УДК 004.056.5
DOI https://doi.org/10.32782/IT/2024-4-6

Андрій БОРИСЕНКО
доктор філософії, молодший науковий співробітник, Інститут транспортних систем і технологій НАН України, вул. Писаржевського, 5, м. Дніпро, Україна, 49005
ORCID: *0009-0001-5083-5255*
Scopus Author ID: *58158469800*

Бібліографічний опис статті: Борисенко, А. (2024). Безпечне програмування: автоматизовані інструменти для виявлення вразливостей в коді. *Information Technology: Computer Science, Software Engineering and Cyber Security,* 4, 43–52, doi: https://doi.org/10.32782/IT/2024-4-6

БЕЗПЕЧНЕ ПРОГРАМУВАННЯ: АВТОМАТИЗОВАНІ ІНСТРУМЕНТИ ДЛЯ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ В КОДІ

З кожним роком виявляється все більше вразливостей програмного забезпечення та вони стають все більш шкідливими для організацій у всьому світі, впливаючи на їхні фінанси, діяльність і репутацію.

Мета роботи. *Вразливості можуть становити серйозний ризик та призвести до зламу системи, витоку інформації або відмови в обслуговуванні. Ця проблема погіршується через нестачу фахівців з кібербезпеки. Автоматизовані інструменти та технології можуть оптимізувати процес виявлення вразливостей, зробивши його ефективнішим. Метою роботи є аналіз ефективності існуючих рішень для виявлення різних типів вразливостей на різних етапах розробки програмного забезпечення, а також визначення переваг і недоліків кожного з інструментів. Робота спрямована на підвищення точності та ефективності виявлення уразливостей в реальних умовах програмування.*

Методологія. *Досліджено наявні автоматизовані інструменти для виявлення вразливостей у коді в контексті безпечного програмування. Проведено порівняльний аналіз популярних рішень, таких як статичний аналізатор коду, динамічний аналіз та інструменти для аналізу залежностей. Ретельно вивчено принципи роботи, функціональні можливості та особливості кожного інструменту. Виокремлено основні переваги та обмеження існуючих рішень, зокрема щодо їх інтеграції у процеси, швидкості виявлення помилок та точності результатів. На основі результатів проведеного аналізу надано рекомендації щодо оптимального вибору та налаштування інструментів для виявлення, а також пропозиції щодо їх вдосконалення для забезпечення більш ефективного виявлення та усунення уразливостей.*

Наукова новизна. *Великі мовні моделі та інші інструменти штучного інтелекту швидко привертають увагу та застосовуються в усьому світі. Наукова новизна цієї статті полягає в їх комплексному аналізі. У статті вперше проведено детальне порівняння різних типів інструментів для виявлення вразливостей, зокрема з акцентом на новітні рішення, які забезпечують більш точне виявлення вразливостей у сучасних технологіях програмування. Визначено актуальні проблеми, що виникають при використанні автоматизованих інструментів для виявлення вразливостей, зокрема, питання фальшивих спрацьовувань та інтеграційні складнощі, що не завжди висвітлюються в існуючих дослідженнях. Це дозволяє більш глибоко зрозуміти обмеження існуючих рішень і визначити напрямки їх вдосконалення.*

Висновки. *Результати цієї роботи демонструють значний потенціал використання машинного навчання для виявлення вразливостей програмного забезпечення безпосередньо на рівні вихідного коду. Створений набір даних, а також застосування методів машинного навчання, дозволили досягти високої точності в класифікації потенційних вразливостей. Було доведено, що можна значно покращити ефективність аналізу, скорочуючи час, необхідний для виявлення вразливостей, і дозволяючи швидше фокусуватися на потенційних проблемах.*

Ключові слова: *безпека систем, кібербезпека, вразливості, програмне забезпечення з відкритим кодом, автоматизація, тестування безпеки, методологія оцінки.*

Andrii BORYSENKO
Doctor of Philosophy, Junior Research Fellow, Institute of Transport Systems and Technologies of Ukrainian National Academy of Science, 5, Pisarzhevskogo Str., Dnipro, Ukraine, 49005
ORCID: *0009-0001-5083-5255*
Scopus Author ID: *58158469800*

To cite this article: Borysenko, A. (2024). Bezpechne prohramuvannia: avtomatyzovani instrumenty dlia vyiavlennia vrazlyvostei v kodi [Secure programming: automated tools for detecting vulnerabilities in code]. *Information Technology: Computer Science, Software Engineering and Cyber Security*, 4, 43–52, doi: https://doi.org/10.32782/IT/2024-4-6

SECURE PROGRAMMING: AUTOMATED TOOLS FOR DETECTING VULNERABILITIES IN CODE

The aim. *Vulnerabilities can pose a serious risk and lead to system breaches, information leaks or denial of service. This problem is exacerbated by the shortage of cybersecurity professionals. Automated tools and technologies can optimize the vulnerability detection process, making it more efficient. The purpose of the work is to analyze the effectiveness of existing solutions for detecting different types of vulnerabilities at different stages of software development, as well as to identify the advantages and disadvantages of each of the tools. The work is aimed at improving the accuracy and efficiency of vulnerability detection in real programming conditions.*

The methodology. *The existing automated tools for detecting vulnerabilities in code in the context of secure programming are studied. A comparative analysis of popular solutions, such as static code analyzers, dynamic analysis and dependency analysis tools, is conducted. The principles of operation, functionality and features of each tool are carefully studied. The main advantages and limitations of existing solutions are highlighted, in particular regarding their integration into processes, speed of error detection and accuracy of results. Based on the results of the analysis, recommendations are provided for the optimal selection and configuration of detection tools, as well as suggestions for their improvement to ensure more effective detection and elimination of vulnerabilities.*

Scientific novelty. *Large language models and other artificial intelligence tools are quickly attracting attention and are used worldwide. The scientific novelty of this article lies in their comprehensive analysis. The article provides the first detailed comparison of different types of vulnerability detection tools, in particular with an emphasis on the latest solutions that provide more accurate detection of vulnerabilities in modern programming technologies. Current problems that arise when using automated vulnerability detection tools are identified, in particular, the issue of false positives and integration difficulties that are not always covered in existing studies. This allows for a deeper understanding of the limitations of existing solutions and to identify areas for their improvement.*

Conclusions. *The results of this work demonstrate the significant potential of using machine learning to detect software vulnerabilities directly at the source code level. The dataset we created, as well as the application of machine learning methods, allowed us to achieve high accuracy in classifying potential vulnerabilities. It was proven that the efficiency of the analysis can be significantly improved, reducing the time required to detect vulnerabilities and allowing us to focus on potential problems faster.*

Key words: *systems security, cybersecurity, vulnerabilities, open-source software, automation, security testing, assessment methodology.*

Постановка проблеми. Події, пов'язані з кібербезпекою, часто висвітлюються в ЗМІ. Протягом багатьох років спостерігається помітне зростання глобальних кібератак. Дослідження, проведене компанією IBM (International Business Machines – один з найбільших в усьому світі виробників та постачальників апаратного та програмного забезпечення, а також IT-сервісів та консалтингових послуг) показало декілька цікавих ключових висновків. У 2023 році середня вартість одного порушення становила для організацій 4,45 мільйона доларів США. Це на 2,3% більше порівняно з 4,35 мільйона минулого року (Booth, 2023). Майбутні прогнози, отримані в іншому дослідженні, вказують на різке збільшення загальних глобальних витрат, пов'язаних з інцидентами кібербезпеки, за оцінками, глобальні витрати досягнуть 10,5 трильйонів доларів США щорічно до 2025 року. Це становить приблизно 28,8 мільярдів доларів США на добу, або 333 тисячі доларів США щосекунди (Check Point Blog, 2022). Завдання надання організаціям досвіду з кібербезпеки виходить за рамки простого пошуку спеціалістів із безпеки – це також передбачає пошук професіоналів, які мають необхідний досвід. Непривілейовані країни особливо сприйнятливі до викликів через недостатню інфраструктуру кібербезпеки, відсутність міжвідомчої координації та реагування на надзвичайні ситуації, обмежені навички використання інформаційно-комунікаційних технологій (ІКТ) і неадекватний захист критичної національної інфраструктури.

У боротьбі з вищезазначеними проблемами може допомогти автоматизація перевірок безпеки. Автоматизовані технології пропонують методичний підхід до цих перевірок і усувають потреби в підвищенні знань. Дуже важливо, щоб процес автоматизації призначав різні рівні серйозності для різних вразливостей безпеки. Характеристика вразливостей програмного забезпечення є критично важливим кроком у виявленні першопричин вразливості, розуміння її наслідків, механізмів атак і відповідних методів пом'якшення. Національний інститут стандартів та технологій США (НІСТ) розробив стандартизовану онтологію опису вразливостей (ООВ), яка визначає атрибути опису, необхідні для ефективного надання оперативної інформації розробникам програмного забезпечення з метою усунення/зменшення вразливості.

Рисунок 1 (Booth, 2016) демонструє різні атрибути онтології ООВ, що використовуються для характеристики вразливостей програмного забезпечення. Наприклад, категорія

характеристики методу впливу представляє, як можна використати вразливість. Характеристики в категорії відображають конкретні методи, які зловмисник може використати, щоб скористатися вразливістю: обхід автентифікації, збій довіри, вихід із контексту, атака «людина посередині» та виконання коду. Інші характеристики вказують наслідки, домен, місце та засоби пом'якшення використання вразливостей. Зацікавлені сторони та постраждалі користувачі вразливого продукту покладаються на характерні дані, щоб визначити, як експлойт впливає на їхні системи та як запобігти експлуатації.

На жаль, визначення характеристик уразливості, як описано в OOB, є ручним, трудомістким і асинхронним процесом.

Правильна характеристика вразливості вимагає перегляду її описів з достатнім досвідом безпеки та знайомства з онтологією. Цей інтенсивний процес призвів до проблем із якістю звітів про вразливості. На рисунку 2 показано графічне представлення зв'язку між скануванням та ідентифікацію всіх пристроїв, оцінкою вразливостей та генеруванням детальних звітів, що підсумовують результати. Дослідження виявили, що звіти про вразливості НІСТ часто залишаються неповними або не мають конкретних характеристик.

Аналіз останніх досліджень і публікацій. Останні дослідження в галузі безпечного програмування та автоматизованих інструментів для виявлення вразливостей в коді показують значний прогрес у розробці інструментів, які не тільки покращують ефективність виявлення вразливостей, але й інтегруються в сучасні методи розробки програмного забезпечення, забезпечуючи більш високий рівень автоматизації і точності. Зокрема, наукові публікації останніх років зосереджуються на кількох ключових напрямках, зокрема на застосуванні машинного навчання, статичному та динамічному аналізі коду, а також на інтеграції інструментів у CI/CD процеси (процеси неперервної інтеграції та доставки).

Використання машинного навчання в контексті виявлення вразливостей набуває популярності. Дослідження, показують, що машинне навчання може значно підвищити точність автоматизованих інструментів для виявлення вразливостей, здатних адаптуватися до нових типів загроз. Алгоритми на основі нейронних мереж та класифікації, навчання з підкріпленням та кластеризації дозволяють значно поліпшити прогнозування потенційних точок входу для атак. Вони здатні виявляти не лише відомі, але й нові, невідомі вразливості, які раніше могли залишатися поза увагою традиційних інструментів. Особливо популярними є методи, які застосовуються до статичного аналізу: створення моделей на основі вихідного коду, які можуть передбачати потенційні вразливості, аналізуючи синтаксис і логіку програм.

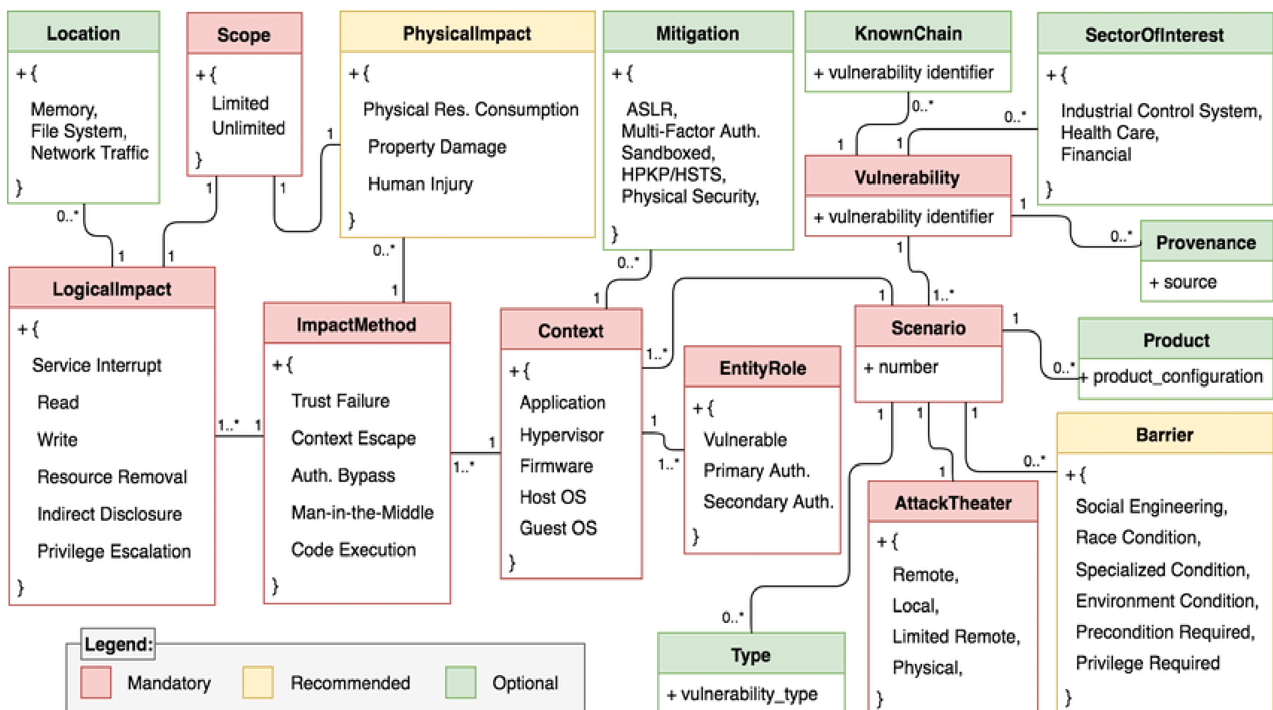

Рис. 1. Онтологія опису вразливості НІСТ (OOB)

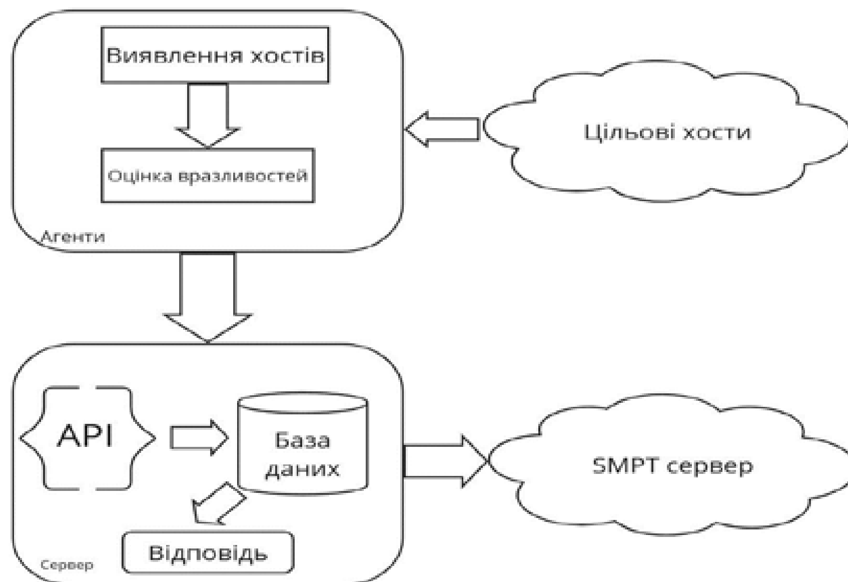

Рис. 2. Графічне представлення зв'язку між модулями оцінки вразливості безпеки (стрілки зображують потік даних)

Використання неструктурованих даних, таких як репозиторії коду, разом з технологіями глибокого навчання дозволяє створювати системи, що більш ефективно виявляють патерни, схожі на вразливості, ніж стандартні алгоритми аналізу (Elmrabit, 2020).

Статичний та динамічний аналіз коду залишаються двома основними підходами для виявлення вразливостей. Останні дослідження демонструють ефективність комбінованого використання обох підходів для більш повного аналізу програмного забезпечення. Статичні інструменти, такі як SonarQube, Checkmarx, та Clang Static Analyzer, стали стандартом для виявлення помилок на етапі розробки. Вони фокусуються на пошуку вразливостей в синтаксисі та структурі коду, що дозволяє розробникам своєчасно виправляти потенційно небезпечні фрагменти.

Водночас, динамічний аналіз, зокрема з використанням інструментів для тестування на проникнення, таких як OWASP ZAP та Burp Suite, зосереджує увагу на перевірці коду в процесі його виконання. Останні дослідження також показують, що інтеграція цих інструментів у сучасні пайплайни CI/CD забезпечує більш швидку верифікацію коду в реальному часі, дозволяючи автоматично виявляти уразливості в тестових середовищах.

Дослідження доводять, що поєднання статичного і динамічного аналізу дає змогу знижувати ймовірність виявлення помилок лише на одному етапі тестування, забезпечуючи багатогранний підхід до безпеки (Acharya, 2023).

Дослідження показують, що інтеграція інструментів виявлення вразливостей у процеси CI/CD стає стандартом практики для безпечного програмування. Наприклад, використання таких інструментів, як GitLab CI/CD, разом із автоматизованими системами для перевірки коду, дає можливість здійснювати регулярні перевірки безпеки на всіх етапах розробки, зменшуючи ймовірність помилок, що потрапляють у кінцевий продукт.

У статтях дослідники вказують, що автоматизація перевірок на етапах розробки й деплою не тільки підвищує якість програмного забезпечення, а й дозволяє виявляти вразливості ще до потрапляння коду у продуктивне середовище. Інтеграція таких інструментів дозволяє швидше реагувати на загрози, знижуючи час, необхідний для виявлення та виправлення вразливостей (Smith, 2023).

У Україні також активно проводяться дослідження в сфері безпечного програмування та автоматизованих інструментів для виявлення вразливостей в коді. Наприклад, українські науковці в своїх дослідженнях використовують нейронні мережі для аналізу великих обсягів коду в реальному часі, що допомагає виявляти нові типи вразливостей і навіть передбачати можливі атаки. Окрім академічних досліджень, низка українських IT-компаній займається розробкою та вдосконаленням автоматизованих інструментів для виявлення вразливостей у програмному забезпеченні. Ці дослідження свідчать про значний прогрес в Україні в галузі безпечного програмування і автоматизованого

виявлення вразливостей. Вчені й фахівці вітчизняних університетів і компаній не тільки розробляють нові методи та інструменти для покращення безпеки, але й активно інтегрують їх у реальні процеси розробки програмного забезпечення.

Таким чином, сучасні методи виявлення вразливостей включають комбінований підхід з використанням різних технік, інтегрованих у розробницькі пайплайни, але, враховуючи думки різних авторів, доводиться констатувати, що в умовах постійного впливу зовнішніх та внутрішніх чинників необхідно приділити більше уваги дослідженню проблем безпеці і це питання є актуальним.

Мета дослідження. Метою статті є надання детального аналізу існуючих інструментів і методів автоматизованого виявлення вразливостей у програмному забезпеченні, охоплюючи як класичні підходи (статичний та динамічний аналіз коду), так і новітні технології, зокрема ті, що використовують машинне навчання і штучний інтелект для підвищення точності виявлення вразливостей. Дана стаття має на меті показати, як інструменти для автоматизованого виявлення вразливостей можуть бути інтегровані у процеси розробки програмного забезпечення. Також важливим є допомогти підвищити рівень кібербезпеки в Україні через популяризацію кращих практик безпечного програмування і розвиток автоматизованих інструментів для виявлення вразливостей.

Для досягнення поставленої мети треба виконати наступні завдання:

– Провести детальний огляд сучасних наукових публікацій, досліджень та технічних звітів з теми безпечного програмування та автоматизованих інструментів для виявлення вразливостей.

– Оцінити ефективність основних підходів до виявлення вразливостей у коді, зокрема статичний і динамічний аналіз, а також нові технології, такі як машинне навчання.

– Проаналізувати майбутні тенденції в розвитку автоматизованих інструментів для виявлення вразливостей і визначити напрямки, які потребують подальшого розвитку.

– Підсумувати основні висновки та зробити рекомендації для практичного використання автоматизованих інструментів для виявлення вразливостей.

Виклад основного матеріалу дослідження. Наразі існує широкий спектр інструментів статичного аналізу, які намагаються виявити типові вразливості. Вони варіюються від реалізацій з відкритим кодом, таких як статичний аналізатор Clang, до пропрієтарних рішень (Kremenek, 2018). Статичні аналізатори роблять це без необхідності виконувати програми. Динамічні аналізатори неодноразово виконують програми з багатьма тестовими входами на реальних або віртуальних процесорах, щоб виявити слабкі місця. Як статичні, так і динамічні аналізатори є інструментами, заснованими на правилах, тому вони обмежені правилами, створеними вручну, і не можуть гарантувати повне тестове покриття кодових баз.

Щоб виявити вразливості в коді було розглянуто два взаємодоповнюючі підходи. Перший використовує функції, отримані з проміжного представлення програм, створених під час процесу компіляції та збірки. Другий працює безпосередньо з вихідним кодом. Ці два підходи пропонують незалежні джерела інформації. Моделі, засновані на вихідному коді, можуть вивчати статистичні кореляції в тому, як пишеться код, використовуючи методи обробки природної мови, але не мають жодних внутрішніх знань про структуру чи семантику вбудованої мови (все це потрібно вивчати з даних). Моделі на основі збірки використовують переваги функцій, отриманих від компілятора, який розуміє структуру мови, але ці функції можуть абстрагувати певні властивості коду, корисні для виявлення вразливостей. Застосовуючи обидва підходи, ми дозволяємо зрештою об’єднати кілька моделей для підвищення ефективності виявлення вразливостей.

Враховуючи складність і різноманітність програм, потрібна велика кількість навчальних прикладів для навчання моделей машинного навчання, які можуть ефективно вивчати шаблони вразливостей безпеки безпосередньо з коду (Gonzalez, 2019). Було зібрано величезний набір даних із мільйонів прикладів функціонального рівня коду C і C++ із SATE IV, дистрибутива Debian Linux і загальнодоступних репозиторіїв Git на GitHub. У таблиці 1 показано зведені дані про кількість функцій, які було зібрано та використано з кожного джерела в наборі даних із понад 12 мільйонів функцій.

Хоча набір даних SATE IV надає позначені приклади багатьох типів вразливостей, він складається з фрагментів синтетичного коду, які недостатньо охоплюють простір природного коду, щоб забезпечити лише відповідний навчальний набір. Випуски пакетів Debian надають вибір дуже добре керованого та підібраного коду, який сьогодні використовується в багатьох системах. Набір даних GitHub надає більшу кількість і більш різноманітний (часто нижчої якості) код.

Таблиця 1

Зведені дані аналізу функцій

	SATE IV	GitHub	Debian
Всього	121,353	9,706,269	3,046,758
Пройшли перевірку	11,896	782,493	491,873
Не вразливі	6,503 (55%)	730,160 (93%)	461,795 (94%)
Вразливі	5,393 (45%)	52,333 (7%)	30,078 (6%)

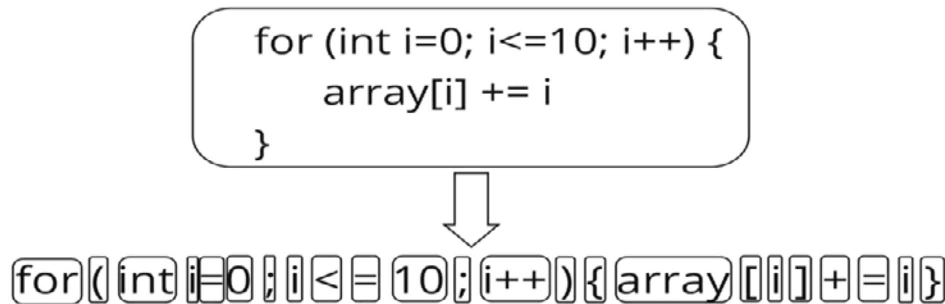

Рис. 3. Приклад, що ілюструє процес лексичного складання

Було реалізовано спеціальний лексер С/С++, який аналізує код і класифікує елементи в різні контейнери: коментарі, рядкові літерали, літерали з одним символом, літерали з кількома символами, числа, оператори, директиви попереднього компілятора та імена. Назви далі поділені на ключові слова (такі як for, if тощо), виклики системних функцій (такі як malloc), типи (такі як bool, int тощо) або імена змінних (Aminu, 2024). Усі назви змінних зіставляються з тим самим загальним ідентифікатором, але кожне унікальне ім’я змінної в одній функції отримує окремий індекс (з метою відстеження того, де змінні знову з’являються). Це відображення змінюється від функції до функції залежно від позиції, у якій ідентифікатор з’являється вперше. Унікальні рядкові літерали, з іншого боку, відображаються на один і той самий ідентифікатор. Нарешті, цілі літерали зчитаються цифра за цифрою, тоді як літерали з плаваючою речовиною відображаються на загальний маркер. На рисунку 2 показаний приклад процесу лексики.

Розроблений лексер зміг скоротити код С/С++ до представлень, використовуючи загальний розмір словника лише 156 токенів. Також одним з дуже важливих кроків підготовки даних було видалення потенційних дублюючих функцій. Сховища з відкритим вихідним кодом часто мають функції, що дублюються в різних пакетах. Таке дублювання може штучно завищити показники продуктивності та приховати повторні визначення, оскільки навчальні дані можуть витікати в тестові набори. Так само є багато функцій, які є майже дублікатами, містять тривіальні зміни у вихідному коді, які суттєво не впливають на виконання функції. Ці майже дублікати важко видалити, оскільки вони часто можуть з’являтися в дуже різних частинах коду та можуть виглядати зовсім по-різному на рівні вихідного коду.

Щоб захиститися від цих проблем, було виконано надзвичайно суворий процес видалення дублікатів. Була видалена будь-яка функцію з дубльованим лексичним представленням вихідного коду або дубльованим вектором функцій на рівні компіляції. Цей вектор функцій на рівні компіляції було створено шляхом вилучення графа потоку керування функції, а також операцій, що відбуваються в кожному базовому блоці (opcode vector, чи op-vec), а також визначення та використання змінних (матриця use-def).

Рядок «Прохідна перевірка» у таблиці 1 відображає кількість функцій, що залишилися після процесу видалення дублікатів, приблизно 10,8% від загальної кількості вилучених функцій. Хоча наш жорсткий процес видалення дублікатів відфільтровує значну кількість даних, цей підхід забезпечує найбільш консервативні результати продуктивності, точно оцінюючи, наскільки добре наш інструмент працюватиме з кодом, якого він ніколи раніше не бачив.

Позначення вразливості коду на функціональному рівні було серйозною проблемою. Щоб створити мітки, було використано три підходи: статичний аналіз, динамічний аналіз і додавання тегів до повідомлень/звітів про помилки.

Незважаючи на те, що динамічний аналіз здатний виявляти тонкі недоліки, виконуючи функції з широким діапазоном можливих вхідних даних, він надзвичайно ресурсомісткий, тому цей підхід не був реалістичним для надзвичайно великого набору даних. Виявилося, що маркування на основі комітів-повідомлень є дуже складним завданням, оскільки надає мітки низької якості. Також був опробований простий пошук за ключовими словами, шукаючи такі слова фіксації, як «buggy», «broken», «error», «fixed», тощо, щоб позначити пари функцій до та після, що дало кращі результати з точки зору релевантності. Однак цей підхід значно зменшив кількість функцій-кандидатів і все ще вимагав значної перевірки вручну, що робило його невідповідним для величезного набору даних. У результаті було вирішенно використовувати три статичні аналізатори з відкритим вихідним кодом, Clang, Cppcheck і Flawfinder (Zhou, 2019), щоб генерувати мітки. Кожен статичний аналізатор різниться за сферою пошуку та виявлення. Наприклад, сфера дії Clang дуже широка, але також включає синтаксис, стиль програмування та інші висновки, які навряд чи призведуть до вразливості. Сфера дії Flawfinder орієнтована на загальний перелік вразливостей і не зосереджується на інших аспектах, таких як стиль. Тому було включено кілька статичних аналізаторів і скоротили їхні результати, щоб виключити знахідки, які зазвичай не пов’язані з уразливими місцями безпеки, щоб створити надійні мітки «вразливий» і «не вразливий» залежно від загального переліку вразливостей. Із 390 загальних типів результатів статичних аналізаторів було визначено, що 149 призводять до потенційної вразливості безпеки. Приблизно 6,8% наших підібраних видобутих функцій C/C++ викликали виявлення вразливості. Таблиця 2 показує статистику частих вразливостей у цих функціях.

При побудові моделей машинного навчання для виявлення помилкових функцій було розглянуто дві цілі. По-перше, оцінка загально можливої продуктивності – наскільки добре кожна модель може передбачити наявність небезпечних методів у коді. По-друге, порівняння продуктивності функцій на основі складання та на основі вихідного коду, щоб виявити, який набір краще прогнозує якість коду.

Реалізований підхід машинного навчання до виявлення вразливостей поєднує представлення нейронних функцій вихідного коду лексичної функції з потужним класифікатором ансамблю, випадковим лісом.

Використано підходи вилучення ознак, подібні до тих, що використовуються для класифікації настрою речення за допомогою згорткових нейронних мереж (CNN) і рекурентних нейронних мереж (RNN) для класифікації вразливості джерела на функціональному рівні:

1) Вбудовування функцій: маркери, що утворюють лексифіковані функції, спочатку вбудовуються у фіксоване k-вимірне представлення (обмежене діапазоном [–1, 1]), яке вивчається під час навчання класифікації через зворотне поширення до лінійного перетворення одноразового вбудовування. Оскільки розроблений словниковий запас набагато менший, ніж словниковий запас природних мов, була змога використати набагато менше вбудовування, ніж це типово для додатків NLP. Експерименти показали, що k = 13 є найкращим для контрольованих розмірів вбудовування, врівноважуючи виразність вбудовування та надмірне припасування.

2) Вилучення функцій: було досліджено як CNN, так і RNN для вилучення функцій із вбудованих представлень джерела. Вилучення згорткових ознак: використовуємо n згорткових фільтрів із формою m × k, тому кожен фільтр охоплює повний простір вбудовування маркера. Розмір фільтра m визначає кількість послідовних маркерів, які розглядаються разом, і було виявлено, що досить великий розмір фільтра m = 9 працює найкраще. Загальна кількість n = 512 фільтрів у поєднанні з пакетною нормалізацією з наступною ReLU була найбільш ефективною.

3) Об’єднання: оскільки довжина функцій C/C++, може різко змінюватися, як згорткові, так

Таблиця 2

Статистика щодо вразливостей, виявлених у аналізованому наборі даних C/C++

ID вразливості	Опис вразливості	Частота, %
120/121/122	Переповнення буфера	38.2%
119	Неправильне обмеження операцій у межах буфера пам’яті	18.9%
476	Розіменування покажчика NULL	9.5%
469	Використання віднімання вказівника для визначення розміру	2.0%
20, 457, 805, тощо	Неналежна перевірка введення, використання неініціалізованої змінної, доступ до буфера з неправильним значенням довжини, тощо	31.4%

і рекурентні функції максимально об'єднуються вздовж довжини послідовності, щоб створити представлення фіксованого розміру (n або n відповідно). У цій архітектурі рівні вилучення функцій повинні навчитися визначати різні сигнали вразливості, тому наявність будь-якого з них у послідовності є важливою.

4) Щільні шари: за шарами вилучення ознак слідує повний зв'язаний класифікатор. Під час навчання було використано 50% випадання з'єднань максимального об'єднаного представлення функцій до першого прихованого шару. Було виявлено, що використання двох прихованих шарів 64 і 16 перед остаточним вихідним шаром softmax дало найкращу ефективність класифікації.

5) Навчання: для зручності пакетування даних було проведено тренування лише на функціях із довжиною маркера 10 ≤ ≤ 500, доповненою до максимальної довжини 500. Оскільки набір даних був сильно незбалансованим, уразливі функції були важливіші у функції втрати.

Було виявлено, що використання нейронних особливостей дало найкращі результати, окрема оптимізація функцій і класифікатора, здається, допомогла протистояти переобладнанню. Цей підхід також робить зручнішим швидко перенавчати класифікатор на нові набори ознак або комбінації ознак.

Перевірка важливості функції пакета слів показує, що класифікатор використовує кореляції міток із індикаторами довжини та складності джерела та комбінаціями викликів, які зазвичай зловживають і призводять до вразливостей. Покращення в порівнянні з базовою лінією можна інтерпретувати як результат більш складних і специфічних моделей індикації вразливості. Загалом розроблені моделі CNN показали кращі результати, ніж моделі RNN як автономні класифікатори та генератори ознак. Крім того, CNN швидше навчалися і вимагали набагато менше параметрів.

Розроблені методи машинного навчання мають деякі додаткові переваги порівняно з традиційними інструментами статичного аналізу. Розроблені користувальницькі лексери та моделі машинного навчання можуть швидко переробляти та оцінювати великі сховища та вихідний код без необхідності компілювати код. Крім того, оскільки машинне навчання обробляє всі вихідні ймовірності, порогові значення можна налаштувати для досягнення бажаної точності та запам'ятовування.

З іншого боку, статичні аналізатори повертають фіксовану кількість результатів, яка може бути надзвичайно великою для величезних кодових баз або занадто малою для критичних програм. У той час як статичні аналізатори можуть краще локалізувати вразливості, які вони знаходять, нами також було використано методи візуалізації, такі як карта активації функції, показана на рисунку 4, щоб допомогти зрозуміти, чому наші алгоритми приймають свої рішення.

Висновки і перспективи подальших досліджень. Було продемонстровано потенціал використання машинного навчання для виявлення вразливостей програмного забезпечення безпосередньо з вихідного коду. Для цього було створено обширний набір даних вихідного коду C/C++, видобутий із сховищ Debian і GitHub, позначений підібраними

```
wchar_t * data;
unionType myUnion;
data = new wchar_t[100];
wmemset(data, L 'A', 100 - 1);
data[100 - 1] = L '\0';
myUnion.unionFirst = data;
{
    wchar_t * data = myUnion.unionSecond;
    {
        wchar_t dest[50] = L "";
        memcpy(dest, data, wcslen(data) * sizeof(wchar_t));
        dest[50 - 1] = L '\0';
        printWLine(data);
        delete[] data;
    }
}
```

Рис. 4. Знімок екрана для інтерактивної демонстрації виявлення вразливостей

виявленими вразливостей із набору інструментів статичного аналізу, і об’єднанням його з набором даних SATE IV. Було розроблено власний лексер C/C++, щоб створити просте загальне представлення вихідного коду функції, ідеальне для машинного навчання. Було застосовано різні методи машинного навчання, які були точно налаштовані для розробленої програми та досягли найкращих загальних результатів, використовуючи функції, отримані за допомогою нейронної мережі. Пройдено до висновків, що підхід, керований даними, може доповнити існуючі інструменти статичного аналізу та скоротити кількість часу, необхідного для виявлення потенційних вразливостей у програмному забезпеченні. Ранжуючи функції за ймовірністю помилок, розробники зможуть швидше завершити перевірку коду та виділити потенційні проблеми, які, можливо, не були виявлені інакше.

Майбутня робота має бути зосереджена на вдосконаленні міток, таких як мітки з інструментів динамічного аналізу або видобуті з патчів безпеки. Це дозволить оцінкам, створеним на основі моделей машинного навчання, більше доповнювати інструменти статичного аналізу. Методи автоматизації, розроблені в цій роботі для навчання безпосередньо на вихідному коді функції, також можуть бути застосовані до будь-якої проблеми класифікації коду, такої як виявлення порушень стилю, категоризація комітів або класифікація алгоритмів/завдань. У міру розробки більших і краще позначених наборів даних глибоке навчання для аналізу вихідного коду стане більш практичним для широкого спектру важливих проблем.

ЛІТЕРАТУРА:

1. B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, «Learning deep features for discriminative localization,» in Computer Vision and Pattern Recognition (CVPR), pp. 2921–2929, IEEE, 2019.

2. Check Point Blog. Check Point Research: Third Quarter of 2022 Reveals Increase in Cyberattacks and Unexpected Developments in Global Trends. checkpoint.com. URL: https://blog.checkpoint.com/2022/10/26/third-quarter-of-2022-revealsincrease-in-cyberattacks/

3. D. Gonzalez, F. Alhenaki, and M. Mirakhorli. Architectural security weaknesses in industrial control systems (ICS) an empirical study based on disclosed software vulnerabilities. In 2019 IEEE International Conference on Software Architecture (ICSA), pages 31–40, March 2019.

4. G. Smith. The intelligent solution: Automation, the skills shortage and cyber-security. Comput. Fraud. Secur, 2018, 6–9.

5. H. Booth, D. Rike H. Booth, D. Rike, and G. Witte. The National Vulnerability Database (NVD): Overview. Technical report, National Institute of Standards and Technology (NIST), 2021.

6. H. Booth. Draft NISTIR 8138, Vulnerability Description Ontology (VDO). Technical report, National Institute of Standards and Technology (NIST), 2016.

7. M. Aminu, S. Anawansedo, Y. Sodiq. Driving Technological Innovation for a Resilient Cybersecurity Landscape. *International Journal of Latest Technology in Engineering, Management & Applied Science*, 13(4), 126–133, 2024.

8. N. Elmrabit, F. Zhou, F. Li. Evaluation of Machine Learning Algorithms for Anomaly Detection. In Proceedings of the 2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security), Dublin, Ireland, 15–19 June 2020; pp. 1–8.

9. S. Acharya, U. Rawat, R. Bhatnagar. A Comprehensive Review of Security: Treats, Vulnerabilities, MalwareDetection, and Analysis. In Security and Communication Net-works, 34 pages, 2023.

10. Z. Kremenek, T. Zhang. A memory model for static analysis of c programs. In International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (2018), Springer, pp. 535–548.

REFERENCES:

1. Zhou, B., Khosla, A., Lapedriza, A., Oliva, A. and Torralba, A. (2019). «Learning deep features for discriminative localization,» in Computer Vision and Pattern Recognition (CVPR), pp. 2921–2929, IEEE,

2. Check Point Blog. Check Point Research: Third Quarter of 2022 Reveals Increase in Cyberattacks and Unexpected Developments in Global Trends. checkpoint.com. Available online: https://blog.checkpoint.com/2022/10/26/third-quarter-of-2022-revealsincrease-in-cyberattacks/ (accessed on 2 February 2024).

3. Gonzalez, D., Alhenaki, F. and Mirakhorli, M. (March 2019). Architectural security weaknesses in industrial control systems (ICS) an empirical study based on disclosed software vulnerabilities. In 2019 IEEE International Conference on Software Architecture (ICSA), pages 31–40,

4. Smith, G. (2018) The intelligent solution: Automation, the skills shortage and cyber-security. Comput. Fraud. Secur, 6–9.

5. Booth, H., Rike, D., Booth, H., Rike, D. and Witte, G. (2021). The National Vulnerability Database (NVD): Overview. Technical report, National Institute of Standards and Technology (NIST),

6. Booth, H. (2016). Draft NISTIR 8138, Vulnerability Description Ontology (VDO). Technical report, National Institute of Standards and Technology (NIST),

7. Aminu, M. Anawansedo, S. Sodiq, Y. (2024). Driving Technological Innovation for a Resilient Cybersecurity Landscape. *International Journal of Latest Technology in Engineering, Management & Applied Science*, 13(4), 126–133,

8. Elmrabit, N. Zhou, F. Li, F. (2020). Evaluation of Machine Learning Algorithms for Anomaly Detection. In Proceedings of the 2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security), Dublin, Ireland, 15–19 June; pp. 1–8.

9. Acharya, S. Rawat, U. Bhatnagar, R. (2023). A Comprehensive Review of Security: Treats, Vulnerabilities, MalwareDetection, and Analysis. In Security and Communication Net-works, 34 pages,

10. Kremenek, Z., Zhang, T. (2018). A memory model for static analysis of c programs. In International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, Springer, pp. 535–548.