Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Міністерство освіти і науки України

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Міністерство освіти і науки України

Кваліфікаційна наукова
праця на правах рукопису

ВОХРАНОВ ІЛЛЯ АНАТОЛІЙОВИЧ

УДК 004.8+004.41

ДИСЕРТАЦІЯ

Метод статичного аналізу вихідного коду програм із використанням великих мовних моделей

122 - Комп’ютерні науки
12 - Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії
Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

І. А. Вохранов

Науковий керівник: Булах Богдан Вікторович, к.т.н

Київ – 2026

АНОТАЦІЯ

Вохранов І. А. Метод статичного аналізу вихідного коду програм із використанням великих мовних моделей. — Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 — Комп'ютерні науки. — Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», Міністерство освіти і науки України, Київ, 2026.

Метою дисертаційного дослідження є розробка, формалізований опис та експериментальна оцінка нейросимвольного методу статичного аналізу програмного коду з використанням великих мовних моделей, який забезпечує контрольовану поведінку LLM через незалежну символьну верифікацію генерованих тверджень, що дозволяє досягти високої повноти виявлення дефектів при збереженні точності та передбачуваності результатів.

Об'єктом дослідження є процеси статичного аналізу програмного коду, спрямовані на виявлення дефектів та вразливостей.

Предметом дослідження є нейросимвольні методи статичного аналізу, що поєднують семантичне розуміння великих мовних моделей з незалежною символьною верифікацією для забезпечення контрольованої та передбачуваної поведінки.

У ході роботи над дисертацією було використано такі **методи дослідження**: аналіз та систематизація наукових джерел для визначення стану досліджень у галузі статичного аналізу коду та нейросимвольних систем; формалізація для опису архітектури та алгоритмів запропонованого методу; моделювання для побудови архітектури запропонованого методу та взаємодії його компонентів; експеримент для емпіричної оцінки ефективності методу на реальних даних; порівняльний аналіз для зіставлення результатів із існуючими підходами; абляційне дослідження для визначення внеску кожного архітектурного

компоненту. Зазначені методи було обрано з огляду на мету та завдання дослідження, а також на їх придатність для вирішення проблем контрольованого використання великих мовних моделей у задачах статичного аналізу коду.

У дисертації *вперше запропоновано* нейросимвольний метод статичного аналізу програмного коду, який відрізняється тим, що *на першому етапі аналізу LLM безпосередньо генерує структуровані твердження про дефекти коду, за чим слідує етап символьного аналізу для їх верифікації*, в той час, як в існуючих нейросимвольних методах статичного аналізу коду (IRIS, LLMSA, MoCQ) навпаки - символьний компонент виконує первинне виявлення або структуризацію дефектів, а нейронний компонент фільтрує чи доповнює його результати. Інвертований напрямок нейросимвольної верифікації "Neural → Symbolic" дозволяє скористатися здатністю великої мовної моделі сприймати семантику коду при одночасному жорсткому контролі згенерованих нею тверджень, що сприяє виявленню ширшого кола дефектів при мінімальних ризиках галюцинації моделі.

Вперше запропоновано спосіб верифікації структурованих тверджень про дефекти коду із застосуванням символьного аналізу Explain-Verify Gate, особливість якого полягає у тому, що атрибути твердження про дефект (тип дефекту, локалізація, пропозиція мінімально задовільного виправлення та обґрунтування) визначають вибір та інтерпретацію подальших символьних перевірок (зіставлення AST-патернів, taint-аналіз, перевірка правдоподібності виправлення), що є відмінним від існуючих способів верифікації у нейросимвольних методах аналізу коду та забезпечує детермінізм верифікації за відсутності потреби повторного звернення до мовної моделі за обґрунтуванням кожного прийнятого нею рішення.

Удосконалено процес прийняття рішень щодо дефектів у нейросимвольному статичному аналізі програмного коду шляхом *застосування механізму Evidence-Weighted Suppression (EWS)*, який зважує докази з кількох незалежних джерел (інструменти статичного аналізу, AST-патерни, показник впевненості базового аналізатора), що відрізняється від процесу визначення дефектів в рамках

існуючих нейросимвольних підходів, в яких відхилення результату на етапі верифікації є остаточним без урахування доказів з інших джерел. Це дозволяє збалансувати точність і повноту аналізу шляхом відновлення валідних виявлень дефектів, що відхиляються символьною верифікацією за відсутності відповідних патернів, але підтверджуються достатніми доказами з інших джерел.

Розроблений метод статичного аналізу коду *дає змогу виявляти додаткові дефекти*, які не виявляються як інструментами на основі правил, так і безпосереднім використанням LLM. Експериментальна оцінка на 1000 зразків із набору даних PySStuBs підтвердила виявлення 247 додаткових дефектів порівняно з базовим LLM-підходом (379 проти 132) та 371 додаткового дефекту порівняно з ансамблем інструментів на основі правил (379 проти 8), що відповідає покращенню recall на 187% та F1-score на 76.4% при збереженні співставного рівня точності (~0.51). Виявлення цих додаткових дефектів на етапі статичного аналізу запобігає їх потраплянню на подальші етапи розробки, де вартість їх виправлення суттєво зростає. Результати аналізу в рамках запропонованого методу *представлені структурованим описом дефекту* (тип, локалізація, обґрунтування, мінімально необхідне виправлення), який завжди *супроводжується результатами символьної верифікації*, що скорочує час розробника на аналіз та усунення виявлених проблем та суттєво полегшує інтеграцію методу в автоматизовані процеси контролю якості коду (CI/CD, code review), де рішення про дефект має спиратися на перевірені докази, а не лише на вихід мовної моделі. Запропонований метод аналізу *не потребує донавчання моделей та збору спеціалізованих датасетів*, що зменшує витрати на впровадження та супровід порівняно з підходами на основі fine-tuning, які вимагають підготовки навчальних даних, обчислювальних ресурсів для тренування та повторного навчання при зміні моделі. Експериментальне застосування GPT-5.2 (RQ2) підтвердило збереження переваг запропонованого методу у порівнянні з існуючими методами статичного аналізу коду при переході на новіше покоління мовної моделі (+132% F1-score, +363% recall), що свідчить про можливість оновлення нейронного компоненту без

модифікації решти складових системи, що суттєво економить витрати. Метод *забезпечує можливість налаштування під різноманітні конкретні задачі аналізу* без зміни загальної архітектури: тип аналізу та цільові класи дефектів визначаються у змісті запиту до нейромережевого компоненту (аналогічно до властивості customizable analysis у LLMSA), а покриття символьної верифікації розширюється додаванням нових правил, AST-патернів та taint-специфікацій для підтримки нових мов програмування та типів дефектів. Абляційне дослідження (RQ3) підтвердило стійкість архітектури до модифікацій окремих компонентів: F1-score між конфігураціями варіюється у діапазоні 0.654–0.668.

Дисертація складається з п'яти розділів.

У **першому розділі** розглянуто теоретичні основи аналізу програмного коду з використанням нейронних мереж. Проаналізовано класичні методи статичного аналізу (аналіз потоку даних, аналіз потоку управління, абстрактна інтерпретація, символьне виконання, model checking, синтаксичний аналіз та pattern matching), визначено їх переваги та фундаментальні обмеження, зокрема проблему вибуху шляхів та обмеження SMT-вирішувачів. Досліджено еволюцію нейронних методів в аналізі коду — від класичного машинного навчання через попередньо навчені моделі (CodeBERT, GraphCodeBERT) до великих мовних моделей (GPT-4, GPT-5.2). Розглянуто концепцію нейросимвольної інтеграції та типи нейросимвольних систем, що обґрунтовує необхідність гібридного підходу, який поєднує семантичне розуміння нейронних моделей з формальними гарантіями символьних методів.

У **другому розділі** проведено аналіз сучасних досліджень у галузі статичного аналізу коду. Систематизовано існуючі нейросимвольні методи аналізу програм: IRIS (символьний → нейронний напрямок, CodeQL → LLM), LLMSA (композиційний нейросимвольний аналіз), MoCQ (ітеративне уточнення патернів). Визначено спільне обмеження існуючих методів — відсутність явного механізму незалежної верифікації тверджень LLM. Сформульовано вимоги до методу, що заповнює цю прогалину, та обґрунтовано позиціонування VERIGATE як першого

методу, що реалізує напрямок Neural → Symbolic у контексті статичного аналізу коду.

У **третьому розділі** детально описано запропонований метод VERIGATE. Представлено архітектуру чотириетапного методу: попередній скринінг (ансамбль інструментів та AST-правила), Explain (генерація структурованих тверджень LLM у форматі JSON), Verify (символьна верифікація через багатоетапне дерево рішень), фінальне рішення з механізмом Evidence-Weighted Suppression. Формалізовано процес верифікації, описано критерії прийняття та відхилення тверджень, механізм вирівнювання спанів та taint analysis. Проведено порівняння архітектурних рішень VERIGATE з існуючими нейросимвольними системами за ключовими характеристиками (напрямок взаємодії, роль LLM, роль символьної системи, пояснюваність). Описано деталі реалізації: технологічний стек, формат промпту для LLM, обробку виходу LLM. Наведено наскрізний приклад роботи VERIGATE на реальному фрагменті коду з датасету PySStuBs, що демонструє нейросимвольну взаємодію компонентів на прикладі виявлення семантичного логічного дефекту.

У **четвертому розділі** представлено результати експериментальної оцінки. Основна оцінка (RQ1) проведена на 1 000 збалансованих зразках із датасету PySStuBs (Python Single-Statement Bugs) з використанням GPT-4. Порівняння трьох методів (VERIGATE, GPT-4 Baseline, Rule-Based) продемонструвало: покращення F1-score на 76.4% (0.612 проти 0.347) та recall на 187% (0.758 проти 0.264) порівняно з GPT-4 Baseline при порівняному precision (~0.51). Інструменти на основі правил продемонстрували дуже низьку ефективність (F1 = 0.031), що підтверджує семантичну складність дефектів у датасеті. Валідація з GPT-5.2 (RQ2, n = 500) підтвердила стабільне покращення: +132% F1-score, +363% recall, що свідчить про стійкість нейросимвольної верифікації як архітектурного принципу при розвитку мовних моделей. Абляційне дослідження (RQ3, n = 500) продемонструвало, що VERIGATE функціонує як інтегрований поетапний процес

з вузьким діапазоном F1 (0.654–0.668) для всіх конфігурацій, де кожен компонент робить незалежний внесок у загальну ефективність.

У **п'ятому розділі** досліджено перспективи розвитку та узагальнення запропонованого методу. Проаналізовано напрямки розширення на нові мови програмування, нові класи дефектів та інтеграцію з виробничими процесами розробки програмного забезпечення (CI/CD, code review, IDE). Визначено обмеження поточної реалізації та шляхи їх подолання.

У дисертації запропоновано нейросимвольний метод VERIGATE для статичного аналізу програмного коду, який через використання незалежної символьної верифікації тверджень LLM дозволяє поєднати семантичне розуміння програмного коду великої мовної моделі із обґрунтуванням кожного виявлення у верифікованих властивостях програми. Основна ідея методу полягає в інверсії напрямку нейросимвольної взаємодії: замість традиційного підходу, де символьний аналіз генерує результати, а нейронна мережа їх фільтрує, у VERIGATE LLM генерує структуровані гіпотези, а символьний аналіз незалежно їх верифікує. У ході експериментальних досліджень було підтверджено ефективність запропонованого методу: покращення F1-score на 76.4% та recall на 187% порівняно з GPT-4 Baseline, стабільне покращення з GPT-5.2, а також збалансований внесок кожного архітектурного компоненту.

Особистий внесок здобувача. Усі основні результати дисертаційного дослідження, представлені до захисту, одержані автором особисто. Здобувачем виконано: аналіз існуючих методів статичного аналізу коду та нейросимвольних систем; розробку нейросимвольного методу VERIGATE з механізмами Explain-Verify Gate та Evidence-Weighted Suppression; формалізацію процесу верифікації структурованих тверджень; розробку програмного прототипу; проведення експериментальної оцінки та аналіз результатів. У публікаціях у співавторстві здобувачеві належать: дослідження існуючих методів, проектування архітектури нейросимвольного методу, розробка механізмів верифікації та

прийняття рішень, експериментальне підтвердження ефективності запропонованого методу.

Апробація матеріалів дисертації. Основні положення та отримані наукові результати, що викладені в даній дисертації, пройшли апробацію через їх презентацію на міжнародній науковій конференції — *4th International Scientific and Practical Conference «Science and Information Technologies in the Modern World» (Афіни, Греція, 24–26 грудня 2025р.)*.

Ключові слова: великі мовні моделі, нейросимвольний аналіз, статичний аналіз коду, верифікація програм, символьний аналіз, виявлення дефектів, аналіз вразливостей, семантичний аналіз, машинне навчання, нейронні мережі, штучний інтелект, інженерія програмного забезпечення, надійність програмного забезпечення, інтелектуальна обробка, життєвий цикл розробки програмного забезпечення.

Список публікацій здобувача за темою дисертації:

[1] Vokhranov, I., & Bulakh, B. (2026). VERIGATE: A verification gate-based neurosymbolic method for static code analysis. *Наука і Техніка Сьогодні, 2*(56), 1488–1507. - DOI 10.52058/2786-6025-2026-2(56)-1488-1507. Здобувачем запропоновано нейросимвольний метод VERIGATE з механізмами Explain-Verify Gate та Evidence-Weighted Suppression, розроблено формальний опис архітектури запропонованого методу та взаємодії його компонентів, здійснено розробку програмного прототипу, проведено експериментальну оцінку на наборі даних PySStuBs та виконано аналіз отриманих результатів. Булах Б.В. надав рекомендації щодо визначення структури методу, брав участь в отриманні та обробці експериментальних даних з використанням програмного прототипу.

[2] Vokhranov, I., & Bulakh, B. (2024). Transformer-based models application for bug detection in source code. *Technology Audit and Production Reserves, 5*(2(79)), 6–15. - DOI 10.15587/2706-5448.2024.310822. Здобувачем проведено аналіз існуючих підходів до застосування моделей на основі архітектури трансформера для виявлення дефектів у програмному коді, розроблено методологію

експериментів із донавчанням моделей BERT, CodeBERT, GPT-2 та CodeT5 на наборі даних RunBugRun, здійснено підготовку даних та проведення експериментів для різних типів дефектів, виконано порівняльний аналіз та інтерпретацію результатів. Булах Б.В. брав участь у визначенні напряму дослідження та обґрунтуванні вибору моделей, надав рекомендації щодо методології проведення експериментів та інтерпретації отриманих результатів.

[3] Вохранов, І., & Булах, Б. (2023). Комбіновані підходи до статичного аналізу коду з використанням нейронних мереж. *Інфокомунікаційні та комп'ютерні технології, 1*(05), 183–193. - DOI 10.36994/2788-5518-2023-01-05-20. Здобувачем проведено систематичний аналіз існуючих підходів до застосування нейронних мереж у статичному аналізі коду за трьома категоріями (коригування специфікацій, визначення специфікацій, аналіз «чорного ящика»), здійснено класифікацію та порівняльний аналіз методів кожної категорії, сформульовано висновки щодо перспективних напрямків комбінування підходів. Булах Б.В. належить постановка задачі дослідження, участь у формуванні структури огляду, надання рекомендацій щодо відбору джерел та уточнення висновків.

[4] Petrenko, A. I., & Vokhranov, I. A. (2023). Neural networks: Studying their decision-making rules. *System Research and Information Technologies, 2023*(2), 74–85. - DOI 10.20535/SRIT.2308-8893.2023.2.06. Здобувачем розроблено методологію дослідження, реалізовано програмний прототип алгоритму декомпозиції DeepRED із запропонованим покращенням, проведено експерименти та виконано аналіз отриманих результатів, підготовлено текст рукопису. Петренко А.І. визначив постановку задачі та напрям дослідження, здійснив наукове керівництво роботою, виконав редагування та підготовку рукопису до публікації.

[5] Вохранов, І. (2025). Нейросимвольні підходи в аналізі програмного коду: огляд сучасного стану та перспективи розвитку. *4th International Scientific and Practical Conference «Science and Information Technologies in the Modern World»*, 161–164. Здобувачем проведено презентацію на міжнародній науковій конференції.

ABSTRACT

Vokhranov I. A. Method of Static Analysis of Program Source Code Using Large Language Models. — Qualifying scientific work as a manuscript.

Dissertation for the degree of Doctor of Philosophy in specialty 122 — Computer Science. — National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Ministry of Education and Science of Ukraine, Kyiv, 2026.

The aim of the dissertation research is the development, formal description, and experimental evaluation of a neurosymbolic method for static analysis of program source code using large language models, which ensures controlled LLM behavior through independent symbolic verification of generated claims, enabling high defect detection recall while preserving precision and predictability of results.

The object of the research is the processes of static analysis of program source code aimed at detecting defects and vulnerabilities.

The subject of the research is neurosymbolic methods for static analysis that combine the semantic understanding of large language models with independent symbolic verification to ensure controlled and predictable behavior.

The following **research methods** were used in the dissertation: analysis and systematization of scientific sources to determine the state of research in the fields of static code analysis and neurosymbolic systems; formalization for describing the architecture and algorithms of the proposed method; modeling for designing the architecture of the proposed method and the interactions of its components; experiment for empirical evaluation of the method's effectiveness on real data; comparative analysis for comparing results with existing approaches; ablation study for determining the contribution of each architectural component. These methods were chosen considering the aim and objectives of the research, as well as their suitability for addressing the problems of controllable usage of large language models in static code analysis tasks.

The dissertation *proposes for the first time* a neurosymbolic method for static analysis of program source code, which is distinguished by the fact that *at the first stage*

of analysis the LLM directly generates structured claims about code defects, followed by a symbolic analysis stage for their verification, whereas in existing neurosymbolic methods for static code analysis (IRIS, LLMSA, MoCQ) the opposite holds — the symbolic component performs primary detection or structuring of defects, while the neural component filters or supplements its results. The inverted direction of neurosymbolic verification "Neural → Symbolic" enables leveraging the large language model's ability to perceive code semantics while simultaneously maintaining strict control over its generated claims, facilitating the detection of a broader range of defects with minimal risks of model hallucination.

For the first time, an approach to verification of structured defect claims using symbolic analysis — the Explain-Verify Gate — is proposed, whose distinctive feature is that the attributes of the defect claim (defect type, location, proposed minimally sufficient fix, and rationale) determine the selection and interpretation of subsequent symbolic checks (AST pattern matching, taint analysis, fix plausibility checking), which differs from existing verification approaches in neurosymbolic code analysis methods and ensures verification determinism without the need to re-query the language model for justification of each decision it has made.

The defect *decision-making process* in neurosymbolic static code analysis is *improved through the application of the Evidence-Weighted Suppression (EWS) mechanism*, which weighs evidence from multiple independent sources (static analysis tools, AST patterns, base analyzer confidence score), differing from the defect determination process in existing neurosymbolic approaches, in which rejection of a result at the verification stage is final without considering evidence from other sources. This enables balancing analysis precision and recall by recovering valid defect detections that are rejected by symbolic verification due to the absence of matching patterns but are supported by sufficient evidence from other sources.

The developed method for static code analysis *enables the detection of additional defects* that are missed both by rule-based tools and by direct use of the LLM. Experimental evaluation on 1,000 samples from the PySStuBs dataset confirmed the

detection of 247 additional defects compared to the baseline LLM approach (379 vs. 132) and 371 additional defects compared to the rule-based tool ensemble (379 vs. 8), corresponding to a recall improvement of 187% and an F1-score improvement of 76.4% while maintaining a comparable precision level (~0.51). Detection of these additional defects at the static analysis stage prevents them from reaching later stages of development, where the cost of their remediation increases substantially. The analysis results produced by the proposed method are *presented as a structured defect description* (type, location, rationale, minimally required fix), which is always *accompanied by symbolic verification results*, reducing the developer's time spent on analyzing and resolving detected issues and substantially facilitating integration of the method into automated code quality control processes (CI/CD, code review), where defect decisions must be based on verified evidence rather than solely on language model output. The proposed analysis method *does not require model fine-tuning or collection of specialized datasets*, which reduces deployment and maintenance costs compared to fine-tuning-based approaches that require training data preparation, computational resources for training, and retraining when the model changes. Experimental application of GPT-5.2 (RQ2) confirmed the preservation of the proposed method's advantages over existing static code analysis methods when transitioning to a newer generation language model (+132% F1-score, +363% recall), indicating the possibility of updating the neural component without modifying the rest of the system's components, which substantially reduces costs. The method provides *the ability to customize for diverse specific analysis tasks* without changing the overall architecture: the analysis type and target defect classes are defined in the content of the query to the neural network component (analogous to the customizable analysis property in LLMSA), while symbolic verification coverage is extended by adding new rules, AST patterns, and taint specifications to support new programming languages and defect types. The ablation study (RQ3) confirmed the architecture's resilience to modifications of individual components: F1-score across configurations varies within the range of 0.654–0.668.

The dissertation consists of five chapters.

The first chapter examines the theoretical foundations of program code analysis using neural networks. Classical static analysis methods are analyzed (data flow analysis, control flow analysis, abstract interpretation, symbolic execution, model checking, syntactic analysis and pattern matching), their advantages and fundamental limitations are identified, including the path explosion problem and SMT solver limitations. The evolution of neural methods in code analysis is explored — from classical machine learning through pre-trained models (CodeBERT, GraphCodeBERT) to large language models (GPT-4, GPT-5.2). The concept of neurosymbolic integration and types of neurosymbolic systems are examined, substantiating the need for a hybrid approach that combines the semantic understanding of neural models with the formal guarantees of symbolic methods.

The second chapter provides an analysis of current research in the field of static code analysis. Existing neurosymbolic program analysis methods are systematized: IRIS (symbolic → neural direction, CodeQL → LLM), LLMSA (compositional neurosymbolic analysis), MoCQ (iterative pattern refinement). A common limitation of existing methods is identified — the absence of an explicit mechanism for independent verification of LLM claims. Requirements for a method that fills this gap are formulated, and the positioning of VERIGATE as the first method implementing the Neural → Symbolic direction in the context of static code analysis is substantiated.

The third chapter provides a detailed description of the proposed VERIGATE method. The architecture of the four-stage pipeline is presented: preliminary screening (tool ensemble and AST rules), Explain (generation of structured LLM claims in JSON format), Verify (symbolic verification through a multi-stage decision tree), final decision with the Evidence-Weighted Suppression mechanism. The verification process is formalized, criteria for accepting and rejecting claims are described, along with the span alignment mechanism and taint analysis. A comparison of VERIGATE's architectural decisions with existing neurosymbolic systems is conducted based on key characteristics (interaction direction, LLM role, symbolic system role, explainability).

Implementation details are described: technology stack, LLM prompt format, LLM output processing. An end-to-end example of VERIGATE operation on a real code fragment from the PySStuBs dataset is provided, demonstrating neurosymbolic component interaction through the detection of a semantic logic defect.

The fourth chapter presents the results of experimental evaluation. The main evaluation (RQ1) was conducted on 1,000 balanced samples from the PySStuBs (Python Single-Statement Bugs) dataset using GPT-4. Comparison of three methods (VERIGATE, GPT-4 Baseline, Rule-Based) demonstrated: an F1-score improvement of 76.4% (0.612 vs. 0.347) and a recall improvement of 187% (0.758 vs. 0.264) compared to the GPT-4 baseline at comparable precision (~0.51). Rule-based tools demonstrated very low effectiveness (F1 = 0.031), confirming the semantic complexity of defects in the dataset. Validation with GPT-5.2 (RQ2, n = 500) confirmed stable improvement: +132% F1-score, +363% recall, indicating the robustness of neurosymbolic verification as an architectural principle across language model generations. The ablation study (RQ3, n = 500) demonstrated that VERIGATE functions as an integrated pipeline with a narrow F1 range (0.654–0.668) across all configurations, where each component makes an independent contribution to overall effectiveness.

The fifth chapter examines the prospects for development and generalization of the proposed method. Directions for extension to new programming languages, new defect classes, and integration with production software development processes (CI/CD, code review, IDE) are analyzed. Limitations of the current implementation and ways to overcome them are identified.

The dissertation proposes the neurosymbolic VERIGATE method for static analysis of program source code, which through the use of independent symbolic verification of LLM claims enables combining the semantic understanding of program source code provided by the large language model with the justification of each detection in verified program properties. The core idea of the method lies in the inversion of the neurosymbolic interaction direction: instead of the traditional approach where symbolic analysis generates results and the neural network filters them, in

VERIGATE the LLM generates structured hypotheses and symbolic analysis independently verifies them. Experimental studies confirmed the effectiveness of the proposed method: an F1-score improvement of 76.4% and a recall improvement of 187% compared to the GPT-4 Baseline, stable improvement with GPT-5.2, and a balanced contribution of each architectural component.

Personal contribution of the applicant. All main results of the dissertation research presented for defense were obtained by the author personally. The applicant performed: analysis of existing methods for static code analysis and neurosymbolic systems; development of the neurosymbolic VERIGATE method with the Explain-Verify Gate and Evidence-Weighted Suppression mechanisms; formalization of the structured claim verification process; development of the software prototype; conducting the experimental evaluation and analysis of results. In co-authored publications, the applicant contributed: research of existing methods, design of the neurosymbolic method architecture, development of verification and decision-making mechanisms, experimental confirmation of the proposed method's effectiveness.

Approbation of dissertation materials. The main findings and scientific results presented in this dissertation were approbated through their presentation at the international scientific conference — *4th International Scientific and Practical Conference "Science and Information Technologies in the Modern World" (Athens, Greece, December 24–26, 2025)*.

Keywords: large language models, neurosymbolic analysis, static code analysis, program verification, symbolic analysis, defect detection, vulnerability analysis, semantic analysis, machine learning, neural networks, artificial intelligence, software engineering, software reliability, intelligent processing, software development life cycle.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API — Application Programming Interface (програмний інтерфейс застосунку)

AST — Abstract Syntax Tree (абстрактне синтаксичне дерево)

BMC — Bounded Model Checking (обмежена перевірка моделей)

BPE — Byte Pair Encoding (побайтове парне кодування)

CBMC — C Bounded Model Checker (інструмент обмеженої перевірки моделей для мови C)

CFG — Control Flow Graph (граф потоку управління)

CI/CD — Continuous Integration / Continuous Delivery (неперервна інтеграція / неперервна доставка)

CPG — Code Property Graph (граф властивостей коду)

CTL — Computation Tree Logic (логіка дерева обчислень)

CWE — Common Weakness Enumeration (перелік поширених слабкостей)

DFG — Data Flow Graph (граф потоку даних)

DSL — Domain Specific Language (предметно-орієнтована мова)

EWS — Evidence-Weighted Suppression (зважене придушення на основі доказів) — механізм прийняття рішень у методі VERIGATE

F1-score — гармонійне середнє precision та recall

FN — False Negative (хибнонегативне спрацювання)

FP — False Positive (хибнопозитивне спрацювання)

GAT — Graph Attention Networks (графові мережі уваги)

GCN — Graph Convolutional Networks (графові згорткові мережі)

GGNN — Gated Graph Neural Network (графова нейронна мережа з вентилями)

GNN — Graph Neural Networks (графові нейронні мережі)

GPT — Generative Pre-trained Transformer (генеративний попередньо навчений трансформер)

IDE — Integrated Development Environment (інтегроване середовище розробки)

JSON — JavaScript Object Notation (формат обміну даними)

LLM — Large Language Model (велика мовна модель)

LSTM — Long Short-Term Memory (довга короткочасна пам’ять) — різновид рекурентної нейронної мережі

LTL — Linear Temporal Logic (лінійна темпоральна логіка)

MPNN — Message Passing Neural Networks (нейронні мережі передачі повідомлень)

PDG — Program Dependence Graph (граф залежностей програми)

Precision — точність (частка істинно позитивних серед усіх позитивних спрацювань)

Recall — повнота (частка виявлених дефектів серед усіх наявних)

RNN — Recurrent Neural Network (рекурентна нейронна мережа)

SAT — Boolean Satisfiability Problem (задача булевої здійсненності)

SMT — Satisfiability Modulo Theories (здійсненність за модулем теорій)

SQL — Structured Query Language (мова структурованих запитів)

SVM — Support Vector Machine (метод опорних векторів)

Taint analysis — аналіз розповсюдження позначених (забруднених) даних

TN — True Negative (істинно негативне спрацювання)

TP — True Positive (істинно позитивне спрацювання)

VERIGATE — VERIfication GATE — нейросимвольний метод статичного аналізу програмного коду, запропонований у даній дисертації

XSS — Cross-Site Scripting (міжсайтове виконання скриптів)

ВСТУП

Актуальність теми

Сучасні програмні системи демонструють зростаючу складність, що супроводжується підвищеним ризиком виникнення дефектів та вразливостей безпеки. Недоліки програмного забезпечення призводять до значних фінансових втрат, компрометації даних та збоїв критично важливих інфраструктур, що створює нагальну потребу в ефективних методах забезпечення якості та безпеки коду.

Статичний аналіз програмного коду є одним з основних механізмів автоматичного виявлення потенційних проблем на ранніх етапах розробки — до етапу виконання програми. Традиційні інструменти статичного аналізу, такі як CodeQL [4], Infer [6], Semgrep [56], Bandit [51] та PMD [49], використовують наперед визначені правила та патерни для виявлення дефектів, забезпечуючи детерміністичну поведінку. Проте ці інструменти потребують значних інженерних зусиль для визначення нових правил аналізу, часто генерують хибнопозитивні спрацювання при аналізі складних кодових баз та мають обмежену ефективність щодо семантичних дефектів, що виходять за межі синтаксичних патернів.

Поява великих мовних моделей (Large Language Models, LLM) відкрила нові можливості для аналізу коду завдяки здатності цих моделей до семантичного розуміння програм [11; 3]. Дослідження демонструють, що сучасні LLM здатні виявляти різноманітні типи вразливостей без спеціалізованого навчання [58; 46]. Однак застосування LLM у статичному аналізі стикається з чотирма фундаментальними проблемами:

1. **Обмежена контрольованість** — LLM можуть виявляти проблеми за межами визначеного scope аналізу або генерувати твердження, що не піддаються програмній верифікації.
2. **Неконсистентна поведінка** — незначні варіації у промптах або структурі коду можуть призводити до суттєво різних результатів між запусками, що підриває надійність.

3. **Неверифіковані твердження** — хоча LLM здатні генерувати розгорнуті пояснення своїх рішень, ці пояснення можуть бути правдоподібними, але фактично невірними ("галюцинації"), що створює потребу в незалежному механізмі валідації відповідності тверджень моделі фактичним структурним властивостям коду.
4. **Відсутність детерміністичних гарантій** — на відміну від систем на основі правил, LLM не здатні забезпечити детерміністичні гарантії щодо виявлення конкретних патернів дефектів.

Ці обмеження колективно перешкоджають досягненню рівня надійності та передбачуваності, необхідного для інтеграції LLM-based інструментів у виробничі процеси CI/CD, де розробники потребують впевненості як у валідності виявлених проблем, так і у консистентності поведінки інструменту.

Нейросимвольні системи, що поєднують здатність нейронних мереж до міркування з символьними перевірками, пропонують перспективний напрям вирішення цих проблем. Нещодавні роботи у галузі нейросимвольного аналізу програм, зокрема IRIS [35], LLMSA [65] та MoCQ [36], демонструють життєздатність поєднання LLM з інструментами статичного аналізу. Проте існуючі системи мають характерні особливості та напрямки: деякі потребують повного контексту репозиторію та значної інфраструктури (IRIS, MoCQ), інші не мають явних механізмів верифікації тверджень, згенерованих LLM (LLMSA).

Ця дисертаційна робота спрямована на розв’язання питання **контролюваної поведінки LLM** у контексті статичного аналізу через розробку нейросимвольного методу VERIGATE. Метод вимагає від LLM генерації структурованих, верифікованих тверджень про потенційні дефекти, які потім незалежно валідуються засобами символьного аналізу програм. Система повідомляє виключно ті проблеми, що одночасно є семантично правдоподібними (за оцінкою LLM) та структурно верифікованими (засобами символьного аналізу), забезпечуючи передбачувану та надійну поведінку.

Ступінь наукової розробленості проблеми

Наукова спільнота накопичила значний обсяг досліджень у сфері статичного аналізу програмного коду. Класичні напрямки — абстрактна інтерпретація, символьне виконання, аналіз потоків даних та управління — розглядаються як основні для забезпечення коректності програм та активно впроваджуються в індустріальні інструменти. Передові інструменти статичного аналізу, такі як CodeQL [4] та Infer [6], застосовують аналіз потоків даних та абстрактну інтерпретацію для виявлення складних дефектів. Інструменти на основі правил, включаючи Semgrep [56], Bandit [51] та PMD [49], забезпечують високу precision для відомих патернів вразливостей, але мають обмежену ефективність щодо нових типів дефектів та семантичних проблем.

Застосування методів машинного навчання до аналізу коду еволюціонувало від традиційних класифікаторів до Graph Neural Networks, застосованих до AST та Program Dependence Graphs [70; 73], та далі до моделей з архітектурою трансформерів, попередньо навчених на корпусах коду: CodeBERT [24], GraphCodeBERT [26]. Ці моделі демонструють потенціал, але потребують донавчання (fine-tuning) на розмічених датасетах вразливостей і можуть не узагальнюватися на типи дефектів, що не представлені у навчальних даних.

Поява великих мовних моделей створила нові парадигми аналізу коду. Дослідження демонструють здатність таких моделей, як GPT, Claude та подібних, до виявлення різноманітних типів вразливостей без спеціалізованого навчання [58; 46; 53]. Однак підходи лише на основі LLM стикаються з фундаментальними обмеженнями: відсутність консистентності між запусками, обмежена прозорість процесу прийняття рішень [44] та відсутність механізму верифікації того, що виявлені проблеми відповідають фактичним властивостям програми.

У сфері нейросимвольного аналізу програм виділяються три ключові роботи. LLMSA [65] пропонує композиційний нейросимвольний аналіз, що покращує виявлення taint-вразливостей. IRIS [35] використовує LLM-керований висновок taint-специфікацій у поєднанні зі статичним аналізом на рівні цілих репозиторіїв.

MoCQ [36] реалізує ітеративний цикл LLM та класичного статичного аналізу для отримання та уточнення патернів вразливостей. Ці роботи демонструють життєздатність нейросимвольних підходів, проте мають свої обмеження.

Незважаючи на значний прогрес, комплексні нейросимвольні підходи з явною верифікацією тверджень LLM та забезпеченням контрольованої поведінки залишаються недостатньо дослідженими. В існуючих системах компоненти переважно працюють у напрямку символьна→нейронна (де LLM фільтрує або розширює результати символьного аналізу), тоді як зворотний напрямок — нейронна→символьна (де символьний аналіз незалежно верифікує твердження LLM) — є менш дослідженим. Ця дисертаційна робота заповнює цю прогалину.

Мета дослідження

Метою дослідження є розробка, формалізований опис та експериментальна оцінка нейросимвольного методу статичного аналізу програмного коду з використанням великих мовних моделей, який забезпечує контрольовану поведінку LLM через незалежну символьну верифікацію генерованих тверджень, що дозволяє досягти високої повноти виявлення дефектів при збереженні точності та передбачуваності результатів.

Завдання дослідження

Для досягнення встановленої мети сформульовано наступні завдання:

1. Проаналізувати класичні та нейронні підходи до статичного аналізу програмного коду, визначити їх переваги та обмеження.
2. Дослідити можливості та обмеження великих мовних моделей у задачах виявлення дефектів та вразливостей.
3. Розробити метод статичного аналізу із використанням великих мовних моделей, що забезпечує контрольованість результатів.
4. Виконати формалізований опис архітектури та компонентів запропонованого методу.

5. Реалізувати програмний прототип для експериментальної перевірки методу.
6. Розробити методику експериментальної оцінки ефективності за показниками точності, повноти та F1-score.
7. Провести експериментальне дослідження та порівняльний аналіз із існуючими підходами на датасетах з реальними дефектами.

Об’єкт і предмет дослідження

Об’єктом дослідження є процеси статичного аналізу програмного коду, спрямовані на виявлення дефектів та вразливостей.

Предметом дослідження є нейросимвольні методи статичного аналізу, що поєднують семантичне розуміння великих мовних моделей з незалежною символьною верифікацією для забезпечення контрольованої та передбачуваної поведінки.

Методи дослідження

У ході роботи над дисертацією було використано такі методи дослідження: аналіз та систематизація наукових джерел для визначення стану досліджень у галузі статичного аналізу коду та нейросимвольних систем; формалізація для опису архітектури та алгоритмів запропонованого методу; моделювання для побудови архітектури запропонованого методу та взаємодії його компонентів; експеримент для емпіричної оцінки ефективності методу на реальних даних; порівняльний аналіз для зіставлення результатів із існуючими підходами; абляційне дослідження для визначення внеску кожного архітектурного компоненту. Зазначені методи було обрано з огляду на мету та завдання дослідження, а також на їх придатність для вирішення проблем використання великих мовних моделей у задачах статичного аналізу коду.

Наукова новизна

Наукова новизна дослідження полягає у розробці нейросимвольного методу VERIGATE для статичного аналізу програмного коду, новизна якого розкривається через три взаємопов'язані аспекти:

1. Вперше запропоновано нейросимвольний метод статичного аналізу програмного коду, який відрізняється тим, що на першому етапі аналізу LLM безпосередньо генерує структуровані твердження про дефекти коду, за чим слідує етап символьного аналізу для їх верифікації, в той час, як в існуючих нейросимвольних методах статичного аналізу коду (IRIS, LLMSA, MoCQ) навпаки - символьний компонент виконує первинне виявлення або структуризацію дефектів, а нейронний компонент фільтрує чи доповнює його результати. Інвертований напрямок нейросимвольної верифікації "Neural → Symbolic" дозволяє скористатися здатністю великої мовної моделі сприймати семантику коду при одночасному жорсткому контролі згенерованих нею тверджень, що сприяє виявленню ширшого кола дефектів при мінімальних ризиках галюцинації моделі.
2. Вперше запропоновано спосіб верифікації структурованих тверджень про дефекти коду із застосуванням символьного аналізу Explain-Verify Gate, особливість якого полягає у тому, що атрибути твердження про дефект (тип дефекту, локалізація, пропозиція мінімально задовільного виправлення та обґрунтування) визначають вибір та інтерпретацію подальших символьних перевірок (зіставлення AST-патернів, taint-аналіз, перевірка правдоподібності виправлення), що є відмінним від існуючих способів верифікації у нейросимвольних методах аналізу коду та забезпечує детермінізм верифікації за відсутності потреби повторного звернення до мовної моделі за обґрунтуванням кожного прийнятого нею рішення.
3. Удосконалено процес прийняття рішень щодо дефектів у нейросимвольному статичному аналізі програмного коду шляхом застосування механізму Evidence-Weighted Suppression (EWS), який зважує докази з кількох

незалежних джерел (інструменти статичного аналізу, AST-патерни, показник впевненості базового аналізатора), що відрізняється від процесу визначення дефектів в рамках існуючих нейросимвольних підходів, в яких відхилення результату на етапі верифікації є остаточним без урахування доказів з інших джерел. Це дозволяє збалансувати точність і повноту аналізу шляхом відновлення валідних виявлень дефектів, що відхиляються символьною верифікацією за відсутності відповідних патернів, але підтверджуються достатніми доказами з інших джерел.

Кількісна оцінка ефективності методу підтверджує його дієвість: покращення F1-score на 76.4% (0.612 проти 0.347) та recall на 187% (0.758 проти 0.264) порівняно з GPT-4 Baseline при порівняному precision (~0.51) на 1 000 зразках із датасету PySStuBs; виявлення 247 додаткових дефектів (379 проти 132); стабільне покращення з GPT-5.2 (+132% F1, +363% recall).

Зв’язок роботи з науковими програмами, планами, темами

Дослідження з розроблення нейросимвольного методу статичного аналізу програмного коду з використанням великих мовних моделей виконувались в рамках тематичного плану науково-дослідних робіт кафедри системного проектування, зокрема в рамках ініціативної теми «Впровадження сервіс-орієнтованого підходу до реалізації процесів діджиталізації суспільства (тема СП – 01/2023)» (№ держреєстрації 0123U101333), що виконується згідно Тематичного плану виконання ініціативних кафедральних науково-дослідних робіт Навчально-наукового інституту прикладного системного аналізу КПІ ім. Ігоря Сікорського. Результати дисертаційного дослідження, зокрема розроблений нейросимвольний метод статичного аналізу програмного коду з використанням великих мовних моделей використано для покращення інструментарію забезпечення якості програмного забезпечення в рамках процесів діджиталізації суспільства.

Практичне значення

1. Розроблений метод статичного аналізу коду *дає змогу виявляти додаткові дефекти*, які не виявляються як інструментами на основі правил, так і безпосереднім використанням LLM. Експериментальна оцінка на 1000 зразків із набору даних PySStuBs підтвердила виявлення 247 додаткових дефектів порівняно з базовим LLM-підходом (379 проти 132) та 371 додаткового дефекту порівняно з ансамблем інструментів на основі правил (379 проти 8), що відповідає покращенню recall на 187% та F1-score на 76.4% при збереженні співставного рівня точності (~0.51). Виявлення цих додаткових дефектів на етапі статичного аналізу запобігає їх потраплянню на подальші етапи розробки, де вартість їх виправлення суттєво зростає.
2. Результати аналізу в рамках запропонованого методу *представлені структурованим описом дефекту* (тип, локалізація, обґрунтування, мінімально необхідне виправлення), який завжди *супроводжується результатами символьної верифікації*, що скорочує час розробника на аналіз та усунення виявлених проблем та суттєво полегшує інтеграцію методу в автоматизовані процеси контролю якості коду (CI/CD, code review), де рішення про дефект має спиратися на перевірені докази, а не лише на вихід мовної моделі.
3. Запропонований метод аналізу *не потребує донавчання моделей та збору спеціалізованих датасетів*, що зменшує витрати на впровадження та супровід порівняно з підходами на основі fine-tuning, які вимагають підготовки навчальних даних, обчислювальних ресурсів для тренування та повторного навчання при зміні моделі. Експериментальне застосування GPT-5.2 (RQ2) підтвердило збереження переваг запропонованого методу у порівнянні з існуючими методами статичного аналізу коду при переході на новіше покоління мовної моделі (+132% F1-score, +363% recall), що свідчить про можливість оновлення нейронного компоненту без модифікації решти складових системи, що суттєво економить витрати.

4. Метод забезпечує *можливість налаштування під різноманітні конкретні задачі аналізу* без зміни загальної архітектури: тип аналізу та цільові класи дефектів визначаються у змісті запиту до нейромережевого компоненту (аналогічно до властивості customizable analysis у LLMSA), а покриття символьної верифікації розширюється додаванням нових правил, AST-патернів та taint-специфікацій для підтримки нових мов програмування та типів дефектів. Абляційне дослідження (RQ3) підтвердило стійкість архітектури до модифікацій окремих компонентів: F1-score між конфігураціями варіюється у діапазоні 0.654–0.668.

Особистий внесок здобувача

Усі основні результати дисертаційного дослідження, представлені до захисту, одержані автором особисто. Здобувачем виконано: аналіз існуючих методів статичного аналізу коду та нейросимвольних систем; розробку нейросимвольного методу VERIGATE з механізмами Explain-Verify Gate та Evidence-Weighted Suppression; формалізацію процесу верифікації структурованих тверджень; розробку програмного прототипу; проведення експериментальної оцінки та аналіз результатів. У публікаціях у співавторстві здобувачеві належать: дослідження існуючих методів, проектування архітектури нейросимвольного методу, розробка механізмів верифікації та прийняття рішень, експериментальне підтвердження ефективності запропонованого методу.

Апробація матеріалів дисертації

Основні положення та отримані наукові результати, що викладені в даній дисертації, пройшли апробацію через їх презентацію на міжнародній науковій конференції — *4th International Scientific and Practical Conference «Science and Information Technologies in the Modern World» (Афіни, Греція, 24–26 грудня 2025р.)*.

Публікації

Статті у наукових фахових виданнях України

[1] Vokhranov, I., & Bulakh, B. (2026). VERIGATE: A verification gate-based neurosymbolic method for static code analysis. *Наука і Техніка Сьогодні, 2*(56), 1488–1507. - DOI 10.52058/2786-6025-2026-2(56)-1488-1507. Здобувачем запропоновано нейросимвольний метод VERIGATE з механізмами Explain-Verify Gate та Evidence-Weighted Suppression, розроблено формальний опис архітектури запропонованого методу та взаємодії його компонентів, здійснено розробку програмного прототипу, проведено експериментальну оцінку на наборі даних PySStuBs та виконано аналіз отриманих результатів. Булах Б.В. надав рекомендації щодо визначення структури методу, брав участь в отриманні та обробці експериментальних даних з використанням програмного прототипу.

[2] Vokhranov, I., & Bulakh, B. (2024). Transformer-based models application for bug detection in source code. *Technology Audit and Production Reserves, 5*(2(79)), 6–15. - DOI 10.15587/2706-5448.2024.310822. Здобувачем проведено аналіз існуючих підходів до застосування моделей на основі архітектури трансформера для виявлення дефектів у програмному коді, розроблено методологію експериментів із донавчанням моделей BERT, CodeBERT, GPT-2 та CodeT5 на наборі даних RunBugRun, здійснено підготовку даних та проведення експериментів для різних типів дефектів, виконано порівняльний аналіз та інтерпретацію результатів. Булах Б.В. брав участь у визначенні напряму дослідження та обґрунтуванні вибору моделей, надав рекомендації щодо методології проведення експериментів та інтерпретації отриманих результатів.

[3] Вохранов, І., & Булах, Б. (2023). Комбіновані підходи до статичного аналізу коду з використанням нейронних мереж. *Інфокомунікаційні та комп'ютерні технології, 1*(05), 183–193. - DOI 10.36994/2788-5518-2023-01-05-20. Здобувачем проведено систематичний аналіз існуючих підходів до застосування нейронних мереж у статичному аналізі коду за трьома категоріями (коригування специфікацій, визначення специфікацій, аналіз «чорного ящика»), здійснено

класифікацію та порівняльний аналіз методів кожної категорії, сформульовано висновки щодо перспективних напрямків комбінування підходів. Булах Б.В. належить постановка задачі дослідження, участь у формуванні структури огляду, надання рекомендацій щодо відбору джерел та уточнення висновків.

[4] Petrenko, A. I., & Vokhranov, I. A. (2023). Neural networks: Studying their decision-making rules. *System Research and Information Technologies, 2023*(2), 74–85. - DOI 10.20535/SRIT.2308-8893.2023.2.06. Здобувачем розроблено методологію дослідження, реалізовано програмний прототип алгоритму декомпозиції DeepRED із запропонованим покращенням, проведено експерименти та виконано аналіз отриманих результатів, підготовлено текст рукопису. Петренко А.І. визначив постановку задачі та напрям дослідження, здійснив наукове керівництво роботою, виконав редагування та підготовку рукопису до публікації.

[5] Вохранов, І. (2025). Нейросимвольні підходи в аналізі програмного коду: огляд сучасного стану та перспективи розвитку. *4th International Scientific and Practical Conference «Science and Information Technologies in the Modern World»*, 161–164. Здобувачем проведено презентацію на міжнародній науковій конференції.

1. ТЕОРЕТИЧНІ ОСНОВИ АНАЛІЗУ ПРОГРАМНОГО КОДУ З ВИКОРИСТАННЯМ НЕЙРОННИХ МЕРЕЖ

1.1. Статичний аналіз програмного коду: загальні положення

Статичний аналіз програмного коду є фундаментальною методологією у сучасній інженерії програмного забезпечення, що забезпечує автоматизовані засоби для виявлення дефектів, вразливостей та проблем якості коду без його безпосереднього виконання [1]. Цей метод відіграє ключову роль у забезпеченні надійності та безпеки програмних систем, особливо в контексті зростаючої складності сучасних програм [12]. Його переваги включають високе покриття коду та незалежність від середовища виконання [48].

Формально статичний аналіз може бути визначений як процес отримання інформації про динамічну поведінку програми шляхом дослідження її тексту [68]. Центральним елементом є апроксимація множини всіх можливих станів виконання програми. У цьому контексті програма розглядається як математичний об'єкт, для якого можуть бути виведені певні властивості, що стосуються її поведінки та відповідності заданим специфікаціям. Специфікація цих властивостей часто ґрунтується на семантичних моделях, які формалізують значення програмних конструкцій.

Розрізняють конкретну та абстрактну семантику. Конкретна семантика точно описує всі можливі стани та переходи програми під час її виконання, надаючи вичерпне, але часто нерозв'язне або обчислювально недосяжне представлення. Абстрактна семантика, натомість, оперує спрощеними, але обчислювально керованими представленнями цих станів, що дозволяє виконувати аналіз. Абстрактна інтерпретація, як формальний метод, обчислює коректні надопроксимації всіх можливих станів виконання програми шляхом відображення конкретної семантики програми в абстрактний домен [30].

Ключовими критеріями оцінки статичного аналізу є коректність (soundness) та повнота (completeness). Коректний аналіз гарантує, що всі виявлені властивості

програми є істинними і що жоден дефект не буде пропущений (відсутність хибнонегативних результатів), хоча при цьому можуть виникати хибнопозитивні спрацювання [13]. Це досягається шляхом надопроксимації, де множина можливих абстрактних станів охоплює всі конкретні стани. Повний аналіз, навпаки, не генерує хибнопозитивних результатів (кожен виявлений дефект є реальним), але може пропускати деякі дійсні дефекти. Повнота часто вимагає недоапроксимації. У більшості практичних застосувань статичний аналіз пріоритезує коректність, щоб мінімізувати ризик пропуску критичних вразливостей, толеруючи при цьому певну кількість хибнопозитивних результатів [25].

Роль семантичного відображення в аналізі полягає у встановленні формального зв'язку між конкретним та абстрактним представленнями поведінки програми. Це відображення дозволяє трансформувати складну конкретну інформацію в більш просту абстрактну форму, яка придатна для обчислювального аналізу, зберігаючи при цьому необхідні властивості коректності або повноти.

Математичні основи статичного аналізу спираються на теорію частково впорядкованих множин (posets) та решіток (lattices). Частково впорядкована множина є множиною з бінарним відношенням, що є рефлексивним, антисиметричним та транзитивним. Решітки — це частково впорядковані множини, де для кожної пари елементів існує найбільший нижній елемент (meet) та найменший верхній елемент (join). Ця структура забезпечує формальний апарат для порівняння та об'єднання інформації про стан програми, що є фундаментальним для абстрактної інтерпретації [30].

Програмні операції моделюються як монотонні функції, що діють на решітках абстрактних доменів. Монотонна функція зберігає порядок: якщо $x \leq y$, то $f(x) \leq f(y)$. Ця властивість гарантує, що ітеративні обчислення на абстрактних доменах сходитимуться до стабільного стану. Теорія нерухомих точок є наріжним каменем статичного аналізу, оскільки багато властивостей програм, таких як досяжні стани або інваріанти циклів, можуть бути виражені як найменші (або

найбільші) нерухомі точки монотонних функцій. Для функції $f: L \to L$ на решітці L нерухомою точкою є елемент $x \in L$ такий, що $f(x) = x$. Найменша нерухома точка (ННТ) для аналізу потоку даних часто обчислюється ітеративно, починаючи з мінімального елемента решітки.

Зв'язок між конкретним доменом C та абстрактним доменом A формалізується за допомогою зв'язків Галуа (Galois connections), які складаються з пари монотонних функцій (1): оператора абстракції α та оператора конкретизації γ.

$$\alpha: C \to A, \gamma: A \to C \quad (1)$$

Ці оператори задовольняють умову: $\alpha(c) \leq a$ тоді і тільки тоді, коли $c \leq \gamma(a)$ для всіх $c \in C$ та $a \in A$. Оператор абстракції α відображає конкретний стан або множину станів у його абстрактне представлення, тоді як оператор конкретизації γ перетворює абстрактний стан назад у множину конкретних станів, яку він апроксимує. Абстрактні домени, такі як домен інтервалів для числових значень або домен знаків, надають конкретні приклади таких абстракцій [2].

Статичний аналіз інтегрується в різні етапи розробки програмного забезпечення: від попереднього аналізу в IDE під час розробки, через оптимізацію та перевірку на етапі компіляції, до безперервного аналізу в CI/CD конвеєрах [52]. Дослідження Google демонструє, що ефективна інтеграція статичного аналізу потребує врахування контексту розробника та мінімізації тертя у робочому процесі [52].

Порівняння статичного та динамічного аналізу дозволяє виділити їхні характерні особливості. Статичний аналіз забезпечує високе покриття коду та аналіз усіх можливих шляхів виконання через надопроксимацію [48], проте стикається з проблемами масштабованості для великих кодових баз через різке зростання кількості станів та шляхів [72]. Він може генерувати хибнопозитивні результати, пріоритезуючи коректність над повнотою [25; 9]. Динамічний аналіз, натомість, забезпечує точність для виконаних шляхів, але обмежений покриттям лише тих шляхів, що фактично виконуються під час тестування.

Незважаючи на свою фундаментальність, класичний статичний аналіз зіштовхується з певними обмеженнями: проблема нерозв'язності (undecidability), проблема вибуху станів (state explosion) та проблема вибуху шляхів (path explosion) при аналізі складних програм [23]. Ці проблеми призводять до значних витрат ресурсів та високого рівня хибнопозитивних спрацювань [25; 34], що знижує довіру розробників до інструментів [21; 61].

Емпіричні дослідження підтверджують серйозність цих проблем. Дослідження використання інструментів статичного аналізу у великих проєктах на C/C++ виявило значні розбіжності у результатах різних інструментів та високий рівень хибнопозитивних спрацювань [47]. Аналогічно, комплексна оцінка інструментів для виявлення вразливостей у Java-коді продемонструвала суттєву варіативність у продуктивності різних аналізаторів [1]. Критичне порівняння шести інструментів статичного аналізу виявило низьку узгодженість між результатами детекції [34].

Для подолання цих обмежень зростає інтерес до інтелектуальної обробки програмного коду через інтеграцію методів машинного навчання (МН) та великих мовних моделей (LLM) [74; 67]. Фундаментальні причини такого поєднання полягають у здатності МН-моделей виявляти складні патерни та кореляції у великих обсягах коду та результатах аналізу, які важко або неможливо формалізувати за допомогою традиційних детермінованих правил [33]. Це дозволяє створювати більш точні механізми для фільтрації хибнопозитивних спрацювань, пріоритизації виявлених проблем та прогнозування вразливостей на основі історичних даних [25].

1.2. Класичні методи статичного аналізу

Класичні методи статичного аналізу програмного коду сформувалися протягом кількох десятиліть і охоплюють широкий спектр підходів, кожен з яких має свої теоретичні основи, переваги та обмеження.

1.2.1. Аналіз потоку даних (Data Flow Analysis)

Аналіз потоку даних є одним з фундаментальних методів статичного аналізу, що досліджує, як значення змінних та інші дані поширюються через програму [48]. Цей метод базується на побудові графа потоку управління (Control Flow Graph, CFG) та ітеративному обчисленні властивостей даних у кожній точці програми.

Формально аналіз потоку даних визначається системою рівнянь. Для кожного базового блоку B у CFG визначаються множини $gen(B)$ — факти, що генеруються блоком, та $kill(B)$ — факти, що знищуються блоком. Вхідна інформація $in(B)$ та вихідна інформація $out(B)$ пов'язані рівняннями:

$$\begin{aligned} out(B) &= gen(B) \cup (in(B) - kill(B)) \\ in(B) &= \cup\_\{P \in pred(B)\} out(P) \end{aligned} \tag{2}$$

де $pred(B)$ — множина попередників блоку B у CFG.

До основних типів аналізу потоку даних належать:

- **Аналіз досяжних визначень (Reaching Definitions)** — визначає, які присвоєння змінних можуть досягти певної точки програми без перевизначення. Цей аналіз є прямим (forward) і використовується для виявлення невизначених змінних та оптимізації коду.
- **Аналіз живих змінних (Live Variables)** — визначає, чи може значення змінної бути використане на подальших шляхах виконання. Це зворотний (backward) аналіз, що застосовується для виявлення мертвого коду та оптимізації використання регістрів.
- **Аналіз доступних виразів (Available Expressions)** — визначає, які вирази вже були обчислені та залишаються незмінними у певній точці програми. Використовується для усунення надлишкових обчислень.
- **Taint-аналіз (Taint Analysis)** — відстежує поширення «забруднених» даних від джерел (sources), таких як користувацький ввід, до приймачів (sinks), таких як небезпечні функції [12]. Цей тип аналізу є критично важливим для

виявлення вразливостей безпеки, включаючи SQL-ін'єкції, XSS та command injection [48].

Практичні інструменти, такі як Semgrep та Bandit, активно використовують аналіз потоку даних для виявлення вразливостей безпеки у коді [7].

1.2.2. Аналіз потоку управління (Control Flow Analysis)

Аналіз потоку управління досліджує можливі шляхи виконання програми та структуру її управляючих конструкцій. Центральним артефактом є граф потоку управління (CFG), вершини якого представляють базові блоки коду, а ребра — можливі переходи між ними.

Граф потоку управління (CFG) формально визначається як орієнтований граф $G = (V, E, entry, exit)$, де V — множина базових блоків, $E \subseteq V \times V$ — множина ребер переходів, *entry* — початкова вершина, *exit* — кінцева вершина. CFG є основою для більшості інших видів статичного аналізу [13].

Граф викликів (Call Graph) представляє відношення викликів між функціями програми. Вершини відповідають функціям, а ребра — можливим викликам. Побудова точного графа викликів ускладнюється наявністю непрямих викликів (через вказівники на функції, віртуальні методи тощо).

Аналіз домінування (Dominance Analysis) визначає відношення домінування між вершинами CFG. Вершина d домінує над вершиною n, якщо кожен шлях від *entry* до n проходить через d. Цей аналіз використовується для оптимізації коду та виявлення циклів.

Виявлення циклів та обчислення інваріантів базується на аналізі домінування та дозволяє ідентифікувати природні цикли програми. Інваріанти циклів — це властивості, що залишаються істинними на кожній ітерації циклу, і їх виявлення є важливим для верифікації програм.

1.2.3. Абстрактна інтерпретація (Abstract Interpretation)

Абстрактна інтерпретація, запропонована Cousot та Cousot [30], є формальною основою для проектування коректних статичних аналізаторів. Цей

метод систематично апроксимує семантику програми, замінюючи конкретні обчислення абстрактними.

Ключовим елементом є вибір абстрактного домену — математичної структури, що представляє апроксимації конкретних значень. Найбільш поширені абстрактні домени включають:

- **Домен знаків (Sign Domain)** — апроксимує числові значення їх знаком: $\{\bot, -, 0, +, \top\}$. Цей простий домен дозволяє виявляти помилки ділення на нуль та переповнення.
- **Домен інтервалів (Interval Domain)** — представляє числові значення як інтервали $[a, b]$. Забезпечує більш точну апроксимацію, ніж домен знаків, але може призводити до значного розширення інтервалів у циклах.
- **Домен октагонів (Octagon Domain)** — представляє обмеження виду $\pm x \pm y \leq c$ між парами змінних. Забезпечує баланс між точністю та ефективністю обчислень.
- **Домен поліедрів (Polyhedra Domain)** — представляє довільні лінійні обмеження між змінними. Найбільш точний серед числових доменів, але має експоненційну складність.

Операція **розширення (widening)** ∇ є критичною для забезпечення збіжності аналізу. Для послідовності абстрактних значень $a_0, a_1, a_2, \ldots$ операція розширення гарантує, що послідовність $a_0, a_0 \nabla a_1, (a_0 \nabla a_1) \nabla a_2, \ldots$ стабілізується за скінченну кількість кроків [2].

Промислові інструменти на основі абстрактної інтерпретації, такі як Astrée та Polyspace, успішно застосовуються для верифікації критичного програмного забезпечення в авіаційній та автомобільній індустріях [68].

1.2.4. Символьне виконання (Symbolic Execution)

Символьне виконання є потужним методом аналізу, що виконує програму з символьними, а не конкретними значеннями вхідних даних [23]. Замість обчислення конкретних результатів, символьне виконання будує символьні вирази, що представляють залежність виходів від входів.

Формально, стан символьного виконання визначається як кортеж *(pc, σ, π)*, де *pc* — програмний лічильник (поточна позиція у коді), *σ* — символьне сховище (відображення змінних на символьні вирази), *π* — обмеження шляху (path constraint) — кон'юнкція умов, що мають бути істинними для досягнення поточної точки.

При досягненні умовного переходу *if(c)* символьне виконання розгалужується на два шляхи: шлях *then* з обмеженням $\pi \wedge c$ та шлях *else* з обмеженням $\pi \wedge \neg c$.

SMT-вирішувачі (Satisfiability Modulo Theories), такі як Z3, STP та CVC4, використовуються для перевірки здійсненності обмежень шляху та генерації конкретних вхідних даних, що активують певні шляхи виконання [69].

Основні застосування символьного виконання включають: автоматичну генерацію тестів, що забезпечує високе покриття коду; виявлення вразливостей шляхом аналізу умов, за яких небезпечні операції можуть бути виконані з контрольованими зловмисником даними [23]; верифікацію властивостей — перевірку, чи існують вхідні дані, що призводять до порушення заданих інваріантів або специфікацій.

Серед відомих інструментів символьного виконання — KLEE для LLVM-програм, SAGE від Microsoft для фаззингу, angr для бінарного аналізу та Symbolic PathFinder для Java [23].

1.2.5. Перевірка на основі моделей (Model Checking)

Перевірка на основі моделей є формальним методом верифікації, що систематично досліджує всі можливі стани системи для перевірки заданих властивостей. Модель програми зазвичай представляється як система переходів (transition system) або автомат Кріпке.

Темпоральні логіки використовуються для специфікації властивостей: LTL (Linear Temporal Logic) описує властивості лінійних послідовностей станів, CTL (Computation Tree Logic) описує властивості дерев обчислень. Типові властивості включають безпеку (safety) — «погана подія ніколи не станеться», живучість

(liveness) — «хороша подія врешті-решт станеться», та відсутність взаємоблокувань.

Обмежена перевірка моделей (Bounded Model Checking, BMC) обмежує глибину пошуку та використовує SAT/SMT-вирішувачі для знаходження контрприкладів. Це дозволяє аналізувати більші системи ціною втрати повноти. Інструменти CBMC (C Bounded Model Checker), SPIN та NuSMV є прикладами успішних реалізацій цього методу.

1.2.6. Синтаксичний аналіз та pattern matching

Найпростіші методи статичного аналізу базуються на синтаксичному аналізі та пошуку патернів у коді без глибокого семантичного аналізу.

Лексичний аналіз — пошук текстових патернів у вихідному коді. Хоча цей метод має обмежену точність через відсутність розуміння структури програми, він є швидким та простим у застосуванні.

Аналіз AST (Abstract Syntax Tree) — побудова та аналіз абстрактного синтаксичного дерева програми. AST представляє синтаксичну структуру коду та дозволяє виявляти патерни на рівні синтаксичних конструкцій [27].

Pattern matching на основі правил — сучасні інструменти, такі як Semgrep, використовують декларативні правила для опису патернів коду, що підлягають виявленню. Ці правила можуть враховувати семантичну інформацію, таку як типи змінних та потоки даних [15].

Популярні інструменти цієї категорії: Semgrep — швидкий pattern-based аналізатор з підтримкою багатьох мов, Bandit — спеціалізований аналізатор безпеки для Python, ESLint/Pylint/Flake8 — лінтери для перевірки стилю та простих помилок.

Порівняльні дослідження показують, що різні інструменти демонструють значну варіативність у результатах детекції [34; 1]. Емпіричне дослідження використання інструментів статичного аналізу для secure code review виявило, що жоден окремий інструмент не забезпечує достатнього покриття, що мотивує використання комбінацій інструментів [7].

1.2.7. Комбіновані методи

Сучасна практика статичного аналізу часто поєднує кілька методів для досягнення кращих результатів.

Конколічне виконання (Concolic Execution) поєднує конкретне та символьне виконання: програма виконується з конкретними вхідними даними, але паралельно збираються символьні обмеження, що дозволяє систематично генерувати нові тестові випадки [2].

Гібридний аналіз комбінує результати статичного та динамічного аналізу для підвищення точності та зменшення хибнопозитивних спрацювань [69].

Ансамблі інструментів — використання кількох статичних аналізаторів з подальшою агрегацією результатів. Дослідження показують, що різні інструменти виявляють різні типи дефектів, і їх комбінація забезпечує краще покриття [34].

1.3. Переваги та обмеження символьних методів

Символьні методи, що охоплюють такі техніки як символьне виконання, перевірка на основі моделей та аналіз на базі SMT-вирішувачів, представляють клас статичного аналізу програм з високим рівнем формальної строгості [23]. Ці методи аналізують програми шляхом представлення обчислень та даних у символьній формі, а не з конкретними значеннями. Така характеристика позиціонує символьні методи як «золоту середину» у спектрі верифікації: вони пропонують більш строгий та точний аналіз, ніж практичні інструменти на основі евристик, проте зазвичай вимагають менших зусиль та експертизи, ніж повна формальна верифікація.

1.3.1. Моделювання простору станів

Ядро символьного аналізу полягає у його здатності моделювати та аналізувати простір станів програми. Стан програми може бути формально визначений як кортеж, що охоплює поточні значення всіх змінних, програмний лічильник та розташування пам'яті. Нехай S позначає множину всіх можливих

станів програми, а $T \subseteq S \times S$ представляє відношення переходів, де пара $(s, s') \in T$ означає можливий перехід зі стану s до стану s' через виконання однієї інструкції програми.

Символьні методи працюють шляхом заміни конкретних вхідних значень символьними змінними. Під час «символьного виконання» програми умови розгалуження та арифметичні операції записуються як логічні формули над цими символьними змінними [23]. Цей процес будує обмеження шляху (path condition, PC) для кожного шляху виконання, яке є кон'юнкцією всіх умов розгалуження:

$$\varphi_p = \bigwedge_i cond_i \tag{3}$$

де $cond_i$ представляє символьну умову, пов'язану з і-м рішенням розгалуження вздовж шляху p. Якщо обмеження шляху є здійсненним (satisfiable), це означає, що шлях є досяжним, і SMT-вирішувач може надати конкретні значення для символьних входів, що ведуть до цього шляху.

Досяжність конкретного стану *s_target* може бути перевірена запитом до SMT-вирішувача:

$$IsSatisfiable(\varphi_p \wedge property(s\_target)) \tag{4}$$

Це строге моделювання надає кілька ключових переваг: **високу точність**, часто мінімізуючи або повністю усуваючи потребу в абстракції [2], та **конкретні контрприклади** — набори вхідних значень, що викликають помилку, що є неоціненним для налагодження та відтворення вразливостей [13]. Проте ця висока точність має значну ціну: обчислювальна складність та проблеми масштабованості [72].

1.3.2. Проблема вибуху шляхів

Символьні методи стикаються зі значною перешкодою, відомою як **проблема вибуху шляхів (path explosion problem)**. Кожна точка умовного розгалуження в програмі потенційно розділяє виконання на кілька різних шляхів [23]. Для програм з глибоко вкладеними умовами або циклами кількість потенційних шляхів виконання зростає експоненційно з розміром програми.

Проблема вибуху шляхів є фундаментальним обмеженням, що випливає з комбінаторної природи виконання програм [72]. Її неможливо повністю усунути не жертвуючи або повнотою аналізу, або його точністю.

Дослідники розробили різні стратегії пом'якшення: обмеження глибини (Depth Limiting) для аналізу обмеженої кількості ітерацій циклів; евристики пошуку, що пріоритезують шляхи з більшою ймовірністю виявлення помилок; комбінування з абстрактною інтерпретацією для відсікання нездійсненних шляхів [2]; конколічне виконання, що поєднує конкретне та символьне виконання [69]. Навіть із цими зусиллями, проблема вибуху шляхів суттєво обмежує застосовність символьних методів до великомасштабних промислових кодових баз [47].

1.3.3. Обмеження SMT-вирішувачів

SMT-вирішувачі відіграють центральну роль у символьному аналізі, визначаючи здійсненність логічних формул над різними теоріями: лінійна арифметика, бітові вектори, масиви та вказівники [23]. Ключові теорії включають лінійну арифметику над цілими та дійсними числами, бітові вектори для низькорівневих мов програмування, та складні теорії масивів і вказівників для моделювання доступу до пам'яті.

Незважаючи на значні досягнення, SMT-вирішувачі мають притаманні обмеження. Багато теорій, особливо при комбінуванні, є NP-повними або навіть нерозв'язними. Час, необхідний для розв'язування SMT-формул, може драматично зростати зі збільшенням розміру формули та кількості задіяних символьних змінних. Повністю автоматизований, вичерпний аналіз великих кодових баз залишається значною мірою поза межами поточних можливостей.

1.3.4. Сценарії переваги символьного аналізу

Незважаючи на обмеження, символьні методи пропонують неперевершені переваги у специфічних контекстах: аналіз критичних модулів (криптографічні бібліотеки, компоненти безпеки) [12]; генерація високоякісних тестових випадків [23]; верифікація системних компонентів (протоколи, драйвери, ядро ОС);

виявлення вразливостей з конкретними контрприкладами (переповнення буфера, цілих чисел, use-after-free) [48].

Символьний аналіз значно покращує класичний статичний аналіз, пропонуючи надійні механізми для підтвердження істинно позитивних результатів, відхилення хибнопозитивних та генерації точних контрприкладів [69].

1.3.5. Порівняльний аналіз символьних та евристичних методів

Таблиця 1.1 — Порівняння символьних та евристичних методів статичного аналізу

Критерій	Символьні методи	Евристичні методи
Точність	Висока, менше хибнопозитивних результатів	Варіативна, залежить від якості евристик
Повнота	Теоретично повні для обмежених доменів	Неповні, можуть пропускати дефекти
Масштабованість	Обмежена проблемою вибуху шляхів та складністю SMT	Висока, лінійна або близька до лінійної
Контрприклади	Автоматична генерація конкретних входів	Зазвичай відсутні
Інтерпретованість	Формальні докази та шляхи виконання	Евристичні пояснення
Застосовність	Критичні компоненти, протоколи	Великі кодові бази, CI/CD

1.3.6. Мотивація гібридних методів

Обмеження символьних методів безпосередньо мотивують дослідження гібридних методів, таких як нейросимвольні системи. Нейронні моделі потенційно можуть діяти як адаптивний шар, навчаючись апроксимувати, фільтрувати або пріоритезувати складні SMT-запити чи шляхи виконання, тим самим направляючи символьний рушій та пом'якшуючи деякі обчислювальні вузькі місця [57].

Символьні методи є надзвичайно потужними при локальному застосуванні, проте масштабування на цілісність великих промислових програмних систем залишається складним. Цей компроміс між точністю та масштабованістю формує переконливий аргумент для інтеграції нейронних мереж та великих мовних моделей як «адаптивного шару» над символьними рушіями [74; 67].

1.4. Нейронні мережі у задачах аналізу коду

Застосування нейронних мереж до задач аналізу програмного коду стало одним з найбільш динамічних напрямків досліджень у програмній інженерії. На відміну від символьних методів, нейронні методи навчаються виявляти патерни безпосередньо з даних, що дозволяє їм адаптуватися до нових типів дефектів та вразливостей без явного програмування правил [39].

1.4.1. Еволюція нейронних методів в аналізі коду

Застосування методів машинного навчання до аналізу програмного коду пройшло кілька етапів. Ранні методи базувалися на ручному конструюванні ознак (feature engineering), де експерти визначали релевантні характеристики коду — метрики складності, статистику токенів, структурні властивості — які використовувалися класичними алгоритмами машинного навчання [55].

Перехід до глибокого навчання дозволив автоматично витягувати ієрархічні представлення коду. Систематичний огляд літератури демонструє значне зростання досліджень, особливо після 2018 року, з фокусом на виявленні вразливостей та дефектів [29]. Найновіший етап характеризується домінуванням попередньо навчених моделей та великих мовних моделей, що навчені на масивних корпусах коду та здатні до transfer learning на специфічні задачі аналізу [64].

1.4.2. Представлення програмного коду для нейронних мереж

Ефективність нейронних моделей критично залежить від способу представлення програмного коду.

Послідовне представлення (Sequence-based) розглядає код як послідовність токенів, аналогічно до обробки природної мови. Код токенізується на рівні лексем або підслів (subword tokenization, наприклад BPE — Byte Pair Encoding). Це представлення природно підходить для рекурентних та трансформерних архітектур [64].

Деревоподібне представлення (Tree-based) використовує абстрактне синтаксичне дерево (AST) програми. Спеціалізовані архітектури, такі як Tree-LSTM та рекурсивні нейронні мережі, можуть обробляти деревоподібні структури, враховуючи батьківсько-дочірні відношення між вузлами [10].

Графове представлення (Graph-based) розширює деревоподібне, додаючи семантичні ребра: потоки даних, потоки управління, залежності між визначенням та використанням змінних. Code Property Graph (CPG) об'єднує AST, CFG та PDG (Program Dependence Graph) в єдину структуру, що забезпечує багате представлення семантики програми [70]. Графові нейронні мережі (GNN) здатні ефективно обробляти такі представлення [41].

Гібридні представлення комбінують кілька модальностей — наприклад, послідовність токенів разом з графом залежностей — для захоплення як локальних синтаксичних патернів, так і глобальних семантичних відношень [41].

1.4.3. Архітектури нейронних мереж для аналізу коду

Рекурентні нейронні мережі (RNN) та їх варіанти — LSTM (Long Short-Term Memory) та GRU (Gated Recurrent Unit) — були серед перших глибоких архітектур, застосованих до аналізу коду. Вони обробляють послідовності токенів, підтримуючи прихований стан, що накопичує інформацію про контекст. Однак RNN мають обмеження щодо моделювання довгострокових залежностей та паралелізації обчислень [39].

Згорткові нейронні мережі (CNN) застосовуються для виявлення локальних патернів у коді. Одновимірні згортки можуть ідентифікувати характерні послідовності токенів, що відповідають типовим помилкам або вразливостям.

TextCNN та подібні архітектури продемонстрували ефективність для класифікації фрагментів коду [55].

Графові нейронні мережі (GNN) спеціально розроблені для обробки графових структур даних. Основні варіанти включають Graph Convolutional Networks (GCN), що узагальнюють операцію згортки на графи; Graph Attention Networks (GAT), що використовують механізм уваги для зважування сусідніх вузлів; Message Passing Neural Networks (MPNN) — загальну парадигму обміну повідомленнями між вузлами. GNN особливо ефективні для аналізу коду, оскільки природно моделюють структурні та семантичні залежності у програмах. Дослідження демонструють, що комбінування мовних моделей з GNN через онлайн-дистиляцію дозволяє досягти кращих результатів у виявленні вразливостей [41].

Трансформери (Transformers) стали домінуючою архітектурою в обробці природної мови та все більше застосовуються до коду. Механізм self-attention дозволяє моделювати залежності між довільними позиціями у послідовності без обмежень RNN. Ключові переваги включають паралелізацію обчислень та здатність захоплювати довгострокові залежності [64].

1.4.4. Попередньо навчені моделі для коду

Парадигма попереднього навчання (pre-training) з подальшим дотренуванням (fine-tuning) революціонізувала застосування нейронних мереж до аналізу коду. Моделі навчаються на великих корпусах коду за допомогою завдань із самонавчанням (self-supervised), а потім адаптуються до конкретних завдань.

CodeBERT [24] — одна з перших великих попередньо навчених моделей для коду, що базується на архітектурі BERT. Навчена на парах код-документація з завданнями Masked Language Modeling (MLM) та Replaced Token Detection.

GraphCodeBERT [26] розширює CodeBERT, додаючи інформацію про потоки даних у процес попереднього навчання, що дозволяє моделі краще розуміти семантичні залежності у коді.

CodeT5 базується на архітектурі T5 (Text-to-Text Transfer Transformer) та підтримує як розуміння, так і генерацію коду. Модель навчена з завданнями span denoising та identifier tagging.

UniXcoder — уніфікована модель, що підтримує множинні модальності коду (текст, AST, коментарі) та множинні задачі (класифікація, генерація, пошук).

Бенчмарк CodeXGLUE [42] забезпечує стандартизовану платформу для оцінки попередньо навчених моделей на широкому спектрі задач, включаючи виявлення дефектів, клонування коду та генерацію.

1.4.5. Великі мовні моделі для аналізу вразливостей

Великі мовні моделі (LLM), такі як GPT-4, Claude та спеціалізовані моделі для коду (CodeLlama, StarCoder), відкрили нові можливості для аналізу програмного коду. Їхня здатність розуміти контекст, генерувати пояснення та навчатися в режимах без прикладів або з кількома прикладами (zero-shot/few-shot) робить їх привабливими для завдань виявлення вразливостей [74].

Систематичний огляд літератури [74] виділяє кілька ключових напрямків: zero-shot та few-shot виявлення без специфічного дотренування [40]; пояснення та обґрунтування виявлених проблем природною мовою [50]; автоматичне виправлення вразливостей.

Проте дослідження також виявляють суттєві обмеження. Емпіричне дослідження [20] показує значний розрив у порівнянні з практичними потребами виявлення вразливостей. Комплексний бенчмаркінг [60] підтверджує варіативність результатів залежно від типу вразливості та способу промптингу.

1.4.6. Застосування нейронних моделей для фільтрації результатів статичного аналізу

Одним з найбільш практичних застосувань нейронних мереж є фільтрація та пріоритизація попереджень від традиційних статичних аналізаторів. Дослідження [33] демонструє ефективність машинного навчання для автоматичного виявлення хибних тривог. Застосування LLM для інспекції тисяч попереджень [67] показує,

що великі мовні моделі можуть ефективно класифікувати попередження, хоча їх продуктивність варіюється залежно від типу дефекту та контексту. DLAP [71] комбінує глибоке навчання з промптингом LLM для виявлення вразливостей, демонструючи переваги гібридних методів.

1.4.7. Обмеження нейронних методів

Незважаючи на значні досягнення, нейронні методи мають суттєві обмеження:

- **Відсутність формальних гарантій** — на відміну від символьних методів, нейронні моделі не надають математичних доказів коректності своїх висновків [10].
- **Проблема галюцинацій** — LLM можуть генерувати правдоподібні, але фактично невірні твердження про код [45].
- **Залежність від навчальних даних** — якість та репрезентативність даних критично впливає на продуктивність; зміщення у даних може призводити до систематичних помилок [29].
- **Вразливість до атак** — нейронні моделі для коду вразливі до adversarial attacks та отруєння даних [37].
- **Обмежена інтерпретованість** — «чорноящиковий» характер глибоких моделей ускладнює розуміння причин рішень [14].
- **Високі обчислювальні вимоги** — тренування та інференс великих моделей потребують значних обчислювальних ресурсів [63].

1.4.8. Порівняння нейронних та символьних методів

Таблиця 1.2 — Порівняння нейронних та символьних методів аналізу коду

Критерій	Нейронні методи	Символьні методи
Формальні гарантії	Відсутні	Надаються за визначенням
Адаптивність	Висока, навчання з даних	Низька, вимагає ручного кодування правил

Інтерпретованість	Обмежена, «чорна скринька»	Висока, формальні докази
Масштабованість	Висока для інференсу	Обмежена вибухом шляхів
Генералізація	На подібні патерни у даних	На формально визначені властивості
Контрприклади	Не генеруються автоматично	Автоматична генерація
Нові типи дефектів	Потребують перенавчання	Потребують нових правил
Обчислювальні вимоги	Високі для тренування	Високі для аналізу

Це порівняння мотивує розробку гібридних нейросимвольних методів, що поєднують переваги обох парадигм. Нейронні компоненти можуть забезпечити адаптивність та масштабованість, тоді як символьні — формальні гарантії та інтерпретованість.

1.5. Нейросимвольні методи

1.5.1. Концепція нейросимвольної інтеграції

Нейросимвольні методи представляють стратегію в галузі штучного інтелекту, що прагне поєднати здатність нейронних мереж до виявлення патернів у великих обсягах даних з логічним виведенням та структурною репрезентацією символьних систем [59]. Ця парадигма інтегрує підсимвольні моделі, які ефективно працюють з нечіткими або неповними даними, з символьними методами, що забезпечують строгість, інтерпретованість та можливість застосування формальних гарантій.

Мотивація для нейросимвольної інтеграції походить з визнання обмежень, властивих кожній парадигмі окремо. Символьні методи, хоч і надають формальні гарантії та високу інтерпретованість, часто страждають від проблем масштабованості, особливо при роботі з великими та складними даними, а також

від жорсткості, що обмежує їхню здатність до адаптації. На противагу цьому, нейронні моделі демонструють виняткову ефективність у навчанні на значних обсягах інформації та виявленні складних залежностей, проте часто функціонують як «чорні скриньки», позбавлені прозорості та формальних гарантій [57].

У контексті статичного аналізу програмного коду нейросимвольні методи відкривають можливості для збільшення точності та зменшення хибнопозитивних спрацювань, автоматичного виведення та уточнення специфікацій, та адаптації аналізу до конкретних проєктів [54].

1.5.2. Типи нейросимвольних систем

Нейросимвольні системи класифікуються за рівнем та способом взаємодії між їхніми нейронними та символьними компонентами [16].

Паралельні (вільно зв'язані) системи. Нейронні та символьні компоненти функціонують відносно незалежно, обмінюючись результатами на високому рівні абстракції. У контексті статичного аналізу: символьний аналізатор генерує потенційні попередження, після чого нейронна модель фільтрує хибнопозитивні або пріоритезує критичні. Переваги: простота інтеграції, можливість використання існуючих інструментів. Обмеження: обмежена глибина взаємодії.

Інтегровані (тісно зв'язані) системи. Нейронні модулі глибоко вбудовані у символьні структури. Нейронна мережа може оцінювати ймовірність успішності різних шляхів виконання у символьному виконанні, спрямовуючи пошук у найбільш перспективні області [59]. Переваги: висока синергія та адаптивність. Обмеження: складність проектування та навчання.

Каскадні (ієрархічні) системи. Вихід однієї компоненти є вхідною інформацією для наступної, формуючи послідовний конвеєр обробки. Нейронна модель може ідентифікувати потенційно вразливі ділянки, а символьний аналізатор виконує їх формальну перевірку; або навпаки — символьний аналізатор попередньо обробляє код, а нейронна модель інтерпретує його вихід. Переваги: чітка архітектура, модульність. Обмеження: помилки на ранніх етапах можуть каскадно поширюватися.

1.5.3. Роль нейронних компонентів у підсиленні статичного аналізу

У контексті статичного аналізу програмного коду нейронні компоненти, включаючи великі мовні моделі, відіграють допоміжну роль, не замінюючи формальну перевірку, а підсилюючи її ефективність.

Нейронні моделі застосовуються для: прогнозування релевантності попереджень — фільтрація хибнопозитивних спрацювань на основі історичних даних; інференції специфікацій та політик — автоматичне виведення кандидатних правил taint-аналізу або інваріантів для формальної перевірки; генерації та ранжування шляхів виконання — направлення символьного рушія у найбільш перспективні області; навчання на історичних даних з систем відстеження проблем та Git-репозиторіїв.

Нейронна компонента не має на меті замінити строгість формальної перевірки. Її призначення полягає в оптимізації процесу аналізу шляхом звуження простору пошуку, вказівки на потенційно небезпечні ділянки коду та допомоги розробникам в інтерпретації результатів.

1.5.4. Приклади нейросимвольних алгоритмів

Нейросимвольне програмування. Концепція нейросимвольного програмування (Neurosymbolic Programming) включає системи, де програми поєднують символьні примітиви з нейронними модулями [8]. Ці системи використовують доменно-специфічні мови (DSL) для символьного представлення програми, де певні компоненти реалізуються як нейронні мережі, створюючи інтерпретовані програми, які добре узагальнюються та ефективно вводять індуктивні зміщення [59].

Neural Program Generation Modulo Static Analysis [43] інтегрує статичний аналіз безпосередньо у процес генерації нейронних програм. Нейронна модель генерує код, а статичний аналізатор виступає як механізм зворотного зв'язку, оцінюючи коректність та відповідність специфікаціям. Це підвищує семантичну коректність згенерованого коду порівняно з чисто нейронними моделями.

LLM-орієнтовані нейросимвольні системи. З розвитком великих мовних моделей виникли гібридні системи: Vul-LMGNNs [41] поєднує мовні моделі з GNN через стратегію злиття; DLAP [71] комбінує глибоке навчання з промптингом LLM для покращення виявлення вразливостей. Ці системи демонструють, що поєднання глибокого семантичного розуміння коду з формальним аналізом покращує точність виявлення.

1.5.5. Основні виклики гібридних систем

Незважаючи на значний потенціал, нейросимвольні системи стикаються з фундаментальними викликами.

Масштабованість. Обчислювальна вартість поєднання моделей глибокого навчання з формальними методами може бути значною. Ефективна інтеграція вимагає оптимізованих алгоритмів, що дозволяють обробляти великі кодові бази без надмірних ресурсних витрат.

Збереження формальних гарантій. Використання евристичних або ймовірнісних нейронних модулів може потенційно підірвати формальні гарантії символьних методів. Розробка механізмів, що дозволяють нейронним компонентам посилювати аналіз, не компрометуючи його обґрунтованість (soundness), є критично важливою.

Інтерпретованість та прозорість. Інтеграція «чорних скриньок» нейронних мереж може ускладнити розуміння причинно-наслідкових зв'язків у роботі гібридної системи. Забезпечення інтерпретованості результатів залишається відкритим питанням.

Якість та репрезентативність даних. Для навчання нейронних компонентів потрібні великі обсяги якісних даних [29]. У вузьких доменах, таких як специфічні аспекти безпеки програмного забезпечення, такі дані можуть бути обмеженими.

Оцінювання та бенчмаркінг. Порівняння нейросимвольних систем з класичними аналізаторами та чисто нейронними моделями вимагає розробки адекватних метрик та стандартизованих бенчмарків.

Інтеграція у реальні процеси розробки. Впровадження нових аналітичних інструментів у існуючі цикли CI/CD вимагає сумісності, надійності та мінімального накладного навантаження.

1.6. Висновки до розділу

У цьому розділі розглянуто теоретичні основи, що формують фундамент для нейросимвольного статичного аналізу програмного коду.

Класичний статичний аналіз спирається на строгий математичний апарат: теорію решіток, абстрактну інтерпретацію та зв'язки Галуа для формалізації відношень між конкретною та абстрактною семантикою програм. Методи аналізу потоку даних та управління, абстрактна інтерпретація, символьне виконання та перевірка на основі моделей забезпечують формальні гарантії коректності, проте стикаються з фундаментальними обмеженнями масштабованості: вибух шляхів, обчислювальна складність SMT-вирішувачів та нерозв'язність загальної задачі.

Нейронні методи — від рекурентних та згорткових мереж, через графові нейронні мережі (GNN), до попередньо навчених трансформерних моделей та великих мовних моделей (LLM) — забезпечують адаптивність, масштабованість та здатність виявляти семантично складні дефекти без явного програмування правил. Водночас вони позбавлені формальних гарантій, страждають від проблеми галюцинацій та мають обмежену інтерпретованість.

Нейросимвольні методи пропонують стратегію інтеграції переваг обох парадигм через паралельні, інтегровані або каскадні архітектури. Головні виклики такої інтеграції — збереження формальних гарантій, забезпечення інтерпретованості та практична масштабованість — визначають простір для нових архітектурних рішень. Детальний аналіз конкретних нейросимвольних методів у контексті виявлення вразливостей, ідентифікація прогалини в існуючих дослідженнях та обґрунтування позиціонування методу VERIGATE представлено у наступному розділі.

2. АНАЛІЗ СУЧАСНИХ ДОСЛІДЖЕНЬ

Цей розділ присвячений систематичному аналізу сучасного стану досліджень у галузі автоматичного виявлення дефектів та вразливостей у програмному коді. Аналіз побудований як послідовне звуження фокусу: від класичних інструментів статичного аналізу, через методи машинного та глибокого навчання (включаючи графові нейронні мережі), до великих мовних моделей та нейросимвольних методів. На кожному етапі ідентифікуються як досягнення, так і фундаментальні обмеження, що мотивують перехід до наступного покоління методів. В кінці розділу розглядається виявлена прогалина — відсутність незалежної символьної верифікації тверджень LLM — яку заповнює метод VERIGATE.

2.1. Класичний статичний аналіз програмного коду

2.1.1. Інструменти на основі правил та їх еволюція

Статичний аналіз програмного коду — дисципліна з понад тридцятирічною історією, що розвивалась від простих перевірок синтаксису до складних формальних методів. Ранні інструменти на кшталт Lint (1979) виконували переважно синтаксичні перевірки: невикористані змінні, потенційно небезпечні конструкції, порушення стилю кодування [12]. Подальша еволюція призвела до створення інструментів, здатних виконувати більш глибокий аналіз: FindBugs для Java, PMD [49], а пізніше — Semgrep [56], Bandit [51] для Python та інші спеціалізовані аналізатори.

Сучасні інструменти на основі правил можна класифікувати за глибиною аналізу. Синтаксичні аналізатори (Semgrep, PMD) працюють переважно на рівні абстрактних синтаксичних дерев (AST) та виявляють патерни, що відповідають відомим типам помилок. Інструменти з підтримкою потоку даних (Bandit, Pylint) враховують залежності між змінними та виконують базовий taint-аналіз. Нарешті, глибокі аналізатори (CodeQL [4], Infer [6]) застосовують абстрактну інтерпретацію

та складний аналіз потоку даних для виявлення таких дефектів, як розіменування нульових вказівників, витоки ресурсів та переповнення буферів [30; 2].

2.1.2. Можливості та фундаментальні обмеження

Головною перевагою інструментів на основі правил є детермінованість: однакові входи завжди дають однакові виходи, що забезпечує відтворюваність та передбачуваність результатів [68]. Кожне виявлення може бути прослідковане до конкретного правила, що забезпечує високу інтерпретованість. Досвід Google [52] та GitHub [15] демонструє успішне впровадження таких інструментів у промислові CI/CD пайплайни.

Водночас інструменти на основі правил мають три фундаментальні обмеження. По-перше, обмежене покриття: інструменти виявляють лише ті типи дефектів, для яких існують наперед визначені правила. Семантичні помилки — неправильні імена змінних, некоректна логіка умовних виразів, помилки в аргументах функцій — залишаються невидимими для синтаксичних патернів [48]. По-друге, високий рівень хибнопозитивних спрацювань: дослідження демонструють, що в реальних проєктах до 80% попереджень статичних аналізаторів є хибнопозитивними, що призводить до втоми від сповіщень (alert fatigue) та ігнорування результатів розробниками [25; 9]. По-третє, значні інженерні зусилля для створення нових правил: адаптація до нових типів вразливостей потребує ручної розробки патернів експертами з безпеки [27].

Дослідження ставлення розробників до інструментів статичного аналізу [21] підтверджує, що інтерпретованість та довіра є критичними факторами прийняття. Розробники використовують інструменти тоді, коли розуміють логіку виявлень і можуть верифікувати їх коректність. Цей висновок є важливим для подальшого аналізу нейронних методів, де інтерпретованість суттєво нижча.

2.2. Методи машинного навчання для аналізу коду

2.2.1. Класичне машинне навчання

Застосування машинного навчання до задач аналізу коду розпочалося з класичних методів — Random Forest, SVM, Naive Bayes — що працювали з сконструйованими вручну ознаками коду: метриками складності (cyclomatic complexity, Halstead metrics), статистикою токенів, структурними характеристиками [29; 18]. Головною перевагою була здатність автоматично виявляти закономірності в даних без експертного визначення правил. Однак якість моделей критично залежала від інженерії ознак (feature engineering) — процесу відбору та конструювання ознак, що вимагав глибоких експертних знань і був специфічним для кожного типу дефектів та мови програмування.

2.2.2. Глибоке навчання на послідовностях

Архітектури глибокого навчання усунули потребу в ручному конструюванні ознак. Рекурентні нейронні мережі (RNN, LSTM) та згорткові мережі (CNN) навчалися автоматично витягувати представлення коду з послідовностей токенів [39; 55]. Системи VulDeePecker та SySeVR продемонстрували здатність виявляти вразливості на основі навчених представлень (learned representations), обробляючи код як послідовність символів або токенів.

Ці методи покращили покриття порівняно з інструментами на основі правил, оскільки навчалися розпізнавати патерни з даних, а не з експертних правил. Водночас послідовне представлення коду ігнорувало його структурні властивості — ієрархію AST, потоки даних та управління — що обмежувало ефективність на складних семантичних дефектах.

2.2.3. Графові нейронні мережі для аналізу коду

Принципово інший підхід запропонували методи на основі графових нейронних мереж (Graph Neural Networks, GNN), що моделюють код як граф — з

вершинами, що відповідають елементам програми, та ребрами, що відображають структурні та семантичні зв'язки.

Devign [73] став одним із перших методів, що застосував GNN до задачі виявлення вразливостей на рівні функцій. Метод конструює комбінований граф коду (Composite Code Graph), що поєднує абстрактне синтаксичне дерево (AST), граф керування потоком (Control Flow Graph, CFG), граф потоку даних (Data Flow Graph, DFG) та природну послідовність коду (Natural Code Sequence). Графова нейронна мережа з керованими вентилями (Gated Graph Neural Network, GGNN) обробляє цей граф, навчаючись виявляти патерни вразливостей, що залежать від структурних зв’язків між елементами коду. Devign продемонстрував перевагу над послідовними моделями, підтверджуючи важливість структурного представлення коду.

Піонерська робота Yamaguchi та ін. [70] з рафами властивостей коду (Code Property Graph, CPG) заклала теоретичне підґрунтя для графового представлення коду, об'єднавши абстрактне синтаксичне дерево (AST), граф керування потоком (CFG) та граф залежностей програми (Program Dependence Graph, PDG) у єдину структуру даних, придатну як для запитів обходу графа, так і для навчання графових нейронних мереж (GNN).

Vul-LMGNNs [41] є прикладом сучасного гібридного методу, що поєднує мовні моделі з графовими нейронними мережами. Метод використовує мовну модель для генерації контекстних представлень коду та графову нейронну мережу для моделювання структурних залежностей, досягаючи покращення завдяки стратегії злиття представлень.

Методи на основі графових нейронних мереж мають кілька вагомих переваг: здатність моделювати нелокальні залежності в коді (data flow, control flow), природне представлення ієрархічної структури програм та можливість враховувати семантику зв'язків між елементами. Водночас вони стикаються з характерними обмеженнями. По-перше, потреба у великих обсягах розмічених даних для тренування — створення якісних датасетів вразливостей є

трудомістким процесом. По-друге, обмежена генералізація на нові типи дефектів, які не представлені у навчальних даних. По-третє, необхідність донавчання (fine-tuning) для кожної конкретної задачі та мови програмування. Нарешті, складність інтерпретації рішень: хоча GNN враховують структуру коду, пояснити, чому конкретний фрагмент класифіковано як вразливий, залишається складним завданням.

2.2.4. Pre-trained моделі для коду

Трансформерні архітектури, попередньо навчені на великих корпусах коду, стали наступним етапом еволюції. CodeBERT [24] був одним з перших, хто продемонстрував ефективність трансферного навчання для задач аналізу коду, використовуючи bimodальне pre-training на парах код-текст. GraphCodeBERT [26] розвинув цю ідею, інтегрувавши інформацію про потоки даних у процес pre-training, що покращило розуміння семантичних зв'язків у коді.

VulBERTa [28] спеціалізований саме для задачі виявлення вразливостей, демонструючи, що pre-trained моделі, адаптовані через fine-tuning на специфічних датасетах, можуть досягати конкурентоспроможних результатів.

Pre-trained моделі суттєво знизили потребу в domain-specific даних порівняно з навчанням "з нуля", проте залишалися залежними від fine-tuning для кожної конкретної задачі та датасету. Це обмежувало їх гнучкість та швидкість адаптації до нових типів дефектів.

Таблиця 2.1 — Еволюція методів виявлення дефектів у коді

Покоління	Представлення коду	Типові методи	Переваги	Обмеження
Rule-based	Синтаксичні патерни	Pattern matching, AST traversal	Детерміністичність, інтерпретованість	Фіксовані правила, низька адаптивність
Класичний ML	Ручні ознаки (метрики)	SVM, Random Forest, Naive Bayes	Автоматичне навчання	Залежність від feature engineering
DL (послідовності)	Token embeddings	RNN, LSTM, CNN	Автоматичне витягування ознак	Ігнорує структуру коду
DL (графи, GNN)	Графові представлення	GGNN, GAT на CPG/CFG/DFG	Моделювання структурних залежностей	Потреба у великих розмічених даних
Pre-trained моделі	Contextual embeddings	CodeBERT, GraphCodeBERT, VulBERTa	Transfer learning	Fine-tuning для кожної задачі
LLM	In-context learning	GPT-4, Claude, CodeLlama	Zero-shot, пояснення, виправлення	Галюцинації, відсутність гарантій

2.3. Великі мовні моделі для аналізу коду

2.3.1. Парадигмальний зсув: від fine-tuning до in-context learning

Великі мовні моделі (LLM) — GPT-4, Claude, CodeLlama та інші — принесли парадигмальний зсув в аналіз коду. На відміну від попередніх моделей глибокого навчання (включаючи GNN та pre-trained трансформери), LLM здатні виконувати задачі виявлення вразливостей у zero-shot та few-shot режимах, тобто без будь-якого специфічного навчання чи fine-tuning [40; 50]. Ця здатність ґрунтується

на масштабі pre-training: сучасні LLM навчені на мільярдах рядків коду з відкритих репозиторіїв, що забезпечує широке покриття мов програмування, фреймворків та патернів помилок [64].

Дослідження демонструють, що LLM, такі як GPT-4 та Claude, можуть виявляти різноманітні типи вразливостей без спеціалізованого навчання [58; 46; 53]. Порівняльна оцінка у zero-shot режимі [40] показує, що LLM конкурентоспроможні з fine-tuned моделями на деяких бенчмарках, хоча демонструють значну варіативність залежно від типу вразливості та стратегії промптингу.

Ключові переваги LLM порівняно з попередніми поколіннями методів: здатність до семантичного розуміння коду, що виходить за межі синтаксичних патернів; мультимовність без окремого тренування; генерація пояснень та виправлень природною мовою; адаптивність через промптинг без зміни ваг моделі [74].

2.3.2. Фундаментальні проблеми LLM у контексті аналізу коду

Попри вражаючі можливості, LLM стикаються з чотирма фундаментальними проблемами, що обмежують їх надійне використання для виявлення вразливостей.

Обмежена контрольованість. LLM позначають проблеми за межами передбаченого обсягу аналізу — модель може повідомляти про стилістичні недоліки, потенційні покращення або теоретичні ризики, коли від неї очікується виявлення лише конкретних типів дефектів. Контроль обсягу виявлень є критичним для практичного застосування, де "alert fatigue" від нерелевантних попереджень знижує ефективність аналітиків безпеки.

Неконсистентна поведінка. Варіації у формулюванні промпту призводять до суттєво різних результатів, що ускладнює відтворюваність та порівняння [60]. Дослідження [20] демонструє значний розрив між поточними можливостями LLM та практичними потребами виявлення вразливостей.

Неверифіковані твердження. Black-box природа LLM означає, що навіть коли модель надає обґрунтування, воно може бути post-hoc раціоналізацією, а не

відображенням фактичного процесу прийняття рішення. Система не має механізму відрізнити валідне твердження від "галюцинації" — правдоподібного, але фактично невірного виходу [10].

Відсутність детерміністичних гарантій. На відміну від інструментів на основі правил, LLM не забезпечують формальних гарантій коректності. Результати є імовірнісними, що фундаментально обмежує застосування у security-critical контекстах, де пропущена вразливість може мати серйозні наслідки.

2.3.3. Емпіричні результати та бенчмарки

Для систематичної оцінки різних методів аналізу коду розроблено кілька стандартних бенчмарків. CodeXGLUE [42] — комплексний бенчмарк, що включає задачі defect detection, clone detection та code summarization, забезпечуючи стандартизовану платформу для порівняння. Defects4J [31] — база реальних дефектів з Java-проєктів для оцінки здатності виявляти та виправляти помилки. PySStuBs [32] — датасет однорядкових помилок у Python (73 013 пар correct/buggy), що оцінює здатність виявляти subtle bugs на рівні окремих statements.

Емпіричні дослідження виявляють суттєву варіативність результатів LLM. Мікробенчмаркове дослідження [62] показує, що LLM демонструють різну ефективність для різних категорій помилок. Комплексний бенчмаркінг [60] підтверджує залежність результатів від типу вразливості, мови програмування та стратегії промптингу. Порівняльна оцінка у zero-shot режимі [40] виявляє, що більші моделі не завжди показують кращі результати.

Систематичний огляд [74] підсумовує тенденцію: LLM перевершують традиційні DL-моделі на деяких задачах, але демонструють значну варіативність між датасетами та типами вразливостей. Це підтверджує потребу в механізмах контролю та верифікації результатів LLM.

Таблиця 2.2 — Емпіричні результати LLM для виявлення вразливостей

Дослідження	Модель	Датасет	Ключові результати	Обмеження
[74]	Огляд LLM	Множинні	LLM перевершують деякі DL-моделі	Варіативність між датасетами
[20]	GPT-4, CodeLlama	Real-world vulns	Значний gap до практичних потреб	Обмежена генералізація
[60]	Multiple LLMs	Security benchmarks	Варіативність за типом вразливості	Залежність від промптингу
[40]	GPT-4, Claude та ін.	Zero-shot evaluation	Більший ≠ кращий; промпт критичний	Нестабільність результатів
[67]	LLM for triage	SA warnings	Ефективна фільтрація FP	Потребує tool integration

2.4. Нейросимвольні методи аналізу програм

2.4.1. Концепція нейросимвольного ШІ

Нейросимвольний штучний інтелект (Neurosymbolic AI) — парадигма, що прагне поєднати сильні сторони нейронних мереж (здатність до навчання з даних, pattern recognition, адаптивність) з перевагами символьних методів (формальні гарантії, інтерпретованість, reasoning) [54; 57]. Систематичний огляд [16] виділяє кілька парадигм інтеграції:

- **Symbolic → Neural:** символьні знання використовуються для направлення або обмеження нейронного навчання.
- **Neural → Symbolic:** нейронні моделі генерують кандидатів або гіпотези, які верифікуються символьними методами.
- **Гібридні архітектури:** нейронні та символьні компоненти тісно інтегровані в єдину систему.

У контексті аналізу програмного коду нейросимвольні методи є особливо перспективними з кількох причин. Код має чітку формальну структуру (AST, CFG, семантика), що природно підходить для символьної обробки. LLM демонструють потужне семантичне розуміння, але не надають гарантій. Комбінація потенційно забезпечує адаптивність LLM з верифікованістю символьних методів. Концепція нейросимвольного програмування [8] формалізує підходи до інтеграції диференційовних нейронних модулів із символьними програмами.

2.4.2. IRIS: символьний аналіз → нейронна фільтрація

IRIS [35] реалізує напрямок Symbolic → Neural, де LLM використовується для покращення та фільтрації результатів класичного статичного аналізатора.

Архітектура IRIS складається з двох компонентів. Спочатку LLM автоматично виводить специфікації taint-аналізу — визначає, які функції є джерелами (sources), приймачами (sinks) та санітизаторами (sanitizers) небезпечних даних. Потім ці специфікації передаються до CodeQL, який виконує формальний taint tracking на рівні цілого репозиторію. Додатково LLM фільтрує результати CodeQL, усуваючи false positives.

IRIS демонструє, що LLM може ефективно конфігурувати символьний аналізатор, зменшуючи потребу в ручному визначенні специфікацій. Водночас метод має характерне обмеження: покриття IRIS обмежене можливостями базового символьного аналізатора (CodeQL). Якщо CodeQL не здатний виявити певний тип дефекту (наприклад, семантичні помилки в іменуванні змінних), IRIS також буде неефективним, оскільки LLM обробляє лише те, що вже знайдене символьним компонентом. Крім того, IRIS потребує повного контексту репозиторію для побудови бази CodeQL та значної інфраструктури для розгортання.

2.4.3. LLMSA: композиційний нейросимвольний аналіз

LLMSA [65] пропонує композиційну архітектуру, де LLM виконує аналіз окремих компонентів програми з подальшою агрегацією результатів. Метод

реалізує багатоетапний процес на основі LLM: програма розбивається на компоненти (функції, модулі), кожен аналізується LLM окремо, а результати агрегуються з урахуванням залежностей.

Ключові характеристики LLMSA: не вимагає компіляції коду (compilation-free), тип аналізу визначається промптом (customizable analysis), обробка на рівні уривкув коду (snippet-level). Автори повідомляють про підвищення F1-міри приблизно на 0,20 у задачі виявлення вразливостей, пов’язаних із taint-аналізом, порівняно з базовим рішенням, що використовує лише LLM.

Однак LLMSA не включає механізму незалежної верифікації тверджень LLM. Кожен етап спирається на міркування, згенеровані LLM, а помилка на одному з етапів поширюється через усю систему. Відсутність символьного верифікатора означає, що "галюцинації" LLM можуть залишатися невиявленими.

2.4.4. MoCQ: ітеративне уточнення патернів

MoCQ [36] використовує ітеративний цикл взаємодії між LLM та класичним статичним аналізом для видобутку та уточнення патернів вразливостей. LLM генерує гіпотези про патерни, статичний аналіз перевіряє їх на кодовій базі, результати повертаються до LLM для уточнення.

MoCQ масштабує виявлення без ручної курації патернів, проте має характерні обмеження: потреба у значній інфраструктурі для ітеративних циклів, повний контекст проєкту для виконання статичного аналізу та відсутність верифікації кожного окремого виявлення — метод зосереджений на уточненні патернів, а не на валідації конкретних тверджень.

2.4.5. Інші гібридні методи

Окрім трьох основних нейросимвольних методів, у літературі представлено кілька суміжних робіт, що досліджують різні аспекти інтеграції нейронних та символьних компонентів.

Neural Program Generation Modulo Static Analysis [43] пропонує метод, де нейронний генератор коду обмежується символьними constraints від статичного аналізатора, забезпечуючи, що згенерований код задовольняє певні властивості (type safety, absence of null dereferences). Хоча цей метод спрямований на генерацію, а не виявлення дефектів, він демонструє принцип символьного обмеження нейронних виходів.

DLAP [71] комбінує deep learning моделі з LLM через augmented prompting: DL-модель генерує додатковий контекст (vulnerability likelihood, relevant code patterns), який включається у промпт для LLM. Метод покращує якість LLM-виявлень через додатковий контекст, але не забезпечує незалежної верифікації.

Combining Foundation Models and Symbolic AI [5] досліджує загальні принципи інтеграції foundation models з символьним ШІ для автоматичного виявлення та виправлення вразливостей, визначаючи перспективні напрямки розвитку без конкретної реалізації процесу верифікації.

Методи верифікації виходів ШІ досліджуються і за межами аналізу коду. Chain-of-Verification [19] та Self-Consistency [66] підвищують надійність через перехресну валідацію виходів самою моделлю. AlphaCode [38] застосовує фільтрацію для відбору якісних рішень серед множини кандидатів. Ці роботи демонструють загальну тенденцію до верифікації виходів нейронних моделей, проте покладаються на додаткові виклики LLM або статистичні механізми, а не на незалежний символьний аналіз.

2.4.6. Порівняння нейросимвольних методів

Аналіз існуючих нейросимвольних методів дозволяє виділити ключові архітектурні відмінності та ідентифікувати незаповнену нішу.

Таблиця 2.3 — Порівняння нейросимвольних методів аналізу коду

Характеристика	**IRIS**	**LLMSA**	**MoCQ**	**VERIGATE**
Архітектура	LLM фільтрує CodeQL	Композиційна	Ітеративне уточнення	**Gate-based**
Напрямок	Symbolic → Neural	Multi-stage LLM	Bidirectional	**Neural → Symbolic**
Snippet-level	Ні (репозиторій)	Так	Ні (проєкт)	**Так**
Training-free	Ні	Так	Ні	**Так**
Незалежна верифікація	Ні (LLM фільтрує)	Ні (LLM reasoning)	Ні (паттерн-рівень)	**Так (символьний gate)**
Structured claims	Ні	Ні	Ні	**Так**
Зовнішні залежності	CodeQL, повний проєкт	Множинні LLM виклики	Повний проєкт	**Мінімальні**

Таблиця 2.3 виявляє чітку закономірність: жоден з існуючих методів не поєднує незалежну символьну верифікацію тверджень LLM з напрямком Neural → Symbolic на snippet-level. IRIS використовує LLM для покращення символьного аналізу (Symbolic → Neural), але не верифікує твердження LLM. LLMSA покладається виключно на LLM reasoning без незалежного символьного верифікатора. MoCQ ітеративно уточнює патерни, але не валідує окремі виявлення.

2.5. Визначення прогалини та позиціонування дослідження

2.5.1. Систематизація обмежень існуючих методів

Аналіз еволюції методів — від інструментів на основі правил до LLM та нейросимвольних систем — дозволяє виділити незаповнену прогалину цієї області, що складається з наступних 5 компонентів:

1. **Відсутність незалежної верифікації тверджень LLM**. Жоден з існуючих нейросимвольних методів не забезпечує незалежну символьну верифікацію кожного конкретного твердження LLM. IRIS верифікує через LLM-фільтрацію (нейронна, а не символьна верифікація). LLMSA покладається на мультиетапний LLM reasoning (нейронна верифікація через додаткові виклики). MoCQ верифікує на рівні патернів, а не окремих виявлень. Результатом є відсутність формальних гарантій того, що конкретне виявлення LLM відповідає фактичним властивостям програми.
2. **Невикористаний напрямок Neural → Symbolic verification**. Існуючі методи переважно реалізують напрямок Symbolic → Neural (IRIS: CodeQL → LLM filter) або складні багатоетапні процеси на основі LLM (LLMSA). Інверсний напрямок — де LLM генерує структуровані гіпотези, а символьний аналіз їх незалежно верифікує — залишається недостатньо дослідженим. Цей напрямок принципово важливий, оскільки знімає обмеження покриття символьного аналізатора: LLM може генерувати гіпотези про довільні типи дефектів, а символьна верифікація забезпечує контроль якості.
3. **Відсутність structured claims від LLM**. Існуючі методи використовують LLM для бінарної класифікації (vulnerable/safe), генерації специфікацій або вільнотекстових обґрунтувань. Ніхто не вимагає від LLM генерації структурованих, символьно перевірюваних тверджень з конкретними полями (тип дефекту, точна локалізація в коді, мінімальне виправлення, обґрунтування), що дозволили б символьному компоненту верифікувати конкретні аспекти кожного твердження.

4. **Розрив між інфраструктурними вимогами та практичністю**. IRIS та MoCQ потребують повного контексту репозиторію та значної інфраструктури. Lightweight нейросимвольний метод, що працює на snippet-level з мінімальними зовнішніми залежностями та одним викликом LLM на аналіз, залишається незаповненою нішею.
5. **Відсутність механізму контрольованої поведінки LLM**. Існуючі методи не адресують проблему контрольованості LLM явно. Система, що повідомляє виключно ті проблеми, які є одночасно семантично правдоподібними (за оцінкою LLM) та структурно верифікованими (засобами символьного аналізу), забезпечила б передбачувану поведінку, необхідну для інтеграції у промислові процеси.

2.5.2. Вимоги до методу, що заповнює прогалину

На основі виявлених прогалин формулюються вимоги до методу, здатного їх адресувати.

Вимога 1: **Генерація структурованих тверджень**. LLM повинен генерувати не вільнотекстові відповіді, а структуровані твердження з конкретними полями: тип дефекту (CWE/категорія), точна локалізація у коді (spans), мінімальне виправлення, обґрунтування. Така структура дозволяє символьному компоненту верифікувати конкретні аспекти кожного твердження незалежно від LLM.

Вимога 2: **Незалежна символьна верифікація**. Метод повинен включати символьний компонент, що незалежно від LLM перевіряє валідність тверджень. Верифікація має базуватися на формальних властивостях коду: структурі AST, потоках даних, відповідності патернам, правдоподібності запропонованих виправлень.

Вимога 3: **Напрямок Neural → Symbolic**. Архітектура має реалізовувати напрямок, де LLM генерує гіпотези, а символьна система їх верифікує, а не навпаки. Це знімає обмеження покриття символьного аналізатора та дозволяє виявляти семантичні дефекти за межами наперед визначених правил.

Вимога 4: **Механізм балансування на основі доказів**. Рішення про класифікацію має враховувати силу доказів від різних джерел: результати верифікації, знахідки інструментів, рівень впевненості LLM. Це дозволяє уникнути як надмірної строгості (відхилення легітимних виявлень), так і надмірної пермісивності (пропуск false positives).

Вимога 5: **Розширюваність без потреби додаткового навчання**. Метод має адаптуватися до нових типів дефектів та моделей без перенавчання. Розширення через модифікацію промптів та додавання символьних правил забезпечує гнучкість та швидке реагування на нові загрози.

2.5.3. Позиціонування методу VERIGATE

Метод VERIGATE заповнює виявлену прогалину як перший нейросимвольний метод статичного аналізу коду з gate-based архітектурою у напрямку Neural → Symbolic, де кожне твердження LLM підлягає незалежній символьній верифікації.

Позиціонування VERIGATE щодо існуючих методів можна описати через три осі. По осі напрямку нейросимвольної взаємодії: IRIS та подібні методи реалізують Symbolic → Neural (символьний аналізатор генерує виявлення, LLM їх фільтрує), тоді як VERIGATE реалізує Neural → Symbolic (LLM генерує гіпотези, символьна система їх верифікує). По осі механізму верифікації: LLMSA покладається на мультиетапний LLM reasoning, MoCQ — на ітеративне уточнення патернів, тоді як VERIGATE використовує explicit verification gate з багатокритеріальним символьним аналізом. По осі практичності: IRIS та MoCQ потребують повного контексту репозиторію, тоді як VERIGATE працює на snippet-level з мінімальними залежностями та одним викликом LLM.

Метод VERIGATE реалізує всі п’ять сформульованих вимог: structured claim generation через JSON-формат тверджень, незалежну символьну верифікацію через Explain-Verify Gate, напрямок Neural → Symbolic, Evidence-Weighted Suppression як механізм балансування, training-free розширюваність та snippet-level аналіз з вбудованою верифікацією.

2.6. Висновки до розділу

Проведений аналіз демонструє послідовну еволюцію методів виявлення дефектів у програмному коді, кожне покоління яких адресує обмеження попереднього, водночас вносячи нові виклики.

Інструменти на основі правил забезпечують детермінованість та інтерпретованість, але обмежені заздалегідь визначеними патернами. Методи ML та DL (включаючи GNN, здатні моделювати структурні залежності через графові представлення коду) розширюють покриття через автоматичне навчання, але потребують значних обсягів розмічених даних та мають обмежену генералізацію. Pre-trained трансформерні моделі знижують потребу в domain-specific даних, але залишаються залежними від fine-tuning. LLM принесли парадигмальний зсув до zero-shot аналізу, проте стикаються з фундаментальними проблемами контрольованості, неверифіковані твердження та відсутності формальних гарантій.

Нейросимвольні методи (IRIS, LLMSA, MoCQ) демонструють перспективність інтеграції нейронних та символьних компонентів. Проте аналіз виявляє конкретну прогалину: жоден з існуючих методів не забезпечує незалежну символьну верифікацію кожного твердження LLM у напрямку Neural → Symbolic на snippet-level без значних інфраструктурних вимог. Цю прогалину заповнює метод VERIGATE, що реалізує gate-based архітектуру з explicit verification через структуровані твердження та багатокритеріальний символьний аналіз. Детальний опис методу представлено у наступному розділі.

3. НЕЙРОСИМВОЛЬНИЙ МЕТОД СТАТИЧНОГО АНАЛІЗУ КОДУ НА ОСНОВІ ВЕРИФІКАЦІЙНОГО ШЛЮЗУ

3.1. Концептуальні засади нейросимвольного статичного аналізу

3.1.1. Обґрунтування нейросимвольного методу

Сучасний стан статичного аналізу програмного коду характеризується фундаментальним протиріччям між двома домінуючими парадигмами. Традиційні інструменти на основі правил (Semgrep [56], Bandit [51], CodeQL [4], Pylint) забезпечують детерміністичну поведінку та високу точність у межах наперед визначених патернів, проте потребують значних інженерних зусиль для визначення нових правил аналізу, часто генерують хибнопозитивні спрацювання при аналізі складних кодових баз та мають обмежену ефективність щодо семантичних дефектів, що виходять за межі синтаксичних шаблонів. Великі мовні моделі (LLM) демонструють здатність до глибокого семантичного розуміння коду та виявлення контекстуально залежних проблем, однак стикаються з чотирма фундаментальними проблемами: обмежена контрольованість, неконсистентна поведінка, неверифіковані твердження та відсутність детерміністичних гарантій (детально розглянуто у Вступі).

Нейросимвольні методи (NeuroSymbolic AI) становлять клас методів штучного інтелекту, що поєднують дві фундаментальні парадигми обчислень:

- Нейронна компонента базується на використанні моделей глибокого навчання, зокрема великих мовних моделей, які ефективно працюють з нечіткою, неструктурованою інформацією. Ці моделі здатні робити узагальнення, знаходити неочевидні закономірності та виконувати семантичний аналіз програмних текстів.
- Символьна компонента ґрунтується на використанні формальних правил, логічних систем та структур даних (абстрактні синтаксичні дерева, графи потоку управління, графи потоку даних). Символьні методи дозволяють

виконувати точні перевірки, формальні виведення та гарантувати строгість результатів.

У контексті статичного аналізу коду нейросимвольна інтеграція реалізується через наступну концептуальну схему: нейронний компонент аналізує код на основі семантичного розуміння, виявляє підозрілі фрагменти та формулює гіпотези про дефекти; символьний компонент виконує формальний аналіз коду (AST, taint analysis, fix plausibility), верифікуючи ці гіпотези; інтеграційний механізм забезпечує взаємодію компонентів для досягнення контрольованої поведінки: основою виявлень є верифіковані твердження, при цьому, твердження, що були відхиленні на етапі верифікації, можуть бути збереженими за наявності вагомих доказів від інструментів.

3.1.2. Аналіз існуючих методів нейросимвольного статичного аналізу

У галузі нейросимвольного аналізу програм виділяються три ключові системи, кожна з яких реалізує відмінний архітектурний підхід.

IRIS (LLM-Assisted Static Analysis) [35] — метод, у якому традиційний символьний аналізатор CodeQL виконує первинне виявлення потенційних вразливостей, а LLM використовується для висновку taint-специфікацій та фільтрації результатів. Напрямок взаємодії: CodeQL → LLM (символьний → нейронний). Система демонструє покращення виявлення порівняно з CodeQL, одночасно зменшуючи кількість хибних спрацювань. Основними обмеженнями є залежність від покриття CodeQL правилами — LLM розширює, але не замінює символьний аналіз — а також потреба у повному контексті репозиторію, що обмежує застосовність до ізольованих фрагментів коду.

LLMSA (LLM-based Static Analysis) [65] — метод композиційного нейросимвольного статичного аналізу, де LLM виконує аналіз потоків даних через багатоетапну обробку. Напрямок: багатоетапний LLM із символьною структуризацією. Система покращує виявлення taint-вразливостей, не потребує компіляції, дозволяє налаштування правил аналізу. Обмеженнями є складність

налаштування та відсутність явного механізму верифікації тверджень LLM — згенеровані виявлення не проходять незалежну символьну валідацію.

MoCQ [36] — метод ітеративного циклу LLM та класичного статичного аналізу для видобутку та уточнення патернів вразливостей. Напрямок: двонаправлений (bidirectional). Система масштабує виявлення без ручної курації правил. Обмеженнями є потреба у значній інфраструктурі, ітеративні цикли зворотного зв'язку та відсутність явних механізмів верифікації для кожного окремого виявлення.

Спільним обмеженням існуючих методів є відсутність явного механізму незалежної верифікації тверджень LLM. Системи або обмежують роль LLM допоміжною функцією фільтрації/розширення символьних результатів (IRIS, MoCQ), або не верифікують згенеровані LLM виявлення через незалежний символьний аналіз (LLMSA). Це залишає проблему галюцинацій та неконтрольованої поведінки LLM невирішеною.

3.1.3. Ключова інновація VERIGATE: інверсія напрямку верифікації

Запропонований метод VERIGATE (VERIfication GATE) реалізує принципово новий напрямок взаємодії між нейронною та символьною компонентами: **нейронний → символьний** (Neural → Symbolic verification).

На відміну від існуючих архітектур, де символьні інструменти генерують виявлення, а LLM їх фільтрує або розширює, у VERIGATE:

1. LLM генерує **структуровані твердження** (claims) про потенційні дефекти у визначеному форматі.
2. Символьна система **незалежно верифікує** ці твердження через зіставлення AST-патернів, taint-аналіз та перевірку правдоподібності виправлення.
3. Приймаються лише верифіковані виявлення, або ті, у яких наявні достатні докази.

Таблиця 3.1 — Порівняння VERIGATE з існуючими нейросимвольними системами

Система	Архітектура	Напрямок	Фрагментний	Навчання	Верифікація
IRIS	LLM фільтрує CodeQL	CodeQL→LLM	Ні	Так	Ні
LLMSA	Композиційний	Багатоетапний LLM	Так	Ні	Ні
MoCQ	Ітеративне уточнення	Двонаправлений	Ні	Так	Ні
VERIGATE	**На основі шлюзу**	**Neural→Symbolic**	**Так**	**Ні**	**Так**

Така інверсія забезпечує контрольовану поведінку LLM: модель розширює покриття аналізу за межі наперед визначених правил, виявляючи семантично складні дефекти, а символьна верифікація гарантує, що кожне виявлення обґрунтоване верифікованими програмними властивостями. Галюцинації, хибні локалізації та неправдоподібні виправлення відсікаються символьною верифікацією, не потрапляючи до кінцевого результату.

3.2. Метод VERIGATE

3.2.1. Загальний огляд методу

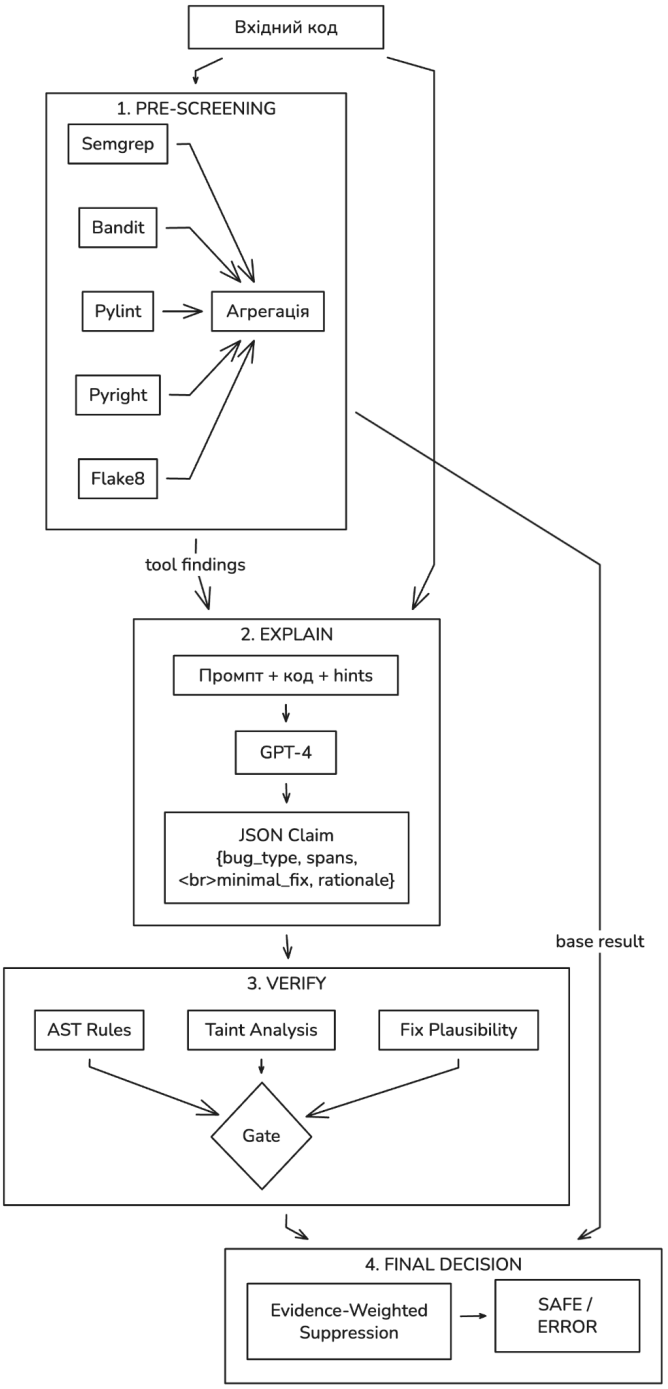
VERIGATE є нейросимвольним методом статичного аналізу коду, що синтезує можливості семантичного розуміння LLM із символьною верифікацією програм. Метод реалізує чотириетапний конвеєр обробки, спроектований для забезпечення **контрольованої поведінки LLM** через вимогу незалежної верифікації всіх тверджень про дефекти засобами символьного аналізу.

Метод працює на основі чотирьох етапів обробки:

1. **Preliminary Screening (попередній скринінг)** — агрегація доказів від кількох інструментів статичного аналізу та формування базової класифікації з показником впевненості. Цей етап надає початкові індикатори вразливостей та контекстну інформацію для подальшого LLM-аналізу.

2. **Explain (генерація структурованих тверджень)** — LLM отримує фрагмент коду та результати інструментів, після чого генерує структуроване твердження (claim), що специфікує: тип дефекту, уражені ділянки коду з номерами рядків, мінімальне виправлення та обґрунтування. Структурований формат робить твердження придатними для верифікації, на відміну від неструктурованих текстових пояснень.
3. **Verify (верифікаційний шлюз)** — структуроване твердження проходить незалежну верифікацію засобами символьного аналізу програм: зіставлення AST-патернів ідентифікує відомі патерни вразливостей, taint-аналіз валідує твердження про потоки даних, перевірка правдоподібності виправлення підтверджує, що запропоноване виправлення дійсно усуне виявлену проблему. Шлюз пропускає виключно ті твердження, що підтверджені символьними доказами.
4. **Final Decision (фінальне рішення)** — логіка зваженої оцінки доказів синтезує результати верифікації з базовим рівнем упевненості (confidence). Верифіковані твердження класифікуються як помилкові (ERROR) з підвищеним рівнем упевненості. Відхилені твердження можуть бути збережені за наявності сильних доказів від інструментів, забезпечуючи баланс між повнотою (recall) та точністю (precision).

Рисунок 3.1 — Блок-схема архітектури VERIGATE



Важливою особливістю архітектури є те, що звернення до LLM відбувається лише один раз на весь аналіз фрагмента коду. Це забезпечує передбачуваність затримки, контрольованість витрат та детермінованість символьної верифікації. Ця архітектура забезпечує контрольовану поведінку LLM: символьна верифікація слугує основним контролем якості для всіх тверджень, згенерованих LLM, тоді як шар фінального рішення, визначає, чи існує достатня кількість підтверджувальних

доказів для збереження виявлень, які контрольний механізм не може символьно підтвердити.

3.2.2. Попередній скринінг

На першому етапі виконується збір базових доказів через ансамбль інструментів статичного аналізу та бібліотеку AST-правил.

Вибір інструментів. VERIGATE інтегрує п'ять інструментів, відібраних за критеріями сумісності з аналізом на рівні фрагментів коду (не потребують повного контексту проєкту чи компіляції), безпековою спеціалізацією та наявністю у дослідницькій практиці:

- *Semgrep* [56] — промисловий стандарт аналізу на основі шаблонів, що використовується у безпекових дослідженнях та підтримує безпосередній аналіз фрагментів коду. Основний інструмент для виявлення патернів вразливостей.
- *Bandit* [51] — Python-специфічний безпековий лінтер, що спеціалізується на виявленні вразливостей у Python-коді. Комплементарний до Semgrep, оскільки фокусується на Python-специфічних паттернах (небезпечна десеріалізація, жорстко закодовані паролі, слабка криптографія).
- *Pylint* — метрики якості коду та виявлення логічних помилок. Надає допоміжні сигнали для оцінки загальної якості.
- *Pyright* — перевірка типів та виявлення невизначених змінних. Корисний для виявлення помилок типізації, не покритих безпековими інструментами.
- *Flake8* — стилістична відповідність та елементарні синтаксичні помилки.

Semgrep та Bandit є первинними (орієнтованими на безпеку) інструментами, тоді як Pylint, Pyright та Flake8 виступають допоміжними (орієнтованими на якість). Система толерантна до недоступності окремих інструментів, адаптуючи обчислення рівня впевненості до наявної бази доказів. Інструменти запускаються паралельно для мінімізації затримок.

Процес скринінгу включає наступні кроки:

- *Фільтрація.* З результатів інструментів відбираються лише сигнали високої серйозності (CRITICAL, HIGH). Низькосерйозні сигнали фільтруються для зменшення шуму.
- *Агрегація.* На основі отриманих сигналів формується базова класифікація $\in$ {SAFE, ERROR} з показником впевненості у діапазоні [0, 1]. Обчислення рівня впевненості (confidence) враховує серйозність знахідок: HIGH/CRITICAL знахідки мають більшу вагу, ніж MEDIUM/LOW; узгодженість між інструментами ($\geq$ 2 інструментів підтверджують проблему) підвищує рівень впевненості; наявність високопріоритетних AST-патернів (undefined variables, missing imports, eval/exec) дає додаткове підвищення.
- *Вихід етапу.* Два результати: (1) базова класифікація з показником впевненості та (2) структуровані сигнали інструментів, що інформують подальший етап Explain.

Важливо зазначити, що базова класифікація не є фінальною — вона слугує вхідними даними для подальших етапів та може бути змінена за результатами символьної верифікації.

3.2.3. Explain — генерація структурованих тверджень

Етап Explain вирішує фундаментальне обмеження аналізу на основі LLM: схильність до генерації нечітких або неверифікованих тверджень. VERIGATE вимагає від LLM строго структурованого виходу — JSON-об'єкту з чотирма обов'язковими полями, як продемонстровано у лістингу 3.1.

Лістинг 3.1 — структура JSON-об’єкту, яку повертає LLM на етапі Explain

```
{
  "bug_type": "Конкретна категорія вразливості",
  "spans": [start_line, end_line],
  "minimal_fix": "Найменша зміна коду, що виправляє проблему",
  "rationale": "Стисле пояснення"
}
```

Поля в JSON-об’єкті мають наступні значення:

- *bug_type* — конкретна категорія дефекту (наприклад, "Command injection via os.system", "Mutable default argument", "Undefined variable"). Ця категорія використовується для вибору відповідних правил верифікації.
- *spans* — номери рядків (1-based), де локалізується проблема. Конкретна локалізація є критично важливою для верифікації — вона дозволяє перевірити, чи дійсно у вказаних рядках присутній заявлений патерн.
- *minimal_fix* — найменша зміна коду, що виправляє проблему. Це поле реалізує принцип пояснення через конструювання (explanation-through-construction): нездатність запропонувати конкретне виправлення свідчить про недостатнє розуміння заявленої проблеми. Вимога мінімального виправлення примушує LLM продемонструвати конкретне розуміння механізму вразливості.
- *rationale* — стисле технічне пояснення.

При відсутності дефекту модель повертає порожній об'єкт {}.

Архітектура промпту передбачає: відповідь виключно у JSON-форматі без Markdown-форматування; знахідки інструментів з попереднього скринінгу надаються як контекстні підказки, що спрямовують, але не обмежують аналіз LLM; явна інструкція щодо повернення порожнього об'єкта при відсутності конкретного дефекту.

Структурований формат виходу створює формальний контракт між нейронною та символьною компонентами: кожне поле є окремим перевірюваним твердженням, а не частиною вільного тексту. Це принципово відрізняється від типового використання LLM, де вихід є неструктурованим текстом без можливості формальної верифікації.

3.2.4. Verify — верифікаційний шлюз

Етап Verify є центральною інновацією VERIGATE. Він отримує структуроване твердження з етапу Explain та валідує його через символьний

аналіз програми, що виконується повністю незалежно від LLM. Ця верифікація забезпечує контрольованість: твердження без символьного підтвердження відхиляються, гарантуючи передбачувану поведінку системи.

Аналітичні техніки верифікації

Верифікація використовує три комплементарні аналітичні техніки:

AST Pattern Matching ідентифікує структурні патерни коду, що відповідають відомим класам вразливостей. VERIGATE підтримує бібліотеку високоточних AST-шаблонів для поширених категорій дефектів:

- *Небезпечне використання API* — eval, exec, os.system з shell=True, небезпечна десеріалізація (pickle.loads, yaml.load без SafeLoader)
- *Типові помилки програмування* — мутабельні аргументи за замовчуванням (list, dict, set), порівняння ідентичності з None/True/False (== замість is), занадто широкі блоки except, помилки у кількості аргументів функцій, невизначені змінні
- *Security-sensitive патерни* — SQL injection вектори, path traversal вразливості, неекрановані template-змінні

AST-детектори реалізовані як методи, що обходять синтаксичне дерево Python-програми (ast.walk), перевіряючи кожен вузол на відповідність визначеним патернам. Кожен детектор повертає список спрацювань правил (RuleHit) з ідентифікатором правила, повідомленням та точною локацією у коді.

Легковагий Taint Analysis відстежує потоки даних від недовірених джерел (sources) до небезпечних точок (sinks):

- *Sources* (джерела недовірених даних): користувацький ввід (input(), sys.argv, os.environ), мережеві дані (request.args, request.form, request.data), зовнішні відповіді (urllib.request, requests.get), вміст файлів.
- *Sinks* (небезпечні точки): виконання команд (os.system, subprocess.call/run), виконання коду (eval, exec), десеріалізація (pickle.loads, yaml.load), файлові операції (open, os.remove), SQL-запити (cursor.execute).

Taint-tracker будує граф залежностей даних, відстежуючи поширення taint через присвоєння, строкові операції (f-strings, конкатенація, .format()) та виклики методів. Консервативні правила поширення забезпечують збереження taint через трансформації, запобігаючи хибнонегативним результатам через недостатнє відстеження. Аналіз є внутрішньопроцедурним — обмежений межами однієї функції. Taint analysis застосовується лише для типів дефектів, що передбачають потоки даних (injection, eval, exec, command, path traversal, deserialization).

Перевірка правдоподібності виправлення (Fix Plausibility) валідує, чи запропоноване виправлення дійсно усуне заявлений дефект. Перевірка включає чотири критерії:

- *Цільовий рядок (target alignment)* — виправлення повинно бути спрямоване на рядки, вказані у полі spans твердження.
- *Усунення проблеми (violation elimination)* — після застосування виправлення до копії коду повторно виконуються AST-аналіз та taint-аналіз; виправлення вважається правдоподібним, якщо кількість спрацювань правил або taint-потоків зменшується.
- *Синтаксична коректність* — виправлений код повинен бути синтаксично валідним.
- *Семантична узгодженість* — виправлення повинно бути узгодженим із заявленим типом дефекту (bug_type).

Ця перевірка є опціональною: помилка застосування виправлення (наприклад, через невалідний синтаксис) не призводить до відхилення твердження, запобігаючи хибнонегативним результатам через технічні артефакти.

Багатоетапне дерево рішень

На відміну від простої формули, верифікаційний шлюз реалізує багатоетапне дерево рішень, що забезпечує баланс між recall (прийняття правдоподібних тверджень) та precision (відхилення необґрунтованих). Шлюз повертає пару (verified: bool, reason: str), де reason — конкретна причина рішення.

Етап 1: Відсутність спрацювань правил (пермісивний шлях). Коли жодне AST-правило не спрацьовує, шлюз є пермісивним: він приймає твердження, якщо дефект є очевидним (наприклад, eval(user_input) у коді при заявленому bug_type "eval injection") або правдоподібним (код містить патерни, що відповідають заявленому типу дефекту). Відхилення відбувається лише за відсутності обох умов. Цей шлях є критично важливим для recall: коли символьні правила не покривають певний клас дефектів, шлюз довіряє семантичному розумінню LLM, розширюючи покриття аналізу за межі наперед визначених патернів.

Етап 2: Вирівнювання спанів (span alignment). При наявності спрацювань правил виконується зіставлення між локаціями спрацювань правил та локаціями (spans), вказаними у твердженні LLM. Вирівнювання перевіряється за чотирма критеріями у порядку зменшення строгості: (1) *exact line match* — правило спрацювало точно на вказаному рядку; (2) *nearby line match* — правило спрацювало у межах ±2 рядків (враховує неточність локалізації LLM); (3) *exact type match* — повідомлення правила містить ті самі ключові слова, що й заявлений bug_type; (4) *general type match* — правило та твердження належать до однієї категорії (security, logic/error, function/attribute). Результатом є набір вирівняних збігів (вирівняних збігів) — спрацювань правил, що підтверджують твердження LLM.

Етап 3: Перевірка невідповідності спанів (strict check). Якщо LLM надав конкретні номери рядків, але жодне правило не вирівнюється з цими рядками, шлюз стає строгим: відхиляє твердження, за винятком очевидних безпекових проблем (eval, exec, shell injection, syntax error). Логіка: коли LLM специфікує конкретну локацію, а правила не підтверджують цю локацію, розбіжність свідчить про можливу помилку LLM.

Етап 4: Taint analysis (для дефектів, пов'язаних з потоками даних). Для типів дефектів, що потребують taint analysis (injection, eval, exec, command, path traversal, deserialization), виконується додаткова перевірка: чи існує taint-потік від source до sink у вказаних рядках. Якщо taint-потік не знайдено та наявний лише

один aligned спрацювання правила, твердження відхиляється. Проте за наявності множинних aligned спрацювань правил (≥ 2) твердження приймається, оскільки множинні незалежні підтвердження компенсують відсутність taint-доказів.

Етап 5: Верифікація виправлення (fix plausibility). Якщо попередні етапи не дали остаточного рішення, виконується перевірка мінімального виправлення за критеріями, описаними вище: виправлення застосовується до коду, повторно виконуються AST-аналіз та taint-аналіз на виправленому коді. Зменшення кількості спрацювань правил або усунення taint-потоку підтверджує, що виправлення дійсно адресує виявлену проблему, і твердження приймається.

Етап 6: Фінальне рішення. Якщо жоден попередній етап не прийняв рішення: (1) наявність узгоджених збігів → прийняття; (2) правдоподібне твердження + будь-які спрацювання правил → прийняття; (3) відсутність доказів → відхилення.

Принципи дизайну верифікаційного шлюзу

Дерево рішень реалізує два ключові принципи:

Пермісивність при відсутності співпадіння патернів. Коли AST-патерни не покривають певний клас дефектів (відсутнє спрацювання правил), шлюз довіряє семантичному розумінню LLM, приймаючи правдоподібні твердження. Це забезпечує розширення покриття аналізу за межі наперед визначених патернів.

Строгість у разі конфлікту доказів. Коли LLM надає конкретні спани, але правила не підтверджують місця їх знаходження, шлюз відхиляє твердження (за винятком очевидних проблем). Тобто, конфлікт між нейронним та символьним компонентами інтерпретується на користь символьних доказів.

Загальна формула верифікації може бути спрощено виражена як:

$$verified = (AST\_match \vee taint\_exists) \wedge fix\_plausible \quad (5)$$

Однак фактичне дерево рішень є дещо складнішим: воно враховує наявність/відсутність спрацювань правил, якість вирівнювання спанів, тип дефекту та силу доказів, забезпечуючи адаптивну верифікацію замість жорсткої формули.

3.2.5. Фінальне рішення та Evidence-Weighted Suppression

На фінальному етапі синтезується інформація з усіх попередніх етапів. Цей етап є критичним для балансування recall (виявлення вразливостей) та precision (уникнення хибнопозитивних спрацювань).

Якщо шлюз верифікує твердження — класифікація ERROR з підвищеним рівнем впевненості: max(baseline_confidence, 0.5). Верифіковане твердження підтримане конкретними доказами символьного аналізу, що обґрунтовують підвищений рівень впевненості.

Якщо шлюз відхиляє твердження — рішення залежить від базової класифікації:

- *Базова класифікація* = ERROR: класифікація ERROR зберігається, якщо базовий показник впевненості ≥ 0.15 або наявні підтверджуючі докази (знахідки інструментів, символьні збіги шаблонів, або заявлений LLM тип дефекту відповідає структурним патернам у коді). Це дозволяє зберегти виявлення, коли попередній скринінг ідентифікував проблему, яку поточне символьне покриття верифікації не здатне підтвердити.
- *Базова класифікація* = SAFE: класифікація підвищується до ERROR лише якщо тип дефекту, заявлений LLM, відповідає патернам, наявним у коді, та базовий показник впевненості є достатнім. Цей консервативний поріг запобігає ситуації, коли спекулятивні твердження LLM перевизначають чисту базову класифікацію.

В іншому випадку класифікація залишається SAFE.

Evidence-Weighted Suppression (EWS) — враховує, що верифікаційний шлюз, попри свою ефективність, має обмежений recall. Деякі валідні дефекти можуть не відповідати наявним AST-патернам або залишатися поза межами виявлення через використання легковагового taint-аналізу. Проте, якщо проблему ідентифікують кілька незалежних інструментів або базовий рівень впевненості є достатньо високим, їх фільтрація може призвести до втрати коректних виявлень (true positives). Тому, у випадках, коли верифікаційний шлюз відхиляє твердження

через відсутність символьних підтверджень, механізм EWS все ще може зберігати їх за наявності достатніх доказів.

3.3. Механізм Explain-Verify Gate

3.3.1. Концептуальна модель

Механізм Explain-Verify Gate є центральним архітектурним елементом VERIGATE, що реалізує контрольовану взаємодію між нейронним та символьним компонентами.

Фаза Explain примушує LLM формулювати конкретні, перевірювані твердження замість вільнотекстових відповідей. Структурований формат виходу (JSON) створює формальний контракт: кожне поле є окремим твердженням, яке може бути незалежно перевірене. Вимога мінімального виправлення (minimal_fix) реалізує принцип пояснення через конструювання — нездатність запропонувати конкретне виправлення є формою доказу недостатнього розуміння проблеми.

Фаза Verify забезпечує незалежну валідацію. Критично важливо, що верифікація виконується без повторного звернення до LLM — це гарантує детермінованість результату та виключає можливість "самопідтвердження" галюцинацій моделлю.

Разом Explain та Verify створюють архітектурний патерн, що обмежує неконтрольовану поведінку LLM: переважна більшість виявлень потребує формального підтвердження в структурі коду, або збережена шаром фінального рішення (EWS) за наявності достатніх доказів від інших джерел.

3.3.2. Формалізація процесу верифікації

Нехай C — вхідний код, T — структуроване твердження LLM (claim), V — набір правил верифікації, $R(C)$ — множина спрацювань правил для коду C.

Твердження $T = (bug\_type, spans, minimal\_fix, rationale)$ верифікується за адаптивною логікою, що враховує наявність або відсутність символьних доказів:

Випадок 1: Наявні спрацювання правил *(R(C) ≠ ∅)*. Твердження вважається верифікованим, якщо виконуються умови:

1. **Rule Alignment:** $\exists r \in R(C) : aligned(r, spans, bug\_type) = true$ — існує спрацювання правила, що вирівнюється з вказаними рядками та типом дефекту.
2. **Location Validity:** $spans \subseteq valid\_lines(C)$ — вказані рядки існують у коді.
3. **Taint Confirmation** (для дефектів потоків даних): $taint\_flow(C, spans) = true \lor |aligned\_hits| \geq 2$ — підтверджено taint-потік або наявні множинні незалежні підтвердження.
4. **Fix Plausibility** (опціонально): $violations(apply(minimal\_fix, C)) < violations(C)$ — застосування виправлення зменшує кількість виявлених проблем.

Випадок 2: Відсутні спрацювання правил *(R(C) = ∅)*. Шлюз є пермісивним: твердження приймається, якщо дефект є очевидним (*obvious_defect(C, bug_type) = true*) або правдоподібним (*plausible(C, bug_type) = true*). Це забезпечує розширення покриття аналізу за межі наперед визначених патернів, довіряючи семантичному розумінню LLM.

Випадок 3: Конфлікт доказів. Якщо LLM надав конкретні spans, але жодне спрацювання правил не співвідноситься з цими рядками, шлюз діє суворо: твердження відхиляється, за винятком очевидних безпекових проблем.

Таким чином, верифікація реалізує адаптивну логіку замість жорсткої кон'юнкції умов, що відповідає багатоетапному дереву рішень, описаному у розділі 3.2.4.

3.3.3. Критерії верифікації

Відповідність патернам перевіряється через механізми з різним рівнем строгості:

- *Exact line match* — AST-патерн знайдено точно у вказаному рядку spans[i]
- *Nearby lines* — патерн знайдено у межах ±2 рядків від вказаного (враховує неточність локалізації LLM)

- *Semantic pattern match* — семантична відповідність типу дефекту, навіть якщо точний рядок не збігається

Evidence Strength оцінюється за критеріями: кількість незалежних правил, що підтверджують проблему (множинні спрацювання правил); серйозність знахідок інструментів (знахідки високого пріоритету); узгодженість результатів між різними інструментами (узгодженість інструментів).

3.4. Відмінності від існуючих методів

3.4.1. Порівняння архітектурних рішень

Порівняння VERIGATE з існуючими методами за ключовими архітектурними характеристиками:

Напрямок взаємодії. IRIS реалізує символьний → нейронний (CodeQL → LLM), LLMSA — багатоетапний LLM із символьною структуризацією, MoCQ — двонаправлений (bidirectional), VERIGATE — нейронний → символьний (LLM → Symbolic verification). VERIGATE є першим методом, що реалізує цей напрямок у контексті статичного аналізу.

Роль LLM. В IRIS — фільтрація та розширення результатів CodeQL. В LLMSA — виконання аналізу потоків даних. В MoCQ — видобуток та уточнення патернів. В VERIGATE — генерація структурованих гіпотез (claims) про дефекти.

Роль символьної системи. В IRIS — первинне виявлення. В LLMSA — структуризація аналізу. В MoCQ — уточнення патернів. В VERIGATE — незалежна верифікація тверджень LLM.

Контроль поведінки LLM через незалежну верифікацію. Більшість існуючих систем (IRIS, LLMSA, MoCQ) не реалізують явного механізму незалежної верифікації тверджень LLM. IRIS та MoCQ покладаються на внутрішню узгодженість або евристичні механізми без символьного підтвердження, тоді як LLMSA забезпечує лише частковий контроль через структуру аналізу. У цих підходах коректність виявлень переважно залежить від якості міркувань моделі. Натомість VERIGATE забезпечує контроль поведінки

LLM шляхом незалежної символьної перевірки: механізм Explain-Verify Gate виконує роль основного контролю якості, підтверджуючи твердження через формальні перевірки, а механізм Evidence-Weighted Suppression (EWS) додатково враховує незалежні докази з інструментів та патернів. Таким чином, поведінка LLM не лише спрямовується структурою запиту, а й обмежується зовнішньою верифікацією, що знижує ризик спекулятивних або непідтверджених тверджень.

Пояснюваність. VERIGATE забезпечує найвищу пояснюваність: кожне рішення супроводжується конкретними спрацюваннями правил (знайдені AST-патерни, taint-потоки, результат перевірки виправлення), що дозволяє аудит та систематичне покращення.

3.4.2. Унікальні властивості VERIGATE

Інверсія напрямку верифікації. VERIGATE реалізує перехід від традиційної парадигми символьний → нейронний до нейронний → символьний, де символьний аналіз незалежно верифікує твердження LLM. Це забезпечує контрольованість через обґрунтування всіх виявлень у верифікованих програмних властивостях (AST-патерни, taint-потоки, правдоподібність виправлення), а не у статистичних патернах.

Структуровані твердження (Structured Claims). Формат виходу LLM у вигляді JSON із конкретними полями (bug_type, spans, minimal_fix, rationale) створює формальний контракт між компонентами. Кожне поле є перевірюваним твердженням, що робить вихід LLM придатним для формальної верифікації. Вимога minimal_fix реалізує пояснення через конструювання — додатковий сигнал якості розуміння проблеми.

Незалежна верифікація. Символьна верифікація виконується без участі LLM, що гарантує детермінованість та виключає можливість "самопідтвердження" галюцинацій.

Evidence-Weighted Suppression. Механізм адаптивного зважування доказів із кількох незалежних джерел дозволяє балансувати precision та recall залежно від

сили наявних підтверджень, зберігаючи виявлення навіть при відхиленні шлюзом за наявності сильних доказів.

Відсутність потреби у навчанні. VERIGATE не потребує дотренування моделей або специфічних датасетів. Розширення функціональності досягається додаванням нових правил верифікації та taint-специфікацій для нових мов програмування та типів дефектів.

3.5. Деталі реалізації

3.5.1. Технологічний стек та архітектура реалізації

Реалізація VERIGATE виконана мовою Python з використанням наступних компонентів:

- *LLM інтеграція:* OpenAI API для генерації структурованих тверджень
- *AST-аналіз:* стандартний модуль ast для Python
- *Інструменти статичного аналізу:* Semgrep, Bandit, Pylint, Pyright, Flake8
- *Оркестрація:* asyncio для паралельного запуску інструментів

Архітектура реалізації побудована на двох основних класах: HybridRuleToolAnalyzer (базовий аналізатор, що об'єднує інструменти та AST-правила для етапу скринінгу) та ExplainVerifyGate (верифікаційний шлюз, що реалізує етапи Explain та Verify). Клас NeuroSymbolicAnalyzer оркеструє взаємодію цих компонентів через чотириетапну обробку. Шлюз використовує окрему інстанцію AST-правил для верифікації, незалежну від базового аналізатора, що забезпечує справжню незалежність символьної перевірки від попередніх етапів.

Характеристики продуктивності. Час аналізу одного фрагмента коду складає 5–15 секунд: 2–5 секунд на базовий аналізатор (інструменти + AST), 3–10 секунд на виклик LLM API (основне вузьке місце продуктивності), < 0.1 секунди на символьну верифікацію.

3.5.2. Формат промпту для LLM

Промпт для етапу Explain сконструйовано з урахуванням наступних принципів:

- *Чітка структура виходу* — явна вимога JSON-формату з визначеними полями (bug_type, spans, minimal_fix, rationale), що створює формальний контракт.
- *Контекст інструментів* — результати попереднього скринінгу передаються як підказки (tool hints), що спрямовують, але не обмежують аналіз LLM. Це дозволяє LLM використовувати сигнали інструментів як відправну точку, зберігаючи здатність виявляти проблеми, не покриті інструментами.
- *Перелік цільових категорій* — промпт специфікує категорії дефектів для пошуку: undefined variables, missing imports, syntax errors, security vulnerabilities, logic bugs, suspicious patterns.
- *Обмеження відповіді* — заборона вільнотекстових пояснень поза JSON-структурою. Вимога повертати порожній об'єкт {} при відсутності конкретного дефекту.

Промпт містить інструкцію для аналізу коду на наявність дефектів та вразливостей, передає код та результати попереднього скринінгу (tool findings), та вимагає відповідь виключно у форматі JSON з полями bug_type, spans, minimal_fix та rationale. У випадку відсутності дефектів модель повертає порожній об'єкт {}.

3.5.3. Обробка виходу LLM

Вихід LLM проходить багаторівневу обробку:

1. **JSON-парсинг** — видалення Markdown-форматування та подальший синтаксичний розбір JSON-структури.
2. **Валідація схеми** — перевірка відповідності структури даних визначеній схемі, зокрема, наявності обов’язкових полів.
3. **Нормалізація** — приведення spans до 1-based індексації та нормалізація значень поля bug_type.

4. **Fallback** — у разі помилки парсингу система повертається до базового результату скринінгу.

Алгоритм обробки спочатку очищує відповідь від можливого Markdown-форматування, потім виконує парсинг JSON. У разі успішного парсингу перевіряється наявність усіх обов'язкових полів. Якщо валідація пройдена, створюється об'єкт Claim з нормалізованими значеннями. У разі будь-якої помилки повертається None, що сигналізує про необхідність використання логіки Fallback.

3.5.4. Наскрізний приклад роботи VERIGATE

Для ілюстрації роботи запропонованого методу розглянемо реальний приклад із експериментальної оцінки на вибірці 1 000 фрагментів Python-коду (PySStuBs). Цей випадок демонструє нейросимвольну взаємодію: VERIGATE коректно ідентифікує семантичний логічний дефект. Приклад вхідного фрагменту коду продемонстровано у лістингу 3.2.

Лістинг 3.2 — Приклад вхідного фрагменту коду з вибірки PySStuBs

```
if len(callsigns) != len(callsigns.values()):
```

Ground truth (еталонна розмітка): дефектний код (buggy) з PySStuBs.

Суть дефекту: Для стандартного словника *dict* вираз *len(d)* завжди дорівнює *len(d.values())*, оскільки кількість ключів дорівнює кількості значень. Умова ніколи не виконається — це логічна помилка. Якщо намір програміста — перевірити наявність дублікатів серед значень, слід використати *len(set(callsigns.values()))*.

Етап 1: Попередній скринінг (Preliminary Screening)

Система запускає ансамбль інструментів статичного аналізу (Bandit, Semgrep, Pylint, Pyright, Flake8) та AST-правила:

Знахідки інструментів продемонстровано у лістингу 3.3. Знахідки AST-правил для цього фрагменту відсутні.

Лістинг 3.3 — Результати попереднього скринінгу

```
   [PYRIGHT] error: "callsigns" is not defined
(reportUndefinedVariable)
   [PYRIGHT] error: Expected indented block
```

Базовий результат: ERROR (скринінг виявив потенційну проблему), base confidence = 0.15.

Інтерпретація: Інструменти виявили лише синтаксичні проблеми — невизначену змінну та неповний блок if. Вони не аналізують семантику умови порівняння та не вказують на логічний дефект. Виявлення логічної помилки потребує розуміння поведінки dict.values() — задача, яка виходить за межі можливостей інструментів на основі правил.

Етап 2: Explain — генерація структурованого твердження

LLM отримує код разом із результатами скринінгу та формулює структуроване твердження (claim) у форматі JSON. Структуроване твердження, яке є результатом роботи LLM продемонстровано у лістингу 3.4

Лістинг 3.4 — Структуроване ствердження, згенероване LLM

```
   {
     "bug_type": "Undefined variable / Syntax error (incomplete if) /
                 Potential logic bug: len(dict) == len(dict.values())
                 always",
     "spans": [
       {"start_line": 1, "start_col": 1, "end_line": 1, "end_col": 52}
     ],
     "minimal_fix": "callsigns: dict = ...\nif len(callsigns) !=
                    len(set(callsigns.values())):\n    ...",
     "rationale": "`callsigns` is referenced but not defined.
                  Additionally, for dicts, `len(callsigns.values())` is
                  always equal to `len(callsigns)`, so the condition
will
                  never detect duplicates unless wrapped in `set()`."
   }
```

Інтерпретація: LLM виходить за межі синтаксичних попереджень скринінгу та явно описує логічну помилку: порівняння *len(d) != len(d.values())* ніколи не спрацює для стандартного словника, а для пошуку дублікатів потрібно використати *set()*. Це демонструє семантичне розуміння, недоступне інструментам

на основі правил. Поле *minimal_fix* пропонує конкретне виправлення, що є додатковим сигналом якості розуміння проблеми.

Етап 3: Verify — символьна верифікація

Верифікаційний шлюз незалежно перевіряє кожне поле твердження:

- Rule alignment: для цього фрагменту AST-правила не спрацювали → активується пермісивний шлях (Етап 1 дерева рішень).
- Plausibility: claim узгоджується з кодом — виклики *len(...)* та *.values()* присутні у фрагменті, тип *dict* є правдоподібним.
- Fix plausibility: запропоноване виправлення *(set(callsigns.values()))* є коректним та адресує виявлену логічну проблему.

Результат верифікації: verified = true, reason = accepted_no_rules (прийнято через пермісивний шлях за відсутності спрацювань правил, оскільки claim визнано правдоподібним).

Етап 4: Фінальне рішення

Оскільки claim пройшов символьну верифікацію:

- *Класифікація:* ERROR (дефект підтверджено)
- *Confidence:* 0.4 (підвищено від базового 0.15)
- *Пояснення:* Verified by gate: Possible logic bug — incorrect use of callsigns.values(). For a dictionary, len(callsigns) equals len(callsigns.values()); to check for duplicate values, use len(set(callsigns.values())).

Підсумок

Підсумок роботи всіх етапів методу VERIGATE на прикладі фрагменту коду зображено в таблиці 3.2.

Таблиця 3.2 — Підсумкова таблиця роботи кожного з етапів методу VERIGATE

Етап	Вхід	Вихід
Screening	Python-фрагмент	Tool findings (Pyright: undefined var, incomplete if); base ERROR, conf. 0.15
Explain	Код + findings → LLM	Структуроване claim: логічна помилка len(d)==len(d.values())
Verify	Код + claim → Gate	verified = true (пермісивний шлях, claim правдоподібний)
Final Decision	Всі етапи	ERROR, confidence 0.4, пояснення

Цей приклад ілюструє переваги нейросимвольного підходу VERIGATE:

1. Інструменти (символьний компонент) фіксують поверхневі синтаксичні проблеми, але не здатні виявити семантичну помилку.
2. LLM (нейронний компонент) додає глибоке семантичне розуміння — розпізнає, що *len(dict) == len(dict.values())* завжди є тотожністю, та пропонує конкретне виправлення.
3. Верифікаційний шлюз (символьний контроль) незалежно підтверджує правдоподібність твердження LLM, забезпечуючи контрольованість рішення.

Результат: VERIGATE коректно класифікує фрагмент як дефектний, демонструючи здатність виявляти семантичні дефекти, які залишаються недосяжними для підходів, що ґрунтуються виключно на правилах, водночас зберігаючи можливість верифікації прийнятого рішення.

3.6. Висновки до розділу

У розділі представлено детальний опис запропонованого нейросимвольного методу VERIGATE для статичного аналізу програмного коду. Основні результати:

1. **Обґрунтовано концептуальні засади нейросимвольного методу**, що поєднує можливості великих мовних моделей із механізмами формальної верифікації для забезпечення контрольованої поведінки LLM.

2. **Проаналізовано існуючі нейросимвольні системи** (IRIS, LLMSA, MoCQ) та визначено їх спільне обмеження — відсутність явного механізму незалежної верифікації тверджень LLM.
3. **Запропоновано інверсію напрямку верифікації** (від нейронного до символьного компонента), що принципово відрізняється від існуючих методів і дозволяє розширити покриття аналізу без втрати контролю над точністю.
4. **Розроблено архітектуру методу VERIGATE** з чотирма послідовними етапами: попередній скринінг, Explain (генерація структурованих тверджень), Verify (символьна верифікація), фінальне рішення.
5. **Продемонстровано відмінності від існуючих методів**: інверсія напрямку верифікації, структуровані твердження, незалежна верифікація, архітектура без потреби у навчанні та аналіз на рівні фрагментів.

Запропонований метод забезпечує поєднання розширеного покриття аналізу (за рахунок семантичних можливостей LLM) з контрольованістю (за рахунок символьної верифікації), що дозволяє отримати передбачувану та надійну поведінку.

4. ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА

4.1. Методологія експериментальної оцінки

4.1.1. Питання дослідження та гіпотези

Експериментальна оцінка методу VERIGATE організована навколо трьох питань дослідження, кожне з яких формалізоване у вигляді гіпотези, що перевіряється.

RQ1: Як VERIGATE порівнюється з чистим LLM-підходом при виявленні реальних дефектів коду?

Гіпотеза H1: Нейросимвольний метод VERIGATE з механізмом Explain-Verify Gate перевершує базовий LLM-підхід (GPT-4 без символьної верифікації) за метриками F1-score та recall при збереженні precision.

RQ2: Чи зберігається ефективність VERIGATE у контексті новіших, більш просунутих мовних моделей?

Гіпотеза H2: VERIGATE забезпечує стабільне покращення результатів незалежно від покоління мовної моделі, зберігаючи покращення.

RQ3: Який внесок кожного архітектурного компоненту (попередній скринінг, верифікаційний шлюз, Evidence-Weighted Suppression) у загальну ефективність?

Гіпотеза H3: Кожен компонент VERIGATE робить незалежний внесок у загальну ефективність, а повна конфігурація забезпечує оптимальний баланс між ефективністю виявлення та інтерпретованістю рішень.

4.1.2. Метрики оцінки

Для оцінки ефективності методів використано стандартні метрики бінарної класифікації:

Precision (точність) — частка справжніх дефектів серед усіх виявлених: $P = \frac{TP}{TP+FP}$. У контексті статичного аналізу високий precision означає мінімальну кількість хибнопозитивних спрацювань (false positives), що зменшує навантаження на розробників під час тріажу результатів аналізу.

Recall (повнота) — частка виявлених дефектів серед усіх наявних: $R = \frac{TP}{TP+FN}$. Високий recall означає, що метод виявляє максимальну кількість реальних дефектів, мінімізуючи пропущені вразливості (false negatives).

F1-score — гармонійне середнє precision та recall: $F_1 = \frac{2 \times P \times R}{P+R}$. F1-score є основною метрикою порівняння, оскільки враховує обидва аспекти ефективності одночасно.

Accuracy (точність класифікації) — частка правильних класифікацій серед усіх зразків: $A = \frac{TP+TN}{TP+TN+FP+FN}$.

У контексті статичного аналізу особливу увагу приділено балансу між precision та recall. Для виробничого середовища критичним є precision, оскільки хибнопозитивні спрацювання підривають довіру розробників до інструменту. Водночас пропущені вразливості (низький recall) несуть безпосередні ризики безпеки. Ідеальний інструмент забезпечує високий recall без втрати precision.

4.1.3. Методи для порівняння

В експериментах порівнюються три категорії методів:

- *VERIGATE* — запропонований нейросимвольний метод з наступними етапами: Preliminary Screening → Explain → Verify → Evidence-Weighted Suppression. Метод вимагає від LLM генерації структурованих тверджень (claims) про дефекти, які потім незалежно верифікуються засобами символьного аналізу програм.
- *LLM Baseline* — чисте використання мовної моделі для класифікації коду на SAFE/ERROR. Модель отримує фрагмент коду та результати інструментів статичного аналізу, надає бінарну класифікацію з обґрунтуванням. Відповідь моделі не проходить через механізм Explain-Verify Gate і не підлягає символьній верифікації. Це порівняння дозволяє оцінити внесок саме нейросимвольної архітектури порівняно з некерованим LLM-підходом.

- *Rule-based* — ансамбль традиційних інструментів (Semgrep, Bandit, Pylint, Pyright, Flake8) без LLM-компоненти. Класифікація ERROR, якщо агрегований коефіцієнт рівня впевненості від інструментів перевищує поріг 0.2; SAFE — в іншому випадку. Коефіцієнт рівня впевненості обчислюється на основі серйозності знахідок та узгодженості між інструментами (аналогічно етапу попереднього скринінгу VERIGATE). Це порівняння дозволяє зрозуміти можливості стандартних інструментів статичного аналізу на обраній вибірці.

Порівняння з *LLM Baseline* дозволяє оцінити внесок механізму Explain-Verify Gate та символьної верифікації. Порівняння з методами на основі правил демонструє переваги нейросимвольної інтеграції над традиційними підходами на семантично складних задачах.

4.1.4. Деталі реалізації

Експерименти проведено з використанням наступної конфігурації:

- Середовище виконання: Python 3.11
- LLM для основної оцінки (RQ1): OpenAI GPT-4
- LLM для валідації (RQ2, RQ3): OpenAI GPT-5.2
- Інструменти статичного аналізу: Semgrep 1.x, Bandit 1.7.x, Pylint 3.x, Pyright 1.x, Flake8 6.x
- Параметри LLM: температура 0.1 (низька для забезпечення консистентності), максимальна кількість токенів 1000

4.2. Датасет та стратегія вибірки

4.2.1. PySStuBs (Python Single-Statement Bugs)

Для всіх експериментів використано датасет PySStuBs (Python Single-Statement Bugs) [32] — колекцію реальних однорядкових помилок, витягнутих із історії комітів відкритих Python-проєктів на GitHub.

Характеристики датасету. PySStuBs містить 73 013 пар buggy/fixed коду з автентичних Python-проєктів. Кожен зразок представляє пару: версія з дефектом (buggy) та виправлена версія (fixed). Датасет охоплює різноманітні типи дефектів, включаючи вразливості, класифіковані за CWE (Common Weakness Enumeration), логічні помилки та типові помилки програмування.

Стратегія вибірки. Для основної оцінки (RQ1) використано стратифіковану випадкову вибірку розміром 1 000 зразків (500 buggy, 500 fixed), що забезпечує різноманітне покриття типів дефектів при збереженні балансу класів. Код із дефектом отримує мітку 1 (ERROR), виправлений код — мітку 0 (SAFE). Збалансований дизайн усуває потенційне зміщення метрик через дисбаланс класів. Для валідації з GPT-5.2 (RQ2) та абляційного дослідження (RQ3) використано окрему вибірку з 500 зразків (250 buggy, 250 fixed).

Обґрунтування вибору. PySStuBs є особливо цінним датасетом для оцінки VERIGATE з кількох причин. По-перше, дефекти у цьому датасеті є семантично складними — вони включають неправильні імена змінних, некоректні оператори, помилки в аргументах функцій, логічні помилки та інші категорії, що потребують глибокого розуміння контексту програми. Традиційні інструменти на основі правил (Semgrep, Bandit) демонструють дуже низьку ефективність ($F1 \approx 0.03$, див. табл. 4.1) на цьому датасеті, оскільки синтаксичні патерни не здатні виявити семантичні дефекти. По-друге, датасет містить реальні дефекти з реальних проєктів, а не штучно створені приклади, що підвищує зовнішню валідність результатів. По-третє, наявність парних buggy/fixed зразків забезпечує однозначну еталонну розмітку для оцінки.

4.3. Основна оцінка: ефективність VERIGATE (RQ1)

Основна оцінка проведена на 1 000 збалансованих зразків із PySStuBs з використанням GPT-4. Порівнюються три методи: *VERIGATE*, *GPT-4 Baseline* та *Rule-based* статичний аналіз.

4.3.1. Результати

Таблиця 4.1 — Порівняння ефективності на датасеті PySStuBs (n = 1 000)

Метод	Precision	Recall	F1	Accuracy	Виявлення
VERIGATE	**0.514**	**0.758**	**0.612**	**0.520**	**379**
GPT-4 Baseline	0.506	0.264	0.347	0.503	132
Rule-Based	0.381	0.016	0.031	0.495	8
Абсолютне покращення	+0.008	+0.494	+0.265	+0.017	+247
Відносний приріст	+1.6%	+187%	+76.4%	+3.4%	+187%

4.3.2. Аналіз результатів

Візуальне порівняння ефективності трьох методів наведено на рис. 4.1.

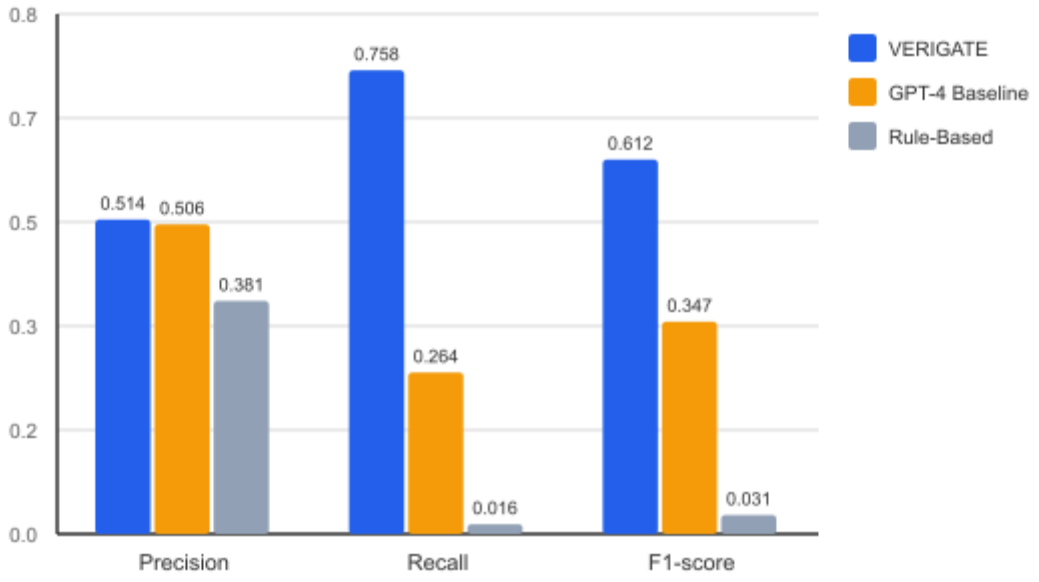


Рисунок 4.1 — Порівняння Precision, Recall та F1-score для VERIGATE, GPT-4 Baseline та Rule-Based (PySStuBs, n = 1 000)

Суттєве покращення recall та F1. VERIGATE виявляє 379 із 500 наявних дефектів (recall = 0.758), порівняно з 132 для GPT-4 Baseline (recall = 0.264). Це відповідає 247 додатково виявленим вразливостям — відносне покращення recall на 187%. F1-score покращується на 76.4% (0.612 проти 0.347). Обидва методи мають порівняний precision (0.514 проти 0.506); верифікаційний шлюз та EWS

дозволяють VERIGATE значно збільшити кількість істинно позитивних результатів при збереженні подібної точності до GPT-4 Baseline.

Порівняний precision. VERIGATE та GPT-4 Baseline досягають близького precision (~0.51), що підтверджує, що верифікаційний шлюз та багатоджерельна агрегація не призводять до надмірного зростання хибнопозитивних спрацювань при суттєвому підвищенні recall.

Нездатність методів на основі правил. Інструменти на основі правил (Semgrep, Bandit, Pylint, Pyright, Flake8) демонструють дуже низьку ефективність на PySStuBs: F1 = 0.031, recall = 0.016, виявляючи лише 8 із 500 реальних дефектів. Це підтверджує, що дефекти у цьому датасеті є семантично складними та виходять за межі можливостей синтаксичних патернів. Для виявлення таких дефектів необхідне семантичне розуміння логіки програми, що забезпечується LLM-компонентом VERIGATE.

4.3.3. Відповідь на RQ1

Чи перевершує VERIGATE чистий LLM-підхід при виявленні реальних дефектів коду?

Так. VERIGATE суттєво перевершує GPT-4 Baseline: F1-score 0.612 проти 0.347 (+76.4%), recall 0.758 проти 0.264 (+187%), при порівняному precision (0.514 проти 0.506). Метод виявляє на 247 дефектів більше (379 проти 132). Гіпотеза H1 підтверджена.

4.4. Валідація з сучасною мовною моделлю (RQ2)

Для оцінки стійкості ефективності VERIGATE при розвитку мовних моделей проведено валідацію з GPT-5.2 — моделлю наступного покоління — на 500 зразках із PySStuBs.

4.4.1. Результати

Таблиця 4.2 — Валідація з GPT-5.2 (n = 500)

Метод	Precision	Recall	F1	Виявлення
VERIGATE	0.500	**0.944**	**0.654**	**236**
GPT-5.2 Baseline	0.455	0.204	0.282	51
Абсолютне покращення	+0.045	+0.740	+0.372	+185
Відносний приріст	+9.8%	+362.7%	+132.0%	+362.7%

4.4.2. Аналіз результатів

Візуальне порівняння результатів VERIGATE та GPT-5.2 Baseline наведено на рис. 4.2.

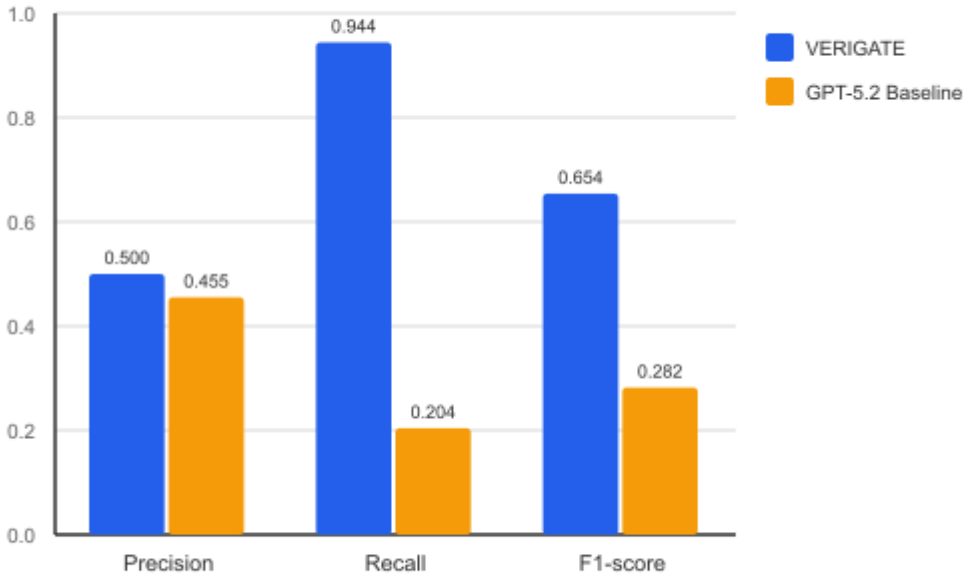


Рисунок 4.2 — Порівняння Precision, Recall та F1-score для VERIGATE та GPT-5.2 Baseline (n = 500)

Незважаючи на використання сучаснішої моделі (GPT-5.2), VERIGATE демонструє суттєві покращення порівняно з GPT-5.2 Baseline: +132% F1-score, +363% recall. GPT-5.2 Baseline виявляє лише 51 із 250 дефектів (recall = 0.204), пропускаючи 199 вразливостей. VERIGATE, натомість, виявляє 236 дефектів (recall = 0.944), що відповідає 185 додатково виявленим вразливостям.

Низька повнота (recall) у GPT-5.2 Baseline в цьому контексті відображає вплив конкретної стратегії промптингу та дизайну задачі: пряма бінарна класифікація (SAFE/ERROR) призвела до суттєвого заниження кількості виявлень. Структурована поетапна послідовність компонентів VERIGATE — з генерацією тверджень, їх символьною верифікацією та зваженою агрегацією доказів — виявляється значно стійкішим до варіацій у промптингу, ніж підхід з некерованою бінарною класифікацією.

Інтерпретація різниці з RQ1. Абсолютні метрики між оцінками з GPT-4 та GPT-5.2 відрізняються, що пояснюється методологічними варіаціями: різні стратегії промптингу, різний розмір вибірки (1 000 vs 500), різні характеристики моделей. Ключовим результатом є стабільна тенденція до покращення: VERIGATE суттєво перевершує базове рішення на основі лише LLM незалежно від покоління моделі та конфігурації експерименту.

4.4.3. Відповідь на RQ2

Чи зберігається ефективність VERIGATE з новішими, більш здібними мовними моделями?

Так. VERIGATE демонструє стабільне покращення і з GPT-4 (RQ1), і з GPT-5.2 (RQ2) порівняно з відповідними базовими рішеннями. Це підтверджує, що нейросимвольна верифікація є стійким архітектурним принципом, який зберігає цінність при розвитку мовних моделей: структурований прохід етапів з символьною верифікацією покращує результати незалежно від базової LLM. Гіпотеза H2 підтверджена.

4.5. Абляційне дослідження: внесок компонентів (RQ3)

Після встановлення ефективності VERIGATE з GPT-5.2 (RQ2) проведено систематичне абляційне дослідження для визначення внеску кожного архітектурного компоненту. Дослідження виконано на тих самих 500 зразках із PySStuBs з GPT-5.2. Послідовно видалялися три ключові компоненти: попередній

скринінг (інструменти та AST-патерни), верифікаційний шлюз (символьна валідація) та Evidence-Weighted Suppression (синтез доказів із кількох джерел).

4.5.1. Результати

Таблиця 4.3 — Результати абляційного дослідження (GPT-5.2, PySStuBs, n = 500)

Конфігурація	**F1**	**Recall**	**Precision**	**TP**	**FN**	**FP**
Full VERIGATE	0.656	0.948	0.501	237	13	236
No Screening	0.654	0.916	0.509	229	21	221
No Verify	0.666	**0.960**	0.510	**240**	**10**	231
No EWS	**0.668**	0.940	**0.518**	235	15	**219**

4.5.2. Аналіз результатів

Результати абляційного дослідження візуалізовано на рис. 4.3.

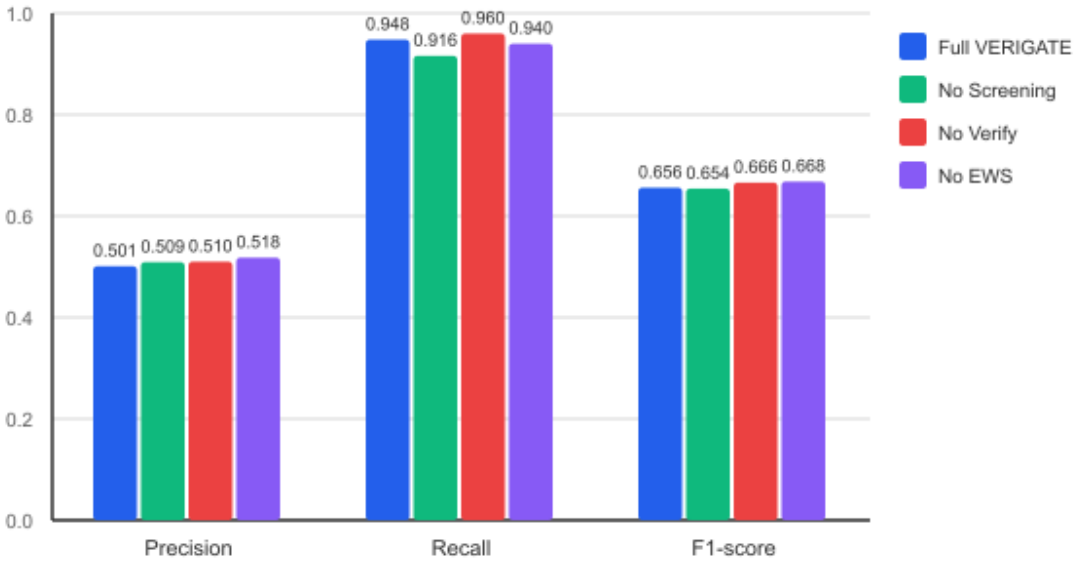


Рисунок 4.3 — Результати абляційного дослідження: вплив видалення компонентів на Precision, Recall та F1-score (GPT-5.2, n = 500)

Абляційне дослідження демонструє, що VERIGATE функціонує як інтегрований потапний процес, де компоненти працюють у поєднанні. На 500 зразках усі конфігурації досягають F1-score між 0.654 та 0.668, що свідчить про

збалансований архітектурний дизайн, а не залежність від одного домінуючого компоненту.

Внесок попереднього скринінгу. Видалення скринінгу (*No Screening*) стабільно знижує recall (0.916 проти 0.948 для *Full VERIGATE*) та збільшує кількість хибнонегативних результатів (21 проти 13). Це підтверджує, що попередній скринінг ефективно збирає докази для подальшого аналізу LLM, надаючи контекстну інформацію, яка покращує якість генерованих тверджень.

Внесок Evidence-Weighted Suppression. Видалення EWS (*No EWS*) знижує recall (0.940 проти 0.948), що демонструє: синтез доказів із кількох джерел успішно відновлює істинно позитивні результати, відхилені на етапі верифікації. EWS є критичним для збереження виявлень, коли шлюз відхиляє твердження LLM, але наявні достатні докази з інших джерел (інструменти статичного аналізу, AST-патерни).

Верифікаційний шлюз: баланс ефективності та контрольованості. Конфігурація *No Verify* досягає дещо вищих F1 (0.666) та recall (0.960), ніж *Full VERIGATE* (0.656, 0.948). Цей результат потребує детального аналізу, оскільки він торкається центральної архітектурної тези роботи.

По-перше, різниця у F1 між *Full VERIGATE* та *No Verify* становить лише 1.5% (0.656 проти 0.666) — усі конфігурації знаходяться у вузькому діапазоні F1 0.654–0.668, що свідчить про збалансований архітектурний дизайн, а не залежність від одного домінуючого компоненту.

По-друге, за агрегованими метриками приховується важливий деталізований результат. Аналіз на 300 зразках (де recall *Full VERIGATE* вищий — 0.953 проти 0.940) показує: *Full VERIGATE* виявляє 7 вразливостей, які *No Verify* пропускає, тоді як *No Verify* виявляє 5 вразливостей, які *Full* пропускає — чисте покращення становить +2 додаткові виявлені вразливості для *Full VERIGATE*. Механізм EWS успішно відновлює 7 тверджень, відхилених верифікаційним шлюзом, при цьому шлюз помилково відхиляє лише 5 валідних тверджень. У контексті безпеки програмного забезпечення, де пропущені вразливості (false negatives) є значно

критичнішими за хибнопозитивні спрацювання (false positives), цей чистий виграш у recall є суттєвим.

По-третє, видалення верифікаційного шлюзу означає втрату контрольованості: без символьної верифікації кожне рішення базується виключно на непрозорому виході LLM, без можливості аудиту чи верифікації. Шлюз забезпечує трасованість рішень — кожне виявлення супроводжується конкретною причиною верифікації (наприклад, verified_with_rules, accepted_no_rules, verified_plausible_with_evidence) та символьними доказами (знайдені AST-патерни, виявлені taint-потоки, результати перевірки виправлень). Ця інтерпретованість є критичною для виробничого середовища: відповідність регуляторним вимогам, можливість аудиту, швидший тріаж хибнопозитивні спрацювання розробниками, та систематичне покращення через розширення бібліотеки верифікаційних правил.

По-четверте, верифікаційний шлюз забезпечує модельну незалежність: символьна верифікація додає стабільність до рішень, що є особливо цінним при використанні менш потужних моделей або при зміні версій LLM. Детермінованість символьної верифікації гарантує, що один і той самий код завжди проходить однакову перевірку, незалежно від стохастичних варіацій LLM.

Таким чином, верифікаційний шлюз є архітектурним рішенням для контрольованості та покращуваності, а не лише для оптимізації одиничної метрики. Мінімальна різниця в ефективності (1.5% F1) є прийнятною платою за якісні переваги, які не вимірюються стандартними метриками classification.

4.5.3. Відповідь на RQ3

Який внесок кожного архітектурного компоненту?

Кожен компонент робить вимірюваний внесок у загальну ефективність VERIGATE, при цьому жоден компонент не є надлишковим. Попередній скринінг покращує recall на 3.2 процентні пункти (0,948 проти 0,916), забезпечуючи контекстну інформацію (tool hints), що суттєво підвищує якість тверджень LLM. Evidence-Weighted Suppression відновлює істинно позитивні результати, відхилені

верифікацією, покращуючи recall на 0.8 процентних пункти (0.948 проти 0.940). Верифікаційний шлюз забезпечує основну ціннісну пропозицію VERIGATE — контрольованість та інтерпретованість рішень — з мінімальним впливом на ефективність виявлення (1.5% F1); при цьому верифікаційний шлюз у поєднанні з EWS забезпечує чисте покращення recall у порівнянні з *No Verify* на менших вибірках та виявляє унікальні вразливості, пропущені без верифікації.

Усі конфігурації знаходяться у вузькому діапазоні F1 (0.654–0.668), що підтверджує збалансованість архітектурного дизайну: продуктивність не колапсує при видаленні одного компоненту, і жоден компонент не виконує всю роботу самостійно. Повна конфігурація VERIGATE є оптимальним балансом між ефективністю виявлення та якістю рішень: вона забезпечує контрольованість, трасованість та можливість систематичного покращення, що є критичними для виробничого застосування у безпековому контексті. Гіпотеза H3 підтверджена.

4.6. Загрози валідності

4.6.1. Внутрішня валідність

Для підвищення відтворюваності результатів використано фіксований random seed для вибірки та низьку температуру LLM (0.1). Тим не менш, деяка варіативність між запусками можлива при використанні методу на основі LLM.

4.6.2. Зовнішня валідність

PySStuBs фокусується на однорядкових Python-дефектах. Узагальнення на інші мови потребує адаптації мовно-специфічних компонентів: парсера та AST, визначень патернів і taint-специфікацій, інтеграції відповідних інструментів статичного аналізу для попереднього скринінгу. Крос-мовна валідація є перспективним напрямком подальших досліджень (розглянуто у Розділі 5).

4.6.3. Конструктивна валідність

Метрики (precision, recall, F1, accuracy) є стандартними для оцінки інструментів статичного аналізу та відповідають прийнятій практиці у

дослідженнях з безпеки програмного забезпечення. Еталонна розмітка базується на анотаціях датасету PySStuBs, де наявність пар buggy/fixed забезпечує однозначну розмітку.

4.6.4. Консистентність оцінок

Різні абсолютні метрики між оцінками з GPT-4 (RQ1) та GPT-5.2 (RQ2) пояснюються варіаціями у стратегіях промптингу та розмірах вибірки, а не неконсистентністю самого методу VERIGATE. Стабільний патерн покращення у обох контекстах підтверджує висновки дослідження.

4.7. Висновки до розділу

Експериментальна оцінка методу VERIGATE підтвердила всі три висунуті гіпотези:

Гіпотеза H1 підтверджена (RQ1). VERIGATE суттєво перевершує GPT-4 Baseline: F1-score 0.612 проти 0.347 (+76.4%), recall 0.758 проти 0.264 (+187%), при порівняному precision (0.514 проти 0.506). Метод виявив на 247 дефектів більше (379 проти 132).

Гіпотеза H2 підтверджена (RQ2). VERIGATE демонструє стабільне покращення і з GPT-4, і з GPT-5.2 порівняно з відповідними базовими рішеннями, що підтверджує стійкість нейросимвольної верифікації. З GPT-5.2: F1 0.654 проти 0.282 (+132%), recall 0.944 проти 0.204 (+363%).

Гіпотеза H3 підтверджена (RQ3). Абляційне дослідження продемонструвало внесок кожного компоненту: попередній скринінг покращує recall через надання контексту для LLM; верифікаційний шлюз забезпечує інтерпретованість рішень з мінімальним впливом на ефективність (1.5% F1); Evidence-Weighted Suppression відновлює істинно позитивні результати, відхилені верифікацією.

Ключові кількісні результати:

- Покращення F1-score: +76.4% (GPT-4, n = 1 000), +132.0% (GPT-5.2, n = 500)
- Покращення Recall: +187% (GPT-4), +362.7% (GPT-5.2)
- Додатково виявлені дефекти: +247 (GPT-4, 379 vs 132), +185 (GPT-5.2)

- Порівняний precision з GPT-4 baseline (~0.51) при суттєво вищому recall

VERIGATE забезпечує контрольовану поведінку LLM у контексті статичного аналізу коду: символьна верифікація слугує основним контролем якості для всіх тверджень, згенерованих LLM, тоді як шар фінального рішення, визначає, чи існує достатня кількість підтверджувальних доказів для збереження виявлень, які контрольний механізм не може символьно підтвердити. Це забезпечує передбачувану та надійну поведінку, що створює передумови для потенційної інтеграції в виробничі конвеєри CI/CD. Стабільне покращення на різних моделях та конфігураціях підтверджує робастність запропонованого нейросимвольного методу.

5. ОБГОВОРЕННЯ РЕЗУЛЬТАТІВ ТА ПЕРСПЕКТИВИ РОЗВИТКУ

5.1. Інтерпретація основних результатів

5.1.1. Ефективність інверсії напрямку верифікації

Центральною тезою дослідження було припущення, що інверсія традиційного напрямку взаємодії між нейронною та символьною компонентами — з Symbolic → Neural на Neural → Symbolic — дозволить досягти контрольованої поведінки LLM у задачах статичного аналізу: розширення покриття без втрати верифікованості результатів.

Експериментальні результати підтверджують цю тезу. На наборі даних PySStuBs, де традиційні інструменти на основі правил (Semgrep, Bandit, Pylint, Pyright, Flake8) демонструють дуже низьку ефективність (F1 = 0.031, recall = 0.016, лише 8 із 500 дефектів), VERIGATE досягає F1 = 0.612 з recall 0.758, виявляючи 379 із 500 наявних дефектів. Це свідчить, що LLM-компонента успішно розширює покриття за межі наперед визначених правил стандартних інструментів, виявляючи семантично складні дефекти — неправильні імена змінних, некоректні оператори, помилки в аргументах функцій та інші категорії, що потребують глибокого розуміння контексту програми. Водночас символьна верифікація забезпечує контрольованість: кожне виявлення супроводжується конкретними доказами (AST-патерни, taint-потоки, результати перевірки виправлень), а не лише непрозорим висновком LLM.

Порівняння з існуючими методами підкреслює значущість інверсії. IRIS [35] реалізує напрямок Symbolic → Neural (CodeQL → LLM), тобто LLM може аналізувати лише ті виявлення, що вже знайдені CodeQL. На датасеті PySStuBs інструменти на базі правил (аналогічні CodeQL) виявляють лише 8 із 500 дефектів (recall = 0.016). VERIGATE, натомість, не обмежений покриттям символьного аналізатора — LLM може генерувати гіпотези про довільні типи дефектів, а символьна система виступає верифікатором, а не генератором.

5.1.2. Валідація механізму Explain-Verify Gate

Механізм Explain-Verify Gate було запропоновано як архітектурне рішення чотирьох фундаментальних проблем LLM у контексті статичного аналізу: обмежена контрольованість, неконсистентна поведінка, неверифіковані твердження та відсутність детерміністичних гарантій. Експериментальні дані підтверджують ефективність цього механізму.

Порівняння VERIGATE з GPT-4 Baseline (та сама LLM без символьної верифікації) демонструє суттєве покращення recall без значної втрати precision: recall зростає з 0.264 до 0.758 (+187%), F1 — з 0.347 до 0.612 (+76.4%), при порівняному precision (0.514 проти 0.506). Цей результат може здаватися контрінтуїтивним: додавання верифікаційного фільтру зазвичай знижує recall.

Пояснення полягає у механізмі роботи шлюзу. По-перше, структурований формат виходу (JSON із полями bug_type, spans, minimal_fix, rationale) примушує LLM формулювати конкретні, перевірювані твердження замість загальної бінарної класифікації. Це покращує якість виявлень, оскільки модель змушена продемонструвати конкретне розуміння проблеми. По-друге, результати інструментів з попереднього скринінгу передаються як контекстні підказки (tool hints), що суттєво підвищує якість тверджень LLM — абляційне дослідження підтверджує це: видалення скринінгу знижує recall на 3.2%. По-третє, багатоетапне дерево рішень верифікаційного шлюзу з пермісивним шляхом при неспрацюванні правил дає LLM можливість виявляти дефекти за межами покриття AST-патернів, а EWS відновлює істинно позитивні спрацювання, відхилені шлюзом.

5.1.3. Роль структурованих тверджень

Формат структурованих тверджень (claims) виявився критичним для ефективності методу. Кожне поле JSON-об'єкту створює окрему точку верифікації: *bug_type* дозволяє вибрати релевантні правила перевірки та визначити, чи потрібен taint-аналіз; *spans* забезпечує локалізацію для AST-аналізу та перевірки

співпадіння зі спрацюванням правил; *minimal_fix* слугує додатковим сигналом правдоподібності через принцип пояснення через конструювання — якщо після застосування виправлення спрацювання правил зменшується на ≥ 20%, це підтверджує розуміння проблеми; *rationale* дозволяє оцінити якість міркувань моделі.

Цей формальний контракт між нейронною та символьною компонентами є ключовою відмінністю від існуючих методів, де LLM генерує вільнотекстові відповіді без можливості систематичної верифікації. LLMSA [65] використовує LLM для аналізу потоку даних, але не вимагає структурованих тверджень, придатних для незалежної валідації. MoCQ [36] ітеративно уточнює патерни, але не верифікує кожне окреме виявлення.

5.1.4. Узагальнення через покоління моделей

Валідація з GPT-5.2 (RQ2) встановлює важливий результат: нейросимвольна верифікація зберігає цінність при розвитку мовних моделей. VERIGATE суттєво перевершує GPT-5.2 Baseline: F1 0.654 проти 0.282 (+132%), recall 0.944 проти 0.204 (+363%).

Хоча абсолютні метрики відрізняються між оцінками з GPT-4 та GPT-5.2 (що пояснюється варіаціями у стратегіях промптингу, розмірах вибірки та характеристиках моделей), патерн покращення є стабільним: VERIGATE суттєво перевершує базове рішення на основі лише LLM незалежно від покоління моделі. Це підтверджує, що нейросимвольна верифікація є стійким архітектурним принципом, а не артефактом конкретної моделі.

Низька повнота GPT-5.2 Baseline (recall = 0.204) підкреслює важливий аспект: стратегія промптингу та дизайн задачі сильно впливають на ефективність прямої LLM-класифікації. Структурований поетапний метод VERIGATE — з генерацією тверджень, їх символьною верифікацією та зваженою агрегацією доказів — виявляється значно стійкішим до таких варіацій, ніж підхід з некерованою бінарною класифікацією.

5.2. Ціннісна пропозиція верифікаційного шлюзу

5.2.1. Баланс ефективності та контрольованості

Абляційне дослідження (RQ3) встановлює нюансований результат щодо верифікаційного шлюзу: конфігурація *No Verify* досягає дещо вищих F1 (0.666) та recall (0.960), ніж Full VERIGATE (0.656, 0.948). Різниця становить лише 1.5% F1 — усі конфігурації знаходяться у вузькому діапазоні 0.654–0.668. Цей результат потребує ретельної інтерпретації, оскільки він торкається центральної архітектурної тези роботи.

Верифікаційний шлюз не є оптимізацією F1-score — він є архітектурним рішенням для забезпечення контрольованості. Шлюз перетворює систему з непрозорого "LLM сказав ERROR" на прозоре "Виявлено: тип дефекту *X*, підтверджений AST-патерном *Y* на рядку *Z*, taint-потік від *A* до *B*, виправлення усуває *N% спрацювань правил*." Ця трансформація критична для production-застосування.

5.2.2. Аналіз на рівні окремих вразливостей

За агрегованими метриками приховується важливий деталізований результат. Аналіз на 300 зразках показує: *Full VERIGATE* виявляє 7 вразливостей, які *No Verify* пропускає, тоді як *No Verify* виявляє 5 вразливостей, які *Full* пропускає — чисте покращення становить +2 додаткові виявлені вразливості для *Full*. Механізм EWS успішно відновлює 7 тверджень, відхилених шлюзом, при цьому шлюз помилково відхиляє лише 5 валідних тверджень.

У контексті безпеки програмного забезпечення пропущена вразливість (false negative) є значно критичнішою за хибнопозитивне спрацювання (false positive). Пропущена вразливість може призвести до порушень безпеки, витоку даних, порушень compliance. Хибнопозитивне спрацювання потребує лише часу розробника для його відхилення. При цьому верифікаційні причини (verification reasons), які надає шлюз, суттєво прискорюють відсіювання хибнопозитивних

реультаів: розробник бачить конкретні спрацювання правил, taint-потоки та результати перевірки виправлень, а не лише "LLM вважає, що тут проблема."

5.2.3. Інтерпретованість та аудитоспроможність

Кожне рішення *Full VERIGATE* супроводжується конкретною причиною. Ця інформація забезпечує:

- Трасованість: розробники можуть зрозуміти, чому конкретне виявлення було зафлаговане або відхилене.
- Аудитоспроможність: security-команди можуть пояснити результати стейкхолдерам; compliance-вимоги потребують обґрунтування кожного рішення.
- Систематичне покращення: аналіз причин відхилень (span_mismatch, taint_check_failed) дозволяє ідентифікувати конкретні слабкі місця у бібліотеці правил та цілеспрямовано їх усувати.

No Verify не надає жодної з цих можливостей — рішення базуються виключно на непрозорому виході LLM.

5.2.4. Модельна незалежність та детермінованість

Gate забезпечує модельну незалежність: символьна верифікація є детерміністичною (один і той самий код завжди проходить однакову перевірку), що додає стабільність до стохастичних рішень LLM. Це є особливо цінним при використанні менш потужних моделей або при зміні версій LLM — шлюз компенсує варіативність нейронної компоненти.

Дані підтверджують цю тезу: при переході від GPT-4 до GPT-5.2 покращення VERIGATE над базовими рішеннями залишається стабільним, тоді як абсолютні метрики базових рішень суттєво варіюються. Структурований процес з символьною верифікацією є більш стійким до змін моделі, ніж пряма бінарна класифікація.

5.3. Порівняння з існуючими методами

5.3.1. Позиціонування відносно IRIS

IRIS [35] є найближчим аналогом VERIGATE у сфері нейросимвольного статичного аналізу. Концептуальне порівняння виявляє принципові архітектурні відмінності.

Напрямок взаємодії. IRIS реалізує Symbolic → Neural (CodeQL знаходить, LLM фільтрує), VERIGATE — Neural → Symbolic (LLM генерує, символьна система верифікує). У IRIS покриття обмежене правилами CodeQL: якщо CodeQL не має правила для певного типу дефекту, IRIS його не виявить. VERIGATE не має такого обмеження — LLM може генерувати гіпотези про довільні дефекти.

Контроль галюцинацій. IRIS не має явного механізму контролю галюцинацій — LLM просто класифікує результати CodeQL як true/false positive. VERIGATE реалізує повний контроль через багатоетапний Verification Gate з конкретними причинами кожного рішення.

Рівень аналізу. IRIS потребує повного контексту репозиторію для роботи CodeQL. VERIGATE працює на рівні окремих фрагментів коду (snippet-level), що робить метод потенційно придатним для впровадження в процеси CI/CD завдяки мінімальним інфраструктурним вимогам.

Результати на PySStuBs ілюструють цю відмінність: інструменти на базі правил (аналогічні CodeQL) демонструють F1 = 0.031 та recall = 0.016 (лише 8 із 500 дефектів). VERIGATE досягає F1 = 0.612 завдяки інверсії напрямку верифікації.

5.3.2. Позиціонування відносно LLMSA

LLMSA [65] є методом композиційного нейросимвольного аналізу, де LLM виконує data-flow аналіз через програмні абстракції.

Верифікація результатів. LLMSA не має механізму незалежної верифікації тверджень LLM — згенеровані LLM виявлення не проходять символьну

валідацію. VERIGATE реалізує повний процес верифікації через Explain-Verify Gate.

Складність налаштування. LLMSA потребує розробки специфічних абстракцій для кожного типу аналізу. VERIGATE не потребує навчання і розширюється додаванням правил верифікації та taint-специфікацій.

Пояснюваність. VERIGATE забезпечує високу пояснюваність завдяки спрацьовуванням правил, структурованим твердженням та обґрунтуванням верифікацій. LLMSA має обмежену пояснюваність — результати є виходом LLM без незалежного підтвердження.

5.3.3. Позиціонування відносно MoCQ

MoCQ [36] реалізує двонаправлений ітеративний цикл між LLM та класичним статичним аналізом для видобутку та уточнення патернів вразливостей.

Підхід до верифікації. MoCQ не верифікує кожне окреме виявлення — метод зосереджений на уточненні патернів, а не на валідації конкретних тверджень. VERIGATE верифікує кожне виявлення через незалежний символьний аналіз.

Інфраструктурні вимоги. MoCQ потребує значної інфраструктури для ітеративних циклів зворотного зв'язку та повного контексту репозиторію. VERIGATE працює на snippet-level без ітеративних циклів — один виклик LLM на аналіз.

5.3.4. Узагальнене порівняння

Аналіз існуючих нейросимвольних методів дозволяє сформулювати узагальнене позиціонування. Методи на основі правил (Semgrep, Bandit, CodeQL) забезпечують детермінованість та високу точність у межах покриття правил, але не здатні виявляти семантичні дефекти за межами наперед визначених патернів — результати на PySStuBs (F1 = 0.031) демонструють це обмеження. Pure LLM методи (GPT-4/5.2 Baseline) мають семантичне розуміння, але страждають від низького recall при прямій класифікації (0.264 для GPT-4, 0.204 для GPT-5.2) та

відсутності верифікованості. Нейросимвольні методи Symbolic → Neural (IRIS) підвищують точність результатів аналізу, що базується на правилах, однак їх можливості залишаються обмеженими покриттям базового символьного аналізатора.

VERIGATE (Neural → Symbolic) поєднує розширене покриття від LLM з формальними гарантіями символьної верифікації, забезпечуючи одночасно високий recall, інтерпретованість рішень та контрольовану поведінку — комбінацію, недосяжну для жодного з існуючих підходів.

5.4. Наукова новизна

Наукова новизна методу VERIGATE розкривається через три взаємопов'язані аспекти, що разом формують цілісний підхід до нейросимвольного статичного аналізу коду.

1. Інверсія напрямку нейросимвольної верифікації. Вперше у контексті статичного аналізу коду продемонстровано, що зміна конвенційного потоку символьний→нейронний, застосованого в існуючих методах (IRIS, LLMSA, MoCQ), на парадигму нейронний→символьний — де нейронне розмірковування генерує твердження, а символьний аналіз їх незалежно валідує — забезпечує вищу контрольованість завдяки обґрунтуванню всіх виявлень у верифікованих програмних властивостях (AST-патерни, taint-потоки, правдоподібність виправлення), а не лише у статистичних патернах. Існуючі методи реалізують інші напрямки взаємодії: IRIS використовує CodeQL для генерації candidates з подальшою фільтрацією LLM, LLMSA структурує аналіз LLM через композиційні промпти, MoCQ ітеративно уточнює патерни між LLM та символьним аналізом. Запропонована інверсія дозволяє використовувати семантичні можливості LLM для розширення покриття аналізу без втрати формальних гарантій, що експериментально підтверджено покращенням recall на 187% при збереженні порівняного precision.

2. Механізм Explain-Verify Gate із примусовою генерацією структурованих тверджень та незалежною символьною валідацією. Механізм вимагає від LLM формулювання верифікованих тверджень у структурованому форматі — тип дефекту, уражені ділянки коду, мінімальне виправлення та обґрунтування. Ці твердження проходять незалежну валідацію засобами символьного аналізу програм через багатоетапне дерево рішень — AST pattern matching, lightweight taint analysis та fix plausibility checking — без повторного звернення до LLM. Тільки символьно підтверджені твердження проходять через шлюз. Ця архітектура забезпечує контрольовану поведінку LLM, фільтруючи неверифіковані галюцинації та зберігаючи семантичні можливості нейронного розмірковування. Патерн є узагальнюваним і може застосовуватися в інших доменах, де потрібно контролювати виходи LLM: генерація коду, автоматичне виправлення, формальна верифікація.

3. Evidence-Weighted Suppression (EWS) для синтезу доказів із кількох джерел. Механізм адресує притаманний верифікаційному шлюзу консерватизм шляхом синтезу доказів із кількох незалежних джерел — інструменти статичного аналізу, зіставлення AST-патернів та базовий рівень впевненості (baseline confidence score) — при прийнятті остаточного рішення, коли верифікаційний шлюз відхиляє твердження LLM. EWS зберігає високо-достовірні класифікації ERROR за наявності сильних доказів (базовий рівень впевненості ≥ 0.40 з підтвердженням від ≥ 2 інструментів, або ≥ 0.30 за наявності символьних патернів), забезпечуючи баланс між precision (фільтрація неверифікованих тверджень) та recall (збереження валідних виявлень, які шлюз самостійно пропустив би). Абляційне дослідження підтверджує ефективність: видалення EWS знижує recall на 0.8 процентних пункти.

Додатково, практичні характеристики методу — training-free архітектура (не потребує дотренування моделей або специфічних датасетів), модульність (LLM може бути замінена на іншу модель, що підтверджено валідацією з GPT-5.2; правила верифікації розширюються без зміни архітектури) та пояснюваність

результатів (кожне виявлення супроводжується конкретними символьними доказами) — створюють передумови для потенційного практичного застосування методу.

5.5. Практичне значення результатів

5.5.1. Для розробників програмного забезпечення

Розширене покриття виявлення. VERIGATE виявляє семантично складні дефекти, недоступні традиційним інструментам на базі правил. На датасеті PySStuBs це означає виявлення 379 дефектів порівняно з лише 8 для Semgrep, Bandit, Pylint, Pyright, Flake8 — дефектів, що потребують розуміння семантики програми, а не лише синтаксичних патернів стандартних інструментів.

Порівняна точність precision з базовим рішенням. VERIGATE та GPT-4 Baseline досягають близького precision (~0.51), що означає: розробники не стикаються з суттєво більшою кількістю хибних спрацювань порівняно з базовим рішенням, але отримують значно більше виявлених реальних дефектів (+247).

Пояснюваність виявлень. Кожне виявлення містить локалізацію (spans), тип дефекту (bug_type), запропоноване виправлення (minimal_fix) та обґрунтування (rationale), доповнені символьними доказами (rule hits, taint-потоки). Це полегшує розуміння та виправлення проблем порівняно з непрозорими виходами LLM.

5.5.2. Потенціал для інтеграції в інструменти та процеси

Архітектурні характеристики VERIGATE — training-free підхід, snippet-level аналіз, один виклик LLM на фрагмент (5–15 секунд) — роблять метод перспективним для практичної інтеграції у існуючі процеси розробки програмного забезпечення. Хоча на поточному етапі дослідження така інтеграція не реалізована та не оцінена експериментально, можна виділити кілька потенційних напрямків застосування.

CI/CD інтеграція. Snippet-level архітектура та прийнятна латентність (5–15 секунд на фрагмент) створюють передумови для використання VERIGATE як додаткового етапу перевірки якості коду у процес безперервної інтеграції.

Code review assistance. Структуровані твердження з verification reasons потенційно можуть допомагати рецензентам зосередитися на проблемних ділянках коду при аналізі pull requests.

Компонент верифікації у системах генерації коду. Архітектура Explain-Verify Gate потенційно може бути адаптована для верифікації автоматично згенерованого коду (наприклад, в системах типу Copilot, CodeLlama), перевіряючи згенерований код на наявність дефектів та вразливостей перед його прийняттям.

5.5.3. Для наукової спільноти

Відтворюваність. Експерименти проведено на публічно доступному датасеті PySStuBs [32] з фіксованими random seeds та чітко описаною методологією.

Архітектурний патерн Explain-Verify. Запропонований патерн може бути узагальнений на інші домени, де потрібно контролювати виходи LLM: генерація коду з верифікацією коректності, автоматичне виправлення дефектів з перевіркою, формальна верифікація програм із LLM-асистованою генерацією доказів.

Framework оцінки. Трикомпонентний framework (ефективність, стійкість до зміни моделей, абляційний аналіз) може слугувати стандартом для порівняння нейросимвольних аналізаторів.

5.6. Обмеження

5.6.1. Обмеження методу

Intra-procedural taint analysis. Реалізований taint-аналіз обмежений межами однієї функції — він не відстежує потоки даних між функціями, класами та модулями. Це може знижувати ефективність на вразливостях, де source та sink знаходяться у різних функціях.

Покриття AST-патернів. Поточна бібліотека AST-патернів покриває поширені категорії дефектів, але не є вичерпною. Абляційне дослідження підтверджує: шлюз інколи відхиляє валідні твердження LLM через недостатнє покриття правил — це є основним напрямком для покращення.

Snippet-level scope. Аналіз оптимізований для ізольованих фрагментів коду. Аналіз повних проєктів з урахуванням залежностей, імпортів та міжмодульного контексту потребує адаптації.

Порівняна точність з базовим рішенням. VERIGATE досягає precision ~0.51 — порівняного з базовим рішенням, але не ідеального. Хоча precision не є основною метою методу (VERIGATE фокусується на recall та контрольованості), FP = 359 при 1 000 зразках означає значну кількість хибних спрацювань, що потребують тріажу.

5.6.2. Обмеження оцінки

Мовне покриття. Основна оцінка проведена виключно на Python-коді. Узагальнення на інші мови програмування потребує адаптації мовно-специфічних компонентів: парсера та AST-правил, taint-специфікацій, інструментів статичного аналізу для попереднього скринінгу.

Характеристики датасету. PySStuBs фокусується на однорядкових Python-дефектах з реальних open-source проєктів. Хоча це забезпечує реалістичність дефектів, узагальнення на інші типи задач (CWE-класифіковані вразливості, багаторядкові дефекти, дефекти у специфічних доменах) потребує додаткової оцінки.

Консистентність між оцінками. Різні абсолютні метрики між оцінками з GPT-4 (RQ1) та GPT-5.2 (RQ2) пояснюються варіаціями у стратегіях промптингу та розмірах вибірки. Хоча стабільний патерн покращення підтверджує висновки, пряме кількісне порівняння між оцінками потребує обережної інтерпретації.

Залежність від комерційних LLM. Експерименти проведені з комерційними моделями OpenAI (GPT-4, GPT-5.2). Ефективність методу з open-source моделями (Llama, CodeLlama, StarCoder) потребує окремої оцінки.

5.7. Перспективи розвитку

5.7.1. Короткострокові напрямки

Розширення бібліотеки верифікаційних правил. Абляційне дослідження показує, що шлюз інколи відхиляє валідні твердження через обмежене покриття AST-патернів. Систематичне розширення правил для додаткових категорій дефектів є найбільш прямим шляхом покращення ефективності без зміни архітектури. Аналіз причин відхилення (span_mismatch, taint_check_failed) дозволяє ідентифікувати конкретні прогалини у бібліотеці правил.

Inter-procedural taint analysis. Розширення taint-аналізу для відстеження потоків даних між функціями дозволить виявляти вразливості, де source та sink знаходяться у різних частинах програми.

Крос-мовна валідація. Розширення підтримки на Java (з використанням CWE-Bench-Java [65]) для демонстрації узагальнюваності методу. Це потребує адаптації мовно-специфічних компонентів: Java AST-парсер, CWE-специфічні правила, taint-специфікації для Java-фреймворків.

5.7.2. Середньострокові напрямки

Адаптивні пороги верифікації. Дослідження динамічної адаптації строгості шлюзу залежно від рівня впевненості LLM, типу дефекту та контексту коду. Це може зменшити кількість хибно відхилених валідних тверджень.

Підтримка повних проєктів. Адаптація архітектури для аналізу повних кодових баз з урахуванням залежностей, конфігурації та міжмодульного контексту.

Локальні LLM. Дослідження ефективності методу з open-source моделями (Llama, CodeLlama, StarCoder) для зниження латентності, вартості та забезпечення конфіденційності. Шлюз може бути особливо цінним для менш потужних моделей, компенсуючи їх обмеження символьною верифікацією.

Зворотний зв’язок від розробників. Інтеграція механізму, за якого відхилення хибнопозитивних виявлень розробниками автоматично уточнює правила перевірки, забезпечуючи самовдосконалювану систему.

5.7.3. Довгострокова перспектива

Автоматичне генерування верифікаційних правил. Система, що автоматично пропонує нові AST-патерни та taint-специфікації на основі верифікованих дефектів, перетворюючи кожне підтверджене виявлення на потенційне нове правило.

Мультимовний аналіз. Розширення на JavaScript, TypeScript, Go, Rust, C++ через розробку мовно-специфічних правил верифікації та taint-конфігурацій.

Інтеграція з IDE. Розробка плагінів для VS Code, IntelliJ для real-time аналізу коду під час написання з миттєвим зворотним зв'язком від Explain-Verify Gate.

Стандартизація формату тверджень. Розробка стандартного формату для structured defect claims, що дозволить інтероперабельність між різними нейросимвольними аналізаторами.

5.8. Висновки до розділу

Аналіз та інтерпретація експериментальних результатів дозволяють сформулювати наступні висновки:

1. Підтверджено ефективність інверсії напрямку верифікації Neural → Symbolic. Експериментальні дані демонструють, що такий напрямок дозволяє поєднати розширене покриття LLM з формальними гарантіями символьної верифікації: VERIGATE досягає F1 = 0.612 на датасеті, де інструменти на базі правил демонструють дуже низьку ефективність (F1 = 0.031).
2. Валідовано ціннісну пропозицію верифікаційного шлюзу як архітектурного рішення для контрольованості. Мінімальна різниця в ефективності з *No Verify* (1.5% F1) є прийнятною платою за якісні переваги: інтерпретованість рішень, аудитоспроможність, модельну незалежність, детермінованість символьної перевірки та можливість систематичного покращення. На рівні окремих вразливостей *Full VERIGATE* виявляє +2 унікальні вразливості порівняно з *No Verify* (300-sample аналіз).

3. Встановлено стійкість нейросимвольної верифікації при розвитку мовних моделей. VERIGATE суттєво перевершує GPT-4 Baseline (+76.4% F1), і з GPT-5.2 (+132% F1), підтверджуючи, що архітектурний принцип є стійким, а не артефактом конкретної моделі.
4. Обґрунтовано наукову новизну через три взаємопов'язані аспекти одного методу: інверсія напрямку взаємодії (Neural → Symbolic), механізм Explain-Verify Gate для контролю виходів LLM та Evidence-Weighted Suppression для балансування precision і recall.
5. Визначено практичне значення результатів для розробників (розширене покриття, пояснюваність) та для наукової спільноти (узагальнюваний архітектурний патерн, framework оцінки). Визначено потенціал методу для інтеграції в інструменти та процеси (CI/CD, code review, верифікація згенерованого коду), що потребує подальшого дослідження.
6. Окреслено обмеження: intra-procedural taint analysis, неповне покриття AST-патернів, snippet-level scope, оцінка лише на Python та одному датасеті. Визначено конкретні перспективи розвитку, від розширення правил та крос-мовної валідації до автоматичного генерування верифікаційних правил та стандартизації формату тверджень.

Загалом, результати дослідження підтверджують, що нейросимвольний метод VERIGATE з механізмом Explain-Verify Gate забезпечує контрольовану поведінку LLM у задачах статичного аналізу коду: система повідомляє виключно ті проблеми, що є одночасно семантично правдоподібними та структурно верифікованими, забезпечуючи передбачувану, надійну та прозору поведінку.

ВИСНОВКИ

У дисертаційній роботі вирішено актуальне наукове завдання розробки нейросимвольного методу статичного аналізу програмного коду, що забезпечує контрольовану поведінку великих мовних моделей через незалежну символьну верифікацію генерованих тверджень. Основні наукові та практичні результати роботи полягають у наступному:

1. Систематизовано класичні, нейронні та нейросимвольні методи статичного аналізу програмного коду. Аналіз еволюції методів — від інструментів на основі правил (Semgrep, Bandit, CodeQL), через методи машинного навчання та графові нейронні мережі (Devign, Code Property Graphs), pre-trained моделі (CodeBERT, GraphCodeBERT) до великих мовних моделей — дозволив ідентифікувати фундаментальні обмеження кожного покоління. Систематизація існуючих нейросимвольних методів (IRIS, LLMSA, MoCQ) виявила конкретну прогалину: відсутність незалежної символьної верифікації тверджень LLM у напрямку Neural → Symbolic. Сформульовано вимоги до методу, що заповнює цю прогалину.
2. Досліджено фундаментальні проблеми застосування великих мовних моделей у статичному аналізі: обмежена контрольованість, неконсистентна поведінка, неверифіковані твердження та відсутність детерміністичних гарантій. Встановлено, що ці проблеми колективно перешкоджають досягненню надійності та передбачуваності, необхідних для інтеграції LLM-інструментів у виробничі процеси розробки програмного забезпечення.
3. Вперше запропоновано нейросимвольний метод статичного аналізу програмного коду, який відрізняється тим, що на першому етапі аналізу LLM безпосередньо генерує структуровані твердження про дефекти коду, за чим слідує етап символьного аналізу для їх верифікації, в той час, як в існуючих нейросимвольних методах статичного аналізу коду (IRIS, LLMSA, MoCQ) навпаки — символьний компонент виконує первинне виявлення або

структуризацію дефектів, а нейронний компонент фільтрує чи доповнює його результати. Інвертований напрямок нейросимвольної верифікації "Neural → Symbolic" дозволяє скористатися здатністю великої мовної моделі сприймати семантику коду при одночасному жорсткому контролі згенерованих нею тверджень, що сприяє виявленню ширшого кола дефектів при мінімальних ризиках галюцинації моделі.

4. Вперше запропоновано спосіб верифікації структурованих тверджень про дефекти коду із застосуванням символьного аналізу Explain-Verify Gate, особливість якого полягає у тому, що атрибути твердження про дефект (тип дефекту, локалізація, пропозиція мінімально задовільного виправлення та обґрунтування) визначають вибір та інтерпретацію подальших символьних перевірок (зіставлення AST-патернів, taint-аналіз, перевірка правдоподібності виправлення), що є відмінним від існуючих способів верифікації у нейросимвольних методах аналізу коду та забезпечує детермінізм верифікації за відсутності потреби повторного звернення до мовної моделі за обґрунтуванням кожного прийнятого нею рішення.
5. Удосконалено процес прийняття рішень щодо дефектів у нейросимвольному статичному аналізі програмного коду шляхом застосування механізму Evidence-Weighted Suppression (EWS), який зважує докази з кількох незалежних джерел (інструменти статичного аналізу, AST-патерни, показник впевненості базового аналізатора), що відрізняється від процесу визначення дефектів в рамках існуючих нейросимвольних підходів, в яких відхилення результату на етапі верифікації є остаточним без урахування доказів з інших джерел. Це дозволяє збалансувати точність і повноту аналізу шляхом відновлення валідних виявлень дефектів, що відхиляються символьною верифікацією за відсутності відповідних патернів, але підтверджуються достатніми доказами з інших джерел.
6. Формалізовано архітектуру та алгоритми запропонованого методу, включаючи чотириетапну обробку (попередній скринінг, Explain, Verify,

фінальне рішення), шестиетапне дерево рішень верифікації, критерії прийняття та відхилення тверджень, механізм span alignment та правила Evidence-Weighted Suppression.

7. Реалізовано програмний прототип методу VERIGATE з модульною архітектурою для мови програмування Python з використанням OpenAI API та ансамблю інструментів статичного аналізу (Semgrep, Bandit, Pylint, Pyright, Flake8). Прототип є training-free та розширюваним через модифікацію промптів і додавання символьних правил.
8. Проведено комплексне експериментальне дослідження на датасеті PySStuBs, що включає три дослідницькі питання. RQ1 (GPT-4, n = 1 000): VERIGATE досягає F1 = 0.612, recall = 0.758, виявляючи 379 із 500 наявних дефектів — покращення F1 на 76.4% та recall на 187% порівняно з GPT-4 Baseline (132 виявлення) при збереженні співставного рівня точності (~0.51). RQ2 (GPT-5.2, n = 500): покращення +132% F1, +363% recall, що підтверджує стійкість нейросимвольної верифікації як архітектурного принципу незалежно від покоління мовної моделі. RQ3 — абляційне дослідження (n = 500): вузький діапазон F1 (0.654–0.668) між конфігураціями підтверджує збалансованість архітектури та незалежний внесок кожного компоненту.

Практичне значення отриманих результатів полягає в тому, що розроблений метод статичного аналізу коду дає змогу виявляти додаткові дефекти, які не виявляються як інструментами на основі правил, так і безпосереднім використанням LLM. Виявлення цих додаткових дефектів на етапі статичного аналізу запобігає їх потраплянню на подальші етапи розробки, де вартість їх виправлення суттєво зростає. Результати аналізу представлені структурованим описом дефекту (тип, локалізація, обґрунтування, мінімально необхідне виправлення), який завжди супроводжується результатами символьної верифікації, що скорочує час розробника на аналіз та усунення виявлених проблем та суттєво полегшує інтеграцію методу в автоматизовані процеси контролю якості коду

(CI/CD, code review). Запропонований метод не потребує донавчання моделей та збору спеціалізованих датасетів, а експериментальне застосування GPT-5.2 підтвердило збереження переваг при переході на новіше покоління мовної моделі, що свідчить про можливість оновлення нейронного компоненту без модифікації решти складових системи. Метод забезпечує можливість налаштування під різноманітні конкретні задачі аналізу без зміни загальної архітектури: тип аналізу та цільові класи дефектів визначаються у змісті запиту до нейромережевого компоненту, а покриття символьної верифікації розширюється додаванням нових правил, AST-патернів та taint-специфікацій.

Достовірність отриманих результатів забезпечується використанням публічного датасету PySStuBs з реальними дефектами з open-source проектів, застосуванням стандартних метрик оцінки (precision, recall, F1-score), проведенням валідації на двох поколіннях мовних моделей (GPT-4 та GPT-5.2), а також абляційним дослідженням, що підтвердило внесок кожного архітектурного компоненту.

Дане дослідження у майбутньому можна розвивати у кількох напрямках. По-перше, розширення символьного аналізу до міжпроцедурного рівня дозволить виявляти вразливості, пов'язані з потоком даних між функціями, що наразі є обмеженням інтрапроцедурного taint-аналізу. По-друге, розширення бібліотеки AST-патернів та taint-специфікацій для підтримки нових мов програмування (JavaScript, TypeScript, C++, Go) та нових класів дефектів збільшить покриття символьної верифікації. По-третє, інтеграція методу у виробничі процеси розробки (CI/CD, code review, IDE) дозволить оцінити його ефективність у реальних умовах промислової експлуатації. Зокрема, перспективним є дослідження адаптивного вибору символьних перевірок залежно від типу дефекту та використання зворотного зв'язку від розробників для автоматичного вдосконалення правил верифікації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Alqaradaghi, M., & Kozsik, T. (2024). Comprehensive Evaluation of Static Analysis Tools for Their Performance in Finding Vulnerabilities in Java Code. *IEEE Access*, 12, 55824-55842. DOI:10.1109/ACCESS.2024.3389955
2. Amadini, R., Gange, G., Schachte, P., Søndergaard, H., & Stuckey, P.J. (2020). Abstract Interpretation, Symbolic Execution and Constraints. *Gabbrielli's Festschrift*. DOI:10.4230/OASIcs.Gabbrielli.7
3. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C.J., Terry, M., Le, Q., & Sutton, C. (2021). Program Synthesis with Large Language Models. ArXiv, abs/2108.07732. DOI:10.48550/arXiv.2108.07732
4. Avgustinov, P., Moor, O.D., Jones, M.P., & Schäfer, M. (2016). QL: Object-oriented Queries on Relational Data. European Conference on Object-Oriented Programming. DOI:10.4230/LIPIcs.ECOOP.2016.2
5. Bharadwaj, R., & Parker, I. (2023). Combining Foundation Models and Symbolic AI for Automated Detection and Mitigation of Code Vulnerabilities. *Naval Research Laboratory.*
6. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., & Rodriguez, D. (2015). Moving Fast with Software Verification. *NASA Formal Methods*. DOI:10.1007/978-3-319-17524-9_1
7. Charoenwet, W., Thongtanunam, P., Pham, V., & Treude, C. (2024). An Empirical Study of Static Analysis Tools for Secure Code Review. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. DOI:10.1145/3650212.3680313
8. Chaudhuri S, Ellis K, Polozov O, Singh R, Solar-Lezama A & Yue Y (2021). Neurosymbolic Programming. *Foundations and Trends in Programming Languages,* Vol. 7 No. 3 pp. 158–243. DOI:10.1561/2500000049

9. Cheirdari, F., & Karabatis, G. (2018). Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools. *2018 IEEE International Conference on Big Data (Big Data)*, 4782-4788. DOI:10.1109/BigData.2018.8622456
10. Chen, Q., Yu, C., Liu, R., Zhang, C., Wang, Y., Wang, K., Su, T., & Wang, L. (2024). Evaluating the Effectiveness of Deep Learning Models for Foundational Program Analysis Tasks. *Proceedings of the ACM on Programming Languages, 8*, 500 - 528. DOI:10.1145/3649829
11. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pondé, H., Kaplan, J., та ін. (2021). Evaluating Large Language Models Trained on Code. *ArXiv, abs/2107.03374*. DOI:10.48550/arXiv.2107.03374
12. Chess, B., & McGraw, G. (2004). Static Analysis for Security. *IEEE Secur. Priv., 2*, 76-79. DOI:10.1109/MSP.2004.111
13. Chess, B., & West, J. (2007). Secure Programming with Static Analysis.
14. Ciatto, G., Sabbatini, F., Agiollo, A., Magnini, M., & Omicini, A. (2024). Symbolic Knowledge Extraction and Injection with Sub-symbolic Predictors: A Systematic Literature Review. *ACM Computing Surveys, 56*, 1 - 35. DOI:10.1145/3645103
15. Clem, T., & Thomson, P. (2022). Static analysis at GitHub. *Communications of the ACM*, 65, 44 - 51. DOI:10.1145/3486594
16. Colelough, B.C., & Regli, W. (2025). Neuro-Symbolic AI in 2024: A Systematic Review. *ArXiv, abs/2501.05435*. DOI:10.48550/arXiv.2501.05435
17. Cui, H., Xie, M., Su, T., Zhang, C., & Tan, S.H. (2024). An Empirical Study of False Negatives and Positives of Static Code Analyzers From the Perspective of Historical Issues. *ArXiv, abs/2408.13855*. DOI:10.48550/arXiv.2408.13855
18. D'Ambros, M., Lanza, M., & Robbes, R. (2011). Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering, 17*, 531 - 577. DOI:10.1007/s10664-011-9173-9
19. Dhuliawala, S., Komeili, M., Xu, J., Raileanu, R., Li, X., Celikyilmaz, A., & Weston, J. (2024). Chain-of-Verification Reduces Hallucination in Large

Language Models. *Findings of the Association for Computational Linguistics: ACL 2024,* 3563–3578. DOI:10.18653/v1/2024.findings-acl.212

20. Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., Wagner, D.A., Ray, B., & Chen, Y. (2024). Vulnerability Detection with Code Language Models: How Far are We? *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 1729-1741. DOI:10.1109/ICSE55347.2025.00038
21. Nguyen, L., Do, Q., Wright, J.R., & Ali, K. (2020). Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations. *IEEE Transactions on Software Engineering*, 48, 835-847. DOI:10.1109/tse.2020.3004525
22. Dunlap, T., Thorn, S., Enck, W., & Reaves, B. (2023). Finding Fixed Vulnerabilities with Off-the-Shelf Static Analysis. *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 489-505. DOI:10.1109/EuroSP57164.2023.00036
23. Duraibi, S., Alashjaee, A.M., & Song, J. (2019). A Survey of Symbolic Execution Tools.
24. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *ArXiv, abs/2002.08155*. DOI:10.18653/v1/2020.findings-emnlp.139
25. Guo, Z., Tan, T., Liu, S., Liu, X., Lai, W., Yang, Y., Li, Y., Chen, L., Dong, W., & Zhou, Y. (2023). Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities. *IEEE Transactions on Software Engineering, 49*, 5154-5188. DOI:10.1109/TSE.2023.3329667
26. Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Yin, J., Jiang, D., & Zhou, M. (2020). GraphCodeBERT: Pre-training Code Representations with Data Flow. *ArXiv, abs/2009.08366*. DOI:10.48550/arXiv.2009.08366

27. Hallem, S., Chelf, B., Xie, Y., & Engler, D.R. (2002). A system and language for building system-specific, static analyses. *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*. DOI:10.1145/512529.512539
28. Hanif, H., & Maffeis, S. (2022). VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. *2022 International Joint Conference on Neural Networks (IJCNN)*, 1-8. DOI:10.1109/IJCNN55064.2022.9892280
29. Shiri Harzevili, N., Boaye Belle, A., Wang, J., Wang, S., Jiang, Z., & Nagappan, N. (2024). A Systematic Literature Review on Automated Software Vulnerability Detection Using Machine Learning. *ACM Computing Surveys, 57*, 1 - 36. DOI:10.1145/3699711
30. Jones, N.D., & Nielson, F. (1995). Abstract interpretation: a semantics-based tool for program analysis. Logic in Computer Science.
31. Just, R., Jalali, D., & Ernst, M.D. (2014). Defects4J: a database of existing faults to enable controlled testing studies for Java programs. *International Symposium on Software Testing and Analysis*. DOI:10.1145/2610384.2628055
32. Kamienski, A.V., Palechor, L., Bezemer, C., & Hindle, A. (2021). PySStuBs: Characterizing Single-Statement Bugs in Popular Open-Source Python Projects. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 520-524. DOI:10.1109/MSR52588.2021.00066
33. Kang, H.J., Aw, K.L., & Lo, D. (2022). Detecting False Alarms from Automatic Static Analysis Tools: How Far are We? *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 698-709. DOI:10.1145/3510003.3510214
34. Lenarduzzi, V., Lujan, S., Saarimäki, N., & Palomba, F. (2021). A Critical Comparison on Six Static Analysis Tools: Detection, Agreement, and Precision. *ArXiv, abs/2101.08832*. DOI:10.2139/ssrn.4044439

35. Li, Z., Dutta, S., & Naik, M. (2024). LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. *ArXiv, abs/2405.17238*. DOI:10.48550/arXiv.2405.17238
36. Li, P., Yao, S., Korich, J.S., Luo, C., Yu, J., Cao, Y., & Yang, J. (2025). Automated Static Vulnerability Detection via a Holistic Neuro-symbolic Approach. *ArXiv, abs/2504.16057*. DOI:10.48550/arXiv.2504.16057
37. Li, J., Li, Z., Zhang, H., Li, G., Jin, Z., Hu, X., & Xia, X. (2023). Poison Attack and Poison Detection on Deep Source Code Processing Models. *ACM Transactions on Software Engineering and Methodology, 33*, 1 - 31. DOI:10.1145/3630008
38. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond та ін. (2022). Competition-level code generation with AlphaCode. *Science, 378*, 1092 - 1097. DOI:10.1126/science.abq1158
39. Lin, G., Wen, S., Han, Q., Zhang, J., & Xiang, Y. (2020). Software Vulnerability Detection Using Deep Neural Networks: A Survey. *Proceedings of the IEEE, 108*, 1825-1848. DOI:10.1109/JPROC.2020.2993293
40. Lin, J., & Mohaisen, D. (2025). From Large to Mammoth: A Comparative Evaluation of Large Language Models in Vulnerability Detection. Network and Distributed System Security Symposium. DOI:10.14722/ndss.2025.241491
41. Liu, R., Wang, Y., Xu, H., Sun, J., Zhang, F., Li, P., & Guo, Z. (2024). Vul-LMGNNs: Fusing language models and online-distilled graph neural networks for code vulnerability detection. *Inf. Fusion, 115*, 102748. DOI:10.1016/j.inffus.2024.102748
42. Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., та ін. (2021). CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *ArXiv, abs/2102.04664*. DOI:10.48550/arXiv.2102.04664
43. Mukherjee, R., Wen, Y., Chaudhari, D., Reps, T., Chaudhuri, S., & Jermaine, C. (2021). Neural Program Generation Modulo Static Analysis. *ArXiv, abs/2111.01633*. DOI:10.48550/arXiv.2111.01633

44. Niu, C., Li, C., Ng, V., Ge, J., Huang, L., & Luo, B. (2022). SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 01-13. DOI:10.1145/3510003.3510096
45. Norheim, J.J., Rebentisch, E., Xiao, D., Draeger, L., Kerbrat, A., & de Weck, O.L. (2024). Challenges in applying large language models to requirements engineering tasks. *Design Science, 10*. DOI:10.1017/dsj.2024.8
46. Pearce, H.A., Tan, B., Ahmad, B., Karri, R., & Dolan-Gavitt, B. (2021). Examining Zero-Shot Vulnerability Repair with Large Language Models. *2023 IEEE Symposium on Security and Privacy (SP)*, 2339-2356. DOI:10.1109/SP46215.2023.10179324
47. Pereira, J.D., & Vieira, M.P. (2020). On the Use of Open-Source C/C++ Static Analysis Tools in Large Projects. *2020 16th European Dependable Computing Conference (EDCC)*, 97-102. DOI:10.1109/EDCC51268.2020.00025
48. Pistoia, M., Chandra, S., Fink, S.J., & Yahav, E. (2007). A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Syst. J., 46*, 265-288. DOI:10.1147/SJ.462.0265
49. PMD Open Source Project. (2025). Pmd source code analyzer project. *Software documentation*. Дата звернення: 15.02.2026. URL: https://docs.pmd-code.org/latest/
50. Purba, M.D., Ghosh, A., Radford, B.J., & Chu, B. (2023). Software Vulnerability Detection using Large Language Models. *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 112-119. DOI:10.1109/ISSREW60843.2023.00058
51. PyCQA. (n.d.). Bandit: A security tool for python code. *Software documentation*. Дата звернення: 16.02.2026. URL: https://bandit.readthedocs.io/
52. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., & Jaspan, C. (2018). Lessons from building static analysis tools at Google. *Communications of the ACM, 61*, 58 - 66. DOI:10.1145/3188720

53. Sandoval, G., Pearce, H.A., Nys, T., Karri, R., Garg, S., & Dolan-Gavitt, B. (2022). Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. USENIX Security Symposium. DOI:10.48550/arXiv.2208.09727
54. Hitzler, P., & Sarker, M. (2025). Neuro-Symbolic Artificial Intelligence. *Studies in Computational Intelligence*. DOI:10.1007/978-981-97-8171-3
55. Semasaba, A.O., Zheng, W., Wu, X., & Agyemang, S.A. (2020). Literature survey of deep learning-based vulnerability analysis on source code. *IET Softw., 14*, 654-664. DOI:10.1049/iet-sen.2020.0084
56. Semgrep, Inc. (n.d.). Semgrep: Lightweight static analysis for many languages. *Software documentation*. Дата звернення: 16.02.2026. URL: https://semgrep.dev/
57. Sheth, A.P., Roy, K., Gaur, M., & Sheth, A.P. (2023). Neurosymbolic Artificial Intelligence (Why, What, and How). *IEEE Intelligent Systems, 38*, 56-62. DOI:10.1109/MIS.2023.3268724
58. Steenhoek, B., Rahman, M.M., Jiles, R., & Le, W. (2022). An Empirical Study of Deep Learning Models for Vulnerability Detection. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2237-2248. DOI:10.1109/ICSE48619.2023.00188
59. Sun, J.J., Tjandrasuwita, M., Sehgal, A., Solar-Lezama, A., Chaudhuri, S., Yue, Y., & Costilla-Reyes, O. (2022). Neurosymbolic Programming for Science. *ArXiv, abs/2210.05050*. DOI:10.48550/arXiv.2210.05050
60. Tamberg, K., & Bahşi, H. (2024). Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study. IEEE Access, 13, 29698-29717. DOI:10.1109/ACCESS.2025.3541146
61. Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., & Gall, H.C. (2018). Context is king: The developer perspective on the usage of static analysis tools. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 38-49. DOI:10.1109/SANER.2018.8330195

62. Venkatesh, A., Sabu, S., Mir, A.M., Reis, S., & Bodden, E. (2024). The Emergence of Large Language Models in Static Analysis: A First Look Through Micro-Benchmarks. *FORGE '24: 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, 35-39.DOI:10.1145/3650105.3652288
63. Viet, N.H., & Vinh, N.T. (2024). Large language models in software engineering. *Journal of Education For Sustainable Innovation*. DOI:10.56916/jesi.v2i2.968
64. Wan, Y., Bi, Z., He, Y., Zhang, J., Zhang, H., Sui, Y., Xu, G., Jin, H., & Yu, P.S. (2023). Deep Learning for Code Intelligence: Survey, Benchmark and Toolkit. *ACM Computing Surveys, 56*, 1 - 41. DOI:10.1145/3664597
65. Wang, C., Gao, Y., Zhang, W., Liu, X., Shi, Q., & Zhang, X. (2024). LLMSA: A Compositional Neuro-Symbolic Approach to Compilation-free and Customizable Static Analysis. *ArXiv, abs/2412.14399*. DOI:10.48550/arXiv.2412.14399
66. Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E.H., & Zhou, D. (2022). Self-Consistency Improves Chain of Thought Reasoning in Language Models. *ArXiv, abs/2203.11171*. DOI:10.48550/arXiv.2203.11171
67. Wen, C., Cai, Y., Zhang, B., Su, J., Xu, Z., Liu, D., Qin, S., Ming, Z., & Cong, T. (2024). Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We? *ACM Transactions on Knowledge Discovery from Data, 18*, 1 - 34. DOI:10.1145/3653718
68. Wichmann, B.A., Canning, A.A., Clutterbuck, D.L., Winsborrow, L.A., Ward, N.J., & Marsh, W. (1995). *Industrial perspective on static analysis. Softw. Eng. J., 10*, 69-75. DOI:10.1049/sej.1995.0010
69. Xu, Y., Zhang, C., & Pu, G. (2025). Revisiting the Combination of Static Analysis Error Traces and Dynamic Symbolic Execution: A Potential Approach for True Positive Confirmation (Registered Report). *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*. DOI:10.1145/3713081.3731720

70. Yamaguchi, F., Golde, N., Arp, D., & Rieck, K. (2014). Modeling and Discovering Vulnerabilities with Code Property Graphs. *2014 IEEE Symposium on Security and Privacy*, 590-604. DOI:10.1109/SP.2014.44
71. Yang, Y., Zhou, X., Mao, R., Xu, J., Yang, L., Zhangm, Y., Shen, H., & Zhang, H. (2024). DLAP: A Deep Learning Augmented Large Language Model Prompting Framework for Software Vulnerability Detection. *J. Syst. Softw., 219*, 112234. DOI:10.48550/arXiv.2405.01202
72. Yao, P., Shi, Q., Huang, H., & Zhang, C. (2021). Program analysis via efficient symbolic abstraction. *Proceedings of the ACM on Programming Languages, 5*, 1 - 32. DOI:10.1145/3485495
73. Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019). Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *ArXiv, abs/1909.03496*. DOI:10.48550/arXiv.1909.03496
74. Zhou, X., Cao, S., Sun, X., & Lo, D. (2024). Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. *ACM Transactions on Software Engineering and Methodology, 34*, 1 - 31. DOI:10.1145/3708522