МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Одеська юридична академія»

Факультет кібербезпеки та інформаційних технологій

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Програмна реалізація процесу відстеження поширення по програмі неперевірених зовнішніх даних»

Рівень «бакалавр»

Галузь знань 12 «Інформаційні технології»,

спеціальність 122 «Комп'ютерні науки»

Виконав здобувач 4 курсу

Колісніченко Руслан Леонідович

Керівник к.т.н., доцент

Чикунов Павло Олександрович

Рецензент к.т.н., доцент

Задерейко Олександр Владиславович

Одеса – 2024

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 Наталія Логінова _____

«____ »_____ 2024 року

З А В Д А Н Н Я

на кваліфікаційну (бакалаврську) роботу

здобувачу Колісніченко Руслану Леонідовичу

1. Тема роботи: «Програмна реалізація процесу відстеження поширення по програмі неперевірених зовнішніх даних».

Керівник роботи: к.т.н, доцент Чикунов Павло Олександрович

затверджені наказом НУ «ОЮА» від «02» квітня 2024 року № 809-06

2. Строк подання здобувачем роботи «20» травня 2024 року

3. Дата видачі завдання «15» вересня 2023 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області та існуючих аналогів	02.10.2023	
2	Формулювання цілей та завдань дослідження	04.12.2023	
3	Проєктування статичного аналізатора	14.02.2024	
4	Програмна реалізація та тестування статичного аналізатора	26.04.2024	
5	Оформлення звітної частини	17.05.2024	

Здобувач _____Руслан Колісніченко_____

(підпис) (ім’я та прізвище)

Керівник роботи _____Павло Чикунов_____

(підпис) (ім’я та прізвище)

АНОТАЦІЯ

Колісніченко Р. Л. Програмна реалізація процесу відстеження поширення по програмі неперевірених зовнішніх даних. – Кваліфікаційна робота бакалавра за спеціальністю 122 «Комп'ютерні науки», освітньою програмою «Комп’ютерні науки».

Кваліфікаційна робота викладена на 42 сторінках основного тексту і 56 сторінках загального тексту, містить 3 розділи, 13 рисунків, 1 таблицю, 2 додатка, 28 використаних джерел.

Метою виконання роботи є проєктування, програмна реалізація та тестування статичного аналізатору Java-коду з використанням техніки taint-аналізу для відстеження поширення по програмі неперевірених зовнішніх даних.

Об’єктом дослідження є процес статичного аналізу текстів програм на мовах високого рівня, який дозволяє відстежувати поширення неперевірених зовнішніх даних за програмою для виявлення потенційних безпекових загроз.

Предметом дослідження є техніки taint-аналізу текстів програм для пошуку потенційно підозрілих змінних, які може бути змінені зовнішнім користувачем.

У кваліфікаційній роботі розроблено плагін для віртуальної машини UnitTestBot та генератора коду IntelliJ IDEA, необхідний для підтримки taint-аналізу відстеження зовнішніх даних. Виконана розробка алгоритмів символьного виконання для пошуку SQL-ін'єкції. Виконана розробка модулю формування звітів про знайдені помилки. Виконано тестування та оцінка якості програмної реалізації. Розроблений застосунок може використовуватись розробниками для пошуку у власних програмах критичних секцій.

Ключові слова: taint-аналіз, вразливості, неперевірені зовнішні данні, символьне виконання, статичний аналіз, UnitTestBot, IntelliJ IDEA, алгоритми обробки правил.

ABSTRACT

Kolisnychenko R. L. Program implementation of the process of tracking the spread according to the program of unverified external data. – Qualification work of a bachelor in specialty 122 "Computer Science", Educational program "Computer Science".

The qualification work consists of 42 pages of main text and 56 pages of total text, contains 3 chapters, 13 figures, 1 table, 2 appendixes and references to 28 sources.

The purpose of the work is the design, software implementation and testing of a static Java code analyzer using the taint analysis technique to track the spread of unverified external data through the program.

The object of the research is the process of static analysis of program texts in high-level languages, which allows tracking the spread of unverified external data within the program to detect potential security threats.

The subject of the research is the techniques of taint analysis of program texts to search for potentially suspicious variables that may be altered by an external user.

In the qualification work, a plugin for the UnitTestBot virtual machine and the IntelliJ IDEA code generator has been developed to support taint analysis for tracking external data. Development of symbolic execution algorithms for searching for SQL injection has been performed. A module for generating reports on found errors has been developed. Testing and evaluation of the quality of software implementation have been carried out. The developed application can be used by developers to search for critical sections in their own programs.

Keywords: taint-analysis, vulnerabilities, unverified external data, symbolic execution, static analysis, UnitTestBot, IntelliJ IDEA, rule processing algorithms.

ЗМІСТ

ВСТУП

Аналіз текстів програм для пошуку потенційних загроз – це важливий етап у виявленні потенційних проблем в програмному забезпеченні. Цей аналіз зазвичай виконується з метою виявлення можливих вразливостей, вбудованих помилок, порушень безпеки та інших проблем, які можуть стати джерелом загроз для безпеки системи чи користувачів. Програми для пошуку загроз можуть аналізувати вихідний код програмного забезпечення з метою виявлення певних конструкцій або використання патернів, які можуть вказувати на потенційні проблеми безпеки.

Існують кілька методів та технік пошуку потенційних загроз в текстах програм, зокрема так званий taint-аналіз – це метод аналізу безпеки програмного забезпечення, який виявляє потенційно небезпечні дії або вразливості, враховуючи потоки даних через програму. Основна ідея полягає в тому, щоб відстежувати, як дані, які можуть бути контрольовані зловмисником, проникають у систему та як вони обробляються програмою.

В процесі виконання програми taint-аналіз визначає, які дані можуть бути забруднені або підозрілими, і як вони можуть вплинути на безпеку програми. Основні заходи taint-аналізу включають відстеження потоку даних, визначення зв'язків між даними та їх джерелами, виявлення можливих точок витоку даних тощо. В результаті він допомагає підвищити безпеку програмного забезпечення шляхом виявлення та усунення можливих шляхів атаки.

Потрапляння неперевірених даних у критичні секції коду може призвести до різних вразливостей, включаючи SQL-ін'єкцію (виконання зловмисником небажаних SQL-запитів) та багато інших. Зловмисники можуть використовувати ці вразливості для порушення коректної роботи системи, отримання конфіденційних даних або проведення інших несанкціонованих операцій.

Наразі відомі дослідження [2-3, 12], у яких проблема пошуку неперевірених зовнішніх даних вирішується за допомогою застосування техніки

символьного виконання коду, яка дозволяє знайти всі набори вхідних даних, що сприяють виконанню кожного з його можливих шляхів.

Символьне виконання (symbolic execution) – це техніка аналізу програмного забезпечення, яка полягає в автоматизованому створенні символьних виразів, які представляють всі можливі шляхи виконання програми. Замість фактичного виконання програми з конкретними вхідними даними, як в традиційному методі тестування, символьне виконання використовує символьні значення для вхідних даних та відстежує, як ці значення впливають на виконання програми через всі можливі шляхи виконання.

Основна ідея полягає у тому, щоб створити символьні вирази для вхідних даних програми та аналізувати вирази, щоб знайти всі можливі шляхи виконання програми, включаючи шляхи, які можуть призвести до помилок або небажаних станів програми. Символьне виконання може бути корисним для виявлення вразливостей програм, таких як витоки пам'яті, ділення на нуль, недоступний код та інші проблеми безпеки.

Актуальність дослідження обумовлена високою затребуваністю програмних систем виявлення дефектів, помилок та вразливостей у програмах на мовах високого рівня, а також необхідністю модернізації процесу відстеження потенційно небезпечних або ненадійних даних, що вводяться у програму зловмисниками і можуть бути використані для атаки.

Метою виконання роботи є проєктування, програмна реалізація та тестування статичного аналізатору Java-коду для відстеження поширення неперевірених зовнішніх даних за програмою.

Для досягнення мети було сформульовано такі завдання:

– провести огляд предметної галузі та підходів існуючих статичних аналізаторів з метою виявлення їх переваг та недоліків;

– реалізувати модуль, який на основі тестів, згенерованих інструментом автоматичної генерації тестів UnitTestBot, буде надаватиме користувачеві звіт із знайденими помилками;

– розробити інтерфейс користувача для перегляду звіту статичного аналізу і вбудувати його в плагін UnitTestBot для IntelliJ IDEA;

– модифікувати ядро символьної віртуальної машини, вбудувавши в нього техніку taint-аналізу, яка дозволить виявляти порушення безпеки в коді;

– протестувати розроблений статичний аналізатор для визначення ефективності знаходження помилок та вразливостей у різних програмах.

Об’єкт дослідження – це процес статичного аналізу текстів програм на мовах високого рівня, який дозволяє відстежувати поширення неперевірених зовнішніх даних за програмою для виявлення потенційних безпекових загроз.

Предметом дослідження є техніки taint-аналізу текстів програм для пошуку потенційно підозрілих змінних у певних вираженнях, які може бути змінені зовнішнім користувачем, що становить потенційну загрозу безпеці.

Методи досліджень: методи системного аналізу для дослідження предметної області, методи математичного моделювання об’єктів пошуку, методи програмної реалізації алгоритмів taint-аналізу, методи управління процесами життєвого циклу програмної системи, методи забезпечення інформаційної безпеки.

Практичне значення здобутих результатів. У символьну віртуальну машину вбудована техніка taint-аналізу для розширення можливостей аналізу текстів програм для пошуку потенційних загроз, а саме випадків використання неперевірених даних у критичних секціях програми.

Апробація кваліфікаційної роботи: публікація тези доповіді «Програмна реалізація процесу відстеження поширення по програмі неперевірених зовнішніх даних» на X Всеукраїнській науково-практичній конференції «Інформаційне суспільство: проблеми та перспективи» (Одеса, 24 травня 2024 р.).

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИБІР ЗАСОБІВ РОЗРОБКИ

1.1 Специфіка процесу відстеження неперевірених зовнішніх даних

Taint-аналіз (taint-checking) [1] – це метод аналізу текстів програм на мовах високого рівня, який дозволяє відстежувати поширення неперевірених зовнішніх даних за програмою для виявлення потенційних безпекових загроз. Ключова ідея полягає в тому, що будь-яка змінна, яка може бути змінена зовнішнім користувачем, становить потенційну загрозу. Якщо ця змінна використовується у певному вираженні, то значення цього виразу також стає підозрілим. Алгоритм відслідковує і сповіщає про ситуації, коли будь-яка з позначених змінних використовується для виконання небезпечних команд до бази даних або операційної системи комп'ютера. Taint-аналіз потребує конфігурації, в якій методам програми можна призначити одну з ролей: taint-source (джерело неперевірених даних) або taint-sink (приймач, деяка критична секція програми).

Джерелом неперевірених даних може бути метод зчитування даних користувача або метод отримання параметра HTTP-запиту. Змінні, до яких записується результат виконання taint-source, називаються позначеними. Приймачем може бути метод прямого звернення до бази даних або файлової системи, а також будь-яка інша потенційно небезпечна операція.

Також є дві допоміжні сутності, які потрібні для розширення можливостей алгоритму. Taint-pass – це функція, яка позначає значення, що повертається з урахуванням міток у її аргументах. Залежно від реалізації, мітки можуть накладатися не тільки на результат методу, а й на сам об'єкт, на вхідні параметри методу також. Taint-cleaner – це функція, яка видаляє заданий набір міток із переданих аргументів. Найчастіше це деякий метод валідації, який перевіряє, чи дійсно користувач ввів дані в очікуваному форматі.

Алгоритм taint-аналізу сканує граф потоку даних, намагаючись виявити маршрут між методом з множини taint-sources і методом з taint-sinks. Основною

проблемою класичного алгоритму, як і в усіх методів неточного аналізу, є велика кількість хибно-позитивних спрацьовувань.

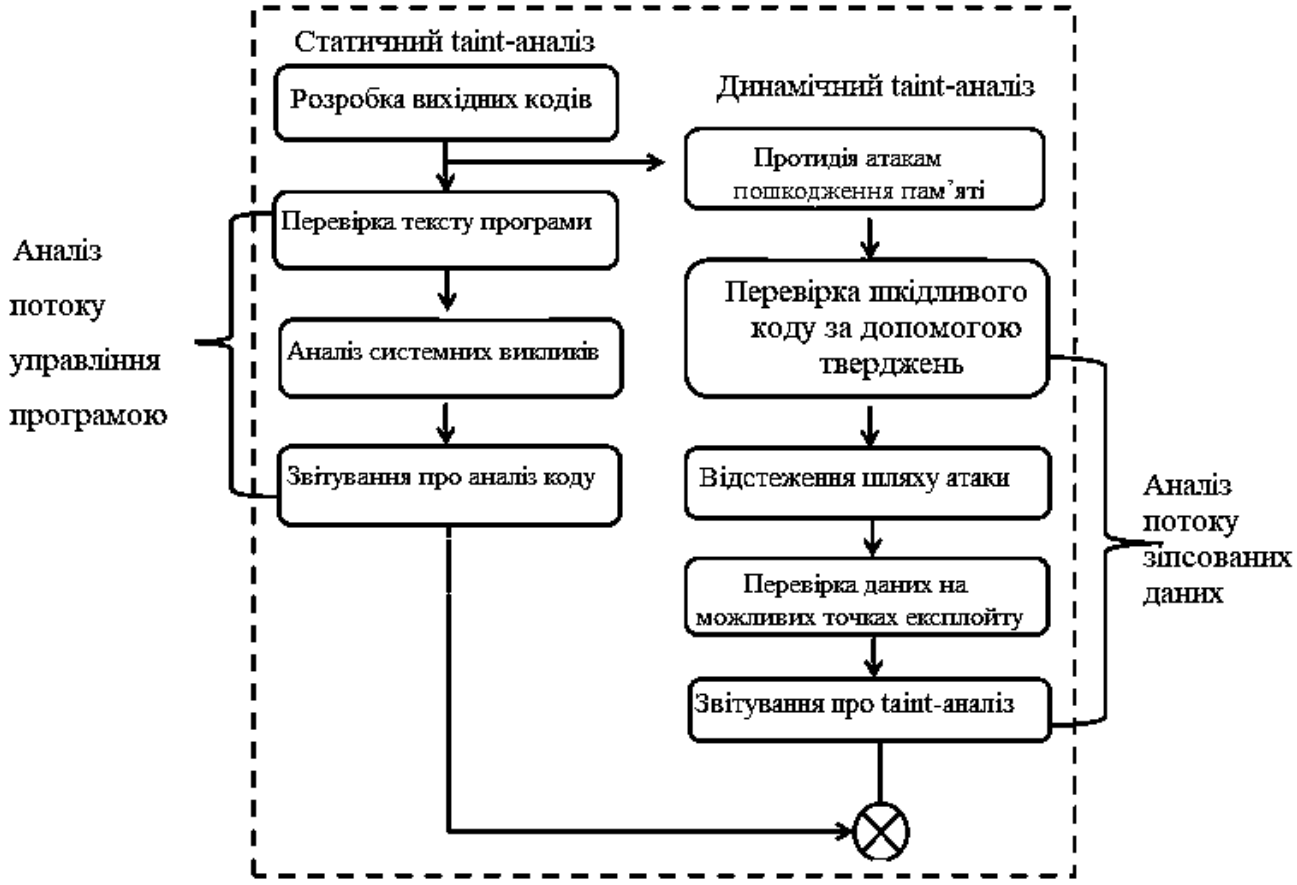

Рисунок 1.1 – Загальний алгоритм проведення taint-аналізу [1]

Для перевірки якості програмного забезпечення доводиться застосовувати багато різних інструментів. Зокрема, до них належать інструменти статичного та динамічного аналізу [14]. Статичний аналіз – це аналіз програмного забезпечення, що проводиться (на відміну динамічного аналізу) без реального виконання досліджуваних програм [4-5]. Найчастіше аналіз проводиться над вихідним кодом продукту, хоча іноді задіяний і якийсь вид об'єктного коду, наприклад, байт-код мови Java.

Для оцінки якості аналізатора коду зазвичай використовується реєстр найнебезпечніших вразливостей програмного забезпечення, що включає зокрема такі помилки, як запис або читання поза межами масиву, SQL-ін'єкція, використання неперевіреного введення користувача, розйменування нульового покажчика, а також багато інших вразливостей. Символьне виконання здатне

знаходити далеко не всі з перелічених помилок, проте його можливості можуть бути суттєво розширені за допомогою методу taint-аналізу.

Зазначені аспекти визначили необхідність розробки статичного аналізатору, який би виконував глибокий аналіз текстів програм на мові Java та знаходив серйозні проблеми, що призводять до несподіваної поведінки програми.

1.2 Аналіз статичних аналізаторів текстів програм

У зв'язку з високою затребуваністю теми автоматичного виявлення дефектів у програмах, на ринку існує безліч різних статичних аналізаторів. Список найвідоміших інструментів, що дозволяють аналізувати Java-код, включає SpotBugs, Coverity, CodeQL, Infer, а також багато інших [6, 9-11]. Перелічені інструменти непогано справляються з пошуком простих помилок або друкарських помилок, проте здебільшого не виконують глибокого дослідження коду, тобто не виявляють рідкісні сценарії виконання, що призводять до некоректної поведінки програми.

Сучасні популярні інструменти аналізу все ще досить погано знаходять уразливості в коді, пропускаючи до половини простих типових уразливостей, не кажучи про складніші, дають хибні результати різного типу і знаходять неоднакові проблеми в певному фіксованому наборі даних [13].

Одним з найпопулярніших аналізаторів вихідного коду для Java є SpotBugs [8], що розповсюджується у вигляді плагіна для IntelliJ IDEA, завдань для систем складання Ant, Gradle та Maven, а також інтерфейсу командного рядка. В ньому реалізовано різні евристики, що базуються на статичному аналізі тексту програми. Добре справляється з пошуком простих помилок або друкарських помилок, а також поганих практик у написанні коду.

Можливості статичного аналізатору суттєво розширюються при встановленні доповнення FindSecurityBugs [9], яке реалізує деякі алгоритми для пошуку вразливостей, у тому числі й taint-аналізу. Разом з цим модулем SpotBugs може знаходити помилки в безпеці, наприклад використання неперевіреного

введення користувача в критичних секціях програми (HTTP-запити, робота з файловою системою і так далі). Основним недоліком даного продукту є велика кількість хибних спрацьовувань (від 40 до 60 відсотків), які виникають через застосування методів, що ґрунтуються на дослідженні вихідного тексту програми.

Сервіс Coverity Scan складається зі статичного та динамічного аналізаторів коду на C++, Java, C#, Scala та деяких інших мовах. У Coverity реалізовано кілька підходів до пошуку помилок, у тому числі taint-аналіз, який, перш за все, використовується для знаходження випадків порушення безпеки, таких як SQL-ін'єкції або XSS (міжсайтовий скриптинг). Сервіс Coverity Scan ініційований Міністерством внутрішньої безпеки США як найбільший у світі дослідницький проект у державно-приватному секторі, зосереджений на якості та безпеці програмного забезпечення з відкритим кодом [11].

Як і у багатьох інструментів, істотний недолік Coverity – часті хибно-позитивні спрацьовування. У різних публікаціях на тему виміру частки помилкових попереджень Coverity виходить від 10 до 35 відсотків.

CodeQL – це інструмент статичного аналізу, який можна використовувати для автоматичного сканування програм на наявність уразливостей і для допомоги в ручному перегляді коду [15]. Наприклад, можна скласти запити, які перевірять чи є шлях у графі потоку даних від місця введення користувача до небезпечної ділянки коду, де ці дані застосовуються.

Найпростіший спосіб використання полягає в тому, щоб просто запустити сканер з набором стандартних запитів, а потім переглядати результати. Зазвичай, вони виявлять проблеми з якістю коду та проблеми з безпекою. Однак більш продуктивний підхід передбачає написання власних запитів, з огляду на специфіку конкретної програми. У цьому полягає основна перевага та головний недолік даного продукту. З одного боку, користувачеві надається велика гнучкість у написанні запитів, що відповідають їхнім потребам. З іншого боку, для ефективного використання інструменту потрібно не лише створити

релевантні запити для конкретного проекту, але й оволодіти складною декларативною мовою та всіма її можливостями.

Точно оцінити частоту хибно-позитивних спрацьовувань може бути складно, адже результат безпосередньо залежить від набору запитів, із якими запускається CodeQL.

Infer Static Analyzer – це статичний програмний аналізатор для мов Java, C і Objective-C. Infer постійно використовують для перевірки вибраних властивостей кожної модифікації коду для основних програм Facebook для Android та iOS, Facebook Messenger, Instagram. Наразі Infer відстежує проблеми, спричинені розіменуванням нульового покажчика та витоками ресурсів і пам’яті, які спричиняють деякі важливіші проблеми на мобільних пристроях [16].

Окремо використовують Infer.AI, загальну структуру аналізу, яка є інтерфейсом до модуля модульного механізму аналізу, який може використовуватися іншими видами програмного аналізу. Ця додаткова загальність була використана для розробки примірників Infer.AI для безпеки, паралелізму та в інших областях.

Використання формальних підходів сприяє меншій кількості видаваних хибно-позитивних результатів порівняно з описаними вище інструментами.

Проведений огляд існуючих інструментів статичного аналізу показав, що багато популярних продуктів, по-перше, допускають велику кількість хибно-позитивних спрацьовувань, а по-друге, проводять лише поверхневий аналіз, не виявляючи рідкісні сценарії виконання, що призводять до некоректної поведінки програми. Одним із способів вирішення зазначених недоліків є використання формальних методів математичної логіки.

1.3 Символьне виконання коду програм

Символьне виконання (symbolic execution) – це техніка аналізу програмного забезпечення, яка полягає в автоматизованому створенні символьних виразів, які представляють всі можливі шляхи виконання програми

[2-3, 17, 18]. Замість фактичного виконання програми з конкретними вхідними даними, як в традиційному методі тестування, символьне виконання використовує символьні значення для вхідних даних та відстежує, як ці значення впливають на виконання програми через всі можливі шляхи виконання.

Фахівці використовують символьне виконання для вирішення таких задач [18]:

– дуже швидкий ріст кількості шляхів для аналізу;

– неповна відповідність модельованого шляху до реального;

– велика складність обчислення для розрахунку вхідних даних;

– відсутність алгоритмів для аналізу деяких символьних обмежень.

Основна ідея полягає у тому, щоб створити символьні вирази для вхідних даних програми та аналізувати вирази, щоб знайти всі можливі шляхи виконання програми, включаючи шляхи, які можуть призвести до помилок або небажаних станів програми. Символьне виконання може бути корисним для виявлення вразливостей програм, таких як витоки пам'яті, ділення на нуль, недоступний код та інші проблеми безпеки.

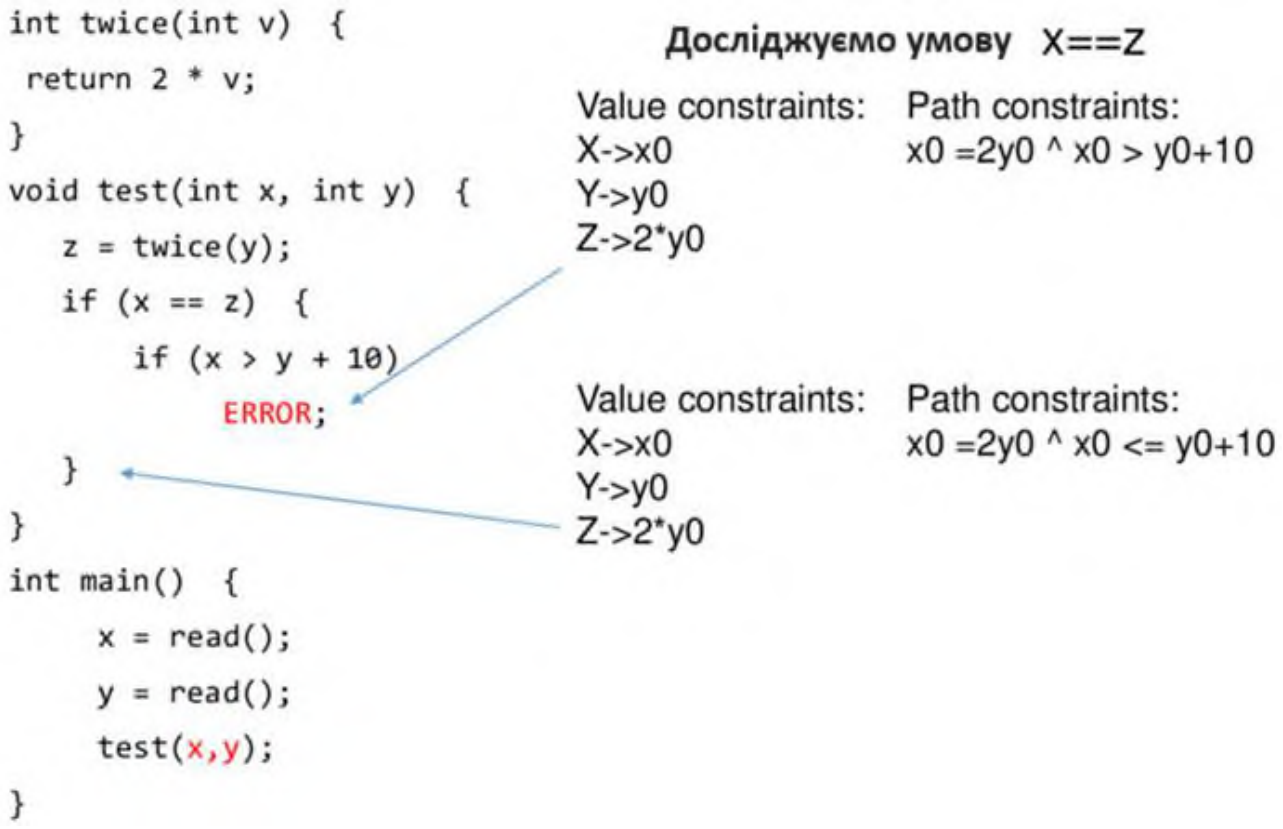

Рисунок 1.2 – Приклад символьного аналізу бінарного коду

Символьне виконання працює в умовах невизначеності вхідних даних. Невизначені дані, тобто символи або вільні змінні, є абстракцією конкретних об'єктів або змінних, якими оперує код під час звичайного виконання. Символи, на відміну від конкретних змінних, можуть набувати одразу безліч значень, які допускає відповідний їм тип даних. Ці абстракції зберігаються у спеціальній символьною пам'яті, стан якої підтримує алгоритм символьного виконання процесі своєї роботи. Крім символьної пам'яті *memory*, в стан, що підтримується, входять такі дані:

– поточна інструкція програми *stmt*, яку обробляє алгоритм;

– умова шляху *path Condition* у вигляді логічної формули, що виражає обмеження на символьні змінні для поточного шляху виконання;

– шлях виконання *path*, за яким алгоритм перейшов у цій стан.

Типовий алгоритм символьного виконання передбачає існування початкового стану, який відповідає стану програми при її запуску. Далі, якщо алгоритм зустрічає інструкцію без розгалуження, він просто оновлює поточний стан з урахуванням отриманої інформації. Наприклад, інструкція привласнення «x:=1» викликає оновлення осередку символьної пам'яті, яка відповідає змінній x.

Якщо ж алгоритм зустрічає інструкцію з розгалуженням, наприклад, if(ξ), то він дублює поточний стан *state*, отримуючи два нових стани *stateif* і *stateelse*. В стані *stateif* оновлюється умова шляху так, начебто ξ виконується, а може *stateelse* умова шляху оновлюється так, ніби ξ не виконується. Таким чином, виходять два стани, що відповідають розгалуженню. Після чого обидві умови перевіряються на виконання, і, у разі нездійсненності умов, відповідний стан видаляється з розгляду.

Робота алгоритму продовжується доти, доки стан не прийде в кінцеву інструкцію або виконання не завершиться внаслідок знаходження помилки у програмі.

Після завершення виконання символьна пам'ять та умова шляху кінцевого стану *statef* визначають обмеження на вхідні дані програми. Для того, щоб знайти конкретні вхідні дані, які задовольняють складені обмеження, використовується

SMT-рішальник (Satisfiability Modulo Theories solver) – це програмне забезпечення, яке використовується для розв'язання задач задоволення умов (SAT) з модульними теоріями [17, 19]. SAT-проблема полягає в тому, щоб визначити, чи існує таке призначення значень для булевих змінних, яке забезпечує задовільність заданого набору логічних умов. SMT-рішальник включає в себе розширення цієї концепції, дозволяючи вирішувати задачі задоволення умов з урахуванням конкретних теорій або обмежень, таких як арифметика, теорія масивів, теорія строк та інші.

SMT-рішальник використовуються в різних областях інформатики, включаючи верифікацію програмного забезпечення, аналіз вбудованих систем, абстрактну інтерпретацію, автоматичну генерацію тестів, синтез програм і т. д. Вони допомагають розв'язувати складні завдання, пов'язані з аналізом та верифікацією програм, шляхом автоматизованого вирішення великих логічних виразів з врахуванням специфічних обмежень, що визначаються теоріями. У результаті програма, запущена на знайдених вхідних даних, пройде той самий шлях виконання, що й кінцевий стан *statef*, з якого вони були отримані.

Для прикладу складемо алгоритм роботи символьної машини для коду:

```
int example(int x, int y)
{
    int a = 1;

    if (x > 27)
    {
        a + = y;
    }
    else
    {
        a += 12;
    }

    return y / a;
}
```

За підсумками аналізу необхідно встановити, чи існують якісь конкретні вхідні аргументи функції *example*, при яких у рядку 10 викидається виняток «ArithmeticException: / by zero», і, якщо є, то хочемо знайти їх.

На рисунку прямокутник – це один стан символьної машини. Він зберігає в собі символьну пам'ять та умову шляху, які для стислості позначені μ і π

відповідно. Пам'ять μ в даному прикладі зберігає кожної змінної її символьне значення, тобто для x – це αx, а для y – αy. Умова шляху π – це логічна форма зі значеннями символьних змінних, що виконується на поточному шляху виконання. Перехід від стану до стану відбувається при обробці деякої інструкції вихідного коду, записаної на відповідному ребрі графа станів.

Спочатку в μ лежать символьні змінні, на які ще не накладено жодне обмеження, а π – це формула, що завжди виконується, то є просто *true*.

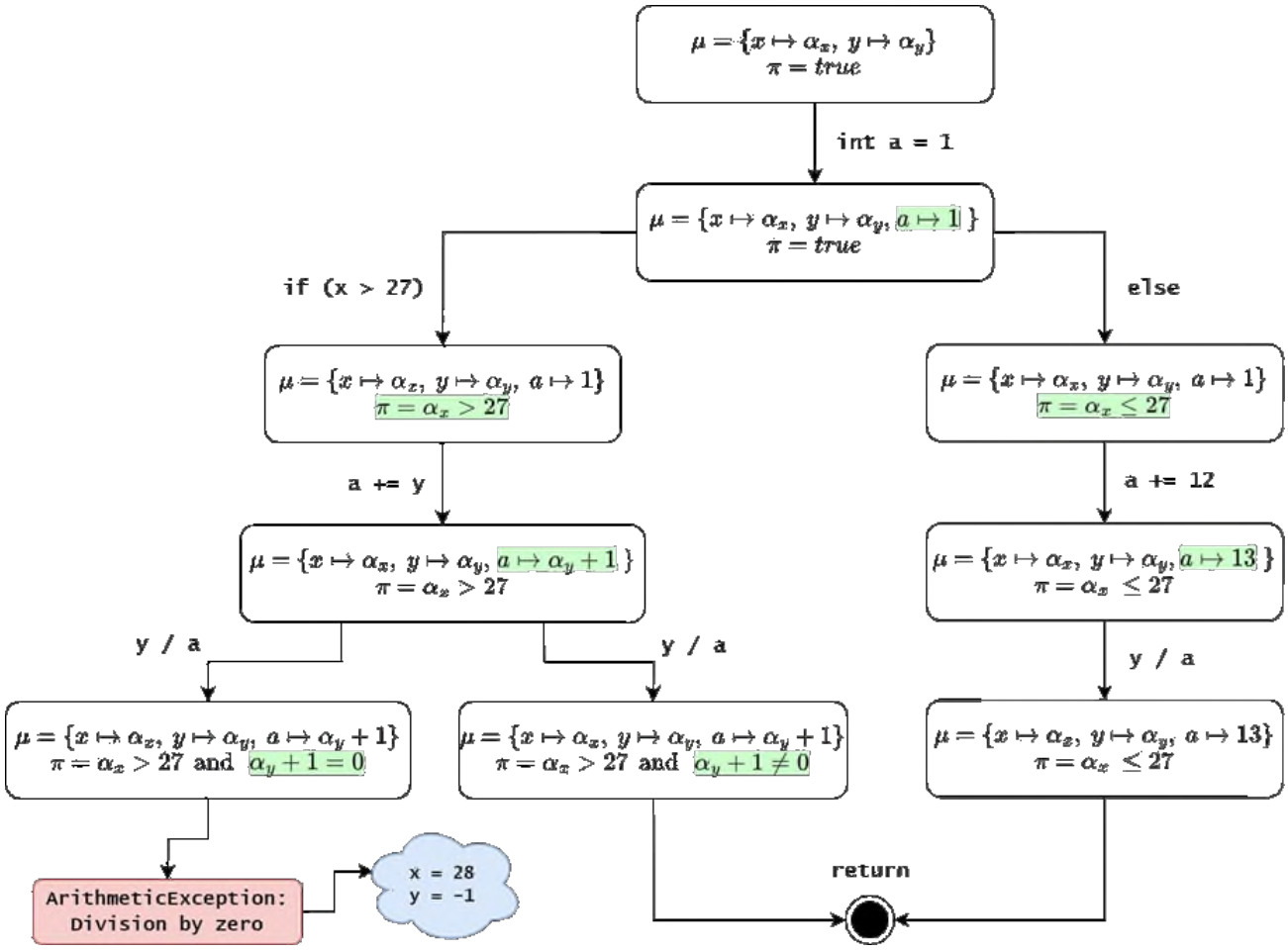

Рисунок 1.3 – Схема символьного виконання функції example

Обробляючи інструкції вихідного коду, символьна віртуальна машина підтримує значення μ і π у актуальному стані. Наприклад, після рядки int a = 1, в μ запам'яталася змінна a та її значення 1. А після обробки умовного оператора if (x > 27), виконання розгалужилося на два шляхи, у кожному з яких оновилося π: у гілці *if* воно одно $\alpha x > 27$, а у гілці *else* одно $\alpha x \leq 27$.

У певний момент алгоритм виявляє стан, що веде до викидання винятку. Після цього він звертається до SMT-рішальник із запитом знайти конкретні значення αx і αy, які виконують формулу $\pi = \alpha x > 27 \wedge \alpha y + 1 = 0$, або повідомити, що рішень немає. У даному прикладі формула виявляється здійсненна, і SMT-рішальник повертає набір αx=28, αy=−1. Таким чином, вдалося автоматично згенерувати тестовий випадок example(28, -1), який призводить до аварійного звершення програми через необроблений виняток.

1.4 Вибір загальної архітектури статичного аналізатора

У рамках дослідження як символьну віртуальну машину взято UnitTestBot [8]. UnitTestBot – це інструмент для автоматизованої генерації модульних тестів і точного аналізу коду, який може знаходити тестові випадки, що призводять до таких дефектів у програмному забезпеченні, як викидання необробленого виключення, переповнення примітивних чисельних типів, зависання методів та деякі інші [20]. Однак самі по собі знайдені значення вхідних параметрів не є прийнятним результатом статичного аналізу, оскільки якість аналізатора вимірюється не тільки в його здатності знаходити неочевидні сценарії виконання, а й у зручності використання для програміста. У зв'язку з цим було вирішено додати функціональність виконання статичного аналізу, а також відображення його результатів безпосередньо до плагіну UnitTestBot.

UnitTestBot – інструмент, який за вихідним кодом Java автоматично генерує модульні тести, намагаючись при цьому максимізувати кількість покритих інструкцій. UnitTestBot використовує для роботи власну символьну віртуальну машину, що дає змогу очікувати від згенерованих тестів теоретично повного покриття коду.

Продукт поставляється як плагін для середовища розробки IntelliJ IDEA [7], що дозволяє запускати генерацію тестів, використовуючи зручний інтерфейс користувача. Інструмент підтримує популярні системи складання Gradle та Maven, тому його досить легко застосувати на будь-якому проекті, написаному

мовою Java. Крім того, UnitTestBot надає користувачам безліч конфігурованих параметрів своєї роботи, гнучко налаштовуючись під потреби тестування конкретного проекту.

Плагін для IntelliJ IDEA задає уявлення інтерфейсу у зазначеному середовищі розробки. Модуль збирає з проекту користувача всі потрібні для генерації тестів дані, наприклад структуру класів або файли з Java байт-кодом. Іншими словами, є точкою входу в застосунок, дозволяючи налаштовувати та запускати генерацію тестів.

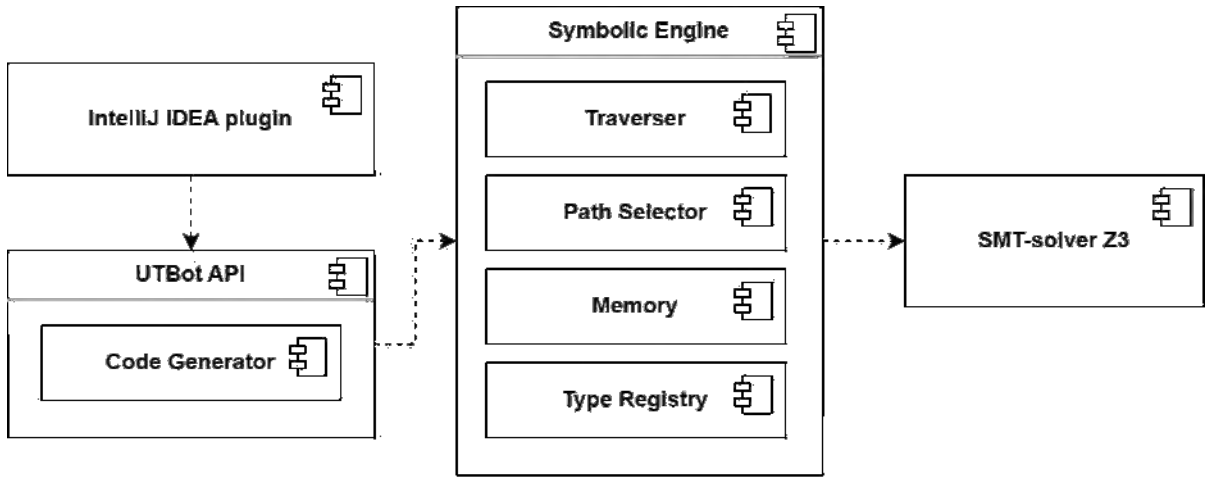

Рисунок 1.4 – Високорівнева діаграма ключових модулів проекту UnitTestBot

Пакет UTBot API відповідає за формування вхідних даних для генерації тестів, запуск символьної віртуальної машини та повернення результату її роботи у зручному для зухвалого коду вигляді. Типовий сценарій використання даного модуля виглядає так. Плагін для IntelliJ IDEA робить запит на генерацію тестів, передаючи байт-код користувальницьких класів та всі інші необхідні дані. Після чого описуваний компонент перетворює дані в потрібний формат і запускає на них символьне виконання. Результат роботи символьної машини надходить у Code Generator, який генерує код тестів мовою Java, що відповідає знайденим тестовим випадкам. У результаті отриманий файл із тестами повертається в модуль плагіна та відображається на екрані користувача.

Модуль Symbolic Engine – це символьна віртуальна машина всередині UnitTestBot, тобто центральний та найскладніший компонент проекту. Як вхідні

дані, він очікує байт-код від методів, а результатом виконання є безліч тестових випадків, тобто наборів вхідних параметрів методу. У процесі роботи модуль робить запити до SMT-рішальнику Z3, як було описано у загальному алгоритмі символьного виконання.

Робота модуля на високому рівні архітектури виглядає таким чином. На кожному кроці поточний стан витягується з Path Selector, потім оброблюється в модулі Traverser, який повертає кілька нових станів. Для кожного з отриманих станів перевіряється, чи воно є термінальним. Якщо так, то воно запам'ятовується як результат виконання, а інакше кладеться в чергу станів усередині Path Selector.

Path Selector – це клас, який приймає рішення про те, яка інструкція програми повинна бути оброблена наступною. Зберігає в собі деякий список доступних станів (при старті лише один початковий), дозволяє додавати туди новий стан, а також знаходити наступний стан. Який саме стан буде вилучено зі списку, визначається на підставі реалізованої всередині евристики.

Компонент Traverser обробляє поданий на вхід стан. Він містить інформацію про графу потоку управління, ієрархії класів у програмі та системі символьних типів. Traverser, залежно від стану та поточної інструкції створює набір нових станів, в яких оновлено символьну пам'ять і додано необхідні логічні обмеження для відповідного шляху виконання.

Компонент Memory відповідає за символьне уявлення пам'яті стану в програму. Компонент Type Registry містить інформацію про систему типів в аналізованому коді. Він дає змогу дізнатися про тип символьного об'єкта під час виконання.

2 ПРОЄКТУВАННЯ СТАТИЧНОГО АНАЛІЗАТОРА

2.1 Модифікація віртуальної машини для підтримки taint-аналізу

Далі наведено опис процесу вбудовування технік taint-аналізу у вибрану символьну віртуальну машину UnitTestBot.

На вхід статичного аналізатору надходить програма як інструкцій байт-коду. Об'єкт PathSelector визначає порядок обходу символьних станів, Traverser відповідає за обробку однієї інструкції, а також підтримує символьну пам'ять Memory та система типів TypeRegistry в актуальному вигляді. У термінальному стані робиться запит до SMT-рішателю з метою визначити здійсненність пройденого шляху і знайти для нього конкретні вхідні дані.

Основна ідея реалізованого підходу полягає в тому, що з кожною символьною змінною зіставляється taint-vector – тобто 64-бітове ціле значення, кожен біт якого відповідає за наявність мітки з номером і в цьому об'єкті. Після чого, в процесі роботи символьної машини, ці зіставлення підтримуються та оновлюються відповідно до класичного алгоритму taint-аналізу.

Зроблені в цій роботі зміни здебільшого зачіпають класи Traverser та Memory. Також були додані нові компоненти TaintModel, TaintMarkRegistry та TaintMarkManager. На рис. 2.1 представлена високорівнева діаграма розробленого модуля. Розберемо зону відповідальності кожної сутності докладніше.

Об'єкт класу TaintMarkRegistry зберігає зіставлення між ім’ям мітки та її порядковим номером від 0 до 63. Бачимо, що кількість міток, які можуть бути одночасно задіяні в аналізі, обмежено числом 64. Однак, по-перше, цього вистачає для майже будь-якого розумного прикладу, а по-друге, рішення обумовлено питаннями продуктивності – операції з типом даних Long відбуваються набагато швидше, ніж якби використали бітовий масив довільного розміру.

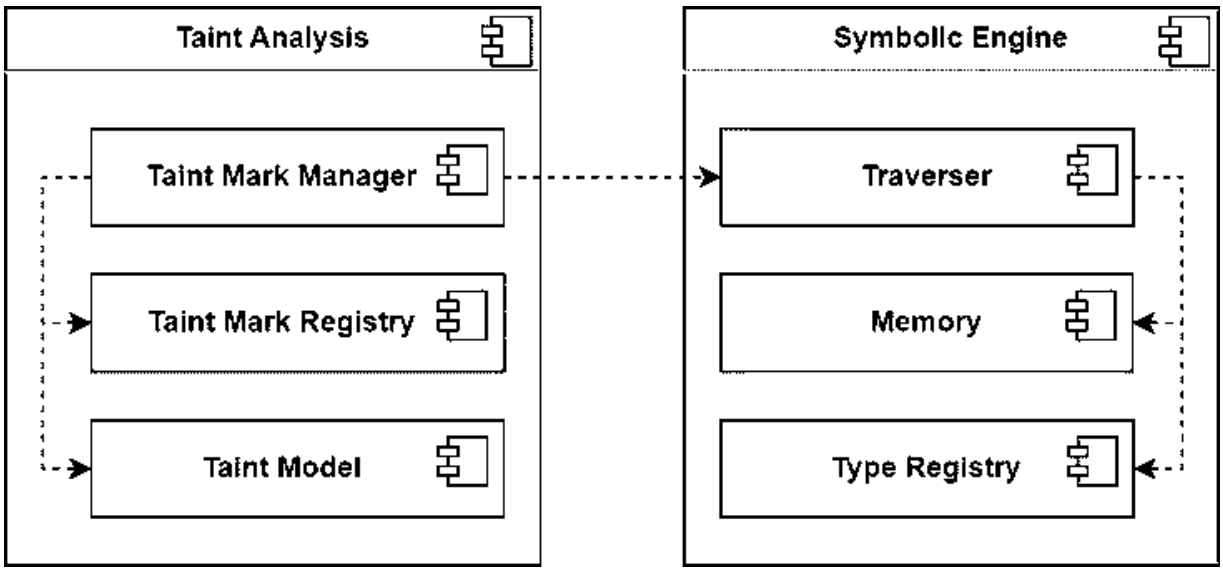

Рисунок 2.1 – Високорівнева архітектура модуля Taint-аналізу

Компонент TaintModel відповідає за надання доступу до конфігурації. Зокрема, він визначає спосіб перетворення умов (значення по ключу conditions в YAML-документі) у логічні вирази над символьними змінними. Зазначимо, що сам TaintModel не виконує жодних дій, а лише надає процедуру, яку можна запустити, вже маючи на руках Memory та TypeRegistry.

Memory займається зберіганням значень taint-vector для символьних змінних. У цьому класі були реалізовані прості функції оновлення векторів та його отримання за адресою символьного об'єкта. Вся складна логіка щодо додавання та видалення міток, що спирається на теорію taint-аналізу, була написана в окремій сутності TaintMarkManager. Тому саме цій клас «обертає» низькорівневу роботу з пам'яттю в зрозумілі з точки зору предметної області операції.

Оновлення інформації про помічені змінні відбувається в процесі роботи Traverser. Перед кожною з invoke інструкцій, які відповідають запуску деякого методу в коді користувача, викликається спеціальний обробник processTaintSink, а вже після інструкції invoke викликаються обробники processTaintSource, processTaintPass та processTaintCleaner. Такий порядок пов'язаний з тим, що всім правилам, Крім приймачів, необхідний результат роботи функції. При цьому

факт передачі заражених даних відбувається в момент запуску функції-приймача, тому повідомити про знайдену вразливість можна ще до виконання.

Перелічені обробники правил звертаються до TaintModel, щоб перевірити, чи є інформація про аналізований метод у конфігурації, і, якщо є, то отримати її. Функція procesTaintSink запитує у TaintMarkManager дані про вже виставлені мітки і додає агро-ничення в SMT-рішатель, виконання яких відповідає виявленню дефекту. Інші обробники змінюють символьну пам'ять Memory через той же TaintMarkManager, додаючи та видаляючи мітки у вибраних символьних об'єктів.

У результаті вдається не тільки виявити можливі вразливості, але і згенерувати конкретні вхідні дані, запуск програми на яких покаже, яким маршрутом заражені змінні можуть проникнути в критичні секції коду.

2.2 Розробка алгоритму символьного виконання коду програми

Далі наведено результат аналізу символьного виконання за допомогою інструменту UnitTestBot функції, що є методом класу Main. Програміст запускає генерацію тестів, використовуючи інтерфейс плагіну для IntelliJ IDEA, при цьому вказавши кількість відведеного на генерацію тестів часу 10 секунд.

Конфігурація, Java байт-код, структура класів, а також вся решта необхідної інформації перетворюється на модулі UTBot API і передається Symbolic Engine в очікуваному для нього вигляді. Той, у свою чергу, за допомогою символьного виконання знаходить 2 набори вхідних параметрів функції.

1. Набір x=27, y=−255 покриває всі рядки, крім сьомого, і повертає значення -19, як результат методу example.

2. Набір x=28, y=−1 покриває рядок номер 7, що залишився непокритим, а також ініціює викид необробленого виключення «ArithmeticException: / by zero» у рядку 12.

```
public class Main
{
    int example(int x, int y)
    {
        int a = 1;

        if (x > 27)
        {
            a += y;
        }
        else
        {
            a += 12;
        }
        return y / a;
    }
}
```

Результат виконання Symbolic Engine передається в Code Generator, який для цього прикладу генерує такий код:

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public final class MainTest
{
@Test
public void testExample1()
    {
        Main main = new Main();
        int actual = main.example(27, -255);
        assertEquals(-19, actual);
    }

 @Test
 public void testExample2()
    {
        Main main = new Main();
        main.example(28, -1);
    }
}
```

Якщо запустити отримані тести, другий завершиться з помилкою. Це є очікуваною поведінкою інструменту, адже програміст не опрацював можливий виняток. Про це повідомляється у вигляді тесту, який не проходить.

2.3 Розробка алгоритму пошуку SQL-ін'єкції

Для реакції на вразливість у вигляді SQL-ін'єкція необхідно реагувати на підозрілі команди. Якщо зловмисник введе у змінну *name* рядок "name'); DROP

TABLE Employees; --", то йому вдасться видалити таблицю *Employees* разом з усіма даними, які у ній зберігалися.

Для роботи taint-аналізу потрібно спочатку встановити правильну конфігурацію для таких параметрів:

– taint-source встановлює метод nextLine класу java.util.Scanner, який додає до значення, що повертається, мітку "user-input".

– taint-pass встановлює метод concat класу java.lang.String, який пропускає через себе мітку ", user-input", отриману або у першого аргументу, чи об'єкта, у якому цей метод викликається (тобто this).

– taint-sink встановлює метод executeUpdate класу java.sql.Statement, который відстежує змінні, помічені "user-input".

Будь-яка коректна реалізація taint-аналізу має знайти в цьому коді помилку: змінна *query* з міткою "user-input" передається до приймача executeUpdate.

В обов'язки алгоритму не входить знаходження конкретних даних, які зміг би ввести зловмисник, потрібно виявити маршрут, що з'єднує джерело-приймач.

Розроблено код плагіна на мові Java для генерації модульних тестів UnitTestBot для пошуку SQL-ін'єкцій:

```
public class SQLInjectionTest
{
    @Test
    public void testSQLInjection()
    {
        // Вхідні дані, що містять SQL запит
        String userInput = "'; DROP TABLE users; --";
        // Перевірка, чи є SQL-ін'єкція у вхідних даних
        boolean isSQLInjection = detectSQLInjection(userInput);
        // Перевірка, чи не знайдено SQL-ін'єкцію
        assertFalse("SQL Injection detected", isSQLInjection);
    }

    // Метод для виявлення SQL-ін'єкцій
    private boolean detectSQLInjection(String userInput)
    {
        Scanner sc = new Scanner(System.in);
        String name = sc.nextLine();
        Statement stmt = connection.createStatement();
        String query =
        "INSERT INTO Employees(name) VALUES('"concat(name).concat("')");
        stmt.executeUpdate(query);
        return userInput.contains(";");
    }
}
```

У цьому коді клас SQLInjectionTest містить метод testSQLInjection(), який тестує наявність SQL-ін'єкції. У методі testSQLInjection() змінна userInput містить вхідний рядок, який може містити SQL-ін'єкцію.

Далі викликається метод detectSQLInjection(), який перевіряє, чи є вхідний рядок підозрілим для SQL-ін'єкції. За допомогою методу assertFalse() перевіряється, чи була виявлена SQL-ін'єкція. Якщо так, тест буде проваленим.

2.4 Розробка звітів зі згенерованих тестів

Результатом роботи будь-якого статичного аналізатора є певний звіт. Інакше кажучи, список виявлених помилок разом із усією додатковою інформацією, яка може допомогти програмісту розібратися у проблемі швидше.

Таким чином, перше завдання, яке вирішується в даній роботі, – створення такого звіту за наявними даними, а саме за згенерованим інструментом UnitTestBot тестами. Причому найбільший інтерес викликають тестові випадки, що призводять до падіння програми, тобто викидання необроблених винятків. Отже, повідомлення про помилку має містити конкретне місце у коді, де виникає виняток, а також шлях виконання, що до нього призводить.

Як формат звіту було вибрано SARIF (Static Analysis Results Interchange Format, формат обміну результатами статичного аналізу), який затверджено як стандарт OASIS [21]. Це формат на основі JSON [22], створений спеціально для одноманітного опису результатів статичного аналізу. Формат SARIF має непогану підтримку у сфері аналізу програм. Зокрема, можна переглядати звіт у зручному вигляді, з навігацією по коду та іншими корисними функціями можна в редакторі Visual Studio Code. Також є інтеграція з CI/CD платформи GitHub, що може бути зручним для великих команд розробників.

Розглянемо структуру формату, а також основні можливості опису помилок, які він надає на прикладі звіту (для стислості всі незначні деталі замінені на крапку).

```
{
"schema" :"https://.../sarif-schema-2.1.0.json", "version" : "2.1.0", "Runs" :
  ["tool" : { ...},
    "results" :
    [ {"ruleId" :"...", "level" :"error",
      "message" : { "text" :"..."},
      "locations" : [...],
      "relatedLocations" : [...],
      "codeFlows" : [...] }
    ]
  ]
}
```

На верхньому рівні формат визначає runs – список результатів різних запусків аналізатора. Кожен запуск містить у собі tool (деяка інформація про інструмент) та results (список виявлених помилок).

Один результат описується ключами:

– ruleId (унікальний ідентифікатор для цього типу помилки);

– level (рівень серйозності дефекту, наприклад *error* або *warning)*;

– message (повідомлення, яке буде показано користувачеві);

– locations (список місць у коді, де було знайдено помилку; кожен окремий location має власний внутрішній пристрій);

– relatedLocations (список місць у коді, якимось чином пов'язаний з них з помилкою; елементами списку є самі об'єкти location);

– codeFlows (список шляхів, якими пройшла програма, перш ніж натрапити на виявлений дефект).

Об'єкт location складається з шляху до файлу artifactLocation і позиції всередині файлу *region*. Приклад об'єкта location представлений нижче.

```
"physicalLocation" : { "artifactLocation" : {"uri":"src/examples/Main.java"},
    "region" : {"startLine":91, "startColumn" :9, "endLine":91, "endColumn":20 } }
```

2.5 Конфігурація аналізатору для taint-аналізу

Одним із завдань дослідження є реалізація taint-аналізу поверх символьної віртуальної машини UnitTestBot, що позбавить техніку, що описується, від хибно-позитивних спрацьовувань. В результаті подібної модифікації єдиним джерелом вразливостей залишиться саме символьне виконання.

У процесі дослідження публічних джерел не було знайдено універсального формату конфігурації taint-аналізу, проте майже усі статичні аналізатори мають власний спосіб налаштування. Тому, по-перше необхідно придумати схему конфігурації, де можна було б описати правила (sources, passes, cleaners і sinks), з якими буде працювати ядро віртуальної машини. Формат повинен бути зручний для користувача, але в той же час надаватиме досить широкі можливості з налаштування алгоритму.

Виходячи з перерахованих вимог, в якості базової мови розмітки було взято YAML [23]. Через те, що YAML має бути легко розширюваним, він в основному інтегрований і побудований на концепціях, описаних у C, Java, Perl, Python. Основні цілі дизайну мови YAML – імітувати власні структури даних сучасних мов програмування, підтримувати статичний аналіз і мати послідовну модель для підтримки загальних інструментів. YAML можна вважати надмножиною формату JSON, що забезпечує синтаксис для покращення читання людиною. Загальна структура конфігураційного документа YAML:

```
sources:
< rule - 1 >
< rule - 2 >
< ... >
passes: [...]
cleaners: [...]
sinks: [...]
```

Тобто це просто списки правил, які стосуються конкретного типу. Кожне правило має певний набір параметрів.

1. Унікальний ідентифікатор методу, який описує це правило. Складається з повного імені методу, що включає ім'я пакета та ім'я класу, а також із сигнатури методу – набору типів його аргументів (ключ *signature* в YAML).

2. Деякі умови *conditions*, які повинні виконуватися під час виконання, щоб правило спрацювало.

3. Безліч міток *marks*, якими оперує правило.

4. Набір конкретних дій з управління мітками, які відбуваються під час спрацьовування правила (add-to, get-from, remove-from або check, залежно від семантики правила).

Таким чином, одне правило taint-source повинно виглядати в такий спосіб:

```
com.abc.ClassName.methodName:
    signature: [<int>,_,<java.lang.Object>]
    conditions:
        arg1:
            not: [-1,1]
    add-to: [this,arg2,return] marks:
    user-input
```

Це правило визначається для методу з ім'ям *methodName* з класу *ClassName*, який знаходиться у пакеті «com.abc». Метод приймає рівно 3 аргументів, перший з яких має тип *int*, другий може бути повністю будь-яким, а останній має тип *java.lang.Object*. Ключ *signature* може бути не вказаний, тоді прийнятним вважається будь-яке перевантаження *methodName*.

Спрацювання відбувається лише у тому випадку, коли перший аргумент (arg1) не дорівнює ні −1, ні 1, що й вказано за ключом *conditions* (список інтерпретується як логічне АБО). Цей параметр є необов'язковим, у разі його відсутності не буде перевірено жодних умов.

Описане джерело накладає мітку *user-input* до змінних, відповідним *this*, *arg2* і *return*. Інакше кажучи, до мітки накладаються до об'єкту класу *ClassName*, у якому викликається *methodName*, другому аргументу функції та значення, що повертається. Причому за ключами *add-to* і *marks* може лежати як список, і одиночне значення – це зроблено зручності використання.

Інші типи правил мають такий самий синтаксис, як і *source*, за винятком ключа *add-to*.

Taint-pass передає мітки з однієї множини об'єктів на іншу, тому для нього визначаються два ключі: *get-from* і *add-to* відповідно. Причому на ключ *add-to* накладаються всі вказані в *marks* мітки, якщо в *get-from* є хоча б одна з них.

Taint-cleaner видаляє мітки з багатьох об'єктів, тому його ключ має назву *remove-from*.

Taint-sink перевіряє наявність деяких міток у змінних, множина яких знаходиться за ключем *check*.

Розроблений формат, з одного боку, є лаконічним та читабельним для кінцевого користувача, а з іншого боку, досить потужним для опису складних правил чи умов спрацьовування.

2.6 Інтеграція технік taint-аналізу та символьного виконання

Ключова ідея поєднання технік taint-аналізу та символьного виконання полягає в тому, щоб для кожної символьної змінної підтримувати набір міток, накладених на неї в процесі роботи символьної машини. Ці набори можуть змінюватися тільки при виклику методу, для якого існує правило конфігурації (source, pass, cleaner або sink).

Загальний вид обробки джерела представлений в алгоритмі на рис. 2.2. Для кожної заданої конфігурації правила *rule*, яке відноситься до поточного методу, і для кожної сутності *entity* всередині нього (*this, return* або аргумент методу) алгоритм отримує відповідну йому символьну змінну *symbol*, після чого додає її набору міток marks[symbol] мітки r_{rule}.

Algorithm 1: Обробка джерела позначених даних

```
ProcessTaintSource (config, stmt)
    Data: config – набір правил taint-аналізу
    Data: stmt – поточна інструкція (виклик методу)
    rules = config.getSourcesBy(stmt);
    foreach rule∈ rules do
        foreach entity∈ rule.entities do
            symbol ← getSymbolicValue(stmt, entity);
            marks[symbol] =marks[symbol]∪rule.marks;
        end
    end
```

Рисунок 2.2 – Приклади алгоритмів обробки джерела

Алгоритм обробки правил taint-sink (див. рис. 2.3) виглядає також з тією лише різницею, що замість додавання символьна машина перевіряє наявність

заборонених міток *rule.marks* у множині *marks[symbol]*, і якщо пересічення множин не порожнє, то повідомляє про знайдену вразливість.

Алгоритми для правил taint-pass і cleaner аналогічні представленим вище, за винятком рядка зі зміною *marks* (відповідно зі своєю семантикою), тому в тексті пояснювальною записки не наводяться.

Algorithm 2: Обробка приймача позначених даних

```
ProcessTaintSink (config, stmt)
    Data:config – набір правил taint-аналізу
    Data:stmt – поточна інструкція (виклик методу)
    rules = config.getSinksBy(stmt);
    foreach rule∈ rules do
        foreach entity∈ rule.entities do
            symbol ← getSymbolicValue(stmt, entity);
            if marks[symbol] ∩ rule.marks≠∅ then
                reportProblem();
            end
        end
    end
```

Рисунок 2.3 – Приклади алгоритмів обробки правил приймача

2.7 Проектування модулю формування звітів про помилки

Далі наведено принципи, за якими модуль формування звітів отримує потрібну інформацію про помилки за результатами символьного виконання.

Після того як UnitTestBot відпрацював на коді користувача і згенерував для нього тести, модуль дивиться на набір знайдених термінальних станів, іншими словами, на тестові випадки.

Модуль формування звітів отримує необхідну інформацію про помилки за результатами символьного виконання шляхом аналізу термінальних станів, які представляють собою тестові випадки, що призводять до викидання винятків.

Кожен термінальний стан *state* містить службову інформацію, зокрема результат виконання, це тип винятку, що виникає під час виконання тестового випадку. Ця інформація дозволяє ідентифікувати конкретну помилку або виняток, який виник у тестовому випадку. Також стан містить вхідні дані

аналізованого методу – це конкретні вхідні дані, що були використані для аналізу методу, і які отримані з SMT-рішальника. Ці дані дозволяють відтворити умови, які спричинили виникнення помилки, і допомагають в розумінні причин, що призвели до виникнення виключення.

Після отримання цієї інформації модуль формує звіт, в якому будуть представлені деталі про кожну знайдену помилку, включаючи тип винятку, вхідні дані, що спричинили помилку, і, можливо, стек викликів, якщо це необхідно для додаткового аналізу.

Цей звіт може бути корисним розробникам для швидкого розуміння і виправлення виявлених помилок, оскільки він надає докладну інформацію про тестові випадки, які призводять до виникнення винятків, і сприяє ефективній відладці програмного коду.

У термінальному стані *state* міститься досить багато корисної інформації. По-перше, там зберігається результат виконання, в нашому випадку це тип винятку, що виникає. По-друге, конкретні вхідні дані аналізованого методу, отримані з SMT-рішальника. Знаючи це, модуль формує повідомлення про помилку за таким шаблоном:

"Unexpected ArithmeticException: / by zero. Test case: example(28, -1)."

У цьому випадку було знайдено поділ на нуль під час запуску методу *example* з аргументами «28» і «−1».

У *state* також міститься шлях з інструкцій *path*, за яким віртуальна машина прийшла у цей стан. Вона працює з байт-кодом, проте всередині UnitTestBot є механізм, який за будь-якою інструкцією може повернути номер відповідного рядка у вихідному коді.

Щоб відновити номер рядка з помилкою *startLine*, достатньо взяти останній елемент зі списку *path*. Далі, модуль зчитує рядок з номером *startLine* з файлу з кодом користувача та обчислює *startColumn* і *endColumn*, обрізаючи пробілові символи з початку та з кінця. Таким чином, формується поле *location* у звіті.

Поле *codeFlows* також відновлюється зі значення *path*. Модуль аналізує граф викликів методів на шляху *path* та призначає для кожного рівня вкладеності

останню на цьому рівні інструкцію. Отриманий список номерів рядків відповідає результатам трасування стека під час запуску код зі знайденими вхідними аргументами.

Останнє, що фіксується у звіті – це поле пов'язаних локацій. Він містить посилання на відповідний даному результату тест у файлі зі згенерованими тестами. Коли стартує розроблений модуль, UnitTestBot вже записав тести у вихідний файл, а отже можна просто знайти позицію потрібного тесту на його ім'я в цьому файлі.

У результаті вся зібрана інформація структурується та серіалізується у JSON у вихідний файл, що дозволяє легко зчитувати та аналізувати інформацію про тести. Кожен тест може бути легко знайдений за його ім'ям у цьому файлі, що спрощує подальшу обробку і використання цих даних:


```
{
    "tests": [
      {
        "name": "testInvalidInput",
      "description": "Test case for handling invalid input",
      "inputData": {
            "inputString": "Invalid input string"
      },
      "expectedException": "IllegalArgumentException"
      },
    {
        "name": "testNullInput",
      "description": "Test case for handling null input",
      "inputData": {
            "inputString": null
      },
      "expectedException": "NullPointerException"
    },
    {
        "name": "testValidInput",
      "description": "Test case for handling valid input",
      "inputData": {
            "inputString": "Valid input string"
      },
      "expectedException": "None"
    }
  ]
}
```

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СТАТИЧНОГО АНАЛІЗАТОРУ

3.1 Огляд програмної реалізації плагіну для середовища IntelliJ

Більшість програмістів використовують різноманітні інтегровані середовища розробки, які відрізняються можливістю підключення плагінів – сторонніх модулів для вирішення конкретних завдань. Це дозволяє кожному користувачеві налаштувати необхідний набір плагінів безпосередньо в IDE для зручності програмування, що часто є вигідніше, ніж використання окремих сервісів.

Багато статичних аналізаторів мають свої плагіни для різних середовищ розробки. У цьому випадку було вирішено створити плагін для IntelliJ IDEA, що дозволить запускати статичний аналіз і переглядати його результати у форматі звіту SARIF. Розробка базується на плагіні для автоматичної генерації модульних тестів UnitTestBot.

Платформа IntelliJ надає широкі можливості для створення нових модулів розширення. Наприклад, існує стандартний механізм Inspections, що дозволяє відображати список виявлених проблем у вигляді панелі Problems View. Для використання цієї можливості було виконано такі кроки.

1. Зареєстровано новий InspectionTool у конфігурації плагіна.

2. Перевизначено метод GlobalSimpleInspectionTool.checkFile, який додає виявлені помилки для конкретного переданого файлу psiFile до загального списку проблем для psiFile.

3. Додано автоматичний запуск механізму Inspections відразу після створення тестів і створення звіту SARIF, щоб відобразити знайдені дефекти в IDE.

Нижче представлений сценарій використання модифікованого плагіна.

1. Користувач запускає генерацію тестів у спеціальному режимі, який передбачає подальше відображення результатів статичного аналізу.

2. Після закінчення роботи символьної машини та створення звіту SARIF на екрані відкривається вкладка Problems View, де перераховані знайдені за допомогою плагіну помилки вразливості у вигляді списку повідомлень (поле message у SARIF).

3. При натисканні на будь-який з елементів цього списку, IDE показує відповідний рядок коду (поле location).

4. У вікні праворуч з'являється кнопка для переходу до потрібного згенерованого тесту (поле relatedLocation).

5. Також, є можливість подивитися трасування стека знайденого виключення (поле codeFlows). Причому ця функціональність підтримує навігацію за кодом для кожного представленого кадру стека.

Досліджено кілька альтернативних шляхів вирішення задачі візуалізації результатів, наприклад засобами розробки окремого. Встановлено, що спосіб використання вже готових Inspections і Problems View буде найпростішим у реалізації та, при цьому, звичний для кінцевого користувача, тому був вибраний саме він.

Крім описаної функціональності, плагін має додаткові можливості підвищення зручності використання. Наприклад, існує можливість запускати аналіз не тільки на одному вибраному класі, а й на кількох, а також на всьому проекті одразу.

Для описаної у п. 2.5 схеми конфігурації реалізовано парсер, а також здійснена його інтеграція з плагіном для IntelliJ IDEA, щоб програміст міг налаштовувати taint-аналіз під свій конкретний проект.

Парсер – це скрипт, який використовується для аналізу тексту з метою отримання конкретної інформації з них згідно з заданими правилами. Парсери широко використовуються у програмуванні для обробки текстових даних, які можуть бути використані для створення структурованих об'єктів або для подальшого аналізу. Враховуючи широкий спектр застосувань, існує багато типів парсерів, включаючи статичні парсери, регулярні вирази, парсери на основі

сканування, парсери для структурованих форматів даних, таких як JSON або XML, та багато інших.

Підсумковий інструмент можна застосовувати, і не складаючи нових правил, для цього було додано поширені джерела та приймачі неперевірених даних:

– java.util.Scanner.nextLine, це taint-source, що накладає мітку user-input на значення, що повертається;

– java.lang.String.concat, це taint-pass, який додає до поверненого значеня всі мітки, що отримані з *this* і *arg1*;

– java.lang.String.isEmpty, це taint-cleaner, який видаляє всі мітки з об'єкту *this* в тому випадку, якщо *return* дорівнює *true* (порожній рядок не може містити шкідливі дані);

– java.sql.Statement.execute, це taint-sink для міток *user-input*.

Плагін підтримує всі стандартні винятки Java, у тому числі AssertionError, тому, один із типових способів використання інструмента виглядає в такий спосіб. Програміст спочатку розставляє assert перевірки в коді, якщо хоче гарантувати виконання деяких інваріантів. Після цього він запускає розроблений статичний аналізатор, який виявляє всі місця, де перевірки не проходять.

Далі наведено приклад реалізації функціоналу перевірки функції *abs*, яка має повертати модуль числа. Її значення, що повертається, має бути не менше нуля, і, щоб це гарантувати, в методі присутній рядок «assert x >= 0».

```
public class Main
{
    int abs(int x)
    {
        if (x < 0)
        {
            x *= -1;
        }
        assert x >= 0;
        return x;
    }
}
```

Однак UnitTestBot виявив, що передача числа -2^{32} призводить до винятку «AssertionError» (відбувається переповнення типу int), що і відображено у звіті

SARIF. Фрагмент звіту для перегляду значення важливих полів без збереження структури формату SARIF виглядає таким чином.

1. Повідомлення про помилку:

```
"message" :
    { "text" :"Unexpected AssertionError." +
       "Test case: abs(-2147483648)" }
```

2. Розташування помилки:

```
"uri":"src/Main.java", "Регіон" : {"startLine":6, "startColumn":9}
```

3. Розташування згенерованого тесту:

```
"uri":"utbot_tests/MainTest.java", "Регіон" :
     { "startLine":29, "startColumn" :5 }
```

4. Трасування стеку:

```
[ "MainTest.testAbs3(MainTest.java:34)", "Main.abs(Main.java:6)" ]
```

3.2. Застосування плагіну для відстеження зовнішніх даних

У середовищі розробки IntelliJ IDEA будемо використовувати плагін для контролю якості коду, що допомагає відслідковувати та аналізувати використання зовнішніх бібліотек та API в проекті. Плагін здатен виявляти використання зовнішніх даних, які можуть бути потенційними джерелами вразливостей у вашому коді. Плагін має надавати рекомендації щодо того, як зменшити ризик використання таких даних і підвищити безпеку вашого проекту.

Щоб встановити плагін у IntelliJ IDEA, зазвичай потрібно перейти до меню "Settings" -> "Plugins", натиснути кнопку "Browse repositories" (або "Marketplace" у новіших версіях IDEA), знайти необхідний плагін за його назвою або описом, встановити його та перезавантажити IDE. Після цього плагін буде активовано та готовий до використання у проекті.

Виконаємо запуск плагіну для перевірки методів Main.example та Util.abs, реалізації яких з'являлися в прикладах раніше, а також на методі Main.sumAbs.

```
int sumAbs(int a, int b)
{
    return Util.abs(a) + Util.abs(b);
}
```

На рис. 3.1 показаний приклад використання плагіну у середовищі IntelliJ IDEA, на якому можна інтерфейс користувача з візуалізацією результатів статичного аналізу. При натисканні на кнопку View generated test, відбувається перехід до відповідного тесту (див. рис. 3.2).

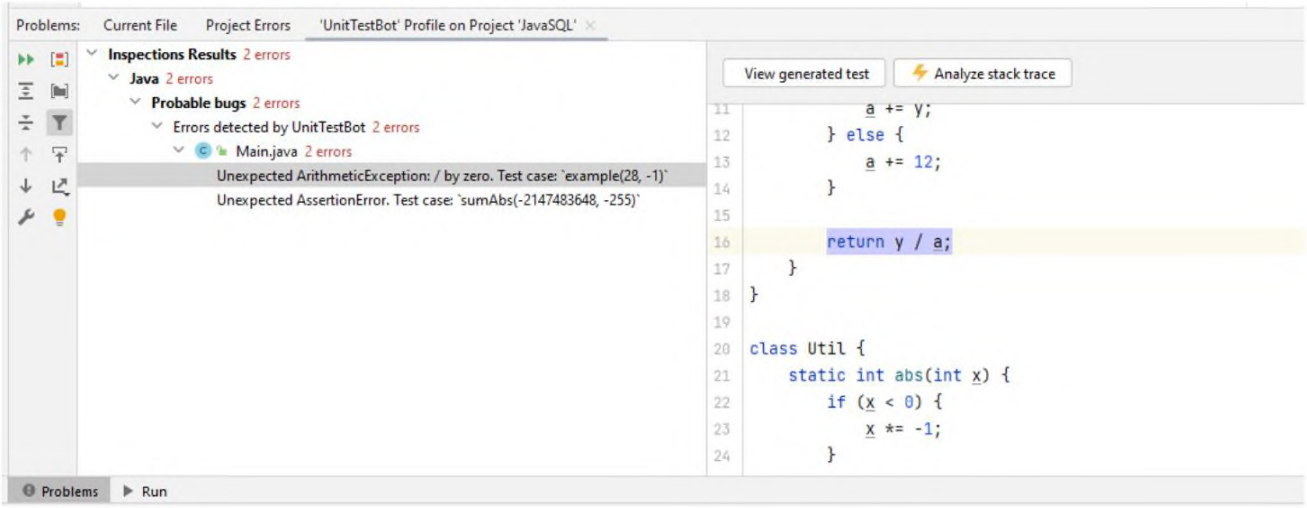

Рисунок 3.1 – Перегляд результатів роботи плагіну

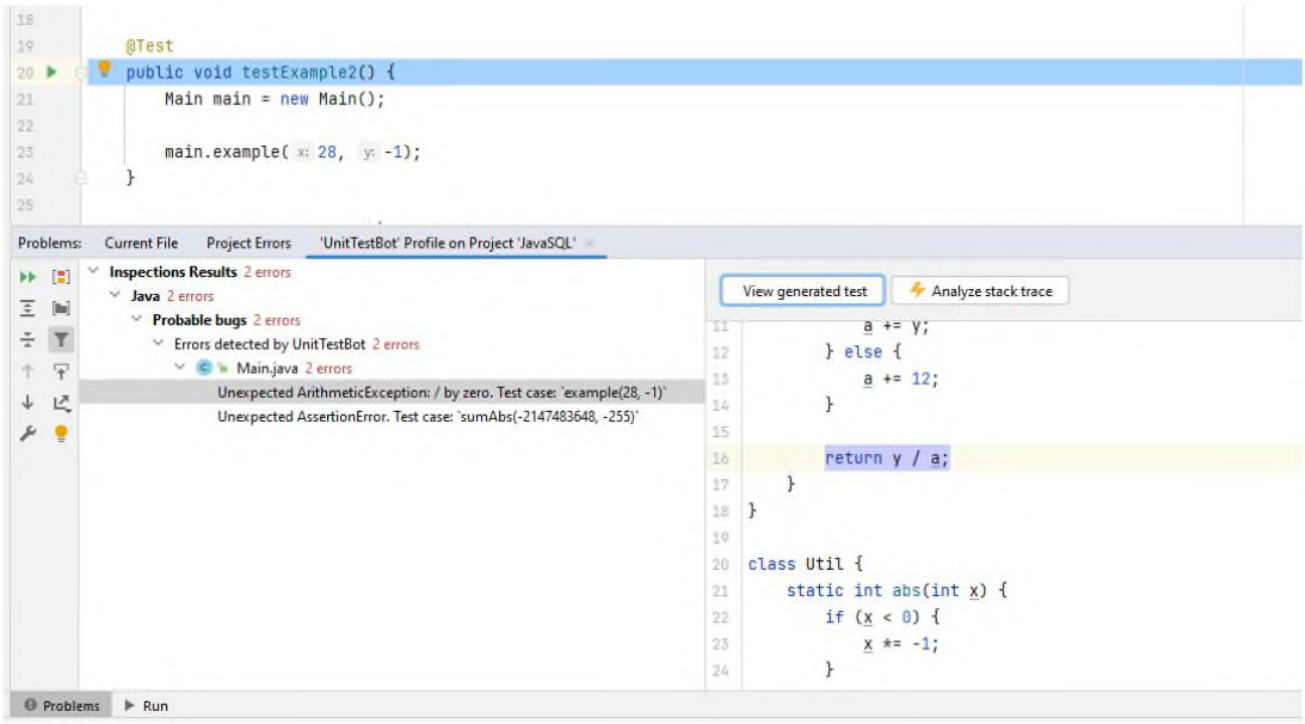

Рисунок 3.2 – Посилання на згенерований тест

При натисканні на кнопку «Analyze stack trace» відкривається вкладка для перегляду результатів трасування стеку (рис. 3.3).

Рисунок 3.3 – Перегляд результатів трасування стеку

3.3 Модифікація генератора коду IntelliJ IDEA

Результатом роботи UnitTestBot окрім звітів SARIF є тести. Іншими словами, на кожному знайденому тестовому випадку запускається CodeGenerator, який повинен оформити тест у вигляді коду мовою Java. Якщо тест веде до виникнення необробленого виключення, він не повинен проходити [28].

Помилки, які виходять в результаті роботи модуля taint-аналізу, не є справжніми з точки зору мови Java, оскільки це не є винятком. Однак хотілося б все одно виділити такі тести як провальні, тому генератор коду був змінений таким чином, щоб наприкінці кожного тесту додавалася штучна помилка, яка б забезпечувала падіння.

```
Assertions.fail(
" 'java.lang.String' marked 'user-input'
was passed into 'Statement.execute' method" );
```

Надане рішення виявилося дуже вдалим, адже воно дозволило автоматично отримати інтеграцію зі звітами SARIF та візуалізацією результатів у вкладці “Problems view” в IntelliJ IDEA. Знайдений тестовий випадок тепер сприймається як справжній виняток, а для них уже написана вся потрібна логіка.

Далі наведена реалізація перевірки чутливих даних у класі User. Метод getPassword повертає чутливі дані, які ні в жодному разі не повинні втекти з програми, а програміст направляє їх у потік виводу, що є серйозною помилкою.

```
public class User
{
    String getLogin() {/*some logic */}

    String getPassword() {/*some logic */}

    String userInfo(String login, String password)
    {
        return login + "#" + password;
    }

    Void printUserInfo()
    {
        var login = getLogin();
        var password = getPassword();
        var info = userInfo(login, password);
        System.out.println(info);
    }
}
```

Спочатку напишемо конфігурацію, яка висловлює сказану думку, і збережемо її у файл ./.idea/utbot-taint-config.yaml, звідки її зможе прочитати аналізатор.

```
sources:
    -User.getPassword:
    add-to: return
    marks: sensitive-data
passes:
    -User.getUserInfo:
    get-from: [arg1, arg2]
    add-to: return
    marks: [] # all
sinks:
    -java.io.PrintStream.println:
    check: arg1
    marks: sensitive-data
```

Далі запускаємо UnitTestBot за допомогою плагіна для IntelliJ IDEA та переглядаємо отриманий результат. По-перше, згенерувався тест, яким зрозуміло, що модуль taint-аналізу знайшов вразливість.

```
@Test
public void testPrintUserInfo1()
{
    User user = new User();
    user.printUserInfo();
    fail(
        "'java.lang.String' marked 'sensitive-data'
        was passed into 'PrintStream.println' method"" +
        ");
}
```

По-друге, з'явилася панель Problems view, де відображено знайдену проблему у вигляді результату статичного аналізу.

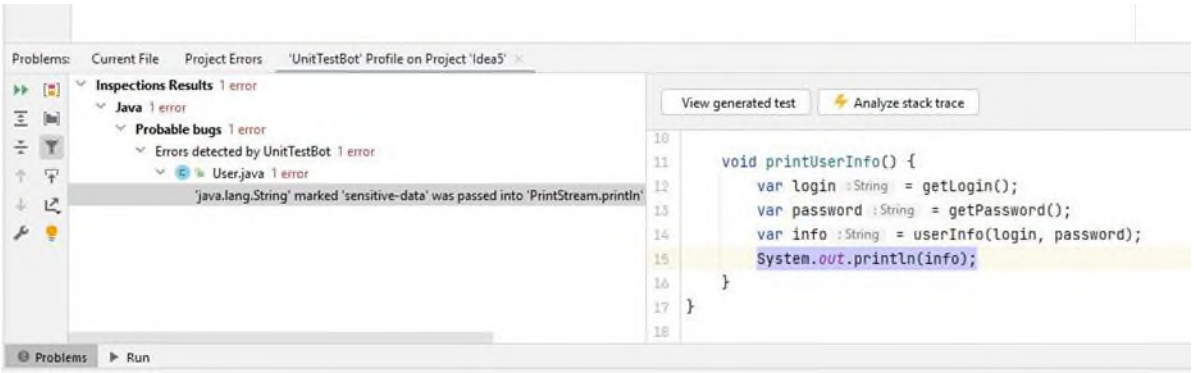

Рисунок 3.4 – Виявлена taint-аналізом проблема

Розберемо докладніше принцип роботи модуля на даному прикладі. Після виконання методу *getPassword* символ, відповідний змінній *password*, позначається як *sensitite-data* (зводиться нульовий битий у його taint-vector).

Після виклику *userInfo* позначається також змінна *info*, так як *userInfo* – це taint-pass, який додає до значення, що повертається, всі мітки, що зібрані з обох своїх аргументів.

Перед друком *info* на консоль, функція-обробник процесу TaintSink додає обмеження в SMT-рішатель, виконання якого відповідає викиданню нашого фіктивного виключення. Логічна формула для даного шляху виявляється здійсненною, тому аналізатор повідомляє про знайдену помилку, що ми в результаті і спостерігаємо.

3.4 Тестування та оцінка якості програмної реалізації

Далі наведено результати тестування розробленого аналізатора для відстеження поширення по програмі неперевірених зовнішніх даних на різних тестових завданнях [28]. Експерименти проводились з метою перевірки якості отриманого продукту. Причому під якістю мається на увазі не тільки кількість і серйозність дефектів, що виявляються ним, але і простота налаштування і запуску, а також зручність використання інструменту в цілому.

Заміри продуктивності не здійснювалися і не були метою тестування, однак для кращої відтворюваності отриманих результатів зазначимо технічні характеристики ноутбука, на якому воно проводилося:

1. OS Windows 11, версія 64x;
2. CPU Intel(R) Core(TM) i5-3570 CPU @ 3.80GHz;
3. 32 ГБ оперативної пам’яті.

Вся взаємодія з налаштуванням, запуском та результатами аналізатора відбувалася виключно засобами модифікованого плагіна для IntelliJ IDEA. Для всіх прикладів було виставлено обмеження часу на аналіз коду в 60 секунд на один клас.

У ході експериментів порівнювалася ефективність знаходження вразливостей між інструментом SpotBugs та технікою taint-аналізу, реалізованою в роботі, за такими показниками:

– загальна кількість знайдених проблем;

– частка виявлених помилок від кількості дефектів у програмі;

– частка хибно-позитивних спрацьовувань від загальної кількості виявлених проблем.

Як тести, на яких проводилися виміри, було взято проект Juliet Java [24] від Центру гарантованого програмного забезпечення NSA – це набір синтетичних програм, у яких міститься понад 100 різних типів уразливостей. Всі програми розподілені за розділами відповідно до назви допущеної в них помилки. Для тестування алгоритмів taint-аналізу було вибрано розділ із SQL-ін'єкціями, в якому знаходиться множина класів з великою кількістю навмисно спеціально зроблених помилок.

У тестових даних всі допущені помилки розмічені, тобто для кожного класу відома вразливість, що знаходиться в ньому, а також її місцезнаходження в коді. Таким чином, потрібно запустити аналізатори і порівняти отримані набори попереджень з правильним.

Результати запусків представлені у табл. 3.1.

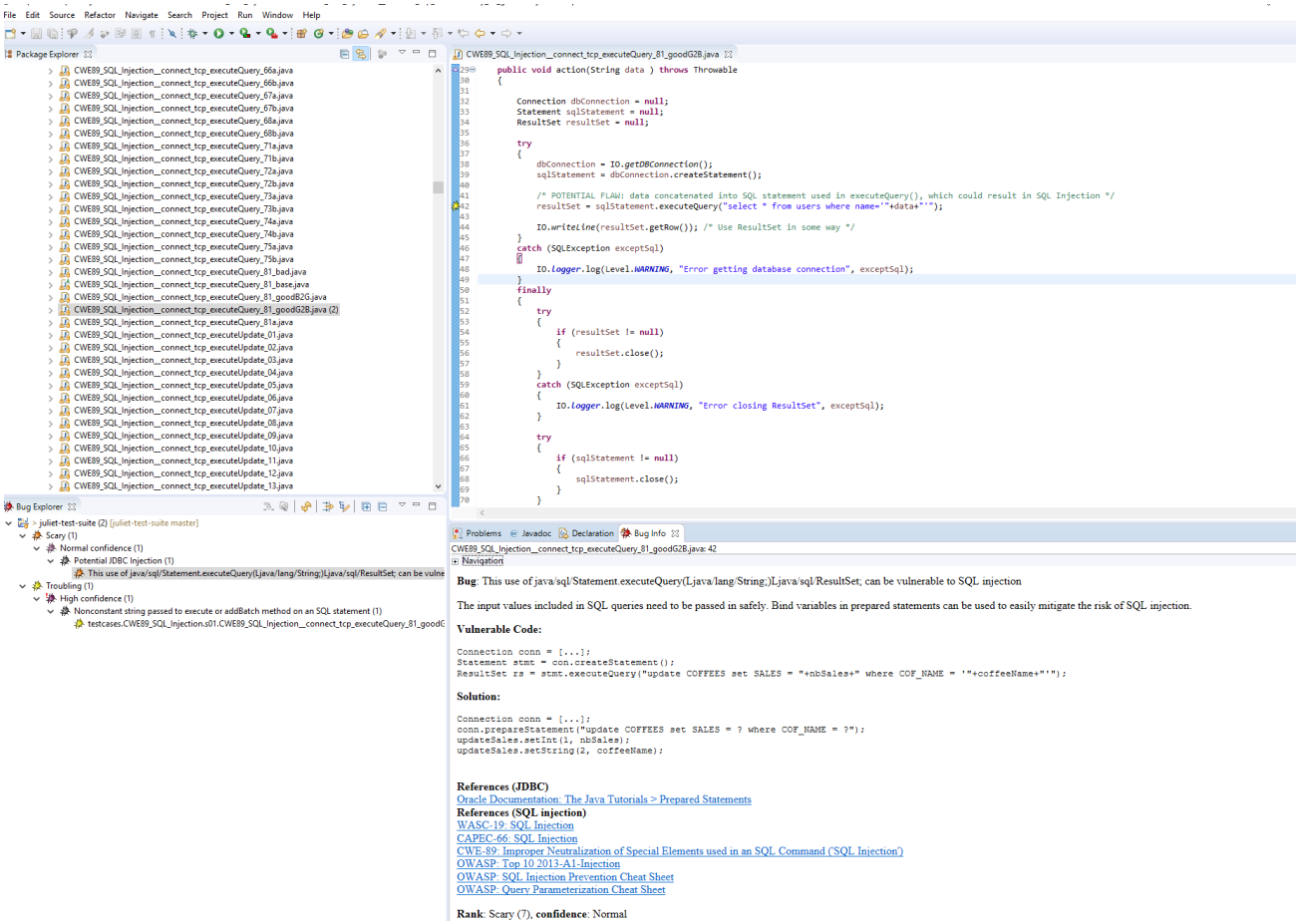

Рисунок 3.5 – Перегляд вмісту набору Juliet Java

Таблиця 3.1 – Результати відстеження неперевірених зовнішніх даних на тестових завданнях Juliet Java.

Показники	**Плагін UnitTestBot**	**Інструмент SpotBugs**
Всього проблем в наборі	**2200**	
Виявлено проблем, всього	1002	5165
З них помилок, вірно знайдених	45%	100%
З них хибних спрацьовувань	0%	57%

Модифікований плагін UnitTestBot зміг знайти 1002 SQL-ін'єкції, що становить 41% з усіх допущених у наборі помилок. При цьому не було видано жодного хибно-позитивного спрацьовування. Інші помилки не були знайдені, в основному, з тієї причини, що інструменту не вдалося згенерувати жодного тесту за наданий йому час. Зокрема, використання масивів або довгих рядків в

програмах, що аналізуються, значно уповільнює символьне виконання в UnitTestBot.

Інструмент SpotBugs виявив усі вразливості, проте разом з цим знайшов ще приблизно 3000 неіснуючих проблем, що значно більше, ніж загальна кількість реальних помилок у програмах. Таким чином, його частота помилкових спрацьовувань становила 57%.

Отримані результати показують, що розроблений модуль taint-аналізу дозволив плагіну UnitTestBot знаходити вразливості виду SQL-ін'єкція практично у половині випадків на відповідному тестовому наборі.

Таким етапом тестування для демонстрації можливості плагіну аналізувати не лише синтетичні програми, а й великі обсяги реального коду, було тестування на кількох популярних проектах на мові Java, код яких знаходиться у відкритому доступі. Перед тим, як запустити статичний аналіз на кожному з кандидатів, проекти були зібрані, тобто їх класи були скомпільовані в Java байт-код, який потрібний для роботи інструменту.

Проект Tape [25] – це набір пов’язаних із чергою класів для Android і Java, який надає можливості транзакційного файлового вводи/виводу. Додавання та видалення з екземпляра є атомарною операцією. Основний файл структуровано таким чином, щоб витримати процес і навіть системні збої.

В результаті запуску аналізатора було згенеровано звіт SARIF розміром понад п’яти тисяч рядків, де було докладно описано понад 50 виявлених проблем:

– виняток розіменувань нульового покажчика NullPointerException виникав 40 разів;

– виняток вводу/виводу IOException виникав 10 раз;

– виняток NegativeArraySizeException виникав два рази;

– виняток NoSuchElementException виникав тільки раз.

Важливо, що деякі знайдені помилки, особливо велика кількість розіменувань нульового покажчика, можуть ніколи не проявитися при реальному використанні проекту Tape. Проблема виникає, якщо передавати до публічних

методів класів «некоректні» значення аргументів, тобто не дотримуватися негласних домовленостей між користувачем та розробником бібліотеки. Основний недолік таких неявних угод полягає в тому, що користувачі, а то й сам розробник через якийсь час після написання коду можуть випадково забути їх виконати і, як наслідок, отримати неправильну поведінку програми. Статичний аналізатор відловлює такі ситуації та повідомляє про них.

Наприклад, у проекті є метод *void add(T entry)*, у якому неявно передбачено, що в метод ніхто не передає значення null, адже інакше буде викинуто виняток *NullPointerException*. Однак плагін UnitTestBot згенерував тестовий випадок, де *entry* дорівнює *null*, а також додав у звіт статичного аналізу цей тест як помилку. В даному випадку, розробнику варто було помітити *entry* анотацією *@NotNull*, що вирішило б описану проблему.

Схожий приклад виник у методі *void remove(int n)*. Плагін UnitTestBot виявив, що його запуск з аргументом *n = 131072* призводить до виникнення виключення *NoSuchElementException*. У цьому випадку поведінка аналізатора очікувана: за допомогою повідомлення про помилку він підказав розробнику задокументувати поведінку методу, щоб уникнути проблем у майбутньому.

Бібліотека Fastjson призначена для роботи з даними у форматі JSON на мові Java. Вона відома своєю високою продуктивністю та простотою використання, забезпечує високу продуктивність, швидку серіалізацію та десеріалізацію JSON даних [26]. Проект має кодову базу близько 50 тисяч рядків.

За підсумками аналізу, було виявлено 1100 проблем, серед яких 800 розіменувань нульового покажчика та 170 виходи за межі масиву. Також було знайдено 55 винятків JSONException, які виникають при роботі з обробкою JSON даних та 75 винятків JSONPathException, які виникають під час виконання операцій з шляхами (JSONPath).

Результати обробки винятків демонструють здатність плагіну працювати з винятками користувача, визначеними безпосередньо в аналізованому проекті.

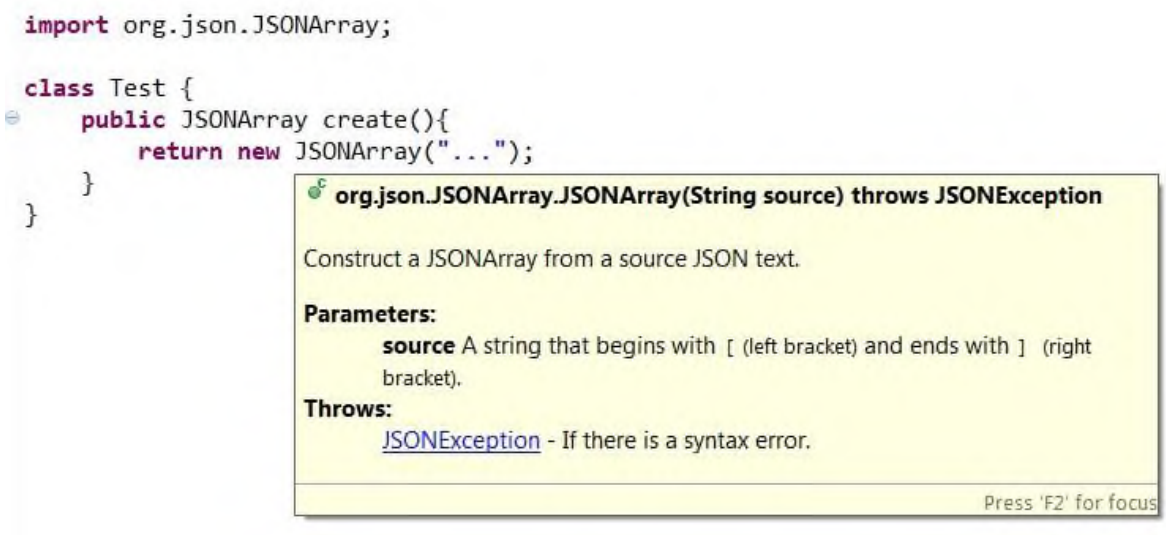

Рисунок 3.6 – Приклад реакції плагіну на виняток JSONException у бібліотеці Fastjson

На великих проектах складно оцінити серйозність помилок, адже для цього потрібно, по-перше, добре розбиратися в самому проекті, а по-друге, вручну проаналізувати кожен результат, що видається, що дуже важко. Тому, щоб оцінити, наскільки неочевидні проблеми здатний знаходити аналізатор, було проведено додаткові експерименти по розв'язання типових задач The International Collegiate Programming Contest – змагання з алгоритмічного програмування для студентів університетів, яке організатори називають «найстарішим, найбільшим і найпрестижнішим» у галузі програмування [27].

У ході роботи було розглянуто 55 завдань із програмування та алгоритмів. Для кожної з них були вивчені відправлені в систему рішення на мові Java, які не пройшли тестування через викинутий виняток (вердикт «Помилка виконання»). З усіх посилок було відібрано лише 20, на яких і планувалося запустити статичний аналіз. У вибірку включалися програми, потребують глибокого аналізу, оскільки допущена у яких помилка була очевидна. В умовах завдань зазвичай встановлюються обмеження на вхідні дані, на які може спиратися програма, що відправляється в систему. Тому, щоб ефективно аналізувати код, потрібно спочатку привести його до такого виду.

Статичний аналіз запускався лише на функції *solve*, яка вже не займається читанням вхідних даних та печаткою результату. У ній викликається спеціальний метод *assume* плагіну UnitTestBot, який дозволяє накласти певні обмеження на символьні змінні. Це потрібно для обліку обмежень з умови завдання.

```
public class Main
{
    static /* тип відповіді */ solve(/* ... */)
    {
        assume(/* обмеження у вигляді предикату */);
        // саме рішення
    }
    public static void main(String[] args)
    {
        // ввід данних
        var result = solve(/* вхідні дані */);
        // Висновок відповіді
    }
}
```

В результаті аналізу для половини з програм вдалося виявити тестовий випадок, що призводить до викидання виключення. Були виявлені такі проблеми, як поділ на нуль, розіменування нульового покажчика, переповнення стека і вихід за межі масиву або рядки, що часто виникають винятки в коді Java. Таким чином, вдалося перевірити аналізатор на здатність знаходити різні дефекти.

ВИСНОВКИ

В роботі вирішено завдання є проєктування, програмна реалізація та тестування статичного аналізатору Java-коду з використанням техніки taint-аналізу для відстеження поширення неперевірених зовнішніх даних у коді програм. У процесі досягнення мети дослідження проведено огляд існуючих на ринку рішень, у ході якого були виявлені їх ключові недоліки – поверховість аналізу, що проводиться, а також велика кількість хибно-позитивних спрацьовувань.

Запропонована інтеграція технік taint-аналіз з інструментами статичного аналізу. У символьну віртуальну машину вбудована техніка taint-аналізу для розширення можливостей аналізу текстів програм для пошуку потенційних загроз, а саме випадків використання неперевірених даних у критичних секціях програми. Виконана програмна реалізація статистичного аналізатора у вигляді плагіна для середовища IntelliJ IDEA, який має низький рівень хибних спрацьовувань та вміє знаходити проблеми в безпеці програм.

Реалізовано модуль створення звітів у форматі SARIF на основі тестів, згенерованих інструментом UnitTestBot. Розроблено зручний інтерфейс користувача для перегляду звітів SARIF у середовищі розробки IntelliJ IDEA. У символьну віртуальну машину вбудована техніка taint-аналізу, що розширює її можливості. Реалізований алгоритм дозволяє знаходити порушення безпеки, а саме випадки використання неперевірених даних у критичних секціях програми.

Статичний аналізатор протестований на відомих проектах з відкритим вихідним кодом Juliet Java, Tape та Fastjson, а також на тестових завданнях олімпіади ICPC-2023.

Апробація кваліфікаційної роботи: публікація тези доповіді «Програмна реалізація процесу відстеження поширення по програмі неперевірених зовнішніх даних» на X Всеукраїнській науково-практичній конференції «Інформаційне суспільство: проблеми та перспективи» (Одеса, 24 травня 2024 р.).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Wang P., Chao W.J., Chao K.M., Lo C.C. Using taint-analysis for threat risk of cloud applications. *In IEEE 11th International Conference*. IEEE, 2014. Pp. 185-190.

2. Baldoni R., Coppa E., D'elia D.C., Demetrescu C., Finocchi I. A survey of symbolic execution techniques. *ACM Computing Surveys. 51(3)*, 2018. Pp. 1-39.

3. Van O., Charles-Henry B. Analysis and classification of malware based on symbolic execution and machine learning. *Doctoral dissertation, LMU Munich*, 2024.

4. Прищепа О., Доценко О. Огляд статичних методів аналізу зловмисного програмного забезпечення. *Комп'ютерні науки та кібербезпека,* 2020. С. 15-24.

5. Білоус І.В., Нагорний П.В. Статичний аналіз коду C# з використанням ROSLYN API. *Інформаційні моделюючі технології, системи та комплекси (ІМТСК-2021)*. Черкаси: Черкаський національний університет ім. Б. Хмельницького, 2021.

6. Вступ до SpotBugs: інструмент для статичного аналізу коду. URL: https://cutt.ly/Hw8jwhPj (дата звернення: 01.04.2024).

7. Sapozhnikov A., Olsthoorn M., Panichella A., Kovalenko V., Derakhshanfar, P. TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion. *arXiv preprint: 2401.06580*, Cornell University, 2024.

8. UnitTestBot pepository. URL: https://github.com/UnitTestBot/UTBotJava.

9. Вступ до SpotBugs: інструмент для статичного аналізу коду. URL: https://javarush.com/ua/groups/posts/uk.3334.vstup-do-spotbugs-nstrument-dlja-statichnogo-analzu-kodu.

10. Find Security Bugs. URL: https://find-sec-bugs.github.io.

11. Coverity Scan Static Analysis. URL: https://scan.coverity.com.

12. Чикунов П.О. Відстеження поширення по програмі неперевірених зовнішніх даних / Матеріали Міжнародної науково-практичної конференції: *Європейські орієнтири розвитку України: науково-практичний вимір в умовах воєнних викликів*. Одеса, вид-во «Гельветика», 2024. С. 100-105.

13. Топ-3 інструменти аудиту смарт-контрактів. URL: https://www.h-x.technology/ua/blog-ua/top-3-smart-contract-audit-tools-ua.

14. Інструменти з відкритим вихідним кодом для аналізу коду. URL: https://hackyourmom.com/servisy/instrumenty-z-vidkrytym-vyhidnym-kodom-dlya-analizu-kodu.

15. CodeQL zero to hero part 2: getting started with CodeQL. URL: https://github.blog/2023-06-15-codeql-zero-to-hero-part-2-getting-started-with-codeql.

16. About Infer. URL: https://fbinfer.com/docs/about-Infer.

17. Bjørner N. Z3 and SMT in industrial R&D. Z3 and SMT in Industrial R&D: 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018. Pp. 675-678.

18. Базилевич Р.П., Франко А.В. Ієрархічна модель систем автоматизованого генерування модульних тестів. *Науковий вісник НЛТУ України*. 2021, т. 31, № 5. С. 96–101.

19. Сценарії, методи та засоби формальної верифікації артефактів процесу проєктування систем критичного призначення: монографія / В. В. Шкарупило, І. В. Блінов. Вінниця: ГО «Європейська наукова платформа», 2021. – 104 с.

20. UnitTestBot. URL: https://plugins.jetbrains.com/plugin/19445-unittestbot.

21. SARIF – The Static Analysis Results Interchange Format. URL: https://sarifweb.azurewebsites.net.

22. Pierre Bourhis. JSON: Data model, Query languages and Schema specification. *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, 2017.* Pp. 123-135. https://doi.org/10.1145/3034786.3056120.

23. Malin Eriksson, Victor Hallberg. Comparison between JSON and YAML for data serialization. *School of Computer Science and Engineering Royal Institute of Technology,* 2011.

24. Charest T., Rodgers N., Wu Y. Comparison of static analysis tools for Java using the Juliet test suite. *In 11th International Conference on Cyber Warfare and Security, 2016*. Pp. 431-438.

25. Tape is a collection of queue-related classes for Android and Java by Square, Inc. URL: https://square.github.io/tape.

26. Порівняння бібліотек Java для серіалізації. URL: https://dou.ua/lenta/articles/java-serialization.

27. ICPC-2023. URL: https://icpc.global.

28. Колісніченко Р.Л. Програмна реалізація процесу відстеження поширення по програмі неперевірених зовнішніх даних / Матеріали IX Всеукраїнської науково-практичній конференції «Інформаційне суспільство: проблеми та перспективи» (Одеса, 24 травня 2024 р.). Одеса, ОНЮА, 2024 – депоновано.

ДОДАТКИ

Додаток А. Приклад програмного коду з вразливостями

Лістинг програми обробки паліндромів, в якому є помилка.

```
public static int solve(String s)
{
    for (int i = 0; i < s.length(); ++i)
    {
        assume('a' <= s.charAt(i) && s.charAt(i) <= 'z');
    }

    int counter = 0;
    int count = 0;
    while (count < s.length())
    {
        counter++;
        char mem1 = s.charAt(count++);
        while (count < s.length() && s.charAt(count) == mem1)
        {
            count++;
        }
        if (count >= s.length())
        {
            break;
        }
        char mem2 = s.charAt(count++);
        while (
        count < s.length() && (s.charAt(count) == mem1 ||
        s.charAt(count) == mem2)
        )
        {
            count++;
        }
        char mem3 = s.charAt(count++);
        while (
        count < s.length() && (s.charAt(count) == mem1 ||
        s.charAt(count) == mem2 || s.charAt(count) == mem3)
        )
        {
            count++;
        }
    }

    return counter;
}
```

Додаток Б. Апробація результатів роботи

ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОЦЕСУ ВІДСТЕЖЕННЯ ПОШИРЕННЯ ПО ПРОГРАМІ НЕПЕРЕВІРЕНИХ ЗОВНІШНІХ ДАНИХ

Колісніченко Р.Л.
студент 4-го курсу факультету кібербезпеки та інформаційних технологій
Національного університету «Одеська юридична академія»

Аналіз текстів програм для пошуку потенційних загроз – це важливий етап у виявленні потенційних проблем в програмному забезпеченні. Цей аналіз зазвичай виконується з метою виявлення можливих вразливостей, вбудованих помилок, порушень безпеки та інших проблем, які можуть стати джерелом загроз для безпеки системи чи користувачів. Програми для пошуку загроз можуть аналізувати вихідний код програмного забезпечення з метою виявлення певних конструкцій або використання патернів, які можуть вказувати на потенційні проблеми безпеки.

Існують кілька методів та технік пошуку потенційних загроз в текстах програм, зокрема так званий taint-аналіз – це метод аналізу безпеки програмного забезпечення, який виявляє потенційно небезпечні дії або вразливості, враховуючи потоки даних через програму. Основна ідея полягає в тому, щоб відстежувати, як дані, які можуть бути контрольовані зловмисником, проникають у систему та як вони обробляються програмою.

Наразі відомі дослідження [1-2], у яких проблема пошуку неперевірених зовнішніх даних вирішується за допомогою застосування техніки символьного виконання коду, яка дозволяє знайти всі набори вхідних даних, що сприяють виконанню кожного з його можливих шляхів.

Символьне виконання (symbolic execution) – це техніка аналізу програмного забезпечення, яка полягає в автоматизованому створенні символьних виразів, які представляють всі можливі шляхи виконання програми. Замість фактичного виконання програми з конкретними вхідними даними, як в традиційному методі тестування, символьне виконання використовує символьні значення для вхідних даних та відстежує, як ці значення впливають на виконання програми через всі можливі шляхи виконання.

Основна ідея полягає у тому, щоб створити символьні вирази для вхідних даних програми та аналізувати вирази, щоб знайти всі можливі шляхи виконання програми, включаючи шляхи, які можуть призвести до помилок або небажаних станів програми. Символьне виконання може бути корисним для виявлення вразливостей програм, таких як витоки пам'яті, ділення на нуль, недоступний код та інші проблеми безпеки.

Актуальність дослідження обумовлена високою затребуваністю програмних систем виявлення дефектів, помилок та вразливостей у програмах на мовах високого рівня, а також необхідністю модернізації процесу відстеження потенційно небезпечних або ненадійних даних, що вводяться у програму зловмисниками і можуть бути використані для атаки.

Метою виконання дослідження є програмна реалізація статичного аналізатору Java-коду для відстеження поширення неперевірених зовнішніх даних за програмою.

Для досягнення мети було сформульовано такі завдання:

– провести огляд предметної галузі та підходів існуючих статичних аналізаторів з метою виявлення їх переваг та недоліків;

– реалізувати модуль, який на основі тестів, згенерованих інструментом автоматичної генерації тестів UnitTestBot [4], буде надаватиме користувачеві звіт із знайденими помилками;

– розробити інтерфейс користувача для перегляду звіту статичного аналізу і вбудувати його в плагін UnitTestBot для IntelliJ IDEA;

– модифікувати ядро символьної віртуальної машини, вбудувавши в нього техніку taint-аналізу, яка дозволить виявляти порушення безпеки в коді;

– протестувати розроблений статичний аналізатор для визначення ефективності знаходження помилок та вразливостей у різних програмах.

Більшість програмістів використовують різноманітні інтегровані середовища розробки, які відрізняються можливістю підключення плагінів – сторонніх модулів для вирішення конкретних завдань. Це дозволяє кожному користувачеві налаштувати необхідний набір плагінів безпосередньо в IDE для зручності програмування, що часто є вигідніше, ніж використання окремих сервісів.

Багато статичних аналізаторів мають свої плагіни для різних середовищ розробки. У цьому випадку було вирішено створити плагін для IntelliJ IDEA [3], що дозволить запускати статичний аналіз і переглядати його результати у форматі звіту SARIF. Плагін здатен виявляти використання зовнішніх даних, які можуть бути потенційними джерелами вразливостей у вашому коді. Плагін має надавати рекомендації щодо того, як зменшити ризик використання таких даних і підвищити безпеку вашого проекту. Розробка базується на плагіні для автоматичної генерації модульних тестів UnitTestBot.

Платформа IntelliJ надає широкі можливості для створення нових модулів розширення. Наприклад, існує стандартний механізм Inspections, що дозволяє відображати список виявлених проблем у вигляді панелі Problems View. Для використання цієї можливості було виконано наступні кроки.

1. Зареєстровано новий InspectionTool у конфігурації плагіна.

2. Перевизначено метод GlobalSimpleInspectionTool.checkFile, який додає виявлені помилки для конкретного переданого файлу psiFile до загального списку проблем для psiFile.

3. Додано автоматичний запуск механізму Inspections відразу після створення тестів і створення звіту SARIF, щоб відобразити знайдені дефекти в IDE.

Нижче представлений сценарій використання модифікованого плагіна.

1. Користувач запускає генерацію тестів у спеціальному режимі, який передбачає подальше відображення результатів статичного аналізу.

2. Після закінчення роботи символьної машини та створення звіту SARIF на екрані відкривається вкладка Problems View, де перераховані знайдені за

допомогою плагіну помилки вразливості у вигляді списку повідомлень (поле message у SARIF).

3. При натисканні на будь-який з елементів цього списку, IDE показує відповідний рядок коду (поле location).

4. У вікні праворуч з'являється кнопка для переходу до потрібного згенерованого тесту (поле relatedLocation).

5. Також, є можливість подивитися трасування стека знайденого виключення (поле codeFlows). Причому ця функціональність підтримує навігацію за кодом для кожного представленого кадру стека.

Виконано тестування розробленого плагіну для відстеження поширення по програмі неперевірених зовнішніх даних на різних тестових завданнях. Вся взаємодія з налаштуванням, запуском та результатами аналізатора відбувалася виключно засобами модифікованого плагіна для IntelliJ IDEA.

У ході експериментів порівнювалася ефективність знаходження вразливостей між інструментом SpotBugs та технікою taint-аналізу, реалізованою в роботі, за такими показниками:

– загальна кількість знайдених проблем;

– частка виявлених помилок від кількості дефектів у програмі;

– частка хибно-позитивних спрацьовувань від загальної кількості виявлених проблем.

У якості тестового проєкту обрано Juliet Java [5] від Центру гарантованого програмного забезпечення NSA – це набір синтетичних програм, у яких міститься понад 100 типів уразливостей. Для тестування алгоритмів taint-аналізу обрано розділ із SQL-ін'єкціями. Модифікований плагін зміг знайти 1002 SQL-ін'єкції, що становить 41% з усіх допущених у наборі помилок. При цьому не було видано жодного хибно-позитивного спрацьовування. Інші помилки не були знайдені, в основному, з тієї причини, що інструменту не вдалося згенерувати жодного тесту за наданий йому час.

Інструмент SpotBugs виявив усі вразливості, проте разом з цим знайшов ще приблизно 3000 неіснуючих проблем, що значно більше, ніж загальна кількість реальних помилок у програмах. Таким чином, частота помилкових спрацьовувань становила 57%.

Отримані результати показують, що розроблений модуль taint-аналізу дозволив плагіну UnitTestBot знаходити вразливості виду SQL-ін'єкцій практично у половині випадків на відповідному тестовому наборі.

Список використаних джерел:

1. Чикунов П.О. Відстеження поширення по програмі неперевірених зовнішніх даних / *Європейські орієнтири розвитку України: науково-практичний вимір в умовах воєнних викликів : Матеріали Міжнародної науково-практичної конференції (Одеса, 26 квітня 2024 р.)*. Одеса, вид-во «Гельветика», 2024. С. 100-105.
2. Wang P., Chao W.J., Chao K.M., Lo C.C. Using taint-analysis for threat risk of cloud applications. In IEEE 11th International Conference. IEEE, 2014. Pp. 185-190.

3. Sapozhnikov A., Olsthoorn M., Panichella A., Kovalenko V., Derakhshanfar, P. TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion. *arXiv preprint*: 2401.06580, Cornell University, 2024.
4. UnitTestBot. URL: https://plugins.jetbrains.com/plugin/19445-unittestbot.
5. Charest T., Rodgers N., Wu Y. Comparison of static analysis tools for Java using the Juliet test suite. *In 11th International Conference on Cyber Warfare and Security, 2016*. Pp. 431-438.

Ключові слова: taint-аналіз, неперевірені зовнішні данні, символьна машина, IntelliJ IDEA, генератор тестів, плагін, SQL-ін'єкція.
Keywords: taint-analysis, unverified external data, symbol engine, IntelliJ IDEA, test generator, plugin, SQL injection.

Науковий керівник: доцент Чикунов П.О.