратами ресурсів при обробці великих обсягів коду. Це створює величезний попит на нові методи, здатні ефективно і швидко працювати з великими обсягами даних і враховувати постійно змінювану природу загроз.

Машинне навчання (ML) надає потужні інструменти для автоматизації

процесу виявлення вразливостей. Одним із важливих досягнень останніх років є використання моделей великого масштабу (LLM), таких як GPT і BERT, для аналізу кодових баз і виявлення потенційних проблем у програмному забезпеченні. Ці моделі, навчені на величезних масивах текстових даних, можуть не лише генерувати код, але й виявляти помилки, які можуть бути важкими для традиційних інструментів. Проте використання LLM для виявлення уразливостей у програмному забезпеченні має свої труднощі. Тонке налаштування цих моделей для специфічних завдань безпеки, зокрема для виявлення конкретних вразливостей у різних мовах програмування, є складним і вимагає значних зусиль і ресурсів.

Мотивація для цієї роботи полягає в необхідності дослідження того, як можна оптимізувати тонке налаштування моделей LLM для виявлення вразливостей саме в тих мовах програмування, які рідко є об’єктами існуючих досліджень LLM. Оскільки мов програмування постійно еволюціонують і з’являються нові підходи до їх використання, а також через швидкий розвиток самих загроз безпеки, вивчення підходів, що базуються на даних, може стати ключем до того, щоб подолати існуючі розриви між теоретичними методами безпеки і реальними вимогами сучасних програмних систем. Використання LLM дозволяє не тільки підвищити ефективність виявлення вразливостей, а й знизити витрати на обробку великих обсягів коду, забезпечуючи більш швидку реакцію на нові загрози і підвищуючи загальний рівень безпеки.

## 1.2 Застосування великих мовних моделей для виявлення уразливостей у високорівневих мовах програмування

Ця робота досліджує потенціал тонкої настройки LLM для виявлення - вразливостей безпеки в мовах програмування, відмінних від C/C++. Хоча велика частина існуючих досліджень зосереджена на вразливостях, пов’язаних із C і C++, мови вищого рівня, які широко використовуються в сучасних середовищах розробки, також становлять значні ризики. Python, JavaScript і Java, зокрема, все

частіше використовуються в різних сферах, від веб-розробки до машинного навчання, що робить їх привабливими цілями для зловмисників.

Першою метою цієї роботи є використання можливостей LLM для аналізу коду на наявність загальних уразливостей безпеки в цих мовах вищого рівня. Це включає в себе підготовку наборів даних уразливих і невразливих зразків коду, точне налаштування попередньо навчених LLM для виконання завдання та оцінку продуктивності моделі на невидимих даних. Виявляючи вразливості на ранніх стадіях циклу розробки, налаштовані моделі можуть стати корисним інструментом для розробників, підвищуючи безпеку програмного забезпечення без значного збільшення часу розробки.

Використання моделі В якості останнього кроку буде створено веб-сайт і конвеєр для безперервної інтеграції (CI) / безперервного розгортання (CD) для подальшого підвищення зручності використання моделей. Для цього необхідно створити API, і користувачі повинні мати можливість автентифікуватися, щоб контролювати використання моделей. API має мати можливість приймати функції коду та робити прогноз щодо того, чи є дана функція коду вразливою чи ні. Причина надання кодових функцій моделі, на відміну від цілої кодової бази, пов’язана з технічними обмеженнями щодо того, скільки даних може обробити LLM. З іншого боку, також неможливо зробити прогноз щодо вразливості на основі одного рядка коду, оскільки модель повинна мати достатньо контексту, щоб зробити точний прогноз. Таким чином, збереження його на рівні функції коду забезпечує хороший баланс між технічними обмеженнями, надаючи достатній контекст.

Питання дослідження Мета дипломної роботи може бути зведена до наступних питань дослідження:

- ЗП1: Наскільки ефективні моделі ШІ у виявленні вразливостей у мовах програмування високого рівня? — Як результати порівнюються з традиційними методами виявлення вразливостей щодо швидкості, точності та простоти використання?
- Запитання 2: Які проблеми та переваги інтеграції моделей

виявлення вразливостей штучного інтелекту в конвеєри безперервної інтеграції/безперервного розгортання в GitHub і GitLab?

### 1.3 Виявлення вразливостей програмного забезпечення

Виявлення вразливостей — це превентивний захід, який вживається для зменшення ризиків безпеки програмного рішення, спрямований на виявлення потенційних недоліків безпеки в коді, якими можуть скористатися зловмисники. Відповідно до Chu et. ін., трьома найпоширенішими методами виявлення вразливості є фаз-тестування, символічне виконання та формальна перевірка [15]. Ці підходи, як правило, нелегко виконати, оскільки ефективне тестування фазз вимагає як часу, так і глибоких знань про програмне забезпечення, яке фазізується [6], символічне виконання потребує великих обсягів пам’яті та іноді неможливе для великих кодових проектів [4], а формальна перевірка є складним, трудомістким і дорогим процесом [37]. Графік різних підходів до аналізу програм можна побачити на рисунку 1.1 [16], де показано співвідношення між статичністю/динамічністю та обсягом ручної роботи, необхідної для використання різних інструментів. Деякі з цих підходів будуть обговорені далі.

Динамічний аналіз, також відомий як динамічне тестування безпеки додатків (DAST), — це набір методів перевірки програм під час виконання для виявлення потенційних вразливостей [46]. Оскільки існує збіг між тестуванням функціональності/помилок і тестуванням безпеки, деякі з цих інструментів і методів добре відомі в індустрії розробників, наприклад модульне тестування [35], аналіз покриття коду за допомогою gcov 1 і детектори помилок пам’яті, такі як AddressSanitizer [61] і Valgrind [47].

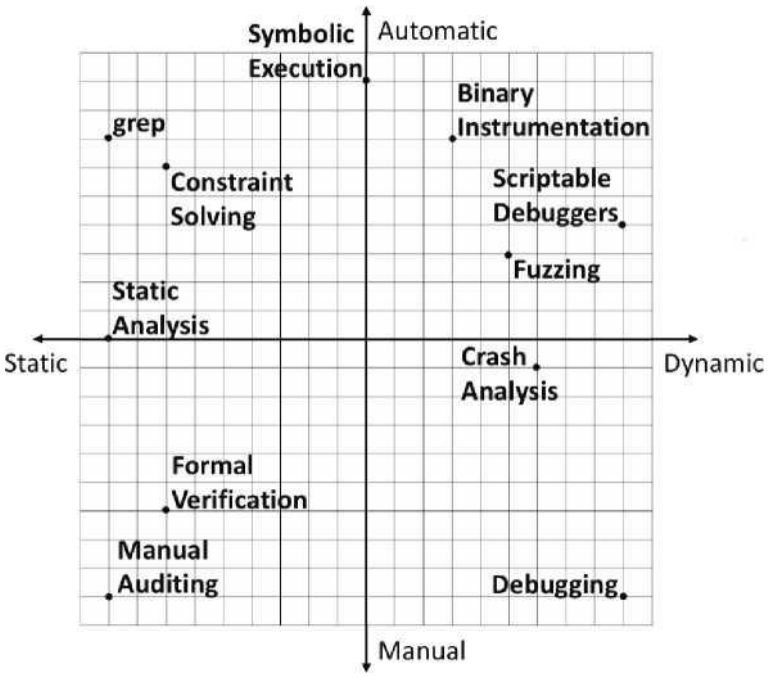


Рисунок 1.1 - Інструменти статичного/динамічного автоматичного/ручного аналізу програм [16].

Фаз-тестування, також відоме як фаззинг, — це концепція багаторазового надсилання ¬вхідних даних, згенерованих машиною, у функцію, щоб підтвердити відсутність проблем із пам’яттю незалежно від вхідних даних [7]. Механізм фаззингу, як правило, AFL або libFuzzer, буде безперервно контролювати охоплення гілки для кожного згенерованого вводу та на основі покриття коду змінюватиме вхідні дані, намагаючись ще більше збільшити охоплення гілок. Якщо новий вхід викликав більше граничних випадків у коді, він зберігається в корпусі, який механізм використовує в майбутніх сеансах фаззингу. Якщо трапиться збій, введені дані буде збережено разом із інформацією про збій. Ця процедура зображена на рисунку 2.2. фаззинг є ефективною та надійною технікою для пошуку вразливих місць безпеки, у нього є свої труднощі. Фаззинг - це трудомісткий процес, ідея полягає в тому, щоб фаззер працював якомога довше, щоб знайти всі крайні випадки в програмі. Оскільки фаззер надсилає напіврандомізовані дані, він може ніколи не знайти певну помилку, якщо вона досить складна. Крім того, розробник, який створює джгути фаззингу, вимагає розуміння рішення в цілому.

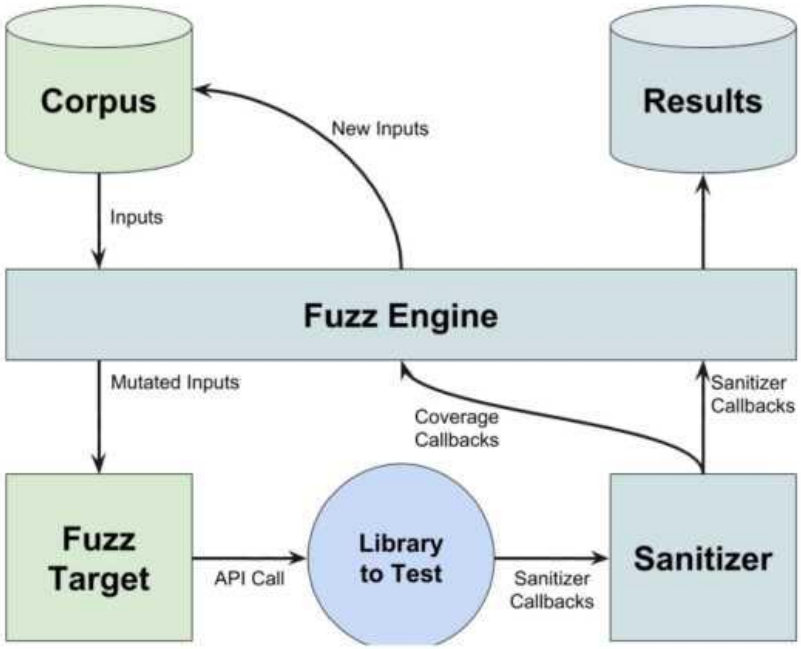


Рисунок 1.2 - Фазинг-тестування

**Статичний аналіз** — це загальний термін для кількох методів аналізу коду, таких як аналіз потоку даних, аналіз потоку керування, зіставлення шаблонів для поширених уразливостей і аналіз абстрактного синтаксичного дерева (AST). Усі ці методи об'єднують той факт, що код ніколи не виконується, що робить їх доступними для статичного аналізу.

**Аналіз потоку даних.** Ключова концепція аналізу потоку даних полягає в аналізі графіка потоку керування (CFG) програми, щоб визначити, які значення можуть мати певні змінні. Аналізуючи потік даних, можна виявити потенційні вразливості, такі як неініціалізовані змінні (досягнення визначень [64]) і зіпсовані дані користувача, які потенційно можуть призвести до ін'єкційних атак.

**Аналіз потоку керування**. За допомогою аналізу потоку керування моделюються можливі шляхи виконання, що дає змогу виявляти такі проблеми, як нескінченні цикли, недоступний код або неправильна обробка винятків [48], які можуть призвести до вразливості безпеки.

LLM є масштабованими та здатними аналізувати великі кодові бази з відносно швидкими результатами. Вони також спрощують процес виявлення вразливості, оскільки не потребують розширених налаштувань з точки зору користувача, що робить його більш доступним для розробників. Оскільки більше розробників застосують цей підхід, можна буде опублікувати більш безпечний код, зменшуючи ризик атак. Керовані даними підходи до виявлення вразливостей Керований даними підхід до виявлення вразливостей за допомогою LLM значною мірою залежить від наявності правильно позначених наборів даних коду. Такі проблеми, як незбалансованість даних, коли захищений код значно перевищує кількість незахищених зразків, і правильність міток, необхідно враховувати при створенні надійної моделі [56, 10]. Наприклад, якщо 99% даних є невразливим кодом і лише 1% є вразливим, модель класифікує весь код як невразливий, оскільки вона отримує високу точність своїх прогнозів. Під час обробки дисбалансу класів важливо підтримувати достатню кількість даних. Це накладає ряд проблем на використання даних. У цій роботи досліджується доцільність використання LLM для -виявлення вразливостей у мовах, відмінних від C/C++, шляхом їх точного налаштування на відповідних наборах даних та оцінки їх ефективності в реальних програмах.

**Системи класифікації вразливостей**. Для стандартизації ідентифікації та класифікації вразливостей існує кілька структур, насамперед CVE [10] і CWE . Ці структури забезпечують структури для виявлення та класифікації вразливостей у різних типах програмного забезпечення та середовищах розробки, допомагаючи у виявленні, запобіганні та виправленні. CVE є універсальним довідником для загальновідомих вразливостей. Система CVE, якою керує MITRE, призначає унікальні ідентифікатори вразливостям продуктів і програмного забезпечення, забезпечуючи точне відстеження. Кожен запис CVE ¬описує конкретну вразливість, включаючи її вплив, уражені версії та інформацію про виправлення, що полегшує швидке визначення ризиків у кодових базах. CWE каталогізує типи вразливостей на основі основних недоліків, які призводять до недоліків безпеки. На відміну від CVE, які

документують окремі випадки, CWE зосереджені на шаблонах і категоріях уразливостей, таких як CWE-89 (SQL Injection) і CWE-787 (Out-of-bound Write). Ця система класифікації є загальновизнаною та критично важливою для виявлення та групування вразливостей для цільових стратегій виправлення. Окрім CVE та CWE, інші системи сприяють управлінню вразливістю. Національна база даних про вразливості (NVD) надає детальні метадані для CVE, включаючи оцінки серйозності (з використанням балів CVSS ) і показники впливу. Крім того, Open Web Application Security Project (OWASP) перераховує найпопулярніші вразливості за поширеністю та ризиком, пропонуючи ресурси для пом'якшення вразливостей веб-додатків. Інструменти автоматичного виявлення вразливостей часто посилаються на ці фреймворки, зіставляючи виявлені недоліки зі списками CVE або CWE. Таке узгодження дозволяє розробникам розглядати ризики відповідно до серйозності та відповідати галузевим стандартам безпеки програмного забезпечення.

**Дилема C /C++** Наскільки нам відомо, немає жодного опублікованого дослідження, у якому велика мовна модель була б навчена для досягнення статистично хороших результатів у виявленні вразливості коду. Більшість досліджень у цій галузі базується на C та C++. Перевірка доступних даних уразливого коду вказує на деякі можливі причини. Однією з основних проблем є транзитивні вразливості [44], наприклад, функція може бути класифікована як вразлива в наборі даних, якщо функція викликає іншу вразливу функцію. Хоча виклик уразливої функції є проблемою безпеки, нерозумно стверджувати, що функція, відповідальна за здійснення виклику, є вразливою - виправлення фактичної здатності вразливості призведе до того, що функція виклику стане невразливою, не вимагаючи жодних змін. Крім того, кілька CWE частіше зустрічаються в мовах нижчого рівня, таких як C і C++, через пряме керування пам'яттю та відсутність вбудованого захисту. У мовах вищого рівня ці вразливості можуть бути зменшені або навіть не існувати. Деякі приклади включають CWE-416 Use After Free і CWE-476 Double Free, оскільки часто використовуються автоматизовані збирачі сміття, або CWE-787 Out-ofbounds

запис через вбудовані перевірки під час виконання, які створюють винятки . Для цієї роботи C/C++ не досліджуватиметься через попередню сувору перевірку кількома дослідниками. Натомість ця робота буде обмежена вивченням JavaScript, PHP, Java, Python і Go, оскільки існує велика кількість даних, що охоплюють ці мови, як описано в наступних розділах.

Концепція використання штучного інтелекту в безпечному кодуванні та в кібербезпеці загалом наразі ретельно перевіряється дослідниками. Цей розділ має на меті пролити світло на поточну дослідницьку ситуацію в цій галузі.

Збирати дані для цілей машинного навчання складно, оскільки машинне навчання значною мірою покладається на правильні та надійні дані. Найпоширеніший спосіб зібрати вразливий код – це скопіювати дані з бази даних CVE, наприклад NVD, і визначити зміну git, яка усунула вразливість. Цей підхід був використаний як Bhandari et.al., опублікованим як CVEFixes [5], так і Fan et.al., опублікованим як Big-Vul [23]. Чакраборті та ін. запропонували іншу методологію збору даних, де вони вибрали проекти Linux Debian Kernel і Chromium, а також перевірили -проблеми безпеки та їх виправлення відповідно до проблем GitHub і репозиторію Bugzilla відповідно, опублікованих як набір даних ReVeal [12]. Нікітопулос та ін. об’єднав методології ReVeal і CVEFixes/Big-Vul, як аналізуючи NVD, так і використовуючи аналіз проблем, випускаючи дані під набором даних CrossVul [20].

Хоча CVEFixes містить здебільшого точні дані (оскільки він збирає CVE та їхні передбачувані виправлення через історію git), він не має стільки даних, скільки Big-Vul. Однак Big-Vul містить дубльований код і погано підібраний, оскільки для своїх даних використовує кілька джерел. Аналіз наборів даних, виконаний Chen et.al. шляхом випадкової вибірки коду та ручного дослідження правильності мітки показали, що зі згаданих наборів даних CVEFixes мають найвищу коректність для мітки вразливості [14].

Машинне навчання (ML) у контексті виявлення вразливостей є відносно новою галуззю, яка зараз ретельно вивчається дослідниками. Дослідження, в якому запити API класифікувалися як шкідливі або не використовували різні

моделі ML, було проведено Піюшем Ранджаном [25]. Дослідження Ранджана порівняло деякі підходи до ML та їх ефективність у правильному виявленні зловмисних запитів. Порівнювали моделі: Дерево рішень , Випадковий ліс [27], Машина опорних векторів (SVM) [22] і Довга короткочасна пам'ять (LSTM) [33]. Дослідження показало, що LSTM є найефективнішим підходом ML для виявлення зловмисних запитів, правильно класифікуючи 95,6% запитів. Інше дослідження використання машинного навчання для виявлення вразливостей було виконано Welearegai et. al. [27], де кілька моделей намагалися класифікувати запити до різних пристроїв Raspberry Pi 11 як доброякісні або такі, що піддаються атаці, і діяти відповідно до класифікації. Вони досягли точності 75% для онлайн-моніторингу атак. У дослідженні, проведеному Ma et. al. [11], вони проаналізували AST програм JavaScript для виявлення шкідливого коду за допомогою Random Forest [27], SVM [12], XGBoost [13] і багаторівневого перцептрона (MLP) [1]. Використовуючи зворотне поширення через вузли AST, вони позначали сегменти коду як зіпсовані чи ні, вказуючи, чи було значення, надіслане до функції, зіпсовано (наприклад, змінено) цією функцією, а потім навчали різні архітектури моделей, щоб отримати порівняння того, як різні моделі виявляли вразливості. Використовуючи цю методологію, вони досягли точності виявлення шкідливого коду 85,60% за допомогою архітектури MLP і точності 96,40% за допомогою XGBoost.

**Тонка настройка -** це процес у машинному навчанні, коли попередньо навчена модель, спочатку навчена на великому загальному наборі даних, адаптується для конкретного завдання шляхом додаткового навчання на більш спеціалізованому наборі даних. Цей підхід використовує узагальнені знання, вбудовані в модель з попереднього навчання, і налаштовує її для більш вузьких застосувань. У контексті LLM тонке налаштування включає в себе коригування параметрів моделі для підвищення продуктивності цільового завдання, наприклад виявлення вразливостей у коді, шляхом навчання шаблонів, що стосуються домену, присутніх у даних міток. В останні роки тонке налаштування великих мовних моделей для виявлення вразливостей показало різні результати,

особливо для виявлення ризиків безпеки в мовах низького рівня, таких як C і C++. Тонка настройка дозволяє LLM, наприклад тим, що базуються на архітектурі трансформаторів, адаптуватися до конкретних контекстів безпеки. Дослідження Ding та ін. з набором даних PrimeVul [18], наприклад, продемонстрував ефективність таких моделей у виявленні вразливостей коду, використовуючи тонке налаштування підібраних наборів даних уразливих і невразливих фрагментів коду. PrimeVul є одним із найновіших, кращих результатів досліджень у цій галузі порівняно з іншими згаданими раніше дослідженнями. Помітні спроби тонкого налаштування були зосереджені на класифікації функцій як вразливих або невразливих. Однак також були спроби застосувати інші варіанти класифікації вразливості. Дослідження Atiiq та ін. [2] підкреслив, що точне налаштування LLM спеціально для типів уразливостей, класифікованих за ідентифікаторами CWE, значно покращує точність виявлення та дозволяє моделям розпізнавати вразливості за допомогою певних атрибутів, таких як впровадження SQL, індексування за межами або міжсайтовий сценарій. Незважаючи на ці досягнення, точне налаштування моделей для універсального виявлення вразливостей у кількох CWE та мовах залишається складним через відмінності в синтаксисі, проблеми безпеки та доступність даних для різних мов.

У дослідженні Dozono et. al. [19], вони виявили прогалину в дослідженнях у виявленні вразливостей, відмінних від C/C++, за допомогою LLM. Використовуючи техніку, відому як швидка праймінг , яка передбачає встановлення тону, стилю або структури для відповідей LLM, у поєднанні зі звичайним запитом, чи є код вразливим чи ні, вони запитували моделі GPT [8], моделі CodeLlama [28] і Gemini 1.5 Pro на мовах Python, C, C++, JavaScript і Java. Вони експериментували з різними підказками та досягли порівняно хороших результатів, мабуть, найкращою комбінацією підказок було: "Ви бінарний класифікатор уразливості штучного інтелекту, який визначає, чи є наданий код вразливим чи ні. Ви повинні відповісти лише "вразливим" або "не вразливим". + "Класифікуйте наступний код як вразливий або невразливий". Вихідні дані або лише «вразливий», або «не вразливий» . Вони вирішили не виконувати тонке

налаштування через вимогу значних обчислювальних ресурсів, але згадали, що точно налаштовані моделі зазвичай перевершують інші підходи машинного навчання. Після експериментів вони досягли точності виявлення вразливості коду від 47% (C) до 88% (JavaScript). Bae et. al. [3] спробував класифікувати код C++, Java і Python за допомогою GPT і Claude [8, 20]. Зокрема, вони використовували GPT-3.5 Turbo, GPT-4 Turbo, GPT-4o та Claude-3.5 Sonnet, щоб класифікувати код і визначити, чи містить він уразливість у певному класі CWE. Вони оцінювали три типи підказок; стислі підказки, підказки щодо налаштування наконечників і покрокові підказки. Вони визначають стислі підказки як: "Будь ласка, оцініть, чи містить наведений вище код уразливість CWE-***. Чи має цей код уразливість CWE-***? (Так/Ні). Оцініть свою впевненість у цій відповіді за шкалою від 0 до 100, де 100 означає "дуже впевнено", а 0 означає "зовсім не впевнено". У підказках щодо встановлення чайових вони додають перед підказкою «Я збираюся дати 500 тис. доларів за краще рішення», а в покрокових підказках додають «Давайте подумаємо крок за кроком». Використовуючи цю методологію, вони досягли максимальної точності 90,22% за допомогою GPT-4o та покрокових підказок, і мінімальної точності 54,95% за допомогою GPT-3.5 Turbo зі стислими підказками . У середньому Claude-3.5 Sonnet показав найкращі результати серед методологій, тоді як GPT-4o з покроковими підказками був найперспективнішим.

### 1.4 Висновок по розділу

З огляду на зростаючу складність сучасних програмних систем та стрімке розвиток цифрової інфраструктури, традиційні методи виявлення вразливостей виявляються недостатньо ефективними для масштабних, динамічних середовищ. Великі мовні моделі (LLM), зокрема GPT і подібні архітектури, демонструють значний потенціал у сфері автоматизованого аналізу безпеки, завдяки своїй здатності розуміти семантику та контекст програмного коду. Їх застосування для

виявлення уразливостей у високорівневих мовах програмування (JavaScript, Python, Java, PHP, Go) відкриває нові можливості для раннього виявлення загроз із мінімальним втручанням людини.

У дослідженні наголошено на важливості тонкого налаштування моделей LLM для підвищення точності прогнозів, а також представлено ефективні практики побудови датасетів та розгортання моделей у реальних середовищах, в тому числі в CI/CD-конвеєрах.

Розгляд сучасних підходів (DAST, фазинг, статичний аналіз) показує, що LLM здатні не лише доповнити існуючі інструменти, а й у перспективі перевершити їх завдяки гнучкості, масштабованості та здатності адаптуватися до нових типів загроз. Проте підкреслюються й обмеження, включаючи залежність від якісних анотованих даних, потребу в обчислювальних ресурсах та ризик неправильної інтерпретації контексту.

Таким чином, LLM - це не універсальне рішення, але потужний інструмент, здатний значно підвищити ефективність забезпечення безпеки коду в поєднанні з традиційними підходами та людською експертизою.

# РОЗДІЛ 2 . ОРГАНІЗАЦІЯ ДАНИХ І ПІДХІД ДО ВПРОВАДЖЕННЯ МОДЕЛІ В РЕАЛЬНІЙ СЕРЕДОВИЩІ

У цьому розділі описано процес збору, аналізу та підготовки даних для навчання та оцінювання. Він містить детальну інформацію про критерії вибору набору даних, пошуковий аналіз даних (EDA) і етапи попередньої обробки, щоб переконатися, що дані придатні для завдань машинного навчання.

## 2.1 Підготовка та обробка даних для навчання моделей виявлення уразливостей

У цьому розділі розглядається процес збору, аналізу та попередньої обробки даних, який є основою для подальшої роботи виявлення вразливостей безпеки за допомогою великих мовних моделей (LLM). Початкові набори даних, використані в цьому дослідженні, складаються з кількох важливих джерел, таких як Big-Vul [23], CVEFixes [5], CrossVul [20], ReVeal [12] і PrimeVul [18]. Кожен з цих наборів містить інформацію про вразливості та зміни в коді програм, однак для цілей цього дослідження був обраний тільки **CVEFixes**, оскільки інші набори мали обмеження або не містили достатньої кількості даних, що стосуються мов програмування високого рівня, як Python, JavaScript або Java.

Початкові набори даних, такі як Big-Vul**,** ReVeal і PrimeVul**,** зосереджені в основному на вразливостях у C і C++, що робить їх менш корисними для дослідження вразливостей у мовах програмування вищого рівня. У процесі збору даних з цих наборів було відфільтровано ті, що не відповідали вимогам дослідження, оскільки вони здебільшого містили лише дані для C і C++. У результаті було залишено два основні набори: CVEFixes та CrossVul**,** з яких CVEFixes був обраний для подальшої роботи через високий рівень повноти та точності міток.

Після початкового аналізу CrossVul було виявлено ряд проблем, таких як численні порожні файли та помилки типу «404: не знайдено», що значно

знижувало якість і корисність цього набору для дослідження. Крім того, за результатами дослідження Нікітопулоса та інших авторів [20] було вказано на можливі проблеми з мітками в наборі даних CrossVul**,** які могли призвести до неточностей у результатах виявлення вразливостей. Це, у поєднанні з аналізом CVEFixes, призвело до вибору цього набору як основного для подальшого використання, оскільки він містить детально мічені вразливості та зміни в коді, які стосуються не тільки C і C++, але й інших мов програмування.

CVEFixes містить базу даних SQL, що містить детальну інформацію про вразливості та виправлення в коді. Структура цієї бази даних зображена на діаграмі ER на рисунку 2.1.

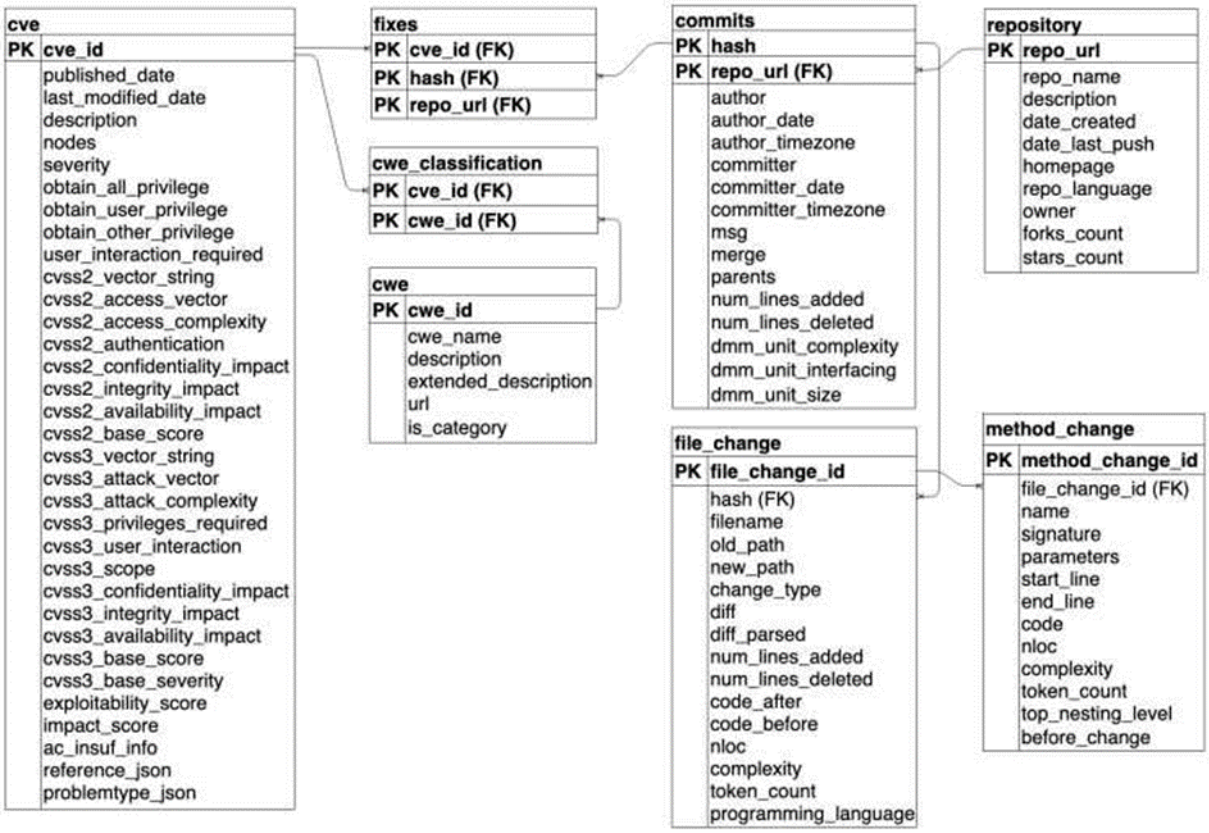


Рисунок 2.1 - Діаграма сутності-зв’язку з набору даних CVEFixes

Для витягу даних з цієї бази було використано SQL-запит (А.1) і бібліотеку Python Pandas [21], що дозволило зберігати дані у зручному форматі кадра даних. Всього в наборі CVEFixes було виявлено 16 123 випадків змін, де

не було вказано мову програмування, що потребували додаткової фільтрації. З них:

- 14 651 випадок змін у мові програмування C
- 5 409 випадків змін у мові програмування C++
- 1 випадок без зазначення мови програмування (None)

Ці непотрібні або некоректно мічені записи були відфільтровані для забезпечення точності подальших аналітичних процесів. Кількість записів для кожної мови програмування детально візуалізована на рисунку 2.2.

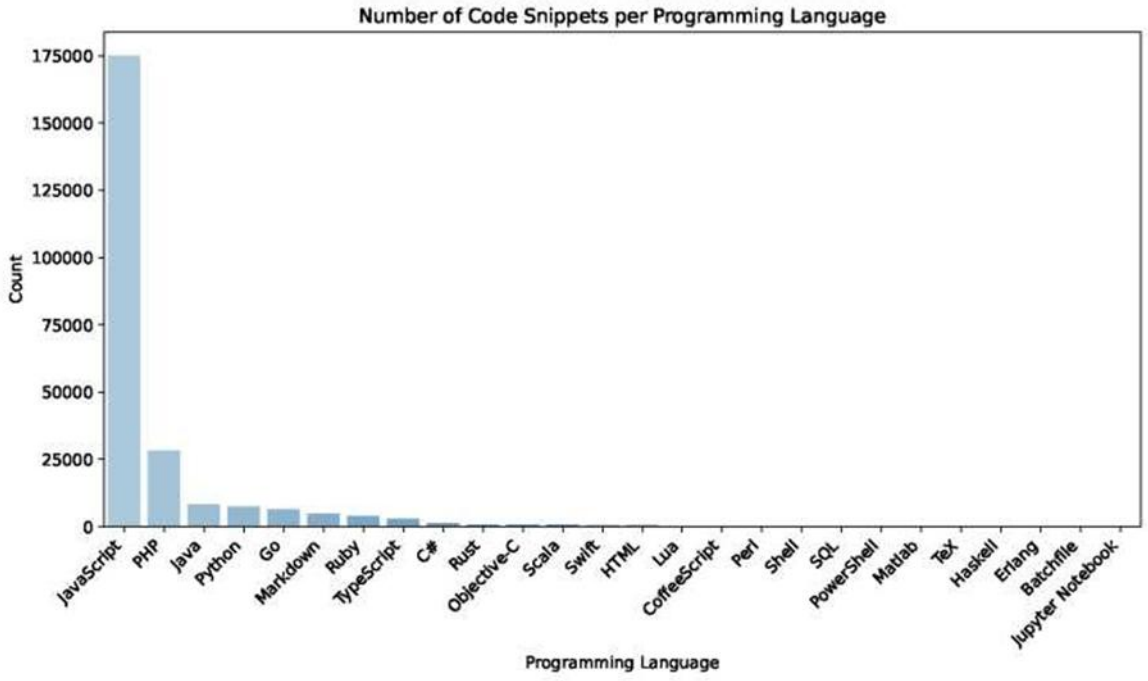


Рисунок 2.2 - Кількість фрагментів коду на мову програмування

Візуалізація даних дозволила з’ясувати, що найпоширенішими мовами у наборі CVEFixes є C і C++, проте значна частина даних стосується саме цих мов. Далі, для того щоб оцінити пропорційний розподіл вразливих та невразливих фрагментів коду серед десяти найбільш поширених мов програмування, була створена додаткова візуалізація на рисунку 2.3. На цьому рисунку показано, як часто зустрічаються вразливості та невразливості в коді для кожної з мов програмування, що дозволяє виявити мови, які потребують балансування даних

для забезпечення рівномірного представлення вразливих і невразливих прикладів.

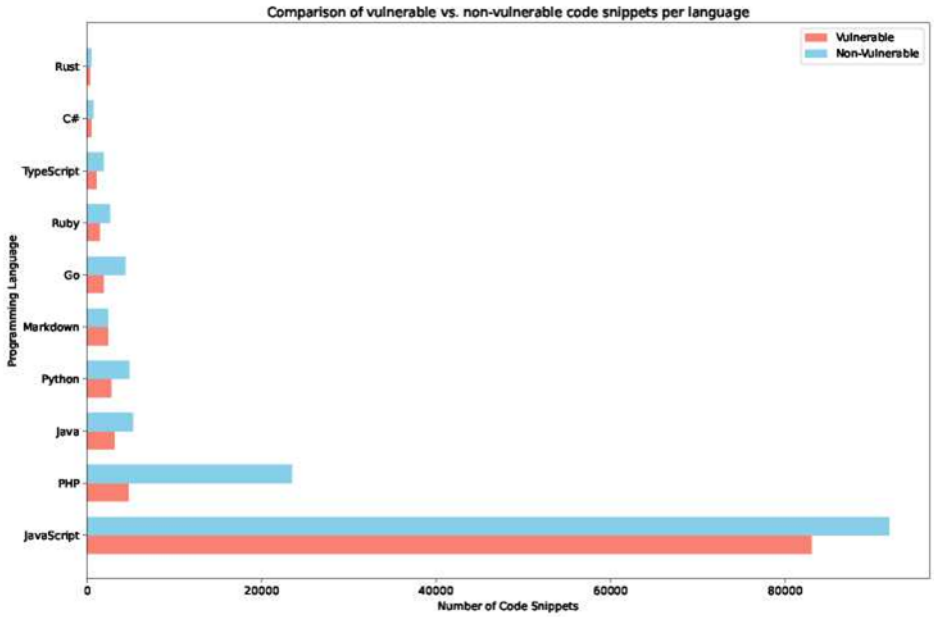


Рисунок 2.3 - Уразливі проти невразливих для десяти найпоширеніших мов.

Оскільки набір даних містить значний дисбаланс між кількістю прикладів вразливих та невразливих фрагментів коду для різних мов, необхідно було застосувати методи балансування даних для забезпечення рівномірного навчання моделей штучного інтелекту. Для цього використовувалися різноманітні техніки, такі як підвищення представлення меншості за рахунок генерації синтетичних прикладів за допомогою технік, таких як SMOTE (Synthetic Minority Over-sampling Technique), а також технології вибірки для зменшення числа більшості, що дозволяє уникнути переобучення на більшості даних.

Після виконання фільтрації та балансування даних наступним кроком була їх підготовка для використання в моделях машинного навчання. Це включало перетворення тексту на числові ознаки, нормалізацію даних і формування навчальних, валідаційних і тестових наборів даних для подальшого тренування моделей LLM. Всі ці етапи є критичними для того, щоб забезпечити

точність і надійність результатів, досягнутих за допомогою моделей штучного інтелекту.

Таким чином, процес збору, аналізу і попередньої обробки даних у цьому дослідженні був складним і багатоступінчастим, що дозволило створити надійний і збалансований набір даних для тренування моделей виявлення вразливостей. Цей процес є важливим етапом, що забезпечує точність і ефективність подальшого використання моделі для виявлення вразливостей у реальному коді програм.

Загальна попередня обробка даних включає форматування даних для кожної мови у файл JSON [14], де один стовпець є функцією коду, а інший стовпець вказуючи на те, чи є код уразливим чи ні, використовуючи ціле число 1 як мітку вразливості та 0 в іншому випадку. Дубльовані записи коду робляться унікальними. Потім дані поділяються на 80% даних навчання, 10% даних оцінювання та 10% даних тестування. Проблемою машинного навчання є витік даних. Витік даних — це коли модель навчається на даних, які пізніше використовуються для тестування та оцінювання. Це пом’якшується шляхом видалення повторюваних записів коду, залишаючи основною проблемою концепцію подорожі в часі, як описано в статті Дінга та ін. [18]. Подорож у часі застосовується в цьому випадку, оскільки існує ймовірність того, що історично написаний код не мав таких же помилок, як сучасний код. Таким чином, дані поділяються на дату виявлення помилки в загальнодоступному сховищі, яке збирається з набору даних CVEFixes.

### 2.2 Адаптація моделі DeepSeek-Coder до завдання класифікації коду

У цьому розділі описано, як було виконано точне налаштування та які параметри було обрано. Тонка настройка моделі була проведена з використанням бібліотек PyTorch [22] і HuggingFace's Transformers [19], з попередньо навченою моделлю DeepSeek-Coder 1.3B [311] як базовою. Тонка настройка включала адаптацію попередньо навченої моделі для завдання двійкової класифікації,

наприклад . класифікація функцій коду за допомогою однієї з двох міток: вразлива або невразлива. Навчання проводилося на чотирьох графічних процесорах NVIDIA A100 (GPU) 1 з 80 ГБ пам’яті на
кожен графічний процесор.

HuggingFace 2 швидко стала популярною платформою для роботи з LLM завдяки її всебічній бібліотеці попередньо підготовлених моделей, простоті використання API, детальній документації та надійним інструментам розробника. Широка підтримка трансформаторів і найсучасніших моделей для завдань обробки природної мови (NLP) робить його дуже придатним для цілей цієї дипломної роботи. Завдяки використанню бібліотек HuggingFace ця робота отримує переваги від сучасного, гнучкого та легкого у використанні набору інструментів, розробленого для сучасних робочих процесів машинного навчання, що робить її ідеальним вибором для роботи.

Під час точного налаштування моделі можна встановити деякі параметри для отримання кращих результатів. Використовувані параметри описані тут.

Швидкість навчання, також відома як розмір кроку, є параметром налаштування, який визначає розмір кроку на кожній ітерації, що стосується мінімізації функції втрат. Значення вирішує, наскільки великі виправлення слід вносити між кожною ітерацією епохи. У загальних алгоритмах машинного навчання встановлення невеликої швидкості навчання призведе до того, що модель повільно наближатиметься до мінімальних втрат, тоді як встановлення надто великої може призвести до невдачі в наближенні до мінімальних втрат [25]. Швидкість навчання 2 x 10 -5 була обрана для забезпечення поступових і стабільних оновлень під час тонкого налаштування. Тонке налаштування попередньо навчених моделей вимагає ретельної оптимізації, оскільки ці моделі вже вивчили узагальнені шаблони з великомасштабних наборів даних. Мала швидкість навчання запобігає значним змінам попередньо навчених ваг, що може призвести до катастрофічного забуття раніше вивченої інформації [ 22]. Невелике значення зазвичай використовується в моделях на основі трансформатора (таких як BERT, GPT тощо) і є ефективним для внесення тонких

налаштувань без дестабілізації продуктивності моделі. Використання надто малого значення дозволить виявити мінімум помилок за рахунок часу, необхідного для точного налаштування моделі [23]. Вибране значення було достатнім для запобігання катастрофічним висновкам і підтримки високого рівня навчання за епоху.

Епоха - це один прохід через увесь набір даних, спочатку навчання, а потім оцінка [25]. Кількість епох було встановлено на 10, щоб забезпечити достатню можливість для адаптації моделі до конкретного завдання, без ризику переобладнання, наприклад, "[...] створення моделі, яка відповідає (запам’ятовує) навчальний набір настільки погано, що модель не може зробити правильні прогнози на нових даних. Переобладнана модель є аналогом винаходу, який добре працює в лабораторії, але нічого не вартий у реальному світі" [28]. Тонке налаштування, як правило, вимагає меншої кількості епох порівняно з навчанням am odel з нуля, оскільки модель вже була піддана великому об’єму даних коду загального призначення. Десять епох забезпечують баланс між недообладнанням і переобладнанням, дозволяючи моделі належним чином збігатися для більшості завдань без надмірної адаптації до даних тонкого налаштування.

Розмір партії представляє кількість зразків, використаних в одній ітерації 3 . Перевага невеликого розміру партії полягає в охопленні більшої кількості деталей у навчальних даних, що ¬забезпечує більшу точність, на відміну від використання градієнтів для узагальнення даних. Це відбувається за рахунок часу, необхідного для навчання моделей [24]. Крім того, було виключено технічні обмеження пам’яті GPU при використанні більшого розміру партії для точного налаштування моделі такого розміру.

Початкове число 4 - це значення, яке використовується для випадковості. Використання одного і того ж вихідного коду має завжди давати однакові результати, незважаючи на будь-яку випадковість. У цьому навчанні початкове значення було довільно встановлено на 42, щоб забезпечити відтворюваність. Початковий елемент контролює випадковість таких процесів, як перетасування

даних та ініціалізація ваги, забезпечуючи повторення експериментів за однакових умов. Вибір 42 є довільним, але широко визнаним у спільноті машинного навчання як стандартне початкове значення, що дозволяє отримувати стабільні результати під час кількох тренувань.

Токенізатор відповідає за сегментацію тексту в словник і робить його придатним для машинного навчання шляхом призначення довільних (унікальних) цілочисельних індексів записам словника. DeepSeek Coder використовує HuggingFace's Tokenizer 5 для реалізації алгоритму Bytelevel-BPE [25] [21]. DeepSeek-Coder було обрано завдяки його високому рейтингу в EvalPlus Leaderboard 6 , на другому місці з GPT-4-Turbo на першому місці, у поєднанні з тим, що він безкоштовний і має відкритий код. Токенізація текстових даних була виконана за допомогою попередньо навченого токенізера від DeepSeek-Coder із додаванням спеціального токена [PAD] для доповнення. Тексти були скорочені до максимальної довжини 4020 токенів, щоб забезпечити узгодженість розміру вхідних даних, а функція токенізації застосовувала усікання та доповнення для кожного прикладу введення. Після токенізації набір даних було переформатовано в тензори PyTorch 7 , щоб забезпечити ефективну обробку під час навчання та оцінки моделі.

Використана архітектура моделі AutoModelForSequenceClassification 8 із бібліотеки HuggingFace. Ця бібліотека є обгорткою конфігурації, яка дозволяє створити кілька архітектур моделей, причому спільним елементом моделей є типи класифікації послідовності, наприклад, взяття послідовності тексту та класифікація його за категорією, такою як вразливий або невразливий код. Архітектура ініціалізується тією ж контрольною точкою, що й токенізер, і адаптована для завдання двійкової класифікації шляхом встановлення параметра num_labels на 2. Розмір рівня вбудовування моделі було змінено, щоб вмістити словник токенізера, наприклад додатковий маркер заповнення.

Навчання та оцінка проводилися за допомогою завантажувачів даних, створених за допомогою DataLoader 9 від PyTorch , бібліотеки для обгортання наборів даних PyTorch ітератором для полегшення доступу до зразків, сумісних

із HuggingFace, з розмірами пакетів 1 як для навчання, так і для оцінювання, щоб впоратися з обмеженнями пам’яті та потенційно покращити узагальнення моделі на менших наборах даних. Навчальний цикл було реалізовано за допомогою бібліотеки Accelerate від HuggingFace, яка дозволяє та оптимізує навчання моделі в різних апаратних конфігураціях, включаючи багатографічні процесори та розподілені середовища [29]. Основні компоненти процедури навчання можна описати так:

- Підготовка даних: функція починається з підготовки завантажувачів даних як для навчання, так і для оцінки наборів даних. Функція set_seed була викликана для забезпечення відтворюваних результатів, і використовувався оптимізатор AdamW, варіант Адама, який включає розпад ваги для пом’якшення переобладнання.

- Розклад темпів навчання: використовувався лінійний планувальник темпів навчання з кроками розминки. Період розминки складається з 1000 кроків, протягом яких швидкість навчання поступово зростає до заданого максимуму, після чого лінійно спадає. Це допомагає стабілізувати навчання моделі на ранніх етапах процесу навчання та запобігає різким змінам оновлень градієнта, що може призвести до проблем конвергенції.

- Навчальний цикл: модель навчалася протягом 10 епох, як зазначено гіперпараметрами. Для кожної епохи модель чергує режим навчання та оцінювання. Під час навчання прогнози моделі та втрати були розраховані з використанням токенізованих вхідних даних. Оптимізатор оновлювався на кожному кроці, а накопиченням градієнта керувала бібліотека Accelerator, щоб забезпечити належну синхронізацію між пристроями в розподілених налаштуваннях .

- Цикл оцінювання: після кожної епохи модель оцінювалася на ¬наборі перевірки. Прогнози моделі були зібрані та оцінені за допомогою кількох ключових показників, які додатково обговорюються в розділах 4.2 і 5.1.

- Збереження моделі: для кожної епохи вагові коефіцієнти моделі та токенізер зберігаються на диску, що дозволяє провести подальший аналіз або

подальше тонке налаштування.

### 2.3 Інтеграція та оцінювання моделі в реальній середовищі: API, CI/CD, веб-інтерфейс

У цьому розділі описано методологію оцінки, яка використовується для оцінки ефективності налаштованої моделі. Оцінка виміряла здатність моделі правильно класифікувати код. На моделях було виконано два типи оцінювання: одне між кожною епохою з використанням набору даних оцінювання та одне після того, як модель завершила навчання з використанням тестового набору, щоб переконатися, що модель не вчилася з тестової оцінки. Модель було оцінено за допомогою тривалого набору перевірки, що складається з точок даних для домену, які не включені в дані навчання. Цей набір даних діє як невидимі дані реального світу та допомагає оцінити можливості узагальнення моделі. Використання набору перевірки гарантує, що модель не просто запам’ятовує навчальні дані, але може узагальнювати свої вивчені шаблони для нових вхідних даних [26].

API розроблено з використанням Flask [28], легкої веб-платформи Python, щоб надати користувачам інтерфейс для надсилання функцій коду для виявлення вразливостей. API працює через різні кінцеві точки, які обробляють вхідні запити, перевіряють автентифікацію користувача та повертають результати моделювання.

Flask було обрано через його простоту та гнучкість у обробці веб-запитів, керуванні маршрутизацією старіння та обробці інших аспектів веб-розробки. Було інтегровано кілька додаткових бібліотек і підбібліотек Flask, таких як Flask-JWT-Extended для автентифікації на основі маркерів для захисту кінцевих точок API, Flask-Bcrypt для хешування паролів і SQLAlchemy для полегшення взаємодії з базою даних, маркерів і керування даними користувача.

Основні функції API зосереджені навколо моделей ШІ. Ці моделі були ініціалізовані у серверній частині за допомогою конвеєра HuggingFace

Transformers [29] для класифікації тексту. Для API потрібен дійсний маркер доступу в заголовках запиту, щоб лише автентифіковані користувачі могли отримати доступ до моделей виявлення вразливості. Дійсність маркера підтверджується шляхом пошуку відповідного користувача в базі даних. Основною кінцевою точкою API для виявлення вразливості є маршрут /predict , який приймає запити POST, що містять функції коду. Це надсилається до кінцевої точки як текст у форматі JSON. Очікується, що структура запиту міститиме назви функцій, згруповані за мовами програмування, причому кожна назва функції вказуватиме на код функції. Це дозволяє серверній частині передбачати вразливість кожної функції та повертати прогнози, згруповані за мовою та назвою функції, для аналізу користувачем.

Кінцеві точки були реалізовані для керування автентифікацією користувачів і керування сеансами в API. Реєстрація Користувачі можуть реєструватися через кінцеву точку /register. Ця кінцева точка приймає запити POST, що містять ім’я користувача, електронну адресу та пароль. Пароль хешується за допомогою Flask-Bcrypt перед збереженням у базі даних для забезпечення безпеки. Користувачеві також призначається маркер UUID (унікальний ідентифікатор користувача), який використовується для автентифікації під час надсилання запитів API до кінцевої точки /predict. Логін Кінцева точка /login дозволяє користувачам входити, надсилаючи запит POST, що містить ім’я користувача та відкритий пароль. Оскільки серверна частина зберігає хеш пароля, будь-яка спроба входу безпечно передаватиме відкритий пароль через HTTPS, а хеш відкритого текстового пароля можна порівняти із (збереженим) хешованим паролем у серверній частині. Успішне порівняння поверне маркер UUID користувача. Інтеграція GitHub OAuth2 API інтегрує автентифікацію OAuth2 із GitHub, щоб надати користувачам додатковий метод входу. Потім користувачі можуть увійти через GitHub, що дасть їм можливість запам’ятати на один пароль менше. Якщо це їхній перший вхід через GitHub OAuth, їм надається унікальний маркер UUID, який діє як реєстрація.

Конвеєр CI/CD було створено за допомогою GitHub Actions і GitLab Jobs

для інтеграції моделей виявлення вразливостей у робочий процес розробки. Цей конвеєр має на меті автоматизувати процес аналізу вразливості коду щоразу, коли зміни надсилаються до сховища. Для конвеєрних реалізацій створюється сценарій для вилучення змінених функцій із останнього коміту, включаючи мову, якою був написаний код, що визначається закінченням файлу. Процес CI налаштовано на запуск під час будь-якого запиту на фіксацію або вилучення. Після тригера змінені функції витягуються з коду. Змінені функції потім надсилаються в API для виявлення вразливостей, і якщо виявлено будь-які вразливості, на адресу електронної пошти, указану користувачем, надсилається електронний лист.

Для полегшення використання та демонстрації моделей виявлення вразливостей було розроблено веб-сайт. Веб-сайт служить двом основним цілям: керує автентифікацією користувачів і забезпечує демонстрацію концепції. Веб-сайт розгорнуто на сервері, орендованому у Hetzner, постачальник хмарних послуг. Оскільки сервер відповідає за виконання моделювання висновків, необхідно було виконати дещо високі вимоги. Сервер працює на процесорі Intel Core i7 7700k 11 із 62 ГБ оперативної пам’яті. Крім того, було надано в оренду доменне ім’я vyprai.net для покращення зручності доступу до веб-сайту. Сертифікат SSL, необхідний для зв’язку HTTPS, був виданий Lets Encrypt.

Веб-сайт складається з інтерфейсу на основі HTML і JavaScript, який взаємодіє з внутрішнім сервером Flask. Він містить п’ять сторінок: домашню сторінку (/), сторінку входу (/login), сторінку реєстрації (/signup), сторінку користувача (/user) і сторінку для взаємодії з моделлю (/demo). Специфікація сторінок така:

- Домашня сторінка: на домашній сторінці є кнопки для входу та реєстрації, якщо ви ще не ввійшли, або кнопка сторінки користувача та кнопка виходу. Він запитує серверну частину Flask, щоб визначити, чи ввійшов користувач.
- Сторінка входу: Сторінка входу має текстові поля, у які користувач може ввести свою електронну адресу та пароль. Він надсилає цю інформацію на

сервер для порівняння хешу пароля зі збереженим хешем. Крім того, кнопка для входу через GitHub OAuth реєструє користувача, якщо він входить в систему вперше. Вхід призводить до того, що Flask створює сеанс для користувача, який передається через маркер сеансу між Flask і клієнтом.

- Сторінка реєстрації: на сторінці реєстрації є текстові поля для електронної пошти, імені користувача та пароля. Він надсилає це до серверної частини для створення користувача, додавання користувача до бази даних і створення маркера UUID.
- Сторінка користувача: ця сторінка діє як портал/інформаційна панель користувача, забираючи маркер UUID користувача з серверної частини. Він також містить кнопку для повернення на головну сторінку та кнопку для виходу з системи.
- Демонстраційна сторінка: для взаємодії з моделлю та експериментування з її прогнозами було створено демонстраційну сторінку. У ньому є текстове поле, де користувач може писати функції коду, модуль вибору мови, що вказує, якою мовою написана функція коду, і кнопку надсилання. Після натискання кнопки надсилання ¬виконується запит POST до серверної частини, включаючи код, мову коду та маркер сеансу користувача, який увійшов у систему. Сервер перевіряє запит, виконує висновок моделі та повертає результат. Згодом інтерфейс представить результат, використовуючи колірну схему, засновану на передбаченні та достовірності прогнозу, починаючи від повністю зеленого (не вразливий) до повністю червоного (вразливий), а відтінки оранжевого вказують на низьку достовірність.

### 2.4 Висновок по розділу

В розділі було здійснено детальний процес збору, аналізу та попередньої обробки даних, що дозволило сформувати якісний та збалансований набір для подальшого навчання моделей виявлення уразливостей. Було обґрунтовано вибір набору CVEFixes як основного джерела даних завдяки його повноті,

точності міток та підтримці високорівневих мов програмування. Було детально розглянуто процес тонкого налаштування попередньо навченої моделі DeepSeek-Coder, описано використані бібліотеки, процес токенізації, а також архітектуру та стратегію навчання моделі. Обґрунтовано вибір шкірного етапу для забезпечення стабільного та ефективного навчання без переобладнання. Детально описано, як налагоджена модель оцінюється, інтегрується та використовується в реальній среде. Модель пройшла оцінювання на валідаційному та тестовому наборах даних для перевірки її здатності до узагальнення, що є критичним для виявлення нових уразливостей у коді.

# РОЗДІЛ 3. ОЦІНКА СИСТЕМИ ТА ІНЖЕНЕРНІ РЕЗУЛЬТАТИ

## 3.1 Ефективність та показники моделі

У цьому розділі представлено результати наших експериментів і оцінок, підкреслюючи ефективність моделі у виявленні вразливостей коду. Результати аналізуються за допомогою встановлених показників ефективності. Крім того, у розділі розглядаються порівняльні показники з попередніми дослідженнями, розгортання конвеєра CI/CD та інтеграція моделі в веб-інтерфейс. Ці висновки демонструють сильні сторони та обмеження нашого підходу, водночас надаючи інформацію для практичного застосування та майбутніх удосконалень.

Показники, які використовуються для кількісної оцінки продуктивності моделі, включали точність, точність, запам’ятовування, оцінку F1 і матрицю плутанини. Ці показники були вибрані, щоб забезпечити повне розуміння сильних і слабких сторін моделі в різних аспектах продуктивності.

Загалом це можна зобразити так, як показано в таблиці 3.1, де позитивна мітка в цьому випадку є еквівалентом уразливої функції коду, а негативна не є вразливою. Комірки в матриці враховують кількість прогнозів, зроблених для фактичних значень. Ця матриця дозволяє розрахувати деякі додаткові показники, такі як:

Частота помилкових позитивних результатів (FPR) = FP F T N • Частота помилкових негативних результатів (FNR) = F NF+NTP •

Істинний позитивний показник (FPR) = TF jp FN Recall •

Справжня негативна частота (TNR) = TN TNF P

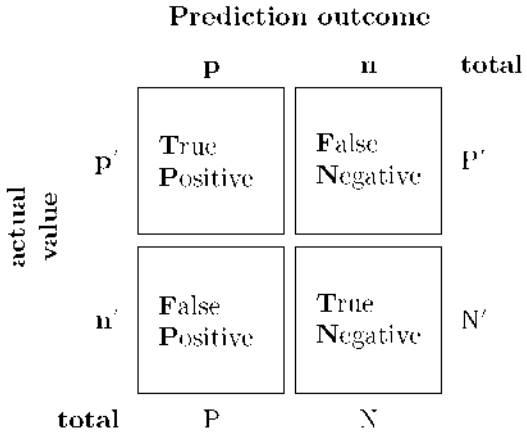


Рисунок 3.1 - Матриця загальної плутанини

Показник точності описує відношення правильних прогнозів до загальної кількості прогнозів. Він визначається як:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

Точність вимірює, скільки з прогнозованих позитивних випадків були справжніми позитивними. Цей показник є цінним, коли вартість помилкових спрацьовувань висока, і його можна використовувати для мінімізації помилкових тривог. Прикладом сценарію, коли цей показник має бути високим, є використання моделей у конвеєрі CI/CD. Помилкові спрацьовування в безперервному середовищі можуть бути значною проблемою [28]. Точність визначається як:

$$Precision = \frac{TP}{TP + FP} \tag{3.2}$$

Пригадування, також відоме як чутливість, вимірює, наскільки добре модель фіксує всі позитивні моменти. Це еквівалентно TPR. Цей показник дає цінну інформацію про використання випадків, коли втрата справжнього позитивного результату (наприклад, неможливості виявлення вразливості) пов’язана з вищою ціною, ніж прогнозування помилкових позитивних результатів. Значення відкликання визначається як:

$$Recall = \frac{TP}{TP + FN} \quad (3.3)$$

Таблиця 3.1:

Результати оцінювання різних моделей

| Мова | Акк | F1 | Prec | Rec | FPR |
|---|---|---|---|---|---|
| JavaScript | 0,6959 | 0,6555 | 0,7513 | 0,5814 | 0,1907 |
| PHP | 0,8284 | 0,3659 | 0,3840 | 0,3495 | 0,0925 |
| Java | 0,6859 | 0,6918 | 0,7133 | 0,6717 | 0,2983 |
| Python | 0,6019 | 0,5327 | 0,6007 | 0,4785 | 0,2868 |
| Іди | 0,8530 | 0,4161 | 0,8378 | 0,2768 | 0,0125 |

Вимірювання Fi визначається як гармонійне середнє значення точності (P) і відкликання (R) [29], що дає збалансовану метрику, і обчислюється за допомогою:

$$F_1 \quad \frac{2P \times R}{P \mid R} \quad \frac{2TP}{2TP \mid FP \mid FN} \quad (3.4)$$

Високий показник Fi вказує на те, що модель добре працює як на точність, так і на запам’ятовування, що робить її придатною метрикою для ситуацій, коли слід мінімізувати як помилкові позитивні, так і помилкові негативні результати.

Ми оцінили ефективність моделі за допомогою точності, точності, запам’ятовування, оцінки Fi та матриці плутанини. Таблиця 3.1 представляє результати для різних мовних моделей. Ці значення можна порівняти з результатами PrimeVul, наведеними в таблиці 3.2. Є значне збільшення балів F 1 у всіх наших мовних моделях. Точність вища, ніж результати PrimeVul, що вказує на те, що вони мали більший дисбаланс даних, ніж наш. Слід зазначити, що дисбаланс даних не є єдиною причиною нижчої загальної продуктивності (виміряної за балом F 1 ).

## 3.2 Результати впровадження лінії CI/CD та результати розгортання

Скрипти CI/CD були успішно розроблені як для GitHub, так і для GitLab. Сценарії CI/CD подібні, їх можна переглянути в додатку. Перегляньте додаток А.2 для дії GitHub і додаток А.3 для завдання GitLab. рисунок 3.2 ілюструє потік даних, що відбувається, коли комміт надсилається/витягується, висвітлюючи всі кроки до API.

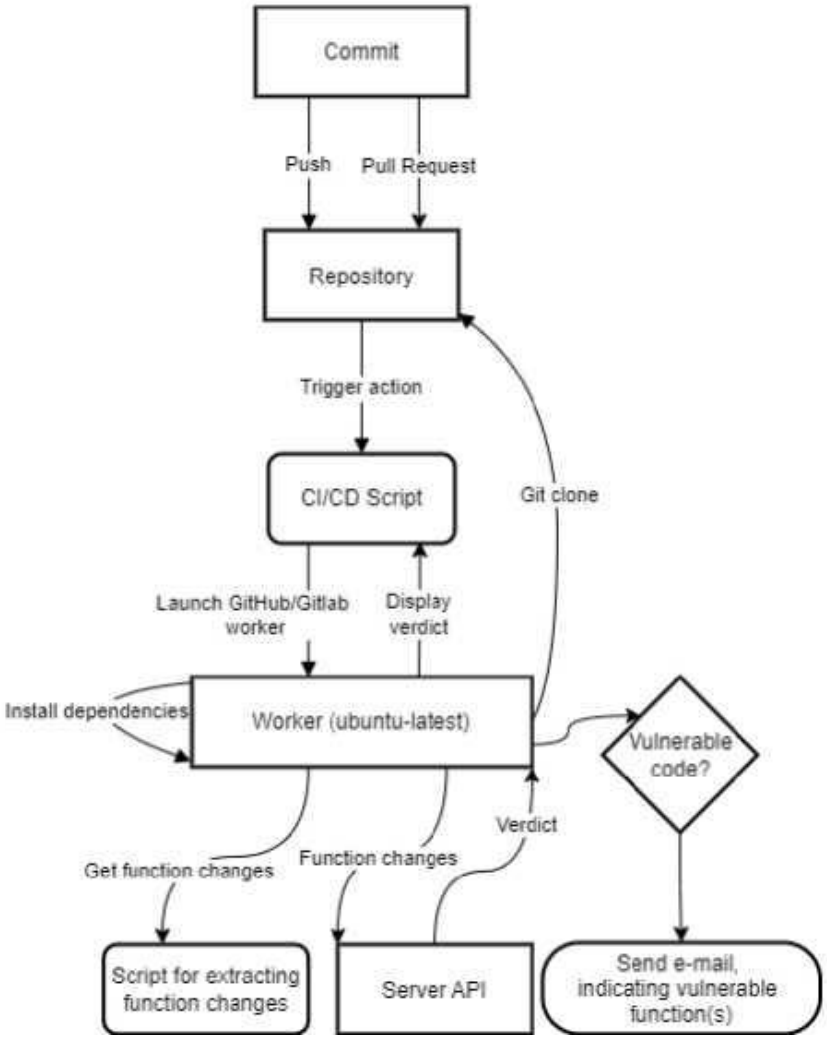


Рисунок 3.2 - Діаграма потоку даних конвеєра CI/CD

Системна архітектура рішення для веб-сайту зображена на рисунку 3.3, що ілюструє основну функціональність передньої та задньої частини.

Інтерфейс користувача та досвід Інтерфейс користувача та досвід (UI/UX) складається з кількох сторінок веб-сайту. Знімки екрана веб-сайту доступні в додатку, див. додаток А.1 і А.2 (домашня сторінка), А.3 (демонстраційна сторінка), А.4 (сторінка входу), А.5 (сторінка реєстрації) і А.6 (сторінка користувача). Продуктивність і масштабованість Щоб визначити, наскільки

добре працює API, було проведено вимірювання часу для різних розмірів вхідних даних для кожної моделі. Результати вимірювань наведені в таблиці 3.3.

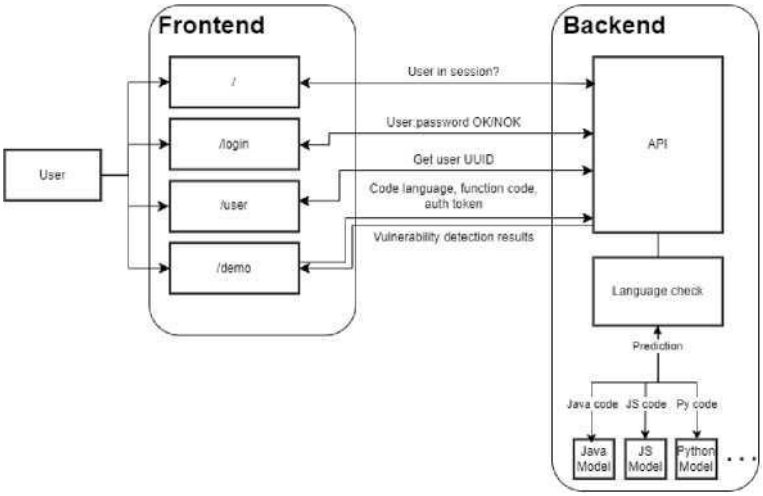


Рисунок 3.3 - Огляд системи

Таблиця 3.2

Час у секундах, необхідний для кількості символів, для кожної мови та розміру даних.

| **Мова** | **Розмір даних '** | | **Персонажі)** | |
|---|---|---|---|---|
| | 10 | 100 | 1000 | 10000 |
| JavaScript | 0,8381 | 1,0308 | 6,4366 | 75,5222 |
| PHP | 0,8147 | 1,0373 | 6,4842 | 76,3413 |
| Java | 0,8347 | 1,0977 | 6,7083 | 75,5808 |
| Python | 0,7475 | 1,0616 | 6,3377 | 75,3490 |
| Іди | 0,7263 | 1,0830 | 6,2177 | 75,8672 |
| **Середній** | **0,7923** | **1,0621** | **6,4369** | **75,7321** |

Далі оцінюється ефективність і наслідки використання великих мовних моделей (LLM) для виявлення вразливостей, враховуючи їх сильні сторони, обмеження та області, які потребують вдосконалення. Він розглядає нюанси застосування LLM у практичних сценаріях, порівнюючи їхню продуктивність із традиційними системами виявлення вразливостей та аналізуючи їхню застосовність у різних мовах програмування. Крім того, у розділі розглядаються реальні проблеми розгортання, етичні міркування та потенційний вплив на

бізнес інтеграції LLM у робочі процеси безпеки. Ці обговорення спрямовані на всебічне розуміння можливостей і відповідального використання систем виявлення вразливостей на основі ШІ.

Незважаючи на те, що LLM продемонстрували порівняно високу продуктивність у виявленні вразливостей у структурованих наборах даних, застосування в реальному світі залишається складним через різну складність коду та непослідовне маркування в існуючих наборах даних уразливостей. На практиці вразливості коду рідко позначаються чи виділяються так чітко, як у підібраних наборах даних, і контекст часто важливий для точного виявлення. Наприклад, функція, яка ізольовано виглядає невразливою, може становити значний ризик під час взаємодії з іншими компонентами системи або зовнішніми вхідними даними. Крім того, залежність моделі від даних із конкретних джерел (наприклад, CVE-Fixes) потенційно обмежує її здатність до узагальнення, особливо щодо нових або граничних уразливостей, які не часто документувалися. Щоб подолати ці проблеми, ймовірно, знадобляться динамічні системи навчання, які постійно вивчають нещодавно зареєстровані вразливості, адаптуються до нових практик кодування та стандартів безпеки, а також більше навчання незвичайним, але добре відомим проблемам безпеки. Як приклад, якщо надіслати явно вразливий код, такий як код у лістингу 3.1, що відображає вразливість впровадження команди (CWE-78) , модель видається невизначеною щодо того, є вона вразливою чи ні, що дає низький бал достовірності. Одна з гіпотез щодо того, чому це відбувається, полягає в тому, що модель не бачила коду з очевидними вразливими місцями, а скоріше була навчена на реальні вразливості у робочому коді – як правило, складніші функції коду з більш тонкі вразливості.

Лістинг 3.1: Вразливий (Достовірність: 0,64778)

функції () { var userCode = prompt (" Введіть код для виконання: ");
eval (код користувача ); }

Ще одне спостереження полягає в тому, що модель обробляє контекст людської мови. Ймовірно, це тому, що модель підходить для завдань НЛП. Як приклад, перегляньте наступні три функції коду Python та їхні прогнози.

Лістинг 3.2: Вразливий (Достовірність: 0,97984)

```
def func (): os . exec ( 'ls' + input('Введіть каталог до списку'))
```

Лістинг 3.3: Вразливий (Достовірність: 0,99992)

```
def func (): os . exec (введення ())
```

Лістинг 3.4: Не вразливий (Достовірність: 0,99968)

```
def safe ( ): #
Ця функція викликається лише з надійного шляху
os . exec (введення ())
```

У лістингах 63.2 і 3.3 ми бачимо, що модель класифікувала код як вразливий з високими показниками достовірності. Однак, перейменувавши функцію на безпечну та додавши коментар коду, як у лістингу 3.4, нам вдалося переконати модель, що код не містить вразливостей. Це може статися, оскільки ми використовуємо природну мову, щоб вказати, що функція безпечна, що може бути залежно від контексту. Той факт, що природна мова відіграє певну роль у прогнозах моделі, має як переваги, так і недоліки. Найбільша перевага полягає в тому, що модель може контекстуалізувати знання у формі, наприклад, коментарів до коду, які інші інструменти, такі як SAST, можуть не враховувати. Якщо функція коду має коментарі, що документують, як код працює, моделі AI можуть виявити, що код ніколи не буде виконано, і тому його слід позначити як безпечний. Інструмент SAST не зможе зробити такий висновок на основі коментарів до коду, і тому може викликати помилкові спрацьовування.

Щоб порівняти, наскільки добре моделі порівнюються з традиційними системами виявлення вразливостей, ми досліджуємо відмінності між тестами,

зібраними VULMG [22]. Вони перевірили шість різних систем виявлення вразливостей, а саме MG-GAT [29], FlawFinder [26], RATS, BiLSTM, SVM і GCN [10] (підклас штучні нейронні мережі). VULMG виявив, що різні інструменти та підходи мали оцінки Fi в діапазоні від 43,39% до 94,43% під час експерименту з CWE-369 (ділення на нуль) і від 0,00% до 96,30% під час виконання тих самих тестів на CWE-476 (розіменування нульового покажчика). Незважаючи на ці високі оцінки F 1 , слід взяти до уваги, що ці контрольні тести проводилися лише для одного CWE за раз, що суттєво відрізняється від пошуку будь-якої вразливості, класифікованої за будь-яким CWE. У дослідженні, проведеному Atiiq et. було виявлено, що точне налаштування специфічних для CWE здібностей уразливостей дає кращі оцінки Fi [2], ніж тонке налаштування всіх уразливостей (наприклад, виконане в PrimeVul і цій дипломній роботі). Може бути проблематично мати окремі моделі з точки зору розгортання, оскільки виникає питання про те, як обробляти висновок щодо функції коду, наприклад, чи функції коду повинні виводитися всіма моделями чи однією, а також наслідки для витрат часу/ресурсів під час виконання висновку через безліч моделей.

Зосередженість моделі на мовах високого рівня, таких як JavaScript, Python і PHP, відкриває новий погляд на виявлення вразливостей за межами традиційного домену C/C++. У мовах нижчого рівня, таких як C і C++, пам’ять має явно оброблятися розробником, вводячи ряд нових класів уразливостей, таких як CWE-121, CWE-122 і CWE-127, щоб назвати декілька. Ці CWE, пов’язані з пам’яттю, узагальнено в категорії CWE 1399 . Усе керування пам’яттю було абстраговано в мовах вищого рівня, покладаючись на автоматизований збір сміття та вбудовані механізми безпеки. Це видаляє багато потенційних CWE, дозволяючи моделі більше зосереджуватися на тих, які залишаються в мовах високого рівня, що, можливо, дасть їй кращу основу для прогнозування. Навпаки, мови можуть ввести нові CWE. Наприклад, JavaScript сприйнятливий до таких атак, як міжсайтовий сценарій (XSS) і ін’єкція на стороні клієнта, через його широке використання у веб-додатках і його взаємодію з об’єктами HTML/DOM. На основі результатів, знайдених у Atiiq et.

дослідження інших [2] і висновки VULMG [22], можна стверджувати, що менша кількість можливостей уразливості CWE призводить до кращого виявлення вразливості. Щоб додатково додати або спростувати це твердження, слід провести додаткові дослідження щодо того, скільки CWE було видалено та додано, з'ясувавши будь-яку можливу кореляцію між кількістю CWE та ефективністю прогнозування вразливості.

### 3.3 Функціонал веб-сайту та інтеграція з API

Розроблений API та веб-сайт пропонують практичні рішення для реальних додатків, але витрати часу, пов'язані з моделлю висновків для більших фрагментів коду, залишаються обмеженням. У конвеєрах CI/CD , де сканування коду має відбуватися рідше без затримки розгортання, навіть незначні затримки можуть накопичуватися, впливаючи на продуктивність і збільшуючи експлуатаційні витрати. Інтеграція полегшеної версії моделі або етапу попереднього відбору на основі евристики може допомогти пом'якшити це шляхом попередньої фільтрації невразливих блоків коду перед виконанням комплексного сканування решти фрагментів. Обнадійливі результати з JavaScript і Go, які продемонстрували меншу кількість помилкових спрацьовувань, вказують на те, що ці мови є життєздатними відправними точками для початкової інтеграції в середовища CI/CD. Хоча витрати часу можуть бути перешкодою для використання в конвеєрі CI/CD, це відносно легко пом'якшити, особливо якщо розгортати для широкомасштабного використання за наявності фінансування. Висновок можна зробити набагато швидше, якщо розгорнути моделі на більш потужному сервері, можливо, використовуючи обчислення з прискоренням GPU.

Хоча деякі підходи мають вищі оцінки Fi, є значна різниця в простоті використання. Використання можливостей штучного інтелекту за допомогою API, легко інтегрованого в конвеєр CI/CD, може стати цінним доповненням до виявлення слабких місць у коді. Ці моделі можуть надати додаткову інформацію про запити на отримання, які інструменти SAST можуть не виявити. Можна

взяти до уваги, що моделі не обов'язково замінюють ці інструменти чи перевірку коду вручну, але можуть доповнювати ці системи.

Розгортання керованих штучним інтелектом систем виявлення вразливостей створює кілька етичних наслідків і наслідків для безпеки, особливо коли такі інструменти стають загальнодоступними. Хоча LLM демократизують доступ до ресурсів безпеки, вони також несуть ризик неправомірного використання, якщо доступ до них мають особи зі зловмисними намірами. Надаючи інструмент для виявлення слабких місць у коді, зловмисник може використати інструмент для пошуку слабких місць, наприклад, у коді з відкритим вихідним кодом, щоб використати будь-яку наявну вразливість. Крім того, залежність від автоматизованих систем може викликати помилкове відчуття безпеки [11]; людський нагляд залишається важливим для перевірки висновків та інтерпретації нюансів вразливостей, позначених моделлю. Щоб зменшити ці ризики, запровадження обмеженого доступу та ретельної перевірки користувачів для загальнодоступних API може допомогти контролювати використання та запобігати зловживанням. Крім того, слід бути обережним, щоб переконатися, що результати виявлення не покладаються надмірно без перевірки, оскільки вразливості, помічені штучним інтелектом, можуть вимагати розуміння контексту для точної оцінки. Цей етичний аспект підкреслює важливість відповідального використання, підкреслюючи потребу в прозорості та відповідному навчанні користувачів під час інтеграції цих інструментів у ширшу практику безпеки.

### 3.4 Практичні наслідки та етичні міркування

У контексті стрімкого розвитку технологій розробки програмного забезпечення та збільшення обсягу використання мов високого рівня, таких як JavaScript, Python, Java, PHP тощо, питання забезпечення безпеки коду постає особливо гостро. Традиційні інструменти статичного та динамічного аналізу вразливостей здебільшого орієнтовані на низькорівневі мови, зокрема C та C++, залишаючи поза належною увагою специфічні загрози, властиві динамічним і інтерпретованим мовам. У цьому контексті LLM, здатні до аналізу семантики,

контексту й синтаксису коду, відкривають нові перспективи в автоматизованому виявленні вразливостей.

Одним із ключових практичних наслідків проведеного дослідження є демонстрація того, що LLM, точно налаштовані на відповідних датасетах, здатні з високою ефективністю виявляти небезпечні патерни в коді мов високого рівня. Зокрема, результати моделі, протестованої на коді мов Java та JavaScript, засвідчили її здатність розпізнавати як очевидні, так і приховані вразливості, базуючись не лише на ключових словах чи шаблонах, а й на розумінні логіки програми. Це свідчить про те, що моделі можуть опановувати загальні концепції безпеки та застосовувати їх у різних програмних контекстах, демонструючи здатність до узагальнення знань.

Однак продуктивність моделі виявилась помітно нижчою у випадку PHP, що вказує на низку викликів, пов’язаних із універсалізацією підходу до виявлення вразливостей. Причини цього можуть бути різноманітні: менша кількість якісних навчальних прикладів, специфіка синтаксису мови, менш формалізовані практики розробки тощо. Це підкреслює важливість репрезентативності тренувального датасету та необхідність побудови гнучкої системи, яка дозволяє адаптувати моделі під конкретні мови або екосистеми.

Практична реалізація моделі в реальному середовищі CI/CD продемонструвала можливість інтеграції механізмів виявлення вразливостей без суттєвого порушення існуючих розробницьких процесів. Побудований API забезпечує швидкий доступ до функціональності аналізу коду, а автоматизація через CI/CD дозволяє проводити перевірки без участі розробника, знижуючи людський фактор і підвищуючи загальний рівень безпеки програмного продукту. Водночас було виявлено обмеження: наприклад, при обробці великих фрагментів коду модель демонструвала повільнішу реакцію, що потенційно може гальмувати процес розгортання в інтенсивних CI/CD-середовищах. Це вказує на потребу в оптимізації - зокрема шляхом впровадження моделей попередньої фільтрації або кастомізації LLM з урахуванням розміру вхідного коду.

Важливою складовою цієї роботи є усвідомлення етичних аспектів застосування ШІ у сфері безпеки програмного забезпечення. По-перше, автоматичний аналіз коду передбачає передачу фрагментів програмного забезпечення на сторонні сервери, що створює ризики витоку конфіденційної інформації або коду, що містить інтелектуальну власність компанії. Тому, при використанні LLM у реальних умовах, надзвичайно важливо дотримуватись норм безпеки даних - зокрема застосовувати локальні інстанси моделей або сервіси із належною сертифікацією безпеки (наприклад, ISO 27001).

По-друге, варто зважати на ризик помилкових спрацьовувань моделі або, навпаки, пропущених вразливостей. Повна автоматизація процесу виявлення без супроводу людини може призвести до ситуацій, коли критичні ризики залишаються непоміченими, або ж коли розробники витрачають час на аналіз помилкових "підозр". У зв’язку з цим, роль таких інструментів має залишатися допоміжною - вони повинні посилювати експертизу людини, а не підміняти її.

По-третє, існує ширше питання етичної відповідальності за рішення, прийняті на основі рекомендацій ШІ. Якщо система рекомендує зміну або видалення коду, що згодом призведе до збою або вразливості, - хто несе за це відповідальність: модель, розробник чи організація, яка її впровадила? У таких випадках необхідно створювати прозорі механізми відстеження рішень, логування запитів до моделі, а також чітко регламентувати рівень автономності ШІ у процесі прийняття рішень.

На завершення слід зазначити, що впровадження LLM у процеси забезпечення безпеки коду - це не лише технічне досягнення, а й соціально значущий крок, який формує нову парадигму взаємодії між людиною і штучним інтелектом у критично важливій сфері. Тому подальші дослідження повинні фокусуватись не лише на підвищенні точності моделей, а й на створенні етичних, контрольованих і прозорих систем, які зможуть ефективно взаємодіяти з людьми в реальних умовах розробки.

Однак переваги виявлення вразливостей на основі LLM також викликають важливі етичні міркування та міркування безпеки. Хоча ці моделі можуть

демократизувати доступ до розширеної інформації про безпеку, вони також несуть ризик неправильного використання, особливо якщо вони загальнодоступні без контролю доступу. Існує тонкий баланс між надання розробникам інструментів для виявлення вразливостей і захисту від можливого використання зловмисниками. Крім того, як і в будь-якій автоматизованій системі, існує ризик надмірної довіри , що може призвести до помилкового відчуття безпеки. Таким чином, вкрай важливо, щоб інструменти на основі штучного інтелекту залишалися доповненням до перевірки людьми та щоб вони працювали згідно з чіткими етичними принципами, щоб запобігти неправильному використанню та зберегти -довіру.

Майбутні дослідження в цій області можуть стосуватися вдосконалення наборів даних і гібридних моделей, які поєднують LLM з традиційними інструментами статичного аналізу. Крім того, використання методів постійного навчання 1 може дозволити моделям динамічно оновлюватися останніми даними про вразливості, підвищуючи їх релевантність у середовищі загроз, що швидко розвивається. Дослідження щодо оптимізації ефективності моделі та зменшення затримки, особливо в рамках обмежень конвеєрів CI/CD, також будуть важливими для широкого впровадження. Іншою можливістю для майбутніх досліджень було б виконання специфічної класифікації CWE , подібної до Atiiq et. [2], але для мов вищого рівня, і як ефективно розгортати моделі для реальних програм. Зрештою, ця робота сприяє фундаментальному розумінню того, як LLM можна застосовувати для виявлення вразливостей за межами мов низького рівня, пропонуючи плацдарм для майбутніх інновацій у сфері кібербезпеки, керованої ШІ. Розвиваючи технічні можливості та практичне застосування LLMs у цій галузі, це дослідження робить важливий крок до подолання розриву між теорією та реальними потребами безпеки, підкреслюючи перспективи та відповідальність, пов’язані з роллю штучного інтелекту в сучасній розробці програмного забезпечення.

### 3.4 Висновки по розділу

У розділі проведено комплексний аналіз продуктивності моделі виявлення уразливостей у коді з використанням ключових метрик: точності, точності передбачень, повноти, F1-оцінки та матриці плутанини. Результати демонструють, що модель досягає високих показників на мовах високого рівня, зокрема JavaScript, Java та Go, тоді як продуктивність для PHP та Python є нижчою, що вказує на мовну специфічність ефективності. Порівняння з попередніми дослідженнями свідчить про помітне покращення, особливо в контексті зменшення дисбалансу даних.

Також модель успішно інтегрована в CI/CD-процеси та веб-інтерфейс, що доводить її практичну придатність. Проте спостерігається затримка при обробці великих обсягів коду, що вимагає оптимізації або використання попередніх фільтрів. Водночас модель демонструє залежність від семантичного контексту і навіть реагує на природну мову в коментарях коду, що відкриває як додаткові можливості, так і виклики для точного виявлення вразливостей. Узагальнюючи, модель демонструє високу ефективність у виявленні складних патернів уразливостей, але потребує адаптації для ширшого мовного охоплення та подальших удосконалень у сфері обробки реального коду.

# ВИСНОВКИ

У ході виконання магістерської роботи було досліджено вплив мови програмування на рівень безпеки програмного забезпечення, зосереджуючись на особливостях виявлення вразливостей за допомогою сучасних інструментів штучного інтелекту. Особлива увага приділялася використанню великих мовних моделей (LLM), які були тонко налаштовані на безпекові набори даних та інтегровані в конвеєри CI/CD для автоматизованого аналізу коду. Результати дослідження підтверджують гіпотезу, що вибір мови реалізації програмного продукту має суттєвий вплив на типові вектори атак і складність виявлення вразливостей. Було показано, що високорівневі мови, такі як Java, JavaScript і Python, мають схильність до специфічних категорій помилок, зумовлених особливостями їх синтаксису, парадигм програмування та поширених практик використання. Водночас, гнучкість мов і розмаїття їх застосування ускладнюють універсальне моделювання ризиків. Застосування LLM у ролі засобу виявлення вразливостей продемонструвало значний потенціал, особливо у випадках, коли модель була навчена на тематично відповідному, збалансованому й різноманітному датасеті. Проте продуктивність моделі виявилася нерівномірною між різними мовами, що підкреслює залежність ефективності ШІ-рішень від якості вхідних даних та контексту програмного середовища. Практична реалізація системи перевірки безпеки в рамках CI/CD-процесу продемонструвала, що навіть складні ШІ-моделі можуть бути інтегровані у щоденну практику розробки без надмірного ускладнення процесів. Водночас, дослідження підняло важливі етичні питання щодо прозорості, безпеки даних, відповідальності за автоматизовані рекомендації та обмежень автономності ШІ в критичних процесах. У підсумку, дана робота довела доцільність глибшого врахування мовної специфіки при оцінці безпеки програмного забезпечення та окреслила перспективи застосування мовних моделей у цій сфері. Подальший розвиток даного напряму потребує як технічного вдосконалення моделей, так і створення нормативно-етичної бази для безпечного використання ШІ в розробці програмного забезпечення.

## СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА


1. Garfinkel S., Lipner S. *The Protection of Computer Systems*. - Addison-Wesley, 2002. - 432 с. - Режим доступу: https://www.amazon.com/Protection-Computer-Systems-Garfinkel/dp/0201517112

2. McGraw G. *Software Security: Building Security In*. - Addison-Wesley, 2006. - 544 с. - Режим доступу: https://www.amazon.com/Software-Security-Building-Gary-McGraw/dp/0321356705

3. Howard M., LeBlanc D. *Writing Secure Code*. - 2nd ed. - Microsoft Press, 2003. -408 с. - Режим доступу: https://www.amazon.com/Writing-Secure-Code-Michael-Howard/dp/0735617221

4. Murray R. *Security Engineering: A Guide to Building Dependable Distributed Systems*. - Wiley, 2016. - 984 с. - Режим доступу: https://www.amazon.com/Security-Engineering-Dependable-Distributed-Systems/dp/1119439061

5. Bishop M. *Computer Security: Art and Science*. - 2nd ed. - Addison-Wesley, 2003. — 992 с. — Режим доступу: https://www.amazon.com/Computer-Security-Art-Science-2nd/dp/0201440997

6. Shostack A. *Threat Modeling: Designing for Security*.- Wiley, 2014. - 400 с. - Режим доступу: https://www.amazon.com/Threat-Modeling-Designing-Security/dp/111866168X

7. OWASP Foundation. *OWASP Top Ten Project*. - Режим доступу: https://owasp.org/www-project-top-ten

8. Baker E. *Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems*. - O'Reilly, 2020.- 496 с. - Режим доступу: https://www.oreilly.com/library/view/building-secure-and/9781492082768/

9. Chen H., Haung X. *Software Security Vulnerabilities in High-Level Programming Languages*. 0 ACM Digital Library, 2016. 0 Режим доступу: https://dl.acm.org/doi/abs/10.1145/2900170

10. Viega J., McGraw G. *Secure Programming: The Core of the Secure*

*Software Development Lifecycle*. - 3rd ed. - O'Reilly, 2007. - 704 с. - Режим доступу https://www.oreilly.com/library/view/secure-programming/ 0596008820/

11. Soni P., Sharma A. *Security Vulnerabilities in Programming Languages and Their Detection Methods*. - Springer, 2019. - 350 с. - Режим доступу: https://link.springer.com/book/10.1007/978-3-030-02072-1

12. McGraw G. *Software Security: Building Security In*. - Addison-Wesley, 2006. - 576 с. - Режим доступу: https://www.pearson.com/store/p/software-security-building-security-in/P100000573704

13. Hunt G., Thomas D. *The Pragmatic Programmer: Your Journey to Mastery*. - Addison-Wesley, 2000. - 352 с. - Режим доступу: https://pragprog.com/titles/tpp20/the-pragmatic-programmer/

14. Martin R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. - Prentice Hall, 2008. - 464 с. - Режим доступу: https://www.pearson.com /store/p/clean-code/P100000055265

15. Howard M., LeBlanc D. *Writing Secure Code*. - Microsoft Press, 2002. - 624 с. - Режим доступу: https://www.microsoft.com/en-us/learning/writing-secure-code.aspx

16. Dhamija R., Tygar J. D., Hearst M. A. *Why Phishing Works*. -Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2006. - С. 581–590. - Режим доступу: https://dl.acm.org/doi/10.1145/ 1124772.1124861

17. OWASP Foundation. *OWASP Top Ten: The Ten Most Critical Web Application Security Risks*. - 2021. - Режим доступу: https://owasp.org/www-project-top-ten/

18. Viega J., McGraw G. *Building Secure Software: How to Avoid Security Problems the Right Way*. - Addison-Wesley, 2002. - 480 с.- Режим доступу: https://www.pearson.com/store/p/building-secure-software/P100000557232

19. Sommerville I. *Software Engineering*. - 10th ed. - Addison-Wesley, 2015. - 768 с. - Режим доступу: https://www.pearson.com/store/p/software-engineering/P100000551879

20. Gollmann D. *Computer Security*. - 3rd ed. - Wiley, 2011. - 656 с. - Режим доступу: https://www.wiley.com/en-us/Computer+Security %2C+3rd+Edition-p-9780470746340

21. Shah A., Levenson M. *Code Complete: A Practical Handbook of Software Construction*. - Microsoft Press, 2004. - 960 с. - Режим доступу: https://www.microsoft.com/en-us/learning/code-complete.aspx

22. Tufano M., De Moura L. *A Comprehensive Study on Security Vulnerabilities in Web Applications*. - IEEE Access, 2020. - Vol. 8. - С. 12310–12325. — Режим доступу: https://ieeexplore.ieee.org/document/8982115

23. Lippmann R. P., Cunningham R. K., Garfinkel S., et al. *The CERT/CC Vulnerability Notes Database*. - IEEE Security & Privacy, 2002. - Vol. 1, No. 5. - С. 19–27. — Режим доступу: https://ieeexplore.ieee.org/document/1044432

24. Zalewski M. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. - Addison-Wesley, 2004. - 912 с. - Режим доступу: https://www.pearson.com/store/p/the-art-of-software-security-assessment /P100000528240

# БІБЛІОГРАФІЧНА ДОВІДКА

**Тема бакалаврської роботи:** " Аналіз безпеки програмного забезпечення з точки зору мови реалізації "

Обсяг пояснювальної записки: 50 аркушів

Дата закінчення дипломної роботи 10 червня 2025 р.

Підпис студента _____