МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«МЕТОДИ ТАІНТ-АНАЛІЗУ ДЛЯ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ У PYTHON-КОДІ»

Виконав: студент 2 курсу

рівень магістр

галузь знань 12 «Інформаційні технології»

спеціальність 122 «Комп’ютерні науки»

Колісніченко Руслан Леонідович

Керівник: к.пед.н., доцент

Логінова Наталія Іванівна

Рецензент: к.т.н., доцент

Гура Володимир Ігорович

Одеса – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

Факультет **кібербезпеки та інформаційних технологій**
Кафедра **інформаційних технологій**
Галузь знань: **12 Інформаційні технології**
Спеціальність: **122 Комп'ютерні науки**
Назва освітньої програми: **Комп'ютерні науки**

ЗАТВЕРДЖУЮ

Завідувач кафедри інформаційних технологій
доцент _____Н.І. Логінова
«____»_____20__року

ЗАВДАННЯ

на кваліфікаційну (магістерську) роботу здобувачу Колісніченко Руслану Леонідовичу

1. Тема роботи: «Методи таінт-аналізу для виявлення вразливостей у Python-коді»

Керівник роботи: к.пед.н, доцент Логінова Наталія Іванівна
затверджені наказом НУ ОЮА № 3182-06 від «10» жовтня 2025 року

2. Строк подання здобувачем роботи «25» листопада 2025 року
3. Дата видачі завдання «02» червня 2025 рік

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1.	Вибір теми, вивчення літературних джерел та складання плану роботи		
2.	Підготовка першого розділу роботи та подання його керівнику		
3.	Підготовка другого розділу роботи та подання його керівнику		
4.	Підготовка третього розділу роботи та подання його керівнику		
5.	Підготовка вступу та висновків		
6.	Доопрацювання роботи з урахуванням зауважень керівника		
7.	Подача електронного варіанту роботи для перевірки на плагіат		
8.	Допуск роботи до захисту завідуючим кафедри		
9.	Захист роботи		

Здобувач _____
(підпис) (прізвище та ініціали)

Керівник роботи _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота студента 2-го курсу магістратури Колісніченка Руслана Леонідовича на тему «Методи таінт-аналізу для виявлення вразливостей у Python-коді» на здобуття другого (магістерського) рівня вищої освіти за спеціальністю 122 «Комп'ютерні науки», освітньою програмою «Комп'ютерні науки».

Робота виконана на 50 сторінках, містить 20 рисунків, 1 таблицю. Перелік посилань включає 34 джерела.

Метою дослідження є розробка та вдосконалення методів таінт-аналізу для виявлення вразливостей у Python-коді з використанням технологій статичного аналізу та машинного навчання.

Об'єктом дослідження є процеси аналізу безпеки вихідного коду програмних систем, зокрема Python-застосунків.

Предметом дослідження є методи таінт-аналізу для виявлення поширення "заражених" даних у Python-програмах.

У кваліфікаційній роботі проведено аналіз існуючих методів виявлення вразливостей у Python-програмах, зокрема статичних, динамічних і гібридних підходів. Розглянуто теоретичні основи таінт-аналізу, побудовано модель розповсюдження даних між джерелами (sources) і приймачами (sinks), формалізовано алгоритми для автоматизованої перевірки коду. Особливу увагу приділено поєднанню класичних методів аналізу з технологіями машинного навчання для класифікації потенційних загроз. На основі проведеного дослідження створено програмний інструмент simple_taint.py, який виконує виявлення небезпечних потоків даних у Python-коді.

Результати тестування підтвердили коректність роботи інструменту на прикладних Python-фрагментах та його здатність виявляти типові уразливості.

ТАІНТ-АНАЛІЗ, PYTHON, AST, УРАЗЛИВОСТІ, SQL INJECTION, XSS, СТАТИЧНИЙ АНАЛІЗ, МАШИННЕ НАВЧАННЯ, БЕЗПЕКА КОДУ

ABSTRACT

Qualification work of the 2nd year master's student Kolisnichenko Ruslan Leonidovych on the topic "Taint Analysis Methods for Detecting Vulnerabilities in Python Code" for obtaining the second (master's) level of higher education in the specialty 122 "Computer Science", educational program "Computer Science".

The work is completed on 50 pages, contains 20 figures, 1 table. The list of references includes 34 sources.

The purpose of the study is to develop and improve taint analysis methods for detecting vulnerabilities in Python code using static analysis and machine learning techniques.

The object of the study is the process of source code security analysis in software systems, particularly Python-based applications.

The subject of the study is taint analysis methods for detecting the propagation of untrusted data in Python programs.

The qualification work analyzes existing approaches to vulnerability detection in Python, including static, dynamic, and hybrid methods. Theoretical foundations of taint analysis are examined, and a data propagation model between sources and sinks is developed. Formal algorithms for automated code inspection are presented. Special attention is paid to combining classical data-flow analysis with machine learning techniques to classify potential threats. As a result, a practical tool simple_taint.py was implemented to detect unsafe data flows in Python code.

The testing results confirmed the correct operation of the tool on application Python fragments and its ability to detect typical vulnerabilities.

TAINT ANALYSIS, PYTHON, AST, VULNERABILITIES, SQL INJECTION, XSS, STATIC ANALYSIS, MACHINE LEARNING, CODE SECURITY

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AST (Abstract Syntax Tree) – абстрактне синтаксичне дерево, структура, що використовується для подання програми у вигляді вузлів і зв’язків між ними.

Taint – «заражені» або недовірені дані, які походять з неперевірених джерел і можуть спричинити вразливість.

Taint Source – джерело потенційно небезпечних даних (input, параметри HTTP-запиту, дані з файлів, змінні середовища тощо).

Taint Sink – критична точка програми, куди потрапляння заражених даних може призвести до виконання атаки (os.system(), eval(), cursor.execute()).

Data Flow – потік даних між змінними та функціями у програмі.

Taint Propagation – процес передавання заражених даних між змінними та структурами програми.

Static Analysis – аналіз програмного коду без його виконання.

Dynamic Analysis – аналіз програмного коду під час його виконання.

ML (Machine Learning) – машинне навчання, підхід до автоматичного виявлення вразливостей за допомогою класифікації шаблонів і поведінки коду.

CodeQL – інструмент статичного аналізу, що дозволяє описувати правила пошуку уразливостей за допомогою спеціальної мови запитів.

Data Dependency Graph – граф залежностей даних, який відображає передачу значень між елементами програми.

Python AST Module – стандартний модуль Python, що дозволяє парсити код і будувати його AST-представлення.

NodeVisitor – базовий клас в Python, який використовується для обходу вузлів AST.

Tainted Variable – змінна, що отримала значення з джерела і вважається потенційно небезпечною.

Vulnerability – вразливість у програмному забезпеченні, яка дозволяє зловмиснику здійснити атаку.

Command Injection – тип атаки, за якої зловмисник може виконати системні команди.

SQL Injection – атака, що дозволяє вставити шкідливий SQL-код у запит до бази даних.

Sanitization – процес очищення та перевірки введених даних, що запобігає їх небезпечному впливу на програму.

Interpreter – середовище виконання коду Python.

PyCharm – середовище розробки, у якому виконувалось тестування інструменту.

ВСТУП

Актуальність дослідження. У сучасних умовах стрімкого розвитку цифрових технологій та масового переходу бізнесу, державних установ і приватних користувачів до електронних сервісів, безпека програмного забезпечення стає критично важливим аспектом функціонування інформаційних систем. На фоні зростання кількості кіберінцидентів та уразливостей у програмних продуктах особливе значення набуває аналіз безпеки вихідного коду, оскільки саме на цьому рівні формується основа стійкості системи до зовнішніх атак.

Python є однією з найпопулярніших мов програмування у галузях машинного навчання, розробки вебзастосунків, автоматизації, аналізу даних та кібербезпеки. Завдяки своїй простоті, динамічній типізації та великій кількості бібліотек Python широко використовується у системах, що працюють з даними користувачів та виконують критично важливі функції. Проте ці ж особливості створюють підвищені ризики виникнення вразливостей, пов’язаних з неконтрольованим потоком даних, небезпечними викликами функцій та динамічним виконанням коду.

Одним із найбільш ефективних підходів статичного аналізу безпеки є **таінт-аналіз** (taint analysis) – метод, що дозволяє відстежувати розповсюдження потенційно небезпечних ("заражених") даних від джерел отримання до критичних точок виконання програми. Саме цей підхід використовується для виявлення таких поширених атак, як SQL-ін’єкції, Remote Code Execution (RCE), Command Injection, XSS та інші види загроз, які можуть призвести до витоку інформації, порушення цілісності даних або повного компрометування системи.

Існуючі інструменти – такі як Bandit, Semgrep, PyT, CodeQL – частково вирішують проблему, проте мають низку обмежень: недостатню гнучкість, складність налаштування, обмежену адаптацію під специфіку проєктів або високу складність інтерпретації результатів. Це формує потребу у створенні власного програмного засобу, який продемонструє фундаментальні принципи таінт-аналізу

саме для Python-коду, буде прозорим, зрозумілим, легко модифікованим та орієнтованим на навчальні та дослідницькі цілі.

Мета кваліфікаційної роботи: розробити та дослідити інструмент статичного таінт-аналізу для автоматичного виявлення потенційних вразливостей у Python-коді на основі механізмів аналізу абстрактного синтаксичного дерева (AST) та правил розповсюдження небезпечних даних.

Для досягнення мети необхідно виконати такі **завдання**:

1. Проаналізувати сучасні методи забезпечення безпеки програмного забезпечення, зокрема статичного та динамічного аналізу коду.

2. Дослідити теоретичні основи таінт-аналізу, моделі поширення заражених даних та класифікацію джерел (sources) і приймачів (sinks).

3. Формалізувати алгоритми таінт-аналізу та адаптувати їх для Python з урахуванням особливостей мови.

4. Розробити інструмент статичного аналізу, що використовує AST для відстеження потоку даних.

5. Провести тестування інструменту на прикладах Python-коду та оцінити ефективність виявлення типових уразливостей.

Об’єкт дослідження – процес виявлення та запобігання вразливостям у Python-коді на етапі статичного аналізу.

Предмет дослідження – методи таінт-аналізу, правила розповсюдження небезпечних даних та принципи побудови інструментів статичного аналізу з використанням AST.

Методи дослідження, У роботі використано методи статичного аналізу коду, аналізу даних, моделювання потоків даних, вивчення структури абстрактного синтаксичного дерева, порівняльний аналіз, експериментальне тестування, а також елементи машинного навчання для класифікації потенційних вразливостей.

Наукова новизна роботи полягає в тому, що класичний таінт-аналіз було пристосовано до специфіки мови Python, створено легкий і прозорий інструмент статичного аналізу, орієнтований на дослідницькі та освітні цілі, а також запропоновано підхід до гібридного аналізу, де традиційні правила поєднуються з

машинними критеріями. Окремо систематизовано джерела та приймачі потенційно небезпечних даних у контексті Python-додатків.

Практична цінність полягає в тому, що розроблений інструмент може слугувати основою для навчання принципам статичного аналізу, використовуватися у процесах аудиту коду на Python, стати платформою для створення більш розширених аналітичних рішень, підтримувати дослідження у сфері безпеки програмного забезпечення та допомагати у реальних проєктах швидко виявляти найпоширеніші вразливості.

Структура кваліфікаційної роботи відповідає меті та завданням дослідження та складається зі вступу, основної частини з трьох розділів, висновків, переліку посилань. Робота викладена на 50 сторінках, містить 20 рисунків, 1 таблицю. Перелік посилань включає 34 джерела.

1 АНАЛІЗ МЕТОДІВ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ У PYTHON-КОДІ

1.1 Основні підходи до аналізу безпеки програмного забезпечення

Безпека програмного забезпечення сьогодні є одним із ключових аспектів у розробці будь-яких інформаційних систем. Особливо це стосується мов, які активно застосовують для веб- та серверних рішень, таких як Python. Значна частина вразливостей виникає через неправильну роботу із зовнішніми даними, тому у процесі створення ПЗ важливо використовувати методи, які дозволяють виявляти проблеми ще до того, як вони стануть критичними. У цьому контексті корисними є спеціалізовані інструменти статичного аналізу, що здатні знаходити небезпечні операції у вихідному коді. Одним із таких рішень є PyT, який дозволяє визначати потенційні загрози у Python-застосунках, зокрема веборієнтованих [1].

Важливу роль у підвищенні безпеки відіграють навчальні та методичні матеріали, які описують типові помилки та сценарії, що часто стають причиною загроз. У рекомендаціях з тестування програмного забезпечення наголошується, що розробник має оцінювати не лише функціональність програми, а й те, наскільки безпечною є взаємодія з даними користувача. Такі посібники формують базове розуміння підходів до захисту програм на різних етапах життєвого циклу [2].

Окремий напрямок досліджень у сфері безпеки стосується аналізу потоків даних у програмі. У роботах, присвячених динамічному таінт-аналізу, показано, що відстеження шляху, яким можуть рухатися потенційно небезпечні дані всередині програми, допомагає своєчасно виявляти складні логічні проблеми та приховані загрози. Сучасні підходи поєднують класичні методи з використанням нейронних мереж, що дає змогу підвищити точність такого аналізу та зменшити кількість хибних спрацювань [3].

Дослідження, присвячені зворотній розробці та аналізу шкідливого програмного забезпечення, також демонструють важливість правильного розуміння, як саме виникають уразливості. Аналіз структури шкідливих програм

дозволяє побачити типові техніки, що використовують зловмисники, і зрозуміти, у яких місцях легітимні програми найчастіше роблять помилки. Це, у свою чергу, формує основу для подальшого вдосконалення інструментів і методів аналізу безпеки [4].

1.2 Методи статичного та динамічного аналізу коду

Одним із найважливіших підходів до забезпечення безпеки програмного забезпечення є статичний аналіз коду. Він дозволяє знаходити вразливості ще до запуску програми, що робить його зручним інструментом для перевірки великих проєктів та автоматизації аудиту. Сучасні інструменти статичного аналізу, такі як Semgrep, працюють на основі правил, які описують типові проблеми безпеки. Завдяки цьому можна швидко визначати потенційно небезпечні конструкції без необхідності виконувати програму. Semgrep використовують у багатьох компаніях через його гнучкість і можливість створювати власні правила для виявлення специфічних загроз [5].

Крім статичного аналізу, у практиці забезпечення безпеки важливу роль відіграє аналіз, орієнтований на конкретні особливості мови програмування. Python активно застосовується у сфері кібербезпеки, і тому дослідження, присвячені можливостям самої мови, допомагають краще розуміти її потенціал і ризики. Наприклад, у роботах, пов’язаних із використанням Python у безпекових цілях, розглядаються його переваги та слабкі сторони при роботі з даними, мережами та системними інтерфейсами. Це формує основу для вибору відповідних методів аналізу в Python-застосунках [6].

Окрему категорію становлять динамічні інструменти, які аналізують поведінку програми в реальному часі. Один з таких прикладів – бібліотека tainted, що дозволяє позначати певні вхідні дані та відстежувати їх рух усередині програми. Цей підхід корисний у випадках, коли потенційна вразливість проявляється лише під час виконання, наприклад, залежно від значення введення або конкретного

порядку викликів функцій. Завдяки цьому динамічний аналіз допомагає виявити логічні проблеми, які важко побачити у вихідному коді [7].

У практичних рекомендаціях для розробників підкреслюється, що поєднання статичного та динамічного аналізу є найбільш ефективним. Статичні інструменти дозволяють швидко перевірити весь код, тоді як динамічні засоби показують, як програма поводиться у “живих” умовах. Комбінований підхід дає змогу виявляти як прості, так і складні вразливості, пов’язані із взаємодією даних та специфікою виконання програми. У матеріалах з аудиту безпеки IT-продуктів наводяться приклади, коли поєднання цих двох підходів істотно підвищує якість і повноту аналізу [8].

1.3 Таінт-аналіз: теоретичні основи і застосування

Таінт-аналіз посідає важливе місце у сучасних підходах до забезпечення безпеки програмного забезпечення, оскільки дає змогу формально визначити траєкторію поширення потенційно небезпечних даних усередині програмної системи. Його основна ідея полягає у тому, що будь-які вхідні значення, отримані із зовнішнього середовища – зокрема користувацьке введення, HTTP-параметри, файли чи мережеві пакети – маркуються як небезпечні (tainted). Далі аналітична система відстежує їх переміщення між змінними, структурами даних і функціями, встановлюючи, чи здатні такі значення у подальшому потрапити до критичних точок виконання (sink). Саме тому таінт-аналіз дозволяє виявляти ризики, що виникають при можливості неконтрольованого використання даних у SQL-запитах, командах ОС, механізмах рендерингу HTML або системних API [9].

З методологічної точки зору, таінт-аналіз ґрунтується на формальній моделі потоків даних, що передбачає класифікацію програмних елементів на джерела (sources), вузли поширення (propagation nodes) та приймачі (sinks). Джерела є точками входу даних у програму, проміжні вузли реалізують передавання чи трансформацію значень, а приймачі – це операції, у яких некоректні або неперевірені дані можуть бути використані у контекстах з високим рівнем ризику.

Для Python-застосунків типовими джерелами є input(), параметри HTTP-запитів, вміст файлів та cookie, а найбільш ризиковими приймачами – виклики os.system(), функції доступу до баз даних, шаблонізатори та інструменти динамічного формування HTML [10].

Схема на рис. 1.1 демонструє класичний ланцюг таінт-поширення, який починається з отримання даних, продовжується шляхом присвоєння та передачі значень між змінними або функціями, та завершується можливим потраплянням до точки виконання небезпечної операції.

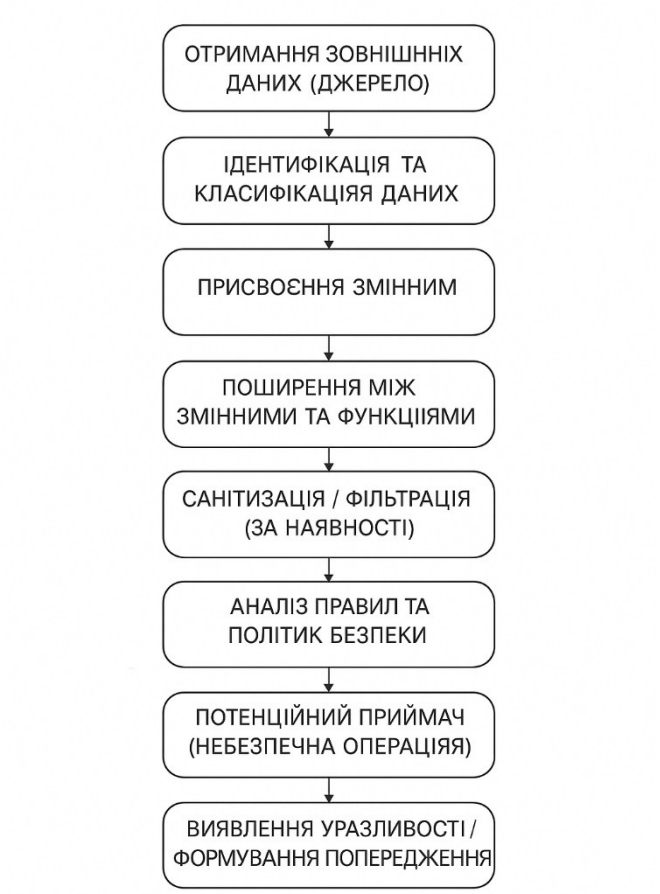

Рисунок 1.1 – Загальна модель таінт-аналізу

Узагальнена модель таінт-аналізу, що зазвичай ілюструється у вигляді послідовності від джерела до приймача, показує, як навіть за наявності простого коду дані можуть проходити складними шляхами. Теоретичною основою цього підходу є графові моделі залежностей та формальний аналіз потоків, які дозволяють описувати програму як спрямований граф, у якому вершини

відповідають операціям, а ребра – передаванню значень. Це забезпечує можливість виявляти приховані ланцюги поширення, які не завжди очевидні з точки зору програміста.

Одним із ключових понять методу є класифікація програмних елементів за ролями. Джерела (sources) – це точки входу зовнішніх даних. Проміжні вузли (propagation) здійснюють передачу даних або трансформаційні операції. Приймачі (sinks) – це операції, виконання яких може призвести до експлуатації вразливостей, якщо туди передані неперевірені дані [11]. У рамках безпеки Python-кодів джерелами можуть бути input(), параметри HTTP, вміст файлів, ключі cookie; а приймачами – os.system(), SQL-запити, функції рендерингу HTML, шаблонізатори та API низькорівневих системних бібліотек.

На рис. 1.2 представлені найбільш типові джерела надходження недовірених даних у Python-застосунках та відповідні точки, у яких використання необроблених значень є найбільш ризиковим.

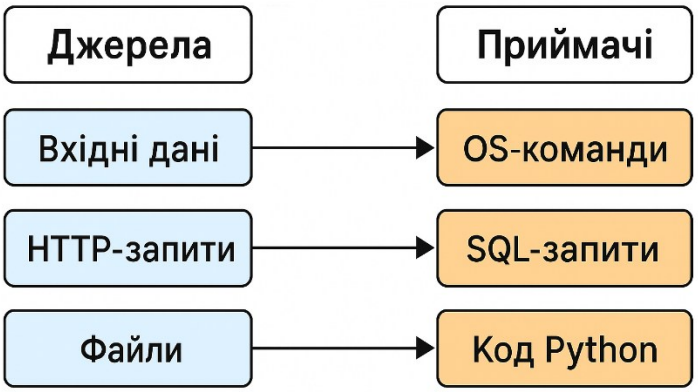

Рисунок 1.2 – Схема взаємодії джерел і приймачів

Щоб підкреслити важливість такого механізму, достатньо розглянути приклад поширення значень у простому однолінійному коді. Навіть якщо програміст впевнений, що змінна не використовується безпосередньо, таінт-аналіз доводить, що фактичний ланцюг передачі може бути складнішим через внутрішні функції та проміжні присвоєння [12].

На рис. 1.3 такий підхід показує, що навіть найпростіший програмний код

може містити логічні вразливості, якщо не контролювати весь шлях передачі даних.

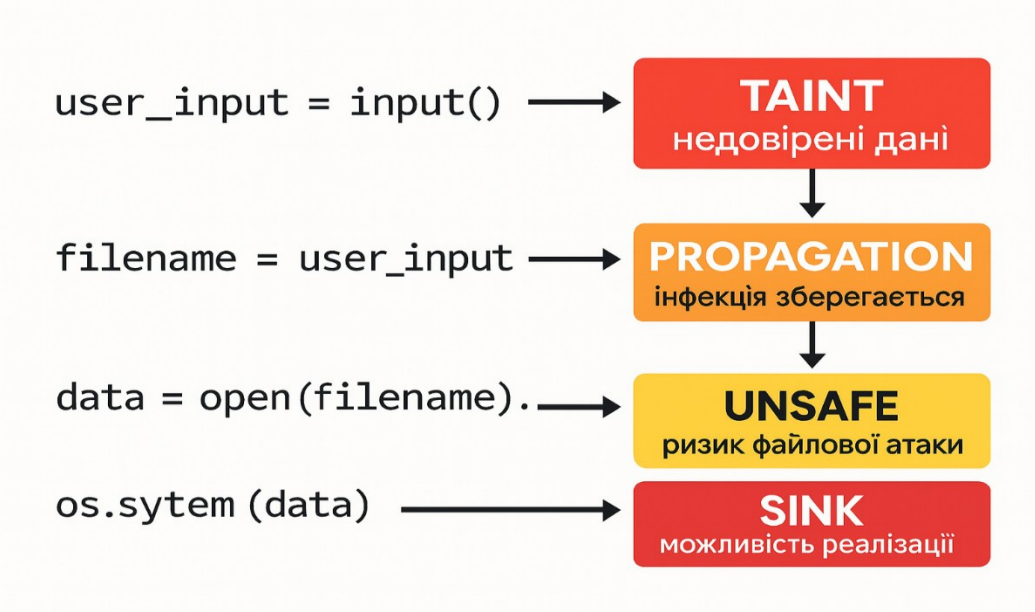

Рисунок 1.3 – Приклад поширення таінтованих даних у фрагменті коду

Таким чином, таінт-аналіз є не лише практичним інструментом, а й концептуальною теоретичною моделлю, що дозволяє формалізувати взаємодію між даними та операціями у програмі, забезпечуючи основу для виявлення і запобігання уразливостей у Python-системах.

1.4 Аналіз моделей таінт-аналізу у Python-програмах

Розвиток сучасних інформаційних систем, зокрема вебзастосунків, платформ електронної комерції, цифрових сервісів та інтегрованих програмних рішень, вимагає впровадження ефективних методів контролю за потоками даних у програмному коді. Особливо це актуально для Python, який активно застосовується у галузях web-розробки, машинного навчання, автоматизації, DevOps та побудови мікросервісних архітектур [13]. Внаслідок цього виникає потреба у використанні методів, здатних виявляти приховані ризики появи шкідливих потоків даних ще до моменту виконання програми. Одним із найбільш результативних та науково обґрунтованих підходів є таінт-аналіз, що базується на відстеженні руху

потенційно шкідливих даних (tainted data) від моменту потрапляння до програми і до точки їх застосування в операціях підвищеного ризику [14].

У науковій літературі та практичних інструментах таінт-аналіз поділяють на декілька моделей, кожна з яких орієнтована на певний рівень точності, масштабованості або продуктивності. Серед найбільш розповсюджених підходів виділяють: статичний таінт-аналіз, динамічний таінт-аналіз, гібридні моделі, контекстно-чутливі моделі та моделі з машинним навчанням, що дозволяють прогнозувати або класифікувати небезпечні сценарії [15].

Порівняння статичного і динамічного підходів наведено у табл.1.1

Таблиця 1.1 – Порівняльна таблиця підходів до таінт-аналізу

Критерій	Статичний аналіз	Динамічний аналіз
Час виконання	До запуску програми	Під час виконання
Продуктивність	Висока	Нижча
Точність	Менша, можливі false positive	Вища точність
Виявлення логічних залежностей	Частково	Так
Застосування	Розробка та CI/CD	Тестування та runtime-захист

Статичний таінт-аналіз виконується без запуску програми, аналізуючи лише її вихідний код, синтаксичну структуру та логіку передачі даних. Такий підхід дозволяє виявляти можливі уразливості на етапі розробки, інтегрувати перевірку у CI/CD та зменшувати витрати на подальшу підтримку системи. У Python статичний таінт-аналіз може здійснюватися шляхом аналізу AST-дерева (Abstract Syntax Tree), що дозволяє виявляти таінтовані елементи навіть за умов багаторівневих присвоєнь або вкладеного виклику функцій (рис. 1.4) [16].

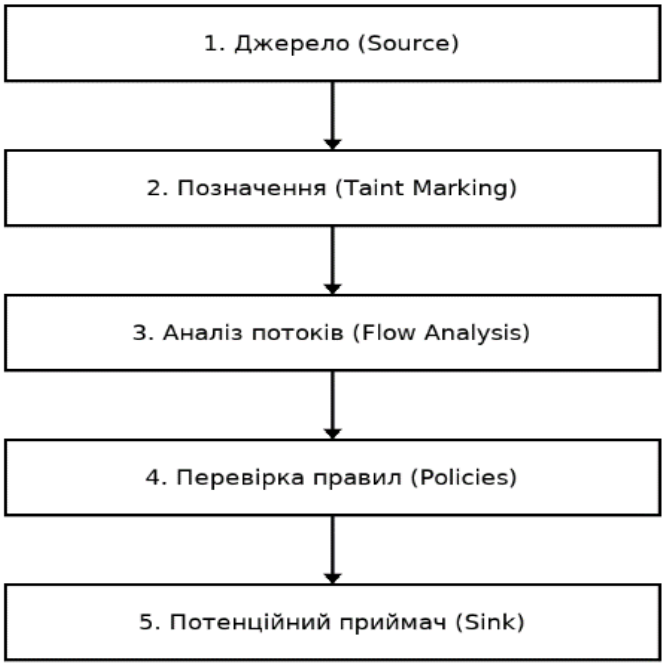

Рисунок 1.4 – Схематичне зображення статичної моделі таінт-аналізу

Динамічний таінт-аналіз здійснюється під час виконання програми, що дозволяє точно виявити реальні шляхи передачі даних, але потребує ресурсів, додаткового інструментування та ізольованого середовища. На відміну від статичного аналізу, динамічний підхід дозволяє відстежувати потоки даних, що формуються внаслідок умовних операторів, циклів, зовнішніх API-викликів та залежностей, недоступних за правилами статичної інтерпретації (рис. 1.5) [17].

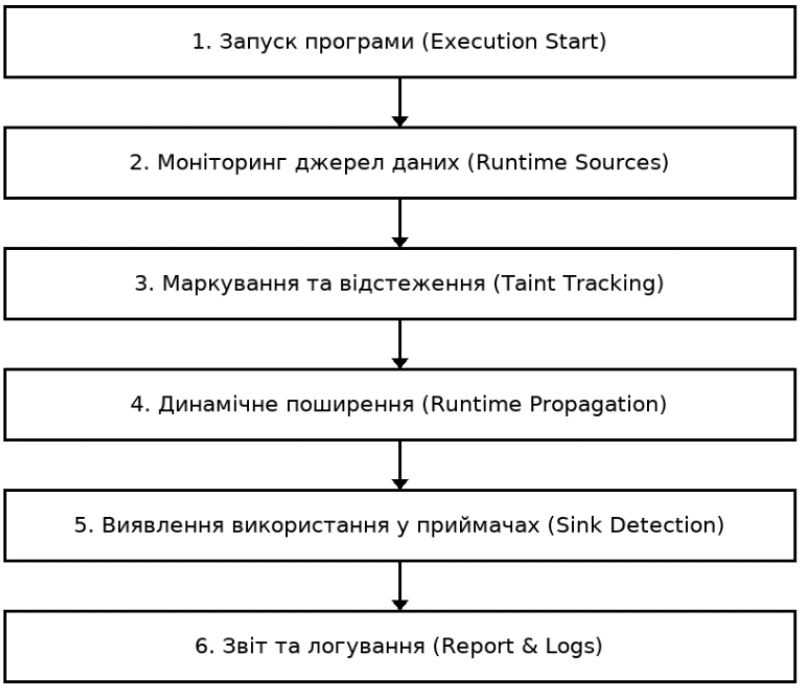

Рисунок 1.5 – Схематичне зображення динамічної моделі таінт-аналізу

Гібридні моделі поєднують можливості статичного та динамічного аналізу, дозволяючи зберегти точність виявлення уразливостей разом з ефективністю аналізу на ранніх етапах розробки. Цей підхід вважається найбільш перспективним у контексті Python-екосистеми, оскільки мова має динамічну природу й може змінювати типи даних або функції виконання під час runtime, що ускладнює повністю статичний підхід [18].

Окремим напрямом розвитку таінт-аналізу є моделі, засновані на алгоритмах машинного навчання (ML), які дозволяють визначати ймовірність ризику не на основі лише синтаксичних залежностей, а через статистичні закономірності. ML-моделі не потребують жорсткого формального опису графів передачі даних і можуть будувати власні закономірності на основі вибірки прикладів шкідливого поведінкового коду [19]. У рамках ML-підходу також можливе використання нейронних моделей перетворення AST у матричні та графові представлення**,** що значно підвищує точність визначення загроз у Python-коді [20].

Таким чином, проведений аналіз моделей таінт-аналізу свідчить, що вибір моделі значною мірою залежить від архітектури застосунку**,** типів потенційних

загроз**,** а також вимог до продуктивності й точності аналізу. Для систем, де пріоритетом є низька вартість експлуатації, доцільно використовувати статичний аналіз; у системах з критичним рівнем безпеки – динамічний або hybrid-аналіз; а ML-моделі можуть бути впроваджені в організаціях із високим рівнем доступності обчислювальних ресурсів та великими наборами навчальних даних.

Висновки до розділу 1

У першому розділі було розглянуто різні підходи до аналізу безпеки програмного забезпечення та те, як саме вони застосовуються у Python-застосунках. Під час дослідження стало зрозуміло, що сучасні програми працюють з великою кількістю даних, отриманих із зовнішніх джерел – від користувацького введення до мережевих запитів. Саме через це вони часто стають вразливими до атак, якщо ці дані не контролювати.

Було проаналізовано три основні методи аналізу: статичний, динамічний та комбінований. Визначено, що методи мають свої плюси, і кожен з них підходить під різні задачі.

Основна увага була приділена саме таінт-аналізу – підходу, який допомагає простежити, як дані рухаються від джерела до кінцевої точки програми. Це важливо, бо саме неконтрольований рух даних часто призводить до таких поширених уразливостей, як SQL-ін’єкції, виконання небезпечних команд чи XSS-атаки. Таінт-аналіз дозволяє побачити повний шлях даних та зрозуміти, де саме може виникнути проблема.

Отже, таінт-аналіз – це потужний спосіб контролю за потоком даних у Python-програмах. Він допомагає вчасно знаходити небезпечні місця у коді ще до того, як ними скористаються зловмисники. Результати цього розділу створюють основу для подальших етапів роботи, де вже буде розроблено власний інструмент таінт-аналізу та проведено його дослідження на практиці.

2 ТЕОРЕТИЧНІ ОСНОВИ ТАІНТ-АНАЛІЗУ PYTHON-КОДУ

2.1 Модель розповсюдження “заражених” даних у Python

Для того щоб зрозуміти, як саме таінт-дані поширюються у Python-програмах та чому цей процес є критичним для виявлення вразливостей, необхідно детально розглянути модель їхнього руху всередині коду. На відміну від традиційних мов, Python дозволяє працювати з даними у максимально гнучкий спосіб – без жорсткої типізації та з динамічною природою змінних. Саме ця гнучкість, з одного боку, робить Python зручним для розробки, а з іншого – створює додаткові ризики у сфері безпеки, пов’язані з передачею даних між різними частинами програми.

Основою будь-якого таінт-аналізу є усвідомлення руху небезпечних даних від джерела до місця використання. Джерелами можуть бути: користувацьке введення, HTTP-параметри, дані з файлів або мережі. Приймачами – функції, що виконують код, формують SQL-запити або взаємодіють з операційною системою. Заражені дані можуть передаватися через присвоєння, виклики функцій, обробку рядків, конкатенацію, форматування та навіть через складні структури (списки, словники, об’єкти).

У Python модель поширення визначається тим, що змінні не зберігають значення жорстко – вони можуть вільно змінюватись і переопределятися. Це створює можливість для небезпечних сценаріїв, коли дані з джерела можуть непомітно пройти через декілька шарів логіки та потрапити у критичні точки програми (рис. 2.1).

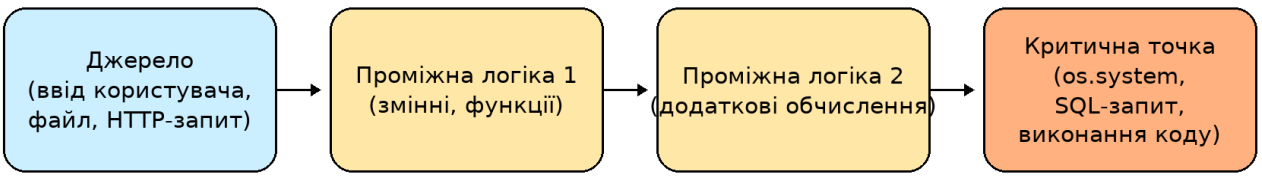

Рисунок 2.1 – Поширення даних через шари логіки у Python

Зображена схема демонструє типовий ланцюг поширення даних у таінт-аналізі – від моменту отримання зовнішнього вводу до потенційно небезпечної операції. Ліва частина містить «джерело» даних, тобто будь-яке зовнішнє надходження інформації: користувацький ввід, файл або HTTP-запит. Далі дані проходять через перший етап проміжної логіки, який може включати змінні, функції чи початкові обчислення. Наступний блок представляє другий шар проміжної логіки – додаткові перетворення або обробку даних. Завершується ланцюг «критичною точкою» (sink), у якій неперевірені дані можуть бути використані в небезпечних контекстах, наприклад у виклику os.system, формуванні SQL-запиту або виконанні фрагмента коду.

Однією з ключових теоретичних моделей, що описує поведінку небезпечних даних, є модель потоку даних (data-flow model). Вона дозволяє прослідкувати увесь маршрут змінної від моменту її формування до моменту використання. Сучасні аналітичні системи, такі як Pysa та CodeQL, будують саме графи потоку даних, де вузлами є операції чи змінні, а ребрами – передача значень [21].

Крім того, у сучасному аналізі Python застосовуються розширені моделі, засновані на графах залежностей. Вони дозволяють відстежувати не тільки прямі присвоєння, а й опосередковані зв’язки між функціями, параметрами, структурами даних та результатами обчислень. Це важливо, оскільки в реальних програмах небезпечні дані майже ніколи не потрапляють у sink напряму – вони проходять через десятки проміжних дій [22].

Ще одна важлива концепція – контекстний таінт-аналіз. Він дозволяє враховувати, у якому саме контексті появляється змінна: у виклику функції, у циклі, в умовному операторі чи всередині об’єкта. Це суттєво підвищує точність виявлення вразливостей, адже рух даних у Python часто залежить від логіки програми, а не тільки від синтаксису. Для Python це особливо актуально, оскільки небезпечні дані можуть потрапити до критичних операцій у непрямий спосіб: через вкладені структури, лямбда-функції, імпортовані модулі, динамічне формування атрибутів об’єктів, замикання, генератори тощо. Тому аналіз простих присвоєнь є

недостатнім – потрібні моделі, що описують цілісну логіку обчислень, до яких відносяться:

1. Поліморфізм і ускладнені шляхи передачі

Python допускає перевантаження операторів і методів, а також різні форми неявного перетворення даних. Це означає, що дані можуть поширюватися не тільки через стандартні операції, а й через спеціальні методи класів (str, add, getitem тощо). У контексті таінт-аналізу це вимагає розширеного моделювання поведінки об’єктів, а інколи – побудови семантичних моделей, які імітують внутрішню логіку класів.

2. Вплив динамічної диспетчеризації

Python визначає, яку саме функцію або метод буде викликано, лише під час виконання. Така динамічна диспетчеризація ускладнює побудову точних графів потоку даних. У теорії аналізу програм це пов’язано з проблемою aliasing – коли кілька змінних посилаються на один і той самий об’єкт. Тому аналітичні інструменти повинні моделювати не тільки потік даних, але й потік посилань.

3. Використання абстрактної інтерпретації

Сучасні таінт-аналізатори застосовують метод абстрактної інтерпретації – математичний апарат, який дозволяє аналізувати програму без виконання, замінюючи конкретні значення абстрактними доменами. Для Python це складно, але необхідно, оскільки точний аналіз усіх можливих станів призводив би до вибуху комбінацій.

4. Мультифункціональність проміжної логіки

Розглянута схема (рис. 2.1) відображає, що «проміжна логіка» не є однорідною. У сучасному підході її можна деталізувати на:

– логіку трансформації (зміна типу, форматування, конкатенація);

– логіку маршрутизації (передача параметрів між функціями);

– логіку контролю (if/else, цикли, винятки);

– логіку структурування даних (списки, словники, об’єкти).

Така деталізація дає змогу точніше моделювати сценарії поширення і зменшувати кількість хибних спрацьовувань.

5. Контекстність – шлях до точності

Контекстний аналіз дозволяє враховувати:

– місце виклику функції (call-site);

– логічний шлях виконання (path-sensitive analysis);

– типи даних у кожному конкретному моменті (type-sensitive analysis).

Ці моделі суттєво підвищують точність порівняно з класичним інтра-процедурним аналізом.

Підсумовуючи, модель розповсюдження «заражених» даних у Python ґрунтується на аналізі шляху, який проходять дані до свого використання. Динамічність мови, відсутність фіксованих типів і велика кількість способів трансформації даних роблять Python одночасно зручним і ризикованим середовищем для розробки. Розуміння особливостей цієї моделі є обов'язковим для побудови ефективних інструментів таінт-аналізу та подальшого виявлення вразливостей.

2.2 Джерела sources та приймачі sinks у Python-програмах

Одним із ключових елементів таінт-аналізу є визначення того, звідки потрапляють небезпечні дані та в яких точках вони можуть спричинити вразливість. Саме тому поняття джерел (sources) і приймачів (sinks) є фундаментальними при побудові ефективної моделі контролю даних у Python-застосунках.

Джерела (sources) – це будь-які точки коду, де програма отримує дані, що не можна вважати надійними. Найчастіше це:

1) input() і інші форми введення користувача;

2) параметри HTTP-запитів;

3) дані з файлів;

4) відповіді API;

5) змінні середовища;

6) дані, отримані через мережеві сокети.

Такі джерела становлять потенційну загрозу, тому що програміст не контролює їх зміст – користувач або зловмисник можуть передати спеціально сформоване значення, яке призведе до небезпечної поведінки коду.

Приймачі (sinks) — це «критичні точки», у яких небезпечні дані вже можуть спричинити реальну атаку. У Python до них належать:

1) виконання системних команд (os.system(), subprocess.Popen());
2) формування SQL-запитів (cursor.execute());
3) динамічне виконання коду (eval(), exec());
4) формування HTML-відповідей у веб-фреймворках;
5) виклики, що впливають на файлову систему або мережу;
6) бібліотеки, які виконують внутрішні операції з високими привілеями.

Завдання аналізатора полягає в тому, щоб відстежити, чи може значення з певного джерела потрапити до небезпечного приймача, і якщо так – на якому саме шляху це відбувається.

Сучасні підходи до визначення джерел і приймачів активно застосовуються в інструментах на кшталт CodeQL та Semgrep. Наприклад, CodeQL має вбудовані модулі, які дозволяють формально описати список джерел/приймачів і моделювати їх поведінку в коді. Це забезпечує точніше визначення уразливостей та мінімізує кількість хибних спрацювань [24].

У свою чергу, Python пропонує більш просту, але гнучку модель, де джерела і приймачі задаються через правила, що дозволяє адаптувати аналіз під конкретний проєкт. Незважаючи на простоту механізму, його ефективність широко підтверджена практикою, особливо у випадках виявлення SQL-ін’єкцій, командних ін’єкцій, XSS та інших загроз, пов’язаних із неконтрольованими даними [25].

Правильне визначення джерел та приймачів дозволяє не лише знайти потенційні вразливості, а й побудувати ефективну стратегію їх попередження. Якщо система аналізу знає, в які точки програми можуть проникнути небезпечні дані, вона здатна відстежувати їх шлях значно точніше і забезпечувати більш надійний захист.

2.3 Використання індексів та агрегатних функцій для оптимізації продуктивності таінт-аналізу

Оптимізація продуктивності при проведенні таінт-аналізу Python-застосунків є критичним аспектом, особливо коли інструмент працює з великими обсягами коду, складними структурами даних та значною кількістю шляхів поширення таінту. Одними з ключових механізмів підвищення ефективності аналізу є використання індексування структур даних та агрегувальних операцій, що дозволяють зменшити кількість зайвих обчислень і прискорити пошук залежностей.

Більшість сучасних статичних аналізаторів, зокрема системи, що досліджують поширення таінту, будують граф залежностей або граф потоку даних. З часом кількість вузлів у такому графі може сягати десятків тисяч, тому пошук необхідних зв’язків без оптимізації стає повільним.

У таких випадках використовуються спеціальні таблиці індексації, що дозволяють швидко знаходити:

1) змінні, що вже помічені як таінтові;

2) вузли з певним типом операції (наприклад, виклики функцій);

3) шляхи, що ведуть до критичних приймачів.

Нижче наведено приклад фрагмента реалізації індексування шляхів у графі поширення даних (рис. 2.2):

```
index_by_type = {}

def add_node(node):
    if node.type not in index_by_type:
        index_by_type[node.type] = []
    index_by_type[node.type].append(node)

function_calls = index_by_type.get("call", [])
```

Рисунок 2.2 – Приклад індексації вузлів у графі даних

У такий спосіб аналізатор уникає повного обходу дерева чи графа, а одразу звертається до готових індексів, що багаторазово прискорює аналіз у великих проєктах [26].

Ще одним важливим елементом оптимізації є агрегування даних під час аналізу.

У таінт-аналізі агрегатні операції використовуються для:

– підрахунку кількості небезпечних шляхів;

– оцінки глибини поширення даних;

– виявлення функцій, які найчастіше виступають у ролі проміжних вузлів;

– визначення рівня критичності певного приймача.

На рис. 2.3 наведено простий агрегувальний приклад, де аналізатор рахує, скільки разів певна функція виступає у ролі проміжного вузла:

```
from collections import Counter

taint_flow_log = [
    ("input", "sanitize"),
    ("sanitize", "execute"),
    ("input", "log"),
    ("log", "execute")
]

counter = Counter([step[1] for step in taint_flow_log])
print(counter)  # {'sanitize': 1, 'execute': 2, 'log': 1}
```

Рисунок 2.3 – Агрегування даних про поширення таінту

Завдяки агрегації можна визначити функції, які потребують додаткового аналізу або рефакторингу – наприклад, ті, що найчастіше опиняються на шляху до небезпечного приймача [27].

У деяких моделях аналізу, особливо тих, що використовують машинне навчання або комбіновані статичні та динамічні методи, необхідно індексувати дані не тільки за типом, а й за контекстом. Наприклад:

– функція, викликана у певному модулі;

– змінна, що була змінена у визначеній гілці умовного оператора;

– виклик, що відбувається всередині циклу.

Нижче наведено приклад складеного індексу (рис. 2.4):

```
index = {}

def add(node):
    key = (node.type, node.context)
    if key not in index:
        index[key] = []
    index[key].append(node)
```

Рисунок 2.4 – Складений індекс за типом і контестом

Такий підхід дозволяє значно зменшити кількість хибнопозитивних результатів і підвищити точність аналізу.

Індексація та агрегування – це одні з найважливіших методів оптимізації продуктивності таінт-аналізаторів. Індекси забезпечують швидкий доступ до ключових вузлів графа залежностей, а агрегатні функції дозволяють виконувати складні обчислення без зайвого навантаження на систему. Їх поєднання дає можливість аналізатору працювати стабільно навіть у великих проєктах з великою кількістю шляхів поширення даних.

2.4 Використання машинного навчання для класифікації потенційних загроз

У сучасних системах захисту Python-застосунків машинне навчання відіграє важливу роль у виявленні потенційно небезпечних шляхів поширення даних. На відміну від класичних статичних методів, ML-моделі здатні аналізувати контекст, поведінкові характеристики коду, внутрішні залежності та приховані шаблони, що дозволяє визначати загрози більш точно та адаптивно.

Фундаментом для застосування машинного навчання у таінт-аналізі є концепція моделювання механізму sources–sinks (джерел та приймачів), детально описана в роботі BreachForce [28]. Ця модель визначає критичні точки програми:

Джерела (sources) – місця, де ненадійні дані потрапляють у систему (користувацький ввід, зовнішні API, файли);

Приймачі (sinks) – критичні операції, де використання неперевірених даних може призвести до вразливостей (SQL-запити, виконання команд, серіалізація).

Машинне навчання ефективно доповнює існуючі засоби статичного аналізу. Наприклад, CodeQL використовує декларативний підхід до таінт-аналізу через модуль TaintTracking, що описує правила поширення даних [29]. Однак такий підхід має обмеження: необхідність ручного опису всіх можливих шляхів, складність адаптації до нових типів вразливостей та висока кількість хибнопозитивних спрацювань.

ML-моделі розширюють можливості статичного аналізу, автоматично:

– виявляючи нові шаблони небезпечної поведінки на основі історичних даних;

– ідентифікуючи відхилення та аномалії в потоках даних;

– адаптуючись до еволюції коду та змін архітектури застосунку.

ML-моделі навчаються розпізнавати не лише явні зв'язки між джерелами та приймачами, але й складні, багатокрокові ланцюги поширення даних через проміжні функції, класи та модулі.

Однією з критичних проблем класичного статичного аналізу є генерація надмірної кількості попереджень, з яких значна частина є хибнопозитивними [30]. У корпоративних системах це створює додаткове навантаження на команди безпеки та знижує довіру до інструментів аналізу.

ML-моделі вирішують цю проблему шляхом:

– контекстної фільтрації – аналіз реального використання даних у бізнес-логіці;

– ранжування загроз – призначення пріоритетів на основі ймовірності реальної експлуатації;

– навчання на помічених даних – використання історії підтверджених і відхилених попереджень для покращення точності.

Однією з найбільш суттєвих проблем класичних засобів статичного аналізу залишається висока частка хибно-позитивних попереджень, що створює значний додатковий робочий тягар у корпоративних системах [31]. ML-моделі зменшують цей ефект за рахунок контекстної інтерпретації бізнес-логіки, ранжування загроз щодо ймовірності реальної експлуатації та навчання на історично підтверджених прикладах. В умовах мультимовних програмних середовищ, де Python взаємодіє з JavaScript, SQL та іншими мовами, машинне навчання демонструє здатність працювати з різними синтаксичними структурами, уявленнями коду та рівнями абстракції, забезпечуючи однаковий аналітичний конвеєр для гетерогенних проектів.

Ефективність ML-підходів у таїнт-аналізі безпосередньо залежить від якості навчальних даних. Результати статичного аналізу та механізми відстеження потоків даних можуть бути основою для автоматичної розмітки вразливостей, яка потім уточнюється експертами та доповнюється синтетично згенерованими прикладами. Це дозволяє формувати репрезентативні вибірки, придатні навчання моделей, здатних розпізнавати широкий спектр загроз – від SQL-ін'єкцій і XSS до складних форм командних впроваджень [32].

Для таінт-аналізу використовуються різні архітектури машинного навчання:

– класичні ML-алгоритми (Random Forest, Gradient Boosting) – для класифікації на основі статистичних характеристик коду;

– глибоке навчання (RNN, LSTM, Transformer) – для аналізу послідовностей токенів і контекстних залежностей;

– Graph Neural Networks (GNN) – для обробки графів потоків даних та CFG/DFG;

– навчання з підкріпленням – для адаптивного пошуку оптимальних шляхів аналізу.

Машинне навчання посилює традиційний таінт-аналіз за трьома ключовими напрямами:

– контекстна класифікація – визначення небезпечних шляхів з глибоким розумінням бізнес-логіки програми, врахуванням санітизації даних та контекстних перевірок;

– зменшення хибнопозитивних спрацювань – інтелектуальна фільтрація попереджень на основі ймовірнісних оцінок та історичного аналізу, що суттєво покращує точність статичного аналізу;

– виявлення складних прихованих загроз – ідентифікація нетривіальних вразливостей, які неможливо описати жорсткими правилами: багатокрокові атаки, складні умовні ланцюги, exploitation через комбінації кількох функцій.

За всіх переваг інтеграції машинного навчання в процеси таїнт-аналізу залишаються невирішеними завдання отримання великих масивів розмічених даних, зрозумілості роботи моделей, своєчасної адаптації до нових типів загроз та забезпечення достатніх обчислювальних ресурсів для аналізу масштабних кодових баз. Тим не менш поєднання ML-підходів з класичними методами статичного аналізу формує більш гнучку, точну і адаптивну систему безпеки, здатну протистояти як відомим, так і раніше не зустрічалися загроз в умовах кіберпростору, що динамічно розвивається.

Висновки до розділу 2

У другому розділі було детально проаналізовано теоретичні засади таінт-аналізу Python-коду, починаючи з ключової моделі розповсюдження «заражених» даних, і завершуючи сучасними методами оптимізації та підсилення аналізу за допомогою машинного навчання. Проведене дослідження продемонструвало, що безпечність Python-застосунків у великій мірі залежить від точності відстеження шляху, яким потенційно небезпечні дані проходять усередині програми.

Було встановлено, що саме динамічна природа Python – відсутність жорсткої типізації, вільне перевизначення змінних, передача значень через різні структури та операції – створює як гнучкість для розробника, так і підвищений ризик потрапляння небезпечних даних у критичні точки. Потім було проаналізовано

фундаментальне значення коректного визначення джерел (sources) та приймачів (sinks), оскільки від цього напряму залежить точність будь-якого механізму статичного аналізу.

Було показано, що оптимізаційні техніки – такі як індексація графів даних, агрегування метаданих та використання складених індексів – відіграють ключову роль у забезпеченні продуктивності аналізатора, особливо при роботі з великими кодовими базами. Без подібних технік аналізатор втрачає швидкість, а якість виявлення вразливостей погіршується.

Використання ML дозволяє враховувати контекст виконання коду, знижувати кількість хибнопозитивних спрацювань, виявляти приховані або складні ланцюги передачі даних та працювати навіть з нестандартними або динамічними конструкціями Python. Інтеграція ML-моделей із класичними механізмами статичного аналізу робить системи захисту значно гнучкішими та ефективнішими.

У цілому, результати розділу підтверджують, що таінт-аналіз Python-коду є складною, багатокомпонентною задачею. Його успіх залежить від правильної побудови моделі руху даних, точного визначення джерел і приймачів, оптимізації обчислювальних процесів та впровадження інтелектуальних методів аналізу. Таке поєднання дозволяє створювати інструменти, здатні виявляти широкий спектр вразливостей – від SQL- та командних ін’єкцій до складних логічних атак, що неможливо виявити лише простим пошуком шаблонів.

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ІНСТРУМЕНТУ АНАЛІЗУ

3.1 Вибір середовища для реалізації та тестування інструменту

Для створення інструменту статичного таінт-аналізу Python-коду необхідно було визначити оптимальне середовище розробки, спосіб організації проєкту та підходи до архітектури майбутнього аналізатора. Оскільки метою є побудова легкого, зрозумілого та швидкодіючого інструменту, важливо було забезпечити його сумісність із будь-яким Python-середовищем та мінімізувати залежність від сторонніх бібліотек.

У якості основного середовища розробки було обрано PyCharm Community Edition 2024. Це зумовлено тим, що PyCharm забезпечує зручний інтерфейс для роботи зі структурою проєкту, коректним підсвічуванням синтаксису, інтегрованим терміналом та інструментами для швидкого налагодження коду. Завдяки цьому розробка інструменту simple_taint.py стала більш організованою та зрозумілою.

Другим важливим аспектом стало визначення архітектури самого аналізатора. Підхід до реалізації був обраний на основі роботи Ahmed K. [33], де розглядається концепція динамічного відстеження небезпечних даних у Python та їх моделювання на рівні виконання. Цей матеріал дозволив краще зрозуміти ключові етапи руху даних та визначити алгоритмічні кроки, які необхідно реалізувати у статичному аналізаторі.

Загальна архітектура *simple_taint.py* будується навколо таких принципів:

1. Використання модуля ast для розбору програми Інструмент не виконує код, а лише аналізує його структуру, що забезпечує безпечність і незалежність від зовнішнього середовища.

2. Відокремлення джерел (sources) та приймачів (sinks) Основним завданням аналізатора є визначення того, чи може значення, отримане з небезпечного джерела, потрапити до критичної функції.

3. Ведення списку «заражених» змінних (tainted variables)
Під час проходу AST дерево постійно оновлюється множина змінних, у яких можуть міститися неконтрольовані дані.

4. Перевірка кожного виклику функції на предмет наявності небезпечного аргументу
Інструмент фіксує рядок, де відбувається потенційно небезпечний виклик.

5. Повна незалежність від бібліотек та фреймворків
Програма використовує лише стандартні можливості Python, завдяки чому може запускатися в будь-якому середовищі.

Структура проєкту складається з двох основних файлів:

simple_taint.py – безпосередньо інструмент аналізу;

test_code.py – тестовий файл, що містить приклади вразливостей, які аналізатор повинен виявляти.

Завдяки чіткій архітектурі та можливостям PyCharm, розробка аналізатора стала більш керованою, а тестування – значно зручнішим. PyCharm також дає можливість швидко перемикатися між різними тестовими файлами, переглядати історію запусків та організовувати структуру проєкту. Це забезпечило ефективне середовище для створення початкової версії інструменту та подальшої його перевірки.

Таким чином, вибір PyCharm Community Edition 2024 та архітектурних рішень на основі підходів, описаних у [33], дозволив сформувати стабільну платформу для реалізації інструменту статичного таінт-аналізу, який є простим у використанні та забезпечує якісне виявлення небезпечних шляхів у Python-коді.

3.2 Алгоритм та структура коду інструменту таінт-аналізу

У цьому підрозділі детально розглянуто структуру, логіку та алгоритми роботи розробленого інструменту статичного таінт-аналізу Python-коду. Аналізатор simple_taint.py побудований на основі стандартного модуля Python – ast, який використовується для побудови абстрактного синтаксичного дерева (AST) та

подальшого обходу вузлів програми. Методика реалізації інструменту включає визначення джерел небезпечних даних (sources), критичних приймачів (sinks), відстеження передачі значень між змінними та виявлення ситуацій, у яких інфіковані значення доходять до критичних точок.

Далі подано детальний опис кожного етапу обробки коду інструментом із покроковим розглядом фрагментів програми та поясненням принципів роботи аналізатора.

```
1  import ast
2  import sys
3
4  SOURCES = {"input", "request_args"}
5  SINKS = {"os.system", "eval", "cursor.execute"}
```

Рисунок 3.1 – Імпорт бібліотек та оголошення sources/sinks

Перший етап роботи інструменту передбачає оголошення множин джерел та приймачів. Множина SOURCES містить функції, виклики яких формують потенційно небезпечні дані (наприклад, input або request_args у веб-додатках). Множина SINKS визначає операції, що можуть бути використані зловмисником для атаки: виконання системних команд, динамічний запуск коду, формування SQL-запитів.

На цьому етапі формується концептуальна модель інструменту: усе подальше моделювання потоків даних здійснюється від джерел до приймачів. У більш складних інструментах ці множини зазвичай містять десятки API-викликів, але базова модель дозволяє виконувати універсальний аналіз навіть у мінімалістичній конфігурації.

```
7  class SimpleTaintAnalyzer(ast.NodeVisitor):
8      def __init__(self):
9          self.tainted_vars = set()
10         self.warnings = []
```

Рисунок 3.2 – Оголошення класу SimpleTaintAnalyzer

Клас SimpleTaintAnalyzer наслідує базовий клас ast.NodeVisitor, який надає інтерфейс для обходу AST-дерева Python-коду.

У конструкторі створюються дві структури:

tainted_vars – множина змінних, які вже мають «таінтований» статус;

warnings – список попереджень про виявлені вразливості.

Цей етап формує основу інструменту: множина tainted_vars є центральним елементом алгоритму, оскільки саме вона відображає побудовану модель передачі даних між змінними.

```
def visit_Assign(self, node):
    if isinstance(node.value, ast.Call) and isinstance(node.value.func, ast.Name):
        if node.value.func.id in SOURCES:
            for target in node.targets:
                if isinstance(target, ast.Name):
                    self.tainted_vars.add(target.id)
```

Рисунок 3.3 – Обробка присвоєнь (visit_Assing)

Цей фрагмент відповідає за найважливіший механізм: відстеження утворення інфікованих змінних на етапі присвоєння значення.

Алгоритм працює так:

1. Перевіряється, чи є правий вираз викликом функції.
2. Аналізується, чи цей виклик належить до джерел.
3. Усі змінні лівої частини присвоєння позначаються як заражені.

Це повністю відповідає моделі типового статичного таінт-аналізу: джерело запускає ланцюжок передачі таінту програмою.

У складних програмах таінт може передаватися через десятки присвоєнь, тому коректна обробка цього кроку є критично важливою для точності інструменту.

```
elif isinstance(node.value, ast.Name):
    if node.value.id in self.tainted_vars:
        for target in node.targets:
            if isinstance(target, ast.Name):
                self.tainted_vars.add(target.id)
```

Рисунок 3.4 – Передача таінту між змінними

Цей фрагмент реалізує модель транзитивного поширення даних: якщо змінна отримує значення іншої змінної, яка вже помічена як небезпечна, нова змінна також стає таінтовою.

Приклад:

a = input()

b = a

c = b

Усі три змінні потраплять у tainted_vars.

Цей етап є важливим, тому що реальні ін’єкції зазвичай відбуваються не одразу після джерела, а через змішані, перетворені та перенесені змінні. Таким чином інструмент формує ланцюжок залежностей у вигляді графа даних.

```
def visit_Call(self, node):
    if isinstance(node.func, ast.Name):
        func_name = node.func.id
    elif isinstance(node.func, ast.Attribute) and isinstance(node.func.value, ast.Name):
        func_name = f"{node.func.value.id}.{node.func.attr}"
    else:
        func_name = ""
```

Рисунок 3.5 – Обробка викликів функцій (visit_Call)

Цей блок коду нормалізує ім’я функції, щоб можна було порівняти його з множиною SINKS.

Python дозволяє викликати функції у різний спосіб:

1. eval()
2. cursor.execute()
3. os.system()

4. obj.method()

Тому інструмент повинен коректно інтерпретувати як прості, так і атрибутні виклики.

Без цього звичайна команда os.system("ls") була б пропущена – інструмент не зміг би розпізнати її як критичний приймач.

```
if func_name in SINKS:
    for arg in node.args:
        if isinstance(arg, ast.Name) and arg.id in self.tainted_vars:
            self.warnings.append(
                f"[!] WARNING: Tainted variable '{arg.id}' flows into {func_name} (line {node.lineno})"
            )
```

Рисунок 3.6 – Перевірка аргументів на наявність таінту

Цей етап є центральною логічною точкою всього аналізатора: він визначає, чи досягають інфіковані дані критичної точки виконання.

Алгоритм:

1. Перевіряється, чи є функція приймачем (sink).
2. Перевіряються всі аргументи виклику.
3. Якщо хоча б один аргумент містить інфіковану змінну – формується попередження.

Наприклад:

cmd = input()

os.system(cmd)

Інструмент виявить командну ін’єкцію.

Цей крок робить simple_taint.py повноцінним аналізатором небезпечних потоків даних.

```python
def analyze_file(filename: str):
    with open(filename, "r", encoding="utf-8") as f:
        code = f.read()
    tree = ast.parse(code, filename=filename)
    analyzer = SimpleTaintAnalyzer()
    analyzer.visit(tree)
    return analyzer.warnings
```

Рисунок 3.7 – Основна функція analyze_file

Функція analyze_file:

1. Зчитує код Python-файлу.
2. Будує абстрактне синтаксичне дерево.
3. Запускає аналізатор, який обходить усі вузли дерева.
4. Повертає сформовані попередження.

Цей етап демонструє, що інструмент працює як класичний статичний аналізатор: не виконує код, не обробляє його динамічно – а лише моделює логіку на основі AST.

```python
if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python simple_taint.py <file_to_analyze.py>")
        sys.exit(1)

    filename = sys.argv[1]
    results = analyze_file(filename)

    if results:
        print(f"Results for {filename}:")
        for r in results:
            print(r)
    else:
        print(f"No taint issues found in {filename}")
```

Рисунок 3.8 – Запуск аналізатора

Це блок, який забезпечує взаємодію користувача з інструментом. Аналізатор зчитує ім’я файлу, переданого аргументом командного рядка, виконує аналіз та виводить результат.

У цьому підрозділі проведено детальний аналіз архітектури інструменту таінт-аналізу, побудованого на основі AST. Розроблена система:

1. коректно відтворює модель передачі даних у Python;
2. визначає джерела небезпечних даних;
3. відстежує транзитивні залежності через змінні;
4. виявляє критичні точки, що можуть призвести до атак;
5. формує точні повідомлення про вразливості.

Ця реалізація демонструє, як на практиці реалізується статичний аналіз поведінки даних у програмі.

3.3 Приклади роботи інструменту

Для демонстрації працездатності розробленого інструменту статичного таінт-аналізу *simple_taint.py* було підготовлено тестовий файл *test_code.py*, у якому реалізовано два характерних сценарії поширення неконтрольованих даних у критичні точки програми. Подібний принцип побудови тестових прикладів відповідає підходам, що застосовуються у сучасних системах аналізу безпеки програмного коду, де спеціально формуються приклади з типовими вразливостями для перевірки коректності роботи інструментів [34]. У наукових та прикладних дослідженнях з кібербезпеки Python-застосунків особлива увага приділяється саме моделюванню руху даних між різними елементами програми, виявленню потенційно небезпечних шляхів та оцінці здатності аналізаторів своєчасно виявляти такі загрози, що узгоджується з логікою використання файлу *test_code.py* у межах даної роботи.

У ньому містяться два відокремлені фрагменти коду, які ілюструють різні сценарії небезпечної поведінки. Перший стосується можливості виконання системних команд на основі введення користувача. Другий – формування SQL-запиту шляхом конкатенації рядків, що створює передумови для класичної SQL-ін’єкції. Обидва приклади є важливими з точки зору безпеки, оскільки вони дозволяють перевірити здатність розробленого аналізатора визначати потенційно

небезпечні шляхи поширення даних.

```
import os

user_input = input("Enter command: ")
cmd = user_input
os.system(cmd)

name = request_args("name")
query = "SELECT * FROM users WHERE name='" + name + "'"
cursor.execute(query)

safe = "ls -la"
os.system(safe)
```

Рисунок 3.9 – Текстовий файл (test_code.py)

Після запуску *simple_taint.py* на тестовому файлі інструмент виконує синтаксичний розбір програми за допомогою модуля AST, ітеративно обходить дерево вузлів і перевіряє, чи може значення, отримане з джерела даних, потрапити до критичного приймача. Якщо шлях передачі даних було успішно встановлено, інструмент формує відповідне попередження про можливу уразливість.

У процесі аналізу було зафіксовано одне попередження, що свідчить про ненадійний шлях даних, який пройшов через джерела та потрапив у небезпечні місця виконання коду.

```
[!] WARNING: Tainted variable 'cmd' flows into os.system (line 5)
```

Рисунок 3.10 – Результат роботи інструменту (simple_taint.py)

Під час виконання тестового файлу інструмент виявив потенційно небезпечний сценарій, пов’язаний із викликом функції *os.system()*. У цьому фрагменті коду програма отримує неконтрольоване введення від користувача через *input()*, що одразу класифікується як потенційно небезпечне джерело даних. Надалі

це значення присвоюється змінній cmd, унаслідок чого стан «зараженості» коректно переноситься на всі пов’язані змінні.

Під час подальшого обходу абстрактного синтаксичного дерева (AST) інструмент визначає, що змінна *cmd* передається як аргумент у виклик *os.system()* – функції, яка є критичним приймачем і може спричинити виконання довільних команд операційної системи. Саме тому інструмент реєструє попередження про можливу командну ін’єкцію. Подібний підхід повністю відповідає сучасним методам статичного аналізу, у яких передачу неконтрольованих даних у системні функції розглядають як одну з найбільш небезпечних категорій загроз.

Отримане повідомлення підтверджує, що інструмент здатний виявляти ланцюги передачі потенційно небезпечної інформації навіть у найпростіших, але критично важливих випадках, де між джерелом і приймачем присутні проміжні змінні.

Виявлена уразливість свідчить про те, що реалізований інструмент коректно аналізує потік даних у Python-коді та здатний визначати небезпечні місця передачі інформації між джерелами та критичними приймачами. Незважаючи на невелику кількість правил і спрощену логіку поширення даних, інструмент демонструє базову точність виявлення реальних загроз, що характерно для більш складних систем статичного аналізу.

Цей результат підтверджує, що розроблений підхід може бути використаний як початкова основа для створення більш функціонального аналізатора з розширеною підтримкою джерел даних, складніших моделей поширення та інтеграцією додаткових типів небезпечних конструкцій у Python-коді.

3.4 Інтеграція машинного навчання для підвищення точності аналізу

Описаний інструмент *simple_taint.py*, працює виключно на основі жорстких правил та статичних множин SOURCES і SINKS. Це створює кілька обмежень. По-перше, неможливість адаптації до нових типів джерел/приймачів без ручного редагування коду. Існує високий рівень false positives – відсутність контекстного

аналізу призводить до попереджень про безпечні операції. Також не враховуються складні багатокрокові трансформації даних. Неможливість виявлення нових патернів. Тому потрібно створити гібридний інструмент з ML-компонентом.

Запропонований підхід поєднує класичний статичний аналіз коду на основі абстрактного синтаксичного дерева (AST) з методами машинного навчання для підвищення точності виявлення вразливостей та зменшення кількості хибних спрацювань. Архітектура рішення складається з трьох основних компонентів (рис. 3.11).

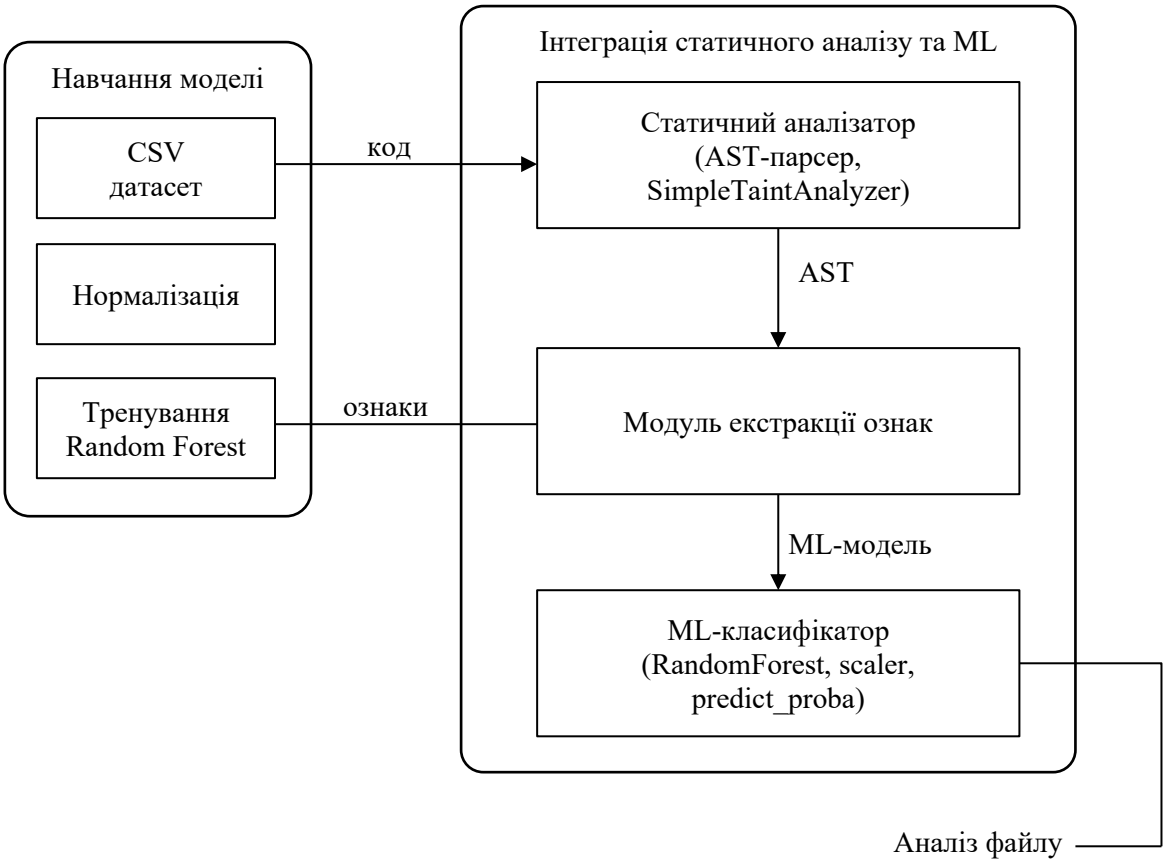

Рисунок 3.11 – Архітектура інтеграції статичного аналізу та ML

Статичний аналізатор призначений для побудови AST, виконує базовий taint-аналіз і формує потенційні шляхи передачі даних від джерел до небезпечних функцій. При цьому зберігається розширений контекст виконання, зокрема кількість блоків try/except, наявність операцій санітизації та валідації, а також застосовані трансформації даних.

Модуль екстракції ознак формує для кожного попередження відповідний вектор ознак, який включає:

– структурні характеристики (глибина викликів, тип вузла AST);

– контекстні фактори (наявність санітизації, валідації, блоків обробки винятків);

– рівень ризику джерела та приймача;

– параметри потоку даних (довжина шляху, кількість трансформацій, конкатенація рядків).

ML-класифікатор на основі моделі Random Forest має здійснювати віднесення попереджень до істинних або хибнопозитивних. Навчання моделі виконується на розміченому наборі даних, що містить приклади як вразливого, так і безпечного коду. Крім класу, модель формує коефіцієнт впевненості, що дозволяє ранжувати виявлені ризики за пріоритетністю.

Додатково реалізовані допоміжні модулі:

– модуль навчання моделі, що забезпечує завантаження датасету, нормалізацію ознак, тренування Random Forest, оцінювання точності та збереження моделі;

– модуль аналізу файлів, який поєднує результати статичного аналізу з ML-класифікацією та виконує сортування попереджень за рівнем ризику.

Застосування методів машинного навчання в такій гібридній архітектурі суттєво зменшує кількість хибнопозитивних спрацювань, автоматизує пріоритезацію вразливостей та підвищує загальну ефективність процесів забезпечення безпеки програмного забезпечення.

Для практичної реалізації ML-компонента необхідне виконання таких етапів:

– створення розміченого датасету вразливого та безпечного коду;

– навчання моделі Random Forest на збалансованій вибірці;

– налаштування гіперпараметрів через крос-валідацію;

– експериментальне порівняння з базовим аналізатором;

– оцінювання метрик точності (Precision, Recall, F1-score).

Ці завдання є предметом подальших досліджень і не входять до обсягу даної роботи.

Описаний підхід має концептуальний характер: архітектуру та фрагменти програмної реалізації наведено в Додатку В, проте повномасштабна реалізація та експериментальна оцінка ефективності залишаються перспективним напрямом подальших досліджень.

Висновки до розділу 3

У третьому розділі виконано проєктування, реалізацію та первинне дослідження інструменту статичного таінт-аналізу simple_taint.py, призначеного для виявлення неконтрольованих потоків даних у Python-коді. Показано, що навіть базові механізми аналізу на основі обходу AST здатні визначати шляхи поширення потенційно небезпечної інформації від джерел до критичних приймачів.

Розроблена архітектура включає обробку присвоєнь, перенесення стану «зараженості» та перевірку аргументів небезпечних функцій, що забезпечує виявлення прямих і простих непрямих шляхів передачі даних. Тестування на файлі test_code.py підтвердило коректність роботи інструменту, зокрема його здатність фіксувати типові загрози, такі як командні ін’єкції через виклики os.system().

Отримані результати свідчать, що навіть початкова реалізація статичного таінт-аналізу є дієвим засобом виявлення базових вразливостей Python-застосунків і може слугувати основою для подальшого розвитку. Перспективними напрямами є розширення правил аналізу, підтримка складніших моделей поширення даних та експериментальна перевірка запропонованої архітектури інтеграції машинного навчання.

Загалом проведена робота підтверджує ефективність статичного таінт-аналізу для виявлення небезпечних конструкцій і підвищення безпеки Python-коду.

ВИСНОВКИ

У кваліфікаційній роботі було всебічно досліджено методи таінт-аналізу для виявлення вразливостей у Python-коді, проведено аналіз сучасних інструментів забезпечення безпеки програмних систем та розроблено власний інструмент статичного аналізу simple_taint.py, який демонструє практичне застосування моделі поширення неконтрольованих даних у програмі. Попри використання спрощеної архітектури, реалізований інструмент повністю відтворює фундаментальні принципи статичного таінт-аналізу, що доводить його ефективність у навчальних, дослідницьких і практичних цілях.

У ході виконання роботи було опрацьовано теоретичні аспекти руху небезпечних даних у Python-застосунках. Встановлено, що динамічна природа мови, відсутність статичних типів, гнучкість присвоєнь та взаємодія зі зовнішніми джерелами даних створюють суттєві ризики виникнення вразливостей. Особлива увага була приділена ролі джерел (sources) та приймачів (sinks), адже саме вони визначають, у яких точках коду можливий небезпечний вплив на систему. Аналіз показав, що більшість критичних атак – SQL-ін’єкції, командні ін’єкції, XSS, виконання довільного коду – виникають саме через неконтрольований потік даних у програмі.

Проведене дослідження сучасних моделей аналізу (статичної, динамічної, гібридної та ML-орієнтованої) дало змогу визначити їхні переваги та обмеження. Статичний аналіз ефективний на етапі розробки, але може пропускати залежності, що формуються під час виконання. Динамічний аналіз є точнішим, проте потребує ресурсів та спеціального середовища. Гібридні методи поєднують переваги обох підходів і вважаються оптимальними для великих Python-проєктів.

Показано, що застосування методів оптимізації – індексації графів даних, агрегування метрик, побудови складених індексів – суттєво підвищує швидкодію та точність інструментів статичного аналізу, що особливо важливо в умовах великих програмних систем. У роботі також обґрунтовано доцільність

використання машинного навчання для зменшення кількості хибнопозитивних результатів та покращення розпізнавання складних шаблонів поширення даних.

Окремою практичною частиною стало створення та тестування інструменту *simple_taint.py*. Проведений аналіз показав, що він коректно обробляє джерела неконтрольованих даних, відстежує їх передавання між змінними та фіксує можливі небезпечні виклики системних функцій. Отримані результати підтверджують правильність реалізованого механізму статичного таінт-аналізу та його ефективність для базового виявлення типових загроз у Python-коді.

Загалом, виконане дослідження доводить, що таінт-аналіз є одним із найбільш ефективних методів виявлення вразливостей у Python-коді, а розроблений інструмент може бути використаний як основа для створення більш розширених, промислових систем аналізу безпеки. Виконана робота дозволила сформувати теоретичне й практичне підґрунтя для подальших досліджень у сфері статичної та динамічної безпеки, автоматизації аудиту коду та інтеграції ML-моделей у сучасні засоби кіберзахисту.

Отримані результати підтверджують актуальність обраної теми та демонструють, що розробка власного інструменту статичного таінт-аналізу є важливим кроком до підвищення якості, надійності та безпеки Python-застосунків, особливо в умовах стрімкого зростання кіберзагроз і масштабування сучасних інформаційних систем

ПЕРЕЛІК ПОСИЛАНЬ

1. Python-security. pyt: A Static Analysis Tool for Detecting Security Vulnerabilities in Python Web Applications // GitHub. – 2020–2025. – URL: https://github.com/python-security/pyt.
2. Testing Guide навч.-метод. посібник / Держ. університет інтелектуальних технологій і зв'язку. – Одеса, 2023. – URL: https://duikt.edu.ua/uploads/l_1823_91992291.pdf.
3. She D., Chen Y., Shah A., Ray B., Jana S. Neutaint: Efficient Dynamic Taint Analysis with Neural Networks // arXiv.org. – 2019. – arXiv:1907.03756. – URL: https://arxiv.org/pdf/1907.03756.
4. Зворотна розробка та аналіз шкідливого програмного забезпечення [Електронний ресурс] : навч. презентація / КПІ ім. Ігоря Сікорського, Інститут ІПСА. – Київ, 2024. – URL: https://infosec-kpi.in.ua/assets/files/re_slides.pdf.
5. Semgrep. Taint analysis overview // Semgrep Documentation. – URL: https://semgrep.dev/docs/writing-rules/data-flow/taint-mode/overview.
6. Глінчук О. Засоби мови програмування Python для кібербезпеки: зб. матеріалів конф. / Волинський національний університет ім. Лесі Українки. – Луцьк ; – С. 87–90. – URL: https://evnuir.vnu.edu.ua/bitstream/123456789/26227/1/hlynchuk_odesa.pdf.
7. Kamodulin. tainted: Dynamic taint analysis for Python // GitHub. – URL: https://github.com/kamodulin/tainted.
8. Безпека коду IT-продукту та виявлення його вразливостей / Wezom Agency. – Херсон, 2024. – URL: https://wezom.com.ua/ua/blog/bezpeka-kodu-it-produktu-yak-viyaviti-vrazlivosti.
9. Meta. Pysa (Static Analysis for Security) // Pyre Documentation. – URL: https://pyre-check.org/docs/pysa-basics/.
10. DjangoChecker: Applying extended taint tracking and server-side scanning for detecting XSS and SQL injection in Django-based web applications // OUCI :

український індекс наукового цитування. – 2023. – URL: https://ouci.dntb.gov.ua/en/works/9jZrrqZ9/.

11. Trail of Bits. Polytracker // PyPI. – URL: https://pypi.org/project/polytracker/.
12. Експериментальне дослідження, програмна реалізація та впровадження методів статичного аналізу безпеки програмного коду / О. О. Кузнецов, В. О. Хорошко // DataProtect. – 2024. – № 1. – URL:https://journals.dut.edu.ua/index.php/dataprotect/article/view/3301.
13. Tracking Information Flow // The Fuzzing Book. – URL: https://www.fuzzingbook.org/html/InformationFlow.html.
14. Conti J. J., Russo A. A taint mode for python via a library // Chalmers University of Technology. – URL: https://www.cse.chalmers.se/~russo/juanjo.htm.
15. How to Make Taint Analysis Precise // OUCI : український індекс наукового цитування. – 2023. – URL: https://ouci.dntb.gov.ua/en/works/loKgrXzl/.
16. Özkan C. What Is Taint Analysis? A Guide for Developers and Security Researchers // Medium. – 2025. – URL: https://can-ozkan.medium.com/what-is-taint-analysis-a-guide-for-developers-and-security-researchers-11f2ad876ea3.
17. Огляд сучасних інструментів автоматизованого статичного аналізу коду програм / О. В. Барановський, О. С. Лозинський // Вісник Національного університету «Львівська політехніка». Серія: Інформаційні системи та мережі. – 2025. – № 1073. – С. 197–205. – URL: https://science.lpnu.ua/sites/default/files/journal-paper/2025/may/38975/k250473-197-205.pdf.
18. Quan H., Wang J., Li X. et al. An Empirical Study of Vulnerabilities in Python Packages and Their Detection // arXiv.org. – 2024. – arXiv:2509.04260v1. – URL: https://arxiv.org/html/2509.04260v1.
19. Triton: A dynamic binary analysis library // Triton Library. – URL: https://triton-library.github.io/.
20. A Python Security Analysis Framework in Integrity Verification // OUCI : український індекс наукового цитування. – 2024. – URL: https://ouci.dntb.gov.ua/en/works/4Y61aOql/.

21. Effendi S. D. B., Pinho X., Dreyer A. M., Yamaguchi F. Scalable Language Agnostic Taint Tracking using Explicit Data Dependencies // arXiv.org. – 2025. – arXiv:2506.06247. – URL: https://arxiv.org/pdf/2506.06247.
22. Campbell G. A. What is “taint analysis” and why do I care? // SonarSource Blog. – URL: https://www.sonarsource.com/blog/what-is-taint-analysis/.
23. Gopinath R., Görz P. Mutation Analysis with Execution Taints // arXiv.org. – 2024. – arXiv:2403.01146. – URL: https://arxiv.org/pdf/2403.01146.
24. Taint Tracking Basics // ICanDoThese. – URL: https://icandothese.com/docs/tech/development/codeql/04_taint_tracking/.
25. Bouzenia I., Bajaj P. K., Pradel M. DyPyBench: A Benchmark of Executable Python Software // arXiv.org. – 2024. – arXiv:2403.00539v1. – URL: https://arxiv.org/html/2403.00539v1.
26. Lyons D., Zahra S. Using Taint Analysis and Reinforcement Learning (TARL) to Repair Autonomous Robot Software // arXiv.org. – 2020. – arXiv:2005.03813. – URL: https://arxiv.org/pdf/2005.03813.
27. Venkatesh A. P. S., Sabu S., Wang J. et al. TypeEvalPy: A Micro-benchmarking Framework for Python Type Inference Tools // arXiv.org. – 2023. – arXiv:2312.16882. – URL: https://arxiv.org/pdf/2312.16882.
28. Understanding Sources and Sinks: A Guide to Taint Analysis // BreachForce. – URL: https://breachforce.net/source-and-sinks.
29. GitHub. TaintTracking Module // CodeQL Documentation. – URL: https://codeql.github.com/codeql-standard-libraries/python/semmle/python/dataflow/old/TaintTracking.qll/module.TaintTracking.html.
30. Taint Tracking in CodeQL // SecurityGrind. – URL: https://securitygrind.com/taint-tracking-in-codeql/.
31. Qian Z., Zhong F., Hu Q. et al. Software Vulnerability Analysis Across Programming Language and Program Representation Landscapes: A Survey // arXiv.org. – 2025. – arXiv:2503.20244v1. – URL: https://arxiv.org/html/2503.20244v1.

32. Boxler D., Walcott K. R. Static Taint Analysis Tools to Detect Information Flows – UCCS, 2018. – URL: https://faculty.uccs.edu/kwalcott/wp content/uploads/sites/49/2024/01/SERP18DanBoxler.pdf.
33. Ahmed K. Dynamic Taint Analysis in Python – URL: https://kamranmahmed.com/tainted.pdf.
34. SecureFixAgent: A Hybrid LLM Agent for Automated Python Static Vulnerability Repair // arXiv.org. – 2025. – arXiv:2509.16275v1. – URL: https://arxiv.org/html/2509.16275v1.

ДОДАТКИ

Додаток А. Фрагменти кодів

```python
import ast
import sys

SOURCES = {"input", "request_args"}
SINKS = {"os.system", "eval", "cursor.execute"}

class SimpleTaintAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.tainted_vars = set()
        self.warnings = []

    def visit_Assign(self, node):
        if isinstance(node.value, ast.Call) and isinstance(node.value.func, ast.Name):
            if node.value.func.id in SOURCES:
                for target in node.targets:
                    if isinstance(target, ast.Name):
                        self.tainted_vars.add(target.id)
        elif isinstance(node.value, ast.Name):
            if node.value.id in self.tainted_vars:
                for target in node.targets:
                    if isinstance(target, ast.Name):
                        self.tainted_vars.add(target.id)
        self.generic_visit(node)

    def visit_Call(self, node):
        if isinstance(node.func, ast.Name):
            func_name = node.func.id
        elif isinstance(node.func, ast.Attribute) and isinstance(node.func.value, ast.Name):
            func_name = f"{node.func.value.id}.{node.func.attr}"
        else:
            func_name = ""

        if func_name in SINKS:
            for arg in node.args:
                if isinstance(arg, ast.Name) and arg.id in self.tainted_vars:
                    self.warnings.append(
                        f"[!] WARNING: Tainted variable '{arg.id}' flows into {func_name} (line {node.lineno})"
                    )
        self.generic_visit(node)

```

Рисунок А.1 – Код simple_taint.py (частина 1)

```python
def analyze_file(filename: str):
    with open(filename, "r", encoding="utf-8") as f:
        code = f.read()
    tree = ast.parse(code, filename=filename)
    analyzer = SimpleTaintAnalyzer()
    analyzer.visit(tree)
    return analyzer.warnings

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python simple_taint.py <file_to_analyze.py>")
        sys.exit(1)

    filename = sys.argv[1]
    results = analyze_file(filename)

    if results:
        print(f"Results for {filename}:")
        for r in results:
            print(r)
    else:
        print(f"No taint issues found in {filename}")
```

Рисунок А.2 – Код simple_taint.py (частина 2)

```python
import os

user_input = input("Enter command: ")
cmd = user_input
os.system(cmd)

name = request_args("name")
query = "SELECT * FROM users WHERE name='" + name + "'"
cursor.execute(query)

safe = "ls -la"
os.system(safe)
```

Рисунок А.3 – Код test_code.py

Додаток Б. Запуск і результат виконання

```
(venv) C:\Users\warrior\PycharmProjects\Diplom\venv>python simple_taint.py test_code.py
Results for test_code.py:
[!] WARNING: Tainted variable 'cmd' flows into os.system (line 5)
```

Рисунок Б.1 – Запуск і результат виконання

Додаток В. Реалізація ML-компонента для simple_taint.py

```
import ast
import pickle
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler

#Крок 1: Розширення класу SimpleTaintAnalyzer
2 usages
class MLEnhancedTaintAnalyzer(ast.NodeVisitor):
    def __init__(self, ml_model_path=None):
        self.tainted_vars = set()
        self.warnings = []
        self.ml_model = None
        self.scaler = None

        # Завантаження ML-моделі
        if ml_model_path:
            self.load_model(ml_model_path)

        # Контекст для аналізу
        self.current_context = {
            'try_blocks': 0,
            'sanitizations': [],
            'validations': [],
            'transformations': []
        }

    1 usage
    def load_model(self, model_path):
        """Завантаження натренованої ML-моделі"""
        with open(model_path, 'rb') as f:
            model_data = pickle.load(f)
            self.ml_model = model_data['model']
            self.scaler = model_data['scaler']
```

```python
#Крок 2: Екстракція ознак
1 usage (1 dynamic)
def extract_features(self, node, warning_info):
    """
    Екстракція ознак для ML-класифікації
    """
    features = {}

    # 1. Структурні ознаки
    features['call_depth'] = self._get_call_depth(node)
    features['num_vars_in_chain'] = len(self._get_taint_chain(warning_info))
    features['ast_node_type'] = self._encode_node_type(node)

    # 2. Контекстні ознаки
    features['in_try_block'] = int(self.current_context['try_blocks'] > 0)
    features['has_sanitization'] = int(len(self.current_context['sanitizations']) > 0)
    features['has_validation'] = int(len(self.current_context['validations']) > 0)

    # 3. Ознаки джерела/приймача
    features['source_risk_level'] = self._classify_source_risk(warning_info['source'])
    features['sink_risk_level'] = self._classify_sink_risk(warning_info['sink'])

    # 4. Ознаки потоку даних
    features['path_length'] = self._calculate_path_length(warning_info)
    features['num_transformations'] = len(self.current_context['transformations'])
    features['has_concatenation'] = self._has_string_concat(node)

    # 5. Семантичні ознаки (опціонально - для deep learning)
    # features['code_embedding'] = self._get_code_embedding(node)

    return np.array(list(features.values())).reshape(1, -1)


1 usage (1 dynamic)
def _classify_source_risk(self, source_name):
    """Класифікація рівня ризику джерела"""
    high_risk = {'input', 'request.args', 'request.form', 'request.json'}
    medium_risk = {'os.environ', 'sys.argv'}
```

```
    if source_name in high_risk:
        return 3
    elif source_name in medium_risk:
        return 2
    return 1
```

1 usage (1 dynamic)

```
def _classify_sink_risk(self, sink_name):
    """Класифікація рівня ризику приймача"""
    critical = {'eval', 'exec', 'os.system', '__import__'}
    high_risk = {'subprocess.call', 'cursor.execute', 'open'}

    if sink_name in critical:
        return 3
    elif sink_name in high_risk:
        return 2
    return 1
```

1 usage (1 dynamic)

```
def _has_string_concat(self, node):
    """Перевірка наявності конкатенації рядків"""
    for child in ast.walk(node):
        if isinstance(child, ast.BinOp) and isinstance(child.op, ast.Add):
            return 1
    return 0
```

```
#Крок 3: ML-класифікація попереджень
1 usage (1 dynamic)
def classify_warning(self, node, warning_info):
    if self.ml_model is None:
        # Якщо модель не завантажена, повертаємо базовий результат
        return True, 0.5

    # Екстракція ознак
    features = self.extract_features(node, warning_info)

    # Нормалізація
    if self.scaler:
        features = self.scaler.transform(features)
    # Передбачення
    prediction = self.ml_model.predict(features)[0]
    confidence = self.ml_model.predict_proba(features)[0][1]
    return bool(prediction), float(confidence)

def visit_Call(self, node):
    """Розширена версія з ML-класифікацією"""
    func_name = self._get_function_name(node)
    if func_name in SINKS:
        for arg in node.args:
            if self._is_tainted(arg):
                # Створення інформації про попередження
                warning_info = {
                    'source': self._get_original_source(arg),
                    'sink': func_name,
                    'line': node.lineno,
                    'arg': ast.unparse(arg) if hasattr(ast, 'unparse') else str(arg)
                }
                # ML-класифікація
                is_vulnerable, confidence = self.classify_warning(node, warning_info)
                if is_vulnerable and confidence > 0.6:  # Поріг впевненості
                    self.warnings.append({
                        **warning_info,
                        'confidence': confidence,
                        'priority': self._calculate_priority(confidence, warning_info)
                    })
    self.generic_visit(node)
```

```python
#Крок 4: Навчання ML-моделі
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import pickle

def train_ml_model(training_data_path, output_model_path):
    """
    Навчання ML-моделі для класифікації вразливостей

    training_data_path: шлях до датасету з розміченими прикладами
    output_model_path: шлях для збереження моделі
    """

    # Завантаження даних
    # Формат: CSV з колонками features + label (1 - вразливість, 0 - false positive)
    import pandas as pd
    data = pd.read_csv(training_data_path)

    X = data.drop('is_vulnerable', axis=1)
    y = data['is_vulnerable']

    # Розділення на train/test
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )

    # Нормалізація
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Навчання Random Forest
    model = RandomForestClassifier(
        n_estimators=100,
        max_depth=10,
        min_samples_split=5,
        class_weight='balanced',
        random_state=42
```

```
)

model.fit(X_train_scaled, y_train)

# Оцінка моделі
y_pred = model.predict(X_test_scaled)
print("Classification Report:")
print(classification_report(y_test, y_pred))

# Збереження моделі
with open(output_model_path, 'wb') as f:
    pickle.dump({
        'model': model,
        'scaler': scaler,
        'feature_names': X.columns.tolist()
    }, f)

print(f"Model saved to {output_model_path}")

# Важливість ознак
feature_importance = pd.DataFrame({
    'feature': X.columns,
    'importance': model.feature_importances_
}).sort_values('importance', ascending=False)

print("\nTop 10 Important Features:")
print(feature_importance.head(10))
```

```python
#Крок 5: Генерація тренувальних даних
def generate_training_dataset(code_samples_dir, output_csv):
    import os
    import pandas as pd
    dataset = []
    for filename in os.listdir(code_samples_dir):
        if not filename.endswith('.py'):
            continue
        filepath = os.path.join(code_samples_dir, filename)
        # Визначення мітки (з імені файлу або з метаданих)
        is_vulnerable = '_vulnerable' in filename
        # Базовий аналіз
        analyzer = MLEnhancedTaintAnalyzer()
        tree = ast.parse(open(filepath).read())
        analyzer.visit(tree)
        # Екстракція ознак для кожного попередження
        for warning in analyzer.warnings:
            features = analyzer.extract_features(tree, warning)
            features_dict = dict(enumerate(features.flatten()))
            features_dict['is_vulnerable'] = int(is_vulnerable)
            dataset.append(features_dict)
    # Збереження у CSV
    df = pd.DataFrame(dataset)
    df.to_csv(output_csv, index=False)
    print(f"Dataset saved: {len(df)} samples")
#Оновлена функція analyze_file
1 usage
def analyze_file(filename, ml_model_path=None):
    
    with open(filename, 'r', encoding='utf-8') as f:
        code = f.read()
    tree = ast.parse(code, filename=filename)
    # Використання ML-розширеного аналізатора
    analyzer = MLEnhancedTaintAnalyzer(ml_model_path=ml_model_path)
    analyzer.visit(tree)
    # Сортування попереджень за пріоритетом
    warnings = sorted(
        analyzer.warnings,
        key=lambda w: w.get('confidence', 0.5) * w.get('priority', 1),
        reverse=True
    )
```

```python
if __name__ == "__main__":
    import sys

    if len(sys.argv) < 2:
        print("Usage: python simple_taint_ml.py <file_to_analyze> [ml_model_path]")
        sys.exit(1)

    file_to_analyze = sys.argv[1]
    ml_model_path = sys.argv[2] if len(sys.argv) > 2 else None

    warnings = analyze_file(file_to_analyze, ml_model_path)

    if warnings:
        print(f"\n Found {len(warnings)} potential vulnerabilities:\n")
        for i, w in enumerate(warnings, 1):
            confidence = w.get('confidence', 0.5)
            priority = w.get('priority', 'MEDIUM')
            print(f"{i}. [{priority}] Line {w['line']}: "
                  f"Tainted data from '{w['source']}' reaches '{w['sink']}' "
                  f"(confidence: {confidence:.2%})")
    else:
        print("No vulnerabilities detected")
```