КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 4 (32), 2026
ISSN 2663 – 4023
DOI 10.28925/2663-4023.2026.32.1165
УДК 004.056.5:004.8:004.4

Ярема Олег Михайлович
аспірант кафедри кібербезпеки
Тернопільський національний технічний університет імені Івана Пулюя, Тернопіль, Україна
ORCID: 0009-0009-8709-7813
yarema.oleh.m@gmail.com

Загородна Наталія Володимирівна
кандидат технічних наук, зав.каф. кафедри кібербезпеки
Тернопільський національний технічний університет імені Івана Пулюя, Тернопіль, Україна
ORCID: 0000-0002-1808-835X
zagorodna_n@tntu.edu.ua

ПОРІВНЯЛЬНИЙ АНАЛІЗ ЕФЕКТИВНОСТІ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ ПРОГРАМНОГО КОДУ З ДОПОМОГОЮ LLM ТА SAST

Анотація. У статті обґрунтовано необхідність впровадження засобів контролю безпеки програмного коду за допомогою великих мовних моделей (LLM), зумовлену стрімким зростанням обсягів програмного коду та появою нових безпекових ризиків, пов'язаних із генерацією коду за допомогою штучного інтелекту та необхідністю інтеграції окремих частин коду у складні архітектурні рішення. Алгоритми існуючих інструментів статичного аналізу коду (SAST) схильні до помилок через нездатність повноцінно враховувати логіку виконання коду та його контекстуальні зв'язки. Використання LLM як верифікатора, який підтверджує або спростовує результати статичного аналізу коду, потенційно може вирішити ці недоліки. У статті проведено порівняльний аналіз ефективності виявлення безпекових вразливостей програмного коду на мові C# за допомогою інструменту статичного аналізу коду Roslyn Analyzers, великих мовних моделей, таких як DeepSeek і Grok та інтеграційного підходу, що поєднує переваги статичних аналізаторів та LLM. Методика дослідження базується на проведенні експериментального дослідження з використанням тестової вибірки фрагментів програмного коду на мові C#, які містять безпекові вразливості різного типу. На першому етапі дослідження тестування фрагментів коду проводилось за допомогою інструменту статичного аналізу коду Roslyn Analyzers. На наступному етапі фрагменти коду було проаналізовано на наявність вразливостей моделями DeepSeek V3 та Grok 4.1. На заключному етапі було оцінено ефективність запропонованого гібридного підходу, що передбачає початкову перевірку коду статичним аналізатором з подальшою передачею його звітів на вхід обраних моделей генеративного інтелекту. Результати дослідження показують. що гібридний підхід з використанням DeepSeek та Roslyn Analyzers забезпечує зростання показників їхньої ефективності в порівнянні з незалежним використанням цих засобів. Порівняльний аналіз показників автономного використання моделей також встановив, що Grok показує гірші результати, ніж модель DeepSeek, і не є найкращим варіантом для застосування в задачах подібного типу. Дослідження демонструє, що інтеграція аналітичних можливостей великих мовних моделей у класичні процеси статичного аналізу програмного коду за рахунок підтвердження або спростовування результатів статичного аналізу є потенційним кроком до самокоригованих процесів аналізу безпеки програмного коду.

Ключові слова: програмне забезпечення; вразливості; тестування; штучний інтелект; LLM; статичний аналіз програмного коду; SAST; гібридний підхід.

ВСТУП

Прискорення темпів діджиталізації в усіх сферах життя людини призводить до стрімкого зростання кількості та складності програмних продуктів, що, свою чергу, зумовлює збільшення обсягів програмного коду. Поява засобів штучного інтелекту для генерації коду, з одного боку, відкрила широкі можливості для суттєвого прискорення

розробки програмного забезпечення, а з іншого боку – значно підвищила ризики потрапляння неперевіреного коду до виробничих систем, а також загострила проблему інтеграції різнорідних фрагментів коду у складних багатокомпонентних архітектурах [1]. Усе це в сукупності створює підґрунтя для виникнення нових безпекових вразливостей у програмному коді, об'єми якого часто перевищують людські можливості щодо когнітивного аналізу цього коду, тому виникає гостра потреба у впровадженні інтелектуальних автоматизованих засобів контролю безпеки. Хоча класичні інструменти статичного аналізу (SAST) забезпечують швидку перевірку при мінімальних витратах ресурсів, емпіричні дослідження демонструють прогалини в їх ефективності через недостатнє покриття можливих вразливостей статичними правилами та фільтрами [2]. Це зумовлено обмеженістю алгоритмів пошуку шаблонів, що ґрунтуються на синтаксичному аналізі, а, отже, не здатні враховувати логіку виконання, потоки даних, динамічні зміни станів та складність архітектури, що призводить до значної кількості хибно позитивних і хибно негативних результатів. Крім того, статичні правила та фільтри потребують постійного оновлення і не завжди справляються з виявленням вразливостей у коді, згенерованому засобами штучного інтелекту [3], [4].

Постановка проблеми. Поява великих мовних моделей (LLM) не лише трансформувала підходи до написання коду, а й відкрила принципово нові можливості для його аналізу, а саме – інтерпретації не лише синтаксису, а й семантики програмного коду на рівні, певною мірою наближеному до людського [5]. Попри ці переваги, використання виключно LLM несе ризики «галюцинацій» або вигаданих результатів, відсутності формальної верифікації та нестабільності результатів, що ускладнює стандартизацію перевірок безпеки [4]. Потенційним рішенням для цих недоліків може стати гібридний підхід, у якому інструменти статичного аналізу коду (SAST) будуть головним рушієм для виявлення підозрілих шаблонів безпекових вразливостей, а LLM будуть забезпечувати додаткову точність та виступати у ролі верифікатора результату, підтверджуючи чи спростовуючи факт виявлення вразливості.

Актуальність цього дослідження зумовлена необхідністю підвищення точності автоматизованого виявлення вразливостей при одночасній мінімізації хибних спрацювань. Завдяки перехресній перевірці результатів детермінованих наборів правил за допомогою аналітичних висновків штучного інтелекту, організації можуть суттєво зменшити обсяг інформаційного «шуму» при перевірці коду, який зазвичай перевантажує команди безпеки.

Дослідження базується на порівняльному аналізі трьох стратегій – використання класичних статичних аналізаторів, методів LLM та їхнього гібридного поєднання. Це дозволить визначити найефективніший шлях для подальших досліджень. Така методологія забезпечує оцінку сильних та слабких сторін використання автономного та гібридного підходів до статичного аналізу та нейромережевої обробки, маючи при цьому пряме практичне значення для створення алгоритмів перевірки коду в реальних проєктах. Практична цінність роботи спрямована на досягнення балансу між швидкістю виявлення вразливостей, яка притаманна статичним аналізаторам коду, та якістю їхнього аналізу, пропонуючи окремий підхід до автоматизації контролю безпеки програмного коду.

Аналіз останніх досліджень і публікацій. Підходи до дослідження безпеки програмного забезпечення демонструють перехід від традиційних методологій, заснованих на статичних правилах і шаблонах, до аналізу на базі штучного інтелекту [6]. Стандарти галузі розробки ПЗ спираються на інструменти статичного тестування,

такі як SonarQube, CodeQL та SnykCode, що інтегруються в конвеєри CI/CD для виявлення дефектів та вразливостей через аналіз та пошук за шаблонами [2], [6], [7]. Ці інструменти довели свою ефективність у зниженні витрат на виправлення коду та виявленні типових загроз, наприклад SQL-ін'єкцій. Однак вони часто пропускають реальні вразливі коміти через жорстку прив'язку до статичних правил, що в свою чергу заважає їм розпізнавати нові або нетипові проблеми [8].

Сучасні дослідження пропонують нову парадигму «інтелектуального аналізу коду» на базі великих мовних моделей (LLM), які здатні розпізнавати складні контекстуальні зв’язки та приховані дефекти в потоках даних [7], [8]. Оцінки таких моделей, як GPT-4, Mistral Large та DeepSeek V3, свідчать про їхню здатність досягати вищого рівня виявлення вразливостей порівняно зі статичними аналогами [2]. Проте дослідження застерігають від специфічних ризиків ШІ, таких як проблема «галюцинацій» та артефакти токенізації, які можуть заважати точному визначенню чи локалізації помилок [3].

Ефективність будь-якого автоматизованого інструменту безпеки значною мірою залежить від якості навчальних даних, і на сьогодні існують відкриті датасети, що містять реальні вразливості та відповідні патчі для їх виправлення. CVEfixes – це відомий датасет, що містить автоматично зібрані дані про реальні вразливості з бази CVE та відповідні коміти-патчі з публічних репозиторіїв, що робить його цінним ресурсом для навчання моделей виявлення вразливостей у програмному коді [9], [10]. Проєкт DiverseVul розширив цю базу за рахунок надання масштабних масивів вихідного коду для глибокого навчання та виявлення складних шаблонів [11]. Дослідження зосереджуються на застосуванні цих даних і великих мовних моделей для імперативних мов програмування, таких як C, C++, Java, а також на оцінці їхньої ефективності у спеціалізованих системах, зокрема операційній системі Android [11], [12], [13], [14]. Проте спостерігається суттєво менша кількість досліджень, які б зосереджувалися на застосуванні великих мовних моделей для мови програмування C# та платформи .Net [2], [15], [16].

Наукові дослідження пропонують використання гібридних підходів до виявлення безпекових вразливостей в коді шляхом інтеграції великих мовних моделей з існуючими інструментами, як один з головних напрямів подальших досліджень та розвідок [2], [17], [18]. Поєднання «контекстного відбору», притаманного мовним моделям, із високою точністю традиційних сканерів могло б створити систему «доповненого інтелекту», яка здатна перехресно перевіряти дані з різних джерел для винесення надійнішого вердикту [19], [20]. Кінцевою метою цього переходу є розробка самокоригованої системи безпеки, здатної покращуватися синхронно зі зростанням складності сучасного ПЗ та реагувати на нові класи загроз [21].

Мета статті. Метою статті є проведення експериментального дослідження та порівняльного аналізу ефективності виявлення вразливостей у програмному коді на мові C# на основі двох підходів: виявлення вразливостей за допомогою автономного використання інструменту Roslyn Analyzers та моделей DeepSeek V3 і Grok 4.1; та виявлення вразливостей за допомогою гібридного підходу, який використовує аналіз програмного коду моделей DeepSeek V3 і Grok 4.1 для підтвердження або спростування результатів статичного аналізу Roslyn Analyzers.

МЕТОДИКА ДОСЛІДЖЕННЯ

Загальна методика експерименту базується на порівняльному аналізі підходів традиційного статичного аналізу та засобів генеративного штучного інтелекту для

виявлення вразливостей у вихідному коді на мові C#. На підготовчому етапі дослідження було сформовано контрольну тестову вибірку з десяти унікальних фрагментів коду, створених на основі реальних репозиторіїв та баз даних вразливостей. Методологія передбачає синтез композитних тестових фрагментів коду шляхом рефакторингу та об'єднання декількох ізольованих патернів вразливостей (SQL-ін'єкції, небезпечна десеріалізація, XSS) в єдині функціональні блоки (див. таблицю 1). Таке підвищення щільності вразливостей у межах одного фрагменту дозволяє перевірити здатність різних підходів зберігати фокус при виявленні накладених безпекових вразливостей, оцінюючи їхню стійкість до інформаційного «шуму» та здатність розпізнавати складні залежності в межах фрагменту програмного коду [18].

Таблиця 1

Характеристики тестових фрагментів програмного коду

Code Snippet No.	Total Vulnerabilities Present	Vulnerabilities Types
1	4	SQL Injection, Cross-Site Scripting (XSS), Hardcoded Credentials, Insecure Cryptography
2	1	Hardcoded Credentials
3	6	SQL Injection, Insecure Deserialization, Broken Authentication, Improper Input Validation
4	3	XSS, Use of Outdated Libraries, Insecure File Handling
5	1	Insecure Direct Object Reference (IDOR)
6	3	Command Injection, Hardcoded Credentials, Insecure File Handling
7	6	SQL Injection, XSS, CSRF, Insecure Randomness, Missing Authorization
8	2	Hardcoded Credentials, Lack of Sanitization
9	3	XML External Entity (XXE), Insecure Session Management, Log Injection
10	1	XSS

На першому етапі дослідження для оцінки ефективності виявлення вразливостей фрагментів коду традиційними підходами було обрано статичний інструмент Roslyn Analyzers, що дозволило оцінити базовий рівень перевірки на основі детермінованих правил.

На наступному етапі ті самі фрагменти коду було проаналізовано на предмет наявності вразливостей моделями DeepSeek V3 та Grok 4.1 через платформу GitHub Models із використанням однакових спеціалізованих промптів для виявлення вразливостей та проведено порівняльний аналіз ефективності двох моделей [22], [23]. На заключному етапі, було протестовано запропонований гібридний підхід, що передбачає початкову перевірку коду статичним аналізатором з подальшою передачею його звітів на вхід обраних моделей генеративного інтелекту (див. рисунок 1). У обох випадках LLM виступає в ролі верифікатора, підтверджуючи або спростовуючи результати Roslyn Analyzers спираючись на логіку виконання програми та контекстуальні зв'язки, що дозволяє порівняти ефективність двох гібридних систем, а саме DeepSeek V3 та Grok 4.1 у поєднанні з Roslyn Analyzers у мінімізації хибних спрацювань.

Для забезпечення аналізу використовуються промпти (див. рисунок 2), які зобов'язують моделі DeepSeek V3 та Grok 4.1 виступати в ролі аналізаторів програмного коду. Запит визначає технічні межі аналізу таким чином, щоб моделі ігнорували стилістичні помилки коду та виявляли лише безпекові вразливості. Для

автоматизованої обробки результатів моделі генеруватимуть відповіді у форматі JSON-масиву, структура якого відповідає заданій схемі (див. рисунок 3).

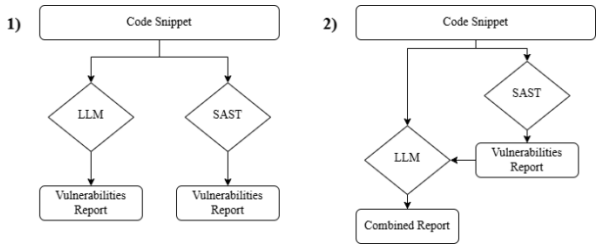

Рис. 1. Алгоритми (1) незалежних підходів на основі LLM та SAST та (2) гібридного підходів на основі комбінації двох інструментів

```
main_prompt = """
Act as a Senior Security Researcher and static analysis engine. Conduct a security-focused audit of the provided source code to
identify high-risk vulnerabilities, specifically targeting:
- Injection Flaws: SQL, Command, and XSS.
- Identity & Access: Insecure AuthN/AuthZ and hardcoded secrets.
- Data Integrity: Insecure deserialization and improper cryptography.
- Environment: Vulnerable dependencies and insecure file/path handling.
Scope: Ignore general bugs, stylistic issues, or code smells. Report only genuine security risks.
Output Format: You must respond exclusively with a JSON array using the following schema:
[{"RuleId": "string", "RuleDescription": "string", "Level": "Error|Warning|Note|None", "Message": "string", "Path": "string",
"Category": "string", "StartLine": integer, "EndLine": integer, "StartColumn": integer, "EndColumn": integer}]
"""
```

Рисунок 2. Приклад основного промпту для моделі

```
[
  {
    "RuleId": "SQL_INJECTION_CONCATENATION",
    "RuleDescription": "The application constructs a SQL query by concatenating user-controlled input directly into the query
    string, leading to SQL Injection.",
    "Level": "Error",
    "Message": "The variable 'username' is concatenated directly into the SQL string. An attacker could provide a malicious
    string (e.g., \"' OR '1'='1\") to bypass authentication or extract sensitive data. Use parameterized queries (SqlParameter)
    instead.",
    "Path": "SourceCode.cs",
    "Category": "Injection Flaws",
    "StartLine": 1,
    "EndLine": 1,
    "StartColumn": 1,
    "EndColumn": 69
  }
]
```

Рисунок 3 – Приклад відповіді моделі

Позначимо загальну кількість наявних вразливостей у фрагменті коду, як Total Present, а кількість правильно ідентифікованих вразливостей, підтверджених експертним висновком, як True Positive. Кількість пропущених загроз (False Negative) розраховується як різниця між загальною кількістю відомих у наборі даних вразливостей (Total Present) та числом знайдених та правильно класифікованих вразливостей (True Positive). Водночас кількість хибно позитивних спрацювань (False Positive) визначається шляхом віднімання кількості правильно ідентифікованих випадків (True Positive) від загальної кількості виявлених моделлю вразливостей (Total Found), тобто знахідка, що не відповідає задокументованій вразливості в тестовому наборі фрагментів коду, вважається хибно позитивним спрацюванням.

Оцінювання кожної стратегії проводиться за допомогою метрик, що характеризують якість виявлення вразливостей. Отримані дані дозволяють розрахувати метрику точності (Precision), яка визначається як частка правильно ідентифікованих вразливостей серед усіх знайдених інструментом вразливостей:

$$\text{Precision} = \frac{\text{True Positive}}{\text{Total Found}} = \frac{\text{True Positive}}{\text{True Positive+False Positive}} \quad (1)$$

Precision є показником чистоти роботи інструмента, тобто чим вищий показник, тим менше хибно позитивних результатів потрапляє до розробника, і тим менше "інформаційного шуму" створює інструмент у робочому процесі. Поряд із цим розраховується метрика повноти (Recall), що відображає вичерпність аналізу, вимірюючи частку реальних вразливостей, які були правильно виявлені:

$$\text{Recall} = \frac{\text{True Positive}}{\text{Total Present}} = \frac{\text{True Positive}}{\text{True Positive+False Negative}} \quad (2)$$

Ці два показники у сукупності дають змогу об'єктивно оцінити баланс між надійністю результатів та здатністю інструмента охопити весь спектр наявних загроз у програмному коді.

Для надання єдиної зваженої оцінки кожної стратегії використовується F1-score, що розраховується як середнє гармонійне значень Precision та Recall, дозволяючи об'єктивно порівнювати інструменти з різними балансами точності та повноти охоплення:

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision+Recall}} \quad (3)$$

У межах дослідження проводиться детальний аналіз цих метрик окремо для Roslyn Analyzer, моделей DeepSeek V3 і Grok 4.1, а також їхніх гібридних поєднань, що завершується систематизацією отриманих даних у порівняльну таблицю та графічні діаграми. Такий підхід забезпечує комплексну оцінку ефективності синергії LLM зі статичним аналізатором, наочно демонструючи переваги або недоліки комбінованого методу.

РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Результати експерименту демонструють ефективність трьох протестованих стратегій, де гібридний підхід виявився найкращою методологією для виявлення вразливостей у середовищі C#. На основі зведених даних, інтеграція детермінованого статичного аналізу з імовірнісним аналізом нейромереж помітно оптимізувала як точність, так і повноту виявлення.

Аналіз результатів роботи Roslyn Analyzer (див. таблицю 2) характеризує цей інструмент як достатньо високоточне, але недостатньо гнучке рішення. Попри успішну ідентифікацію стандартних шаблонів безпеки C# без зайвого «шуму», статичний аналізатор виявився вразливим до пропуску загроз (False Negatives) у фрагментах із нестандартною логікою або перетином різних вразливостей.

Таблиця 2

Результати роботи Roslyn Analyzers

Code Snippet No.	Total Present	Total Found	True Positives	False Negative	False Positive
1	4	4	4	0	0
2	1	1	1	0	0
3	6	5	4	3	1
4	3	2	2	0	0
5	1	1	1	1	0
6	3	3	3	0	0
7	6	5	4	2	1
8	2	2	2	1	0
9	3	3	3	0	0
10	**1**	**2**	**1**	**0**	**1**

Аналіз результатів незалежної роботи моделей DeepSeek та Grok, (див. таблицю 3) демонструє неоднорідний рівень ефективності моделей. Самостійне використання моделі Grok показало набагато гірші показники хибно негативних результатів (False

Negative), в порівнянні з моделлю DeepSeek, але при цьому дещо кращі показники хибно позитивних результатів (False Positive). Модель DeepSeek продемонструвала хороший рівень чутливості з меншими значеннями пропущених вразливостей (False Negative), проте вищими значеннями хибно позитивних випадків (False Positive) в порівнянні з інструментом Roslyn Analyzers. Очевидно, що серед трьох випадків незалежного використання інструментів Roslyn Analyzers, DeepSeek та Grok – саме модель Grok показує найслабші результати з високими показниками пропущених вразливостей.

Таблиця 3

Результати роботи DeepSeek V3 та Grok 4.1

Code Snippet No.	Total Present	DeepSeek V3				Grok 4.1			
		Total Found	True Positive	False Negative	False Positive	Total Found	True Positive	False Negative	False Positive
1	4	5	4	0	1	2	1	3	1
2	1	1	1	0	0	1	1	0	0
3	6	6	5	2	1	6	5	2	1
4	3	2	1	1	1	3	2	0	1
5	1	2	2	0	0	1	1	1	0
6	3	3	3	0	0	3	3	0	0
7	6	6	5	1	1	5	4	2	1
8	2	3	3	0	0	2	2	1	0
9	3	3	2	1	1	2	2	1	0
10	1	1	1	0	0	1	1	0	0

В таблиці 4 продемонстровані результати дослідження з використанням гібридних підходів для виявлення вразливостей. Комбінована робота DeepSeek та Roslyn Analyzers показує покращення результатів виявлення в порівнянні з незалежним використанням інструментів. Комбінація Grok та Roslyn Analyzers теж демонструє позитивну динаміку в порівнянні з незалежним використанням Grok, проте ці результати дещо нижчі за іншу комбінацію. Зокрема в обох комбінованих варіантах значення хибно позитивних результатів (False Positive) приблизно знаходяться на одному рівні, але комбінація Grok + Roslyn Analyzers дає більшу кількість хибно негативних (False Negative) результатів.

Таблиця 4

Результати гібридного підходу моделей Grok та DeepSeek разом з Roslyn Analyzers

Code Snippet No.	Total Present	DeepSeek V3 + Roslyn Analyzers				Grok 4.1 + Roslyn Analyzers			
		Total Found	True Positive	False Negative	False Positive	Total Found	True Positive	False Negative	False Positive
1	4	4	4	0	0	3	2	2	1
2	1	1	1	0	0	1	1	0	0
3	6	6	6	1	0	6	6	1	0
4	3	3	2	0	1	3	2	0	1
5	1	2	2	0	0	1	1	1	0
6	3	3	3	0	0	3	3	0	0
7	6	6	6	0	0	4	4	2	0
8	2	3	3	0	0	3	3	0	0
9	3	3	2	1	1	2	2	1	0
10	1	1	1	0	0	1	1	0	0

Таблиця 5 надає узагальнені показники для усіх використаних підходів. Згідно цих даних пара Grok 4.1 + Roslyn поступається аналогічній зв'язці з DeepSeek за рівнем повноти, але при цьому демонструє ідентичну середню точність результатів. Це вказує на те, що гібридний підхід мінімізує головний недолік моделей LLM – «галюцинації», які, в даному випадку, представлені хибно позитивними (False Positive) результатами. Таким чином поєднання Grok з інструментом статичного аналізу програмного коду перетворює модель з нестабільного аналізатора коду на надійний засіб верифікації, який тим не менш програє комбінації Roslyn Analyzers з DeepSeek.

Таблиця 5

Середні значення показників

Tool	Avg. Precison	Avg. Recall	Avg. F1 score
Roslyn Analyzer	0.91	0.84	0.85
DeepSeek V3	0.86	0.87	0.87
DeepSeek V3 + Roslyn Analyzer	0.93	0.95	0.94
Grok 4.1	0.88	0.75	0.79
Grok 4.1 + Roslyn Analyzer	0.93	0.82	0.86

Рисунок 4 наочно демонструє ефективність інтеграції статичного аналізу в процес роботи LLM, що виражається у позитивній динаміці всіх трьох ключових метрик для пари DeepSeek + Roslyn.

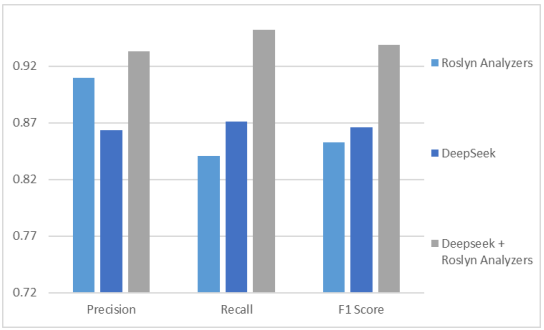

Рисунок 4. Порівняльна діаграма показників для DeepSeek та Roslyn Analyzers

Рисунок 5 демонструє найбільш динамічне зростання показників ефективності Grok при переході до гібридного підходу серед протестованих моделей. Проте показники повноти гібридного підходу демонструють нижчі результати, ніж повнота виявлення вразливостей статичним аналізатором. Також показник F1 Score в гібридному підході виріс менш, ніж на два відсотки в порівнянні з показником лише статичного аналізатора коду.

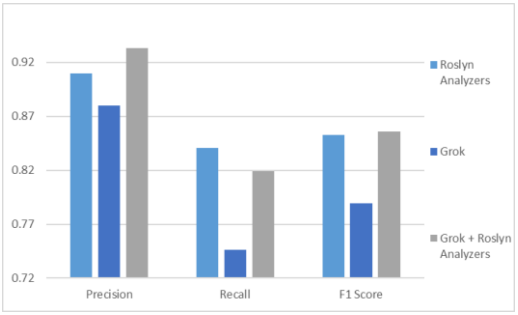

Рисунок 5. Порівняльна діаграма показників для Grok та Roslyn Analyzers

Динаміка показників вказує на те, що Grok може бути не найкращим варіантом для використання в задачах досліджень подібного типу.

ВИСНОВКИ ТА ПЕРСПЕКТИВИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

В межах дослідження було проведено порівняльний аналіз ефективності підходів до виявлення безпекових вразливостей в програмному коді на мові C#. За методикою дослідження було підготовано десять фрагментів коду на мові C# з різними типами безпекових вразливостей. В ході експерименту було проведено аналіз фрагментів коду спочатку за допомогою окремо інструменту Roslyn Analyzers та моделей DeepSeek і Grok, а потім за допомогою гібридного підходу, в межах якого результати роботи статичного сканера передавалися великим мовним моделям для верифікації в якості додаткового контексту.

Порівняльний аналіз експериментальних даних показав, що підхід, який поєднує DeepSeek V3 та Roslyn Analyzer, виявився найкращим. Він досяг кращих результатів за показниками точності та повноти охоплення виявлення загроз у програмному коді. Однак модель Grok 4.1, яка працює разом з Roslyn, хоч і трохи поступається за інтегральною стабільністю, показує більш виражену позитивну динаміку покращення показників порівняно зі своєю автономною версією. Автономне використання моделі Grok 4.1, в свою чергу, показало найгірші результати серед представлених підходів.

Поєднання окремих процесів виявлення вразливостей дозволило покращити їх шляхом перетворення імовірнісних припущень моделей на більш детерміновані та перевірені результати. Це підкреслює потенціал використання сучасних великих мовних моделей як верифікаторів для традиційних інструментів аналізу.

Експеримент також виявив проблему залежності результатів від промпт-інжинірингу. Автономні великі мовні моделі демонструють більш помітний рівень помилок, працюючи без контексту, в порівнянні зі статичними аналізаторами.

Складнощі в інтерпретації деяких складних конструкцій C#, таких як асинхронні патерни, вказують на те, що подальші дослідження в перспективі мають бути спрямовані на пошук і використання більш спеціалізованих моделей. Розширення переліку статичних інструментів за межі Roslyn Analyzer дозволить отримати ширший спектр результатів для перехресної перевірки даних.

Одним із важливих напрямків майбутніх досліджень є створення покращених датасетів, що містять багатошарові вразливості. Також потрібно досліджувати підходи з використанням мультиагентів, де одна велика мовна модель генерує звіт, а інша намагається його спростувати або верифікувати. Кінцевою метою подальших досліджень може бути розробка самокоригованої екосистеми аналізу безпеки, здатної покращуватися синхронно зі зростанням комплексності програмного коду.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Shahana, A., Hasan, R., Farabi, S. F., Akter, J., Mahmud, M. A. A., Johora, F. T., & Suzer, G. (2024). AI-driven cybersecurity: Balancing advancements and safeguards. Journal of Computer Science and Technology Studies, 6(2), 76–85. https://doi.org/10.32996/jcsts.2024.6.2.9
2. Gnieciak, D., & Szandala, T. (2025). Large language models versus static code analysis tools: A systematic benchmark for vulnerability detection. IEEE Access. https://doi.org/10.1109/access.2025.3635168
3. Ferrag, M. A., Battah, A., Tihanyi, N., Jain, R., Maimuț, D., Alwahedi, F., Lestable, T., Thandi, N. S., Mechri, A., Debbah, M., & Cordeiro, L. C. (2025). SecureFalcon: Are we there yet in automated software vulnerability detection with LLMs? IEEE Transactions on Software Engineering, 1–18. https://doi.org/10.1109/tse.2025.3548168

4. Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., Wagner, D., Ray, B., & Chen, Y. (2025). Vulnerability detection with code language models: How far are we? In Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE) (pp. 1729–1741). IEEE. https://doi.org/10.1109/icse55347.2025.00038
5. Curaba, C., D'Ambrosi, D., Minisini, A., & Pérez-Campanero, N. (2024). Cryptoformaleval: Integrating LLMs and formal verification for automated cryptographic protocol vulnerability detection. arXiv.
6. Du, X., Wen, M., Zhu, J., Xie, Z., Ji, B., Liu, H., Shi, X., & Jin, H. (2024). Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. In Findings of the Association for Computational Linguistics (ACL 2024) (pp. 10507–10521). https://doi.org/10.18653/v1/2024.findings-acl.625
7. He, J., & Vechev, M. (2023). Large language models for code: Security hardening and adversarial testing. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS '23). ACM. https://doi.org/10.1145/3576915.3623175
8. Beljulji, E., & Matta, I. (2026). Large language models in security code review and testing. Journal of Systems Research, 5(1). https://doi.org/10.5070/sr3.62177
9. Shvyrov, V. V., Kapustin, D. A., Sentyay, R. N., & Shulika, T. I. (2024). Analysis of datasets and large language models for vulnerability detection in imperative programming languages. Programmnaya Ingeneria, 15(11), 555–569. https://doi.org/10.17587/prin.15.555-569
10. Bhandari, G., Naseer, A., & Moonen, L. (2021). CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software. In Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21). ACM. https://doi.org/10.1145/3475960.3475985
11. Chen, Y., Ding, Z., Alowain, L., Chen, X., & Wagner, D. (2023). DiverseVul: A new vulnerable source code dataset for deep learning-based vulnerability detection. In Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2023). ACM. https://doi.org/10.1145/3607199.3607242
12. Kouliaridis, V., Karopoulos, G., & Kambourakis, G. (2025). Assessing the effectiveness of LLMs in Android application vulnerability analysis. In Lecture Notes in Computer Science (pp. 139–154). Springer. https://doi.org/10.1007/978-3-031-85593-1_9
13. Ajiga, D., Okeleke, P. A., Folorunsho, S. O., & Ezeigweneme, C. (2024). The role of software automation in improving industrial operations and efficiency. International Journal of Engineering Research Updates, 7(1), 22–35. https://doi.org/10.53430/ijeru.2024.7.1.0031
14. Venkatasubramanyam, R. D., Gupta, S., & Uppili, U. (2015). Assessing the effectiveness of static analysis through defect correlation analysis. In 2015 IEEE International Conference on Global Software Engineering (ICGSE). IEEE. https://doi.org/10.1109/icgse.2015.18
15. Singh, D., Sekar, V. R., Stolee, K. T., & Johnson, B. (2017). Evaluating how static analysis tools can reduce code review effort. In 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE. https://doi.org/10.1109/vlhcc.2017.8103456
16. Simões, I. R. S., & Venson, E. (2024). Evaluating source code quality with large language models: A comparative study. In Proceedings of the Brazilian Symposium on Software Quality (SBQS 2024) (pp. 103–113). ACM. https://doi.org/10.1145/3701625.3701650
17. Cheirdari, F., & Karabatis, G. (2018). Analyzing false-positive source code vulnerabilities using static analysis tools. In 2018 IEEE International Conference on Big Data. IEEE. https://doi.org/10.1109/bigdata.2018.8622456
18. Khater, H. M., Khayat, M., Alrabaee, S., Serhani, M. A., Barka, E., & Sallabi, F. (2023). AI techniques for software vulnerability detection and mitigation. In 2023 IEEE Conference on Dependable and Secure Computing (DSC). IEEE. https://doi.org/10.1109/dsc61021.2023.10354233
19. Adebayo, A. O. (2025). Automating security compliance in DevSecOps through AI-driven policy enforcement. International Journal of Science and Research Archive, 15(2), 670–675. https://doi.org/10.30574/ijsra.2025.15.2.1457
20. Mohammed, A. (2023). Elevating cybersecurity audits: How AI is shaping compliance and threat detection. Zenodo. https://doi.org/10.5281/zenodo.14760068
21. Babatunde, L. A., Etim, E. D., Essien, I. A., Cadet, E., Ajayi, J. O., Erigha, E. D., & Obuse, E. (2020). Adversarial machine learning in cybersecurity: Vulnerabilities and defense strategies. Journal of Frontiers in Multidisciplinary Research, 1(2), 31–45. https://doi.org/10.54660/.jfmr.2020.1.2.31-45
22. GitHub. (n.d.). GitHub models marketplace. https://github.com/marketplace?type=models

23. Ponta, S. E., Plate, H., & Sabetta, A. (2020). Detection, assessment and mitigation of vulnerabilities in open-source dependencies. Empirical Software Engineering, 25(5), 3175–3215. https://doi.org/10.1007/s10664-020-09830-x

КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 4 (32), 2026
ISSN 2663 – 4023

Oleh Yarema
PhD student of Cybersecurity Department
Ternopil Ivan Puluj National Technical University, Ternopil, Ukraine
ORCID: 0009-0009-8709-7813
yarema.oleh.m@gmail.com

Nataliya Zagorodna
PhD (Technical Sciences), Head of the Cybersecurity Department
Ternopil Ivan Puluj National Technical University, Ternopil, Ukraine
ORCID: 0000-0002-1808-835X
zagorodna_n@tntu.edu.ua

COMPARATIVE ANALYSIS OF THE EFFECTIVENESS OF SOFTWARE CODE VULNERABILITY DETECTION USING LLM AND SAST

Abstract. This article justifies the need to implement software code security controls using large language models (LLMs), driven by the rapid growth in the volume of software code, the emergence of new security risks associated with AI-generated code, and the need to integrate individual code components into complex architectural solutions. The algorithms of existing static code analysis (SAST) tools are prone to errors due to their inability to fully account for code execution logic and its contextual relationships. Using LLMs as a verifier that confirms or refutes the results of static code analysis has the potential to address these shortcomings. This paper presents a comparative analysis of the effectiveness of detecting security vulnerabilities in C# code using the Roslyn Analyzers static code analysis tool, large language models such as DeepSeek and Grok, and an integrated approach that combines the advantages of static analyzers and LLMs. The research methodology is based on conducting an experimental study using a test sample of C# code fragments containing various types of security vulnerabilities. In the first stage of the study, the code fragments were tested using the static code analysis tool Roslyn Analyzers. In the next stage, the code fragments were analyzed for vulnerabilities using the DeepSeek V3 and Grok 4.1 models. In the final stage, the effectiveness of the proposed hybrid approach was evaluated, which involves an initial code check by a static analyzer followed by the transmission of its reports to the input of selected generative AI models. The results of the study show that the hybrid approach using DeepSeek and Roslyn Analyzers provides an increase in performance metrics compared to the independent use of these tools. A comparative analysis of the performance metrics of the models' standalone use also established that Grok performs worse than the DeepSeek model and is not the best option for application in tasks of this type. The study demonstrates that integrating the analytical capabilities of large language models into classical static code analysis processes by confirming or refuting the results of static analysis is a potential step toward self-correcting software security analysis processes.

Keywords: software; vulnerabilities; testing; artificial intelligence; LLM; static code analysis; SAST; hybrid approach.

REFERENCES (TRANSLATED AND TRANSLITERATED)

1. Shahana, A., Hasan, R., Farabi, S. F., Akter, J., Mahmud, M. A. A., Johora, F. T., & Suzer, G. (2024). AI-driven cybersecurity: Balancing advancements and safeguards. Journal of Computer Science and Technology Studies, 6(2), 76–85. https://doi.org/10.32996/jcsts.2024.6.2.9
2. Gnieciak, D., & Szandala, T. (2025). Large language models versus static code analysis tools: A systematic benchmark for vulnerability detection. IEEE Access. https://doi.org/10.1109/access.2025.3635168
3. Ferrag, M. A., Battah, A., Tihanyi, N., Jain, R., Maimuț, D., Alwahedi, F., Lestable, T., Thandi, N. S., Mechri, A., Debbah, M., & Cordeiro, L. C. (2025). SecureFalcon: Are we there yet in automated software vulnerability detection with LLMs? IEEE Transactions on Software Engineering, 1–18. https://doi.org/10.1109/tse.2025.3548168

4. Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., Wagner, D., Ray, B., & Chen, Y. (2025). Vulnerability detection with code language models: How far are we? In Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE) (pp. 1729–1741). IEEE. https://doi.org/10.1109/icse55347.2025.00038
5. Curaba, C., D'Ambrosi, D., Minisini, A., & Pérez-Campanero, N. (2024). Cryptoformaleval: Integrating LLMs and formal verification for automated cryptographic protocol vulnerability detection. arXiv.
6. Du, X., Wen, M., Zhu, J., Xie, Z., Ji, B., Liu, H., Shi, X., & Jin, H. (2024). Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. In Findings of the Association for Computational Linguistics (ACL 2024) (pp. 10507–10521). https://doi.org/10.18653/v1/2024.findings-acl.625
7. He, J., & Vechev, M. (2023). Large language models for code: Security hardening and adversarial testing. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS '23). ACM. https://doi.org/10.1145/3576915.3623175
8. Beljulji, E., & Matta, I. (2026). Large language models in security code review and testing. Journal of Systems Research, 5(1). https://doi.org/10.5070/sr3.62177
9. Shvyrov, V. V., Kapustin, D. A., Sentyay, R. N., & Shulika, T. I. (2024). Analysis of datasets and large language models for vulnerability detection in imperative programming languages. Programmnaya Ingeneria, 15(11), 555–569. https://doi.org/10.17587/prin.15.555-569
10. Bhandari, G., Naseer, A., & Moonen, L. (2021). CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software. In Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21). ACM. https://doi.org/10.1145/3475960.3475985
11. Chen, Y., Ding, Z., Alowain, L., Chen, X., & Wagner, D. (2023). DiverseVul: A new vulnerable source code dataset for deep learning-based vulnerability detection. In Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2023). ACM. https://doi.org/10.1145/3607199.3607242
12. Kouliaridis, V., Karopoulos, G., & Kambourakis, G. (2025). Assessing the effectiveness of LLMs in Android application vulnerability analysis. In Lecture Notes in Computer Science (pp. 139–154). Springer. https://doi.org/10.1007/978-3-031-85593-1_9
13. Ajiga, D., Okeleke, P. A., Folorunsho, S. O., & Ezeigweneme, C. (2024). The role of software automation in improving industrial operations and efficiency. International Journal of Engineering Research Updates, 7(1), 22–35. https://doi.org/10.53430/ijeru.2024.7.1.0031
14. Venkatasubramanyam, R. D., Gupta, S., & Uppili, U. (2015). Assessing the effectiveness of static analysis through defect correlation analysis. In 2015 IEEE International Conference on Global Software Engineering (ICGSE). IEEE. https://doi.org/10.1109/icgse.2015.18
15. Singh, D., Sekar, V. R., Stolee, K. T., & Johnson, B. (2017). Evaluating how static analysis tools can reduce code review effort. In 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE. https://doi.org/10.1109/vlhcc.2017.8103456
16. Simões, I. R. S., & Venson, E. (2024). Evaluating source code quality with large language models: A comparative study. In Proceedings of the Brazilian Symposium on Software Quality (SBQS 2024) (pp. 103–113). ACM. https://doi.org/10.1145/3701625.3701650
17. Cheirdari, F., & Karabatis, G. (2018). Analyzing false-positive source code vulnerabilities using static analysis tools. In 2018 IEEE International Conference on Big Data. IEEE. https://doi.org/10.1109/bigdata.2018.8622456
18. Khater, H. M., Khayat, M., Alrabaee, S., Serhani, M. A., Barka, E., & Sallabi, F. (2023). AI techniques for software vulnerability detection and mitigation. In 2023 IEEE Conference on Dependable and Secure Computing (DSC). IEEE. https://doi.org/10.1109/dsc61021.2023.10354233
19. Adebayo, A. O. (2025). Automating security compliance in DevSecOps through AI-driven policy enforcement. International Journal of Science and Research Archive, 15(2), 670–675. https://doi.org/10.30574/ijsra.2025.15.2.1457
20. Mohammed, A. (2023). Elevating cybersecurity audits: How AI is shaping compliance and threat detection. Zenodo. https://doi.org/10.5281/zenodo.14760068
21. Babatunde, L. A., Etim, E. D., Essien, I. A., Cadet, E., Ajayi, J. O., Erigha, E. D., & Obuse, E. (2020). Adversarial machine learning in cybersecurity: Vulnerabilities and defense strategies. Journal of Frontiers in Multidisciplinary Research, 1(2), 31–45. https://doi.org/10.54660/.jfmr.2020.1.2.31-45
22. GitHub. (n.d.). GitHub models marketplace. https://github.com/marketplace?type=models

23. Ponta, S. E., Plate, H., & Sabetta, A. (2020). Detection, assessment and mitigation of vulnerabilities in open-source dependencies. Empirical Software Engineering, 25(5), 3175–3215. https://doi.org/10.1007/s10664-020-09830-x

Отримано редакцією журналу / Received: 21.01.26
Прорецензовано / Revised: 15.02.26
Схвалено до друку / Accepted: 26.03.26