Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет *Інформаційних радіотехнологій і технічного захисту інформації*

Кафедра *Комп'ютерної радіоінженерії та систем технічного захисту інформації*

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти *другий (магістерський)*

Забезпечення інформаційної безпеки
в програмних системах шляхом попереднього статичного аналізу коду

Виконав:
студент *2* курсу, групи СТЗІАм-22-1
Еленгаупт Віталій Володимирович

Спеціальність *125 «Кібербезпека»*
Тип програми *освітньо-професійна*
Освітня програма *«Системи технічого захисту, автоматизація її обробки»*

Керівник *проф. Антіпов І. Є.*

Допускається до захисту
зав. кафедри _____ *проф. Антіпов І. Є.*
(підпис)

2024 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційних радіотехнологій та технічного захисту інформації_

Кафедра Комп'ютерної радіоінженерії та систем технічного захисту інформації

Рівень вищої освіти_____другий (магістерьский)_____

Спеціальність_____125____Кібербезпека _____
(код і повна назва)

Тип програми _____освітньо-професійна_____

Освітня програма системи техінчого захисту інформації, автоматизація її обробки
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)
«03»___11_____2023 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові _____Еленгаупту Віталію Володимировичу____
(прізвище, ім'я, по батькові)

1. Тема роботи Забезпечення інформаційної безпеки в програмних системах шляхом попереднього статичного аналізу коду

затверджена наказом університету від "03" _____11_____2023 р. № 1281 Ст
2. Термін подання студентом роботи до екзаменаційної комісії _"12"_____01____ 2023 р.

3. Вихідні дані до роботи:

Для розроблених мовою програмування Crystal програмних продуктів, призначених для роботи під ОС Linux і які включають в себе роботу з даними, а саме: шифрування, серіалізацію та логування даних, а також обробку інформації при роботі з базами даних MySQL та PostreSQL

4. Перелік питань, що потрібно опрацювати в роботі:

- обґрунтувати актуальність забезпечення інформаційної безпеки у програмних системах
- порівняти існуючі методи забезпечення інформаційної безпеки;
- показати недоліки методів, що застосовуються;
- Оцінити можливості статичного аналізу коду для забезпечення інформаційної безпеки в програмних продуктах, що розробляються мовою програмування Crystal;
- розробити правила статичного аналізу коду для знаходження вразливостей пов'язаних із шифруванням, серіалізацією, логуванням та обробкою інформаційних при роботі з базами даних;
- Продемонструвати ефективність запропонованого методу.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)

1. Титульний слайд
2. Аналіз існуючих методів забезпечення ІБ в програмних системах
3. Загальні відомості про мову програмування Crystal
3. Завдання
4. Модулі статичного аналізатора коду
5. Побудова AST девера та його обхід
6…10. Правила аналізу
11. Застосування розробленого статичного аналізатора
12. Метод експертних оцінок для аналізу ефективності розробленого статичного аналізатора
13. Висновки

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
	Визначення цілей роботи	20.10.23 – 31.10.23	
	Аналіз методів забезпечення ІБ в програмних системах	1.11.23 – 15.11.23	
	Реалізація правил статичного аналізатора	16.11.23 – 20.11.23	
	Створення опитувальника для аналізу ефективності розробленого аналізатора	21.11.23 – 30.11.23	
	Складання пояснювальної записки	1.12.23 – 15.12.23	
	Підготовка графічного матеріалу	16.12.23 – 30.12.23	
	Підготовка до захисту	2.01.24 – 15.01.24	
	Захист	16.01.24	

Дата видачі завдання 3 11 2023 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Антіпов І.Є.
(підпис) (посада, прізвище, ініціали)

Реферат

Пояснювальна записка до кваліфікаційної роботи: 80 стор., 19 рис., 1 додаток, 18 джерел.

ІНФОРМАЦІЙНА БЕЗПЕКА, ПРОГРАМНІ СИСТЕМИ, СТАТИЧНИЙ АНАЛІЗ КОДУ, МОВА ПРОГРАМУВАННЯ CRYSTAL.

У роботі досліджено важливість інформаційної безпеки в програмних системах, проаналізовано існуючі методи забезпечення цієї безпеки, з акцентом на розробці статичного аналізатора коду для мови програмування Crystal. Робота підкреслює значення статичного аналізу коду у виявленні потенційних вразливостей на ранніх етапах розробки, підвищуючи загальну безпеку програмних продуктів.

Abstruct

Explanatory note to the master qualification work: 80 pages, 19 fig., 1 appx., 18 ref.

KEY WORDS: INFORMATION SECURITY, SOFTWARE SYSTEMS, STATIC CODE ANALYSIS, CRYSTAL PROGRAMMING LANGUAGE

This work investigates the importance of information security in software systems and analyzes existing methods of ensuring this security, with a focus on developing a static code analyzer for the Crystal programming language. The thesis emphasizes the value of static code analysis in detecting potential vulnerabilities at the early stages of development, thereby enhancing the overall security of software products.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ПС - програмна система;

ІБ - інформаційна безпека;

CVE - Common Vulnerabilities and Exposures;

SAST - Static Application Security Testing;

DAST - Dynamic Application Security Testing;

SQL - Structured Query Language;

AST - Abstract Syntax Tree.

ВСТУП

В сучасному світі інформація стала одним з найцінніших ресурсів. Щоденно ми користуємося різноманітним програмним забезпеченням, яке обробляє, зберігає та передає величезні обсяги даних, включаючи конфіденційну інформацію. Від особистих даних користувачів до стратегічної інформації компаній - все це потребує надійного захисту в умовах постійно зростаючих кіберзагроз.

На жаль, програмне забезпечення, яке ми використовуємо, не завжди є достатньо надійним. Історія знає численні приклади, коли навіть найбільш неприступні на перший погляд системи піддавалися атакам зловмисників, призводячи до великих матеріальних та репутаційних збитків для організацій та приватних осіб.

У цьому документі ми розглянемо питання інформаційної безпеки в програмних системах. Ми дослідимо існуючі методи та підходи до забезпечення захисту даних, а також розглянемо реальні приклади втручань та їх наслідків. Особлива увага буде приділена підходам до підвищення ефективності інформаційної безпеки на різних етапах розробки програмного забезпечення.

Ключовою темою нашого дослідження буде розробка статичного аналізатора коду для сучасної мови програмування. Цей інструмент має потенціал виявляти потенційні вразливості на ранніх етапах розробки, що дозволяє забезпечити вищий рівень захисту інформації та зменшити ризики її витоку або пошкодження.

Ця робота буде зосереджена на розробці статичного аналізатору коду для мови програмування Crystal для ідентифікації та запобігання потенційним вразливостям. Crystal, молода та перспективна мова програмування, пропонує сучасні засоби для створення надійного програмного забезпечення. В рамках цієї роботи ми розглядаємо, як статичний аналіз може допомогти виявити та виправити слабкі місця в коді, що можуть призвести до витоків інформації, вразливостей шифрування, а також інших потенційних проблем безпеки.

1 ІНФОРМАЦІЙНА БЕЗПЕКА ПРОГРАМНИХ СИСТЕМ - ПРОБЛЕМИ ТА РІШЕННЯ

Для того, щоб зрозуміти важливість інформаційної безпеки, необхідно вивчити існуючі виклики та доступні методи їх вирішення. У цьому розділі ми спершу розглянемо ключові поняття та терміни, які будуть використовуватися в нашому дослідженні. Основними для нас є розуміння терміну програмної системи, поняття інформаційної безпеки та розуміння існуючих методів та технік для забезпечення інформаційної безпеки.

Визначивши основні поняття ми зможемо пояснити актуальність забезпечення інформаційної безпеки програмних систем. Для цього ми розглянемо реальні сценарії, де вразливості в програмних системах призвели до втрати даних або інших негативних наслідків. Аналіз таких ситуацій допоможе нам краще зрозуміти значущість та необхідність вдосконалення методів забезпечення інформаційної безпеки.

Далі ми детально розглянемо, чому ж саме виникає інформаційна небезпека та які на даний момент є наявні інструменти для вирішення цієї проблеми.

1.1 Основні поняття та визначення

Програмна система (ПС) - це сукупність програмного коду, документації, даних та інструкцій, які разом виконують певний набір функцій або задач. Він може бути як окремим застосунком, так і складною системою, що поєднує декілька компонентів. Приклади програмних систем включають веб-додатки, мобільні програми та будь-які інші програми, що спочатку створюються розробниками і пізніше використовуються користувачами для виконання певних завдань та функцій. В даній роботі ми фокусуємося на програмних системах, які працюють із даними, а отже мають ризик витоку, ненадійного зберігання або некоректної обробки цієї інформації, що може призвести до порушень

інформаційної безпеки даних. До таких програм можна віднести системи управління базами даних, які зберігають великі обсяги інформації, включаючи персональні дані користувачів. Якщо така система має вразливості, зловмисники можуть отримати доступ до цих даних. Іншим прикладом є електронні комерційні платформи, де користувачі вводять свої платіжні дані. Ненадійна реалізація таких систем може призвести до втрати фінансової інформації. Також це медичні програмні системи, які обробляють особисту медичну інформацію пацієнтів. Ці дані є особливо цінними і їх витік може мати серйозні наслідки для приватності осіб.

Інформаційна безпека (ІБ) програмних систем визначається як захист інформації та систем, які її обробляють, від будь-яких загроз, що можуть призвести до втрати конфіденційності, цілісності та доступності інформації. Це включає в себе захист від зловмисників та вірусів, витоку інформації, а також від внутрішніх загроз, таких як помилки користувачів або неправильна конфігурація системи. Інформаційна безпека програмних систем полягає в забезпеченні захисту інформації та систем від різноманітних загроз, що можуть мати місце на різних етапах розробки. Основна мета - гарантувати конфіденційність, цілісність та доступність інформації.

Для забезпечення безпеки інформаційної системи вже існують певні методи, техніки та інструменти, що ми визначимо як методи вирішення задачі забезпечення інформаційної безпеки. Дані методи можна класифікувати за кількома категоріями, ось основні з них:

1. Технічні методи. Ці методи включають в себе використання технологій, програмного забезпечення та обладнання для захисту інформації. Наприклад, криптографія, мережеві екрани, системи виявлення вторгнень, попередній аналіз коду або робочої системи.
2. Фізичні методи. Дані методи стосуються захисту фізичних ресурсів, таких як сервери, комп'ютери та інфраструктура. Приклади: блокування серверних приміщень, відеоспостереження.

3. Організаційні методи. Ці методи включають в себе процедури, політики та стандарти, які регулюють поведінку користувачів та систем. Приклади: політики доступу, плани відновлення після аварій.
4. Адміністративні методи. Дані методи включають в себе управлінські рішення та процедури, спрямовані на підтримку інформаційної безпеки. Наприклад, навчання персоналу, регулярні аудити безпеки.

В даній роботі ми сконцентруємось на технічних методах вирішення задачі забезпечення інформаційної безпеки, оскільки вони є найбільш критичними. Однією із ключових підкатегорій таких методів є тестування робочих систем, яке дозволяє виявляти та усувати вразливості вже в реальних умовах експлуатації системи. Однак, цей метод має свої обмеження, оскільки він зосереджений на виявленні вразливостей вже існуючих систем, і не завжди може гарантувати повне виявлення всіх потенційних загроз. Тому, щоб забезпечити більш глибокий та всебічний аналіз безпеки, необхідно використовувати попередній аналіз системи. Зокрема, статичний аналіз коду дозволяє розглядати програмний код на наявність вразливостей ще до його виконання, що може значно підвищити рівень безпеки програмної системи.

Статичний аналіз коду – це процес вивчення програмного коду без його виконання. Він здійснюється з метою виявлення різних типів вразливостей, помилок або невідповідностей рекомендованим стандартам програмування. Основна ідея статичного аналізу полягає в тому, щоб знайти потенційні проблеми на ранніх етапах розробки, ще до тестування або розгортання системи.

Для здійснення статичного аналізу використовуються спеціалізовані інструменти, які автоматично перевіряють код на наявність відомих шаблонів проблем. Ці інструменти можуть виявляти різноманітні типи вразливостей, від простих синтаксичних помилок до складних проблем безпеки, таких як некоректне використання пам'яті або потенційні місця для SQL-ін'єкцій.

1.2 Актуальність та забезпечення інформаційної безпеки програмних систем

У сучасному цифровому світі програмні системи стали невід'ємною частиною нашого повсякденного життя. Вони взаємодіють із різноманітними системами та компонентами, включаючи операційні системи, бази даних, інші програми, які вже встановлені на комп'ютері, а також з ресурсами в мережі Інтернет. Ця взаємодія, хоча і приносить безліч переваг, також відкриває двері для потенційних загроз.

Однією з основних проблем є те, що програмні системи, які взаємодіють з різними компонентами, можуть мати невиявлені вразливості. Ці уразливості можуть бути результатом помилок у коді, недоліків у дизайні або ж неправильної конфігурації.

Історія показала, що навіть незначні уразливості можуть призвести до великих інцидентів з безпекою. Декілька відомих прикладів:

1. У 2014 році помилка під назвою Heartbleed в бібліотеці OpenSSL дозволила зловмисникам читати пам'ять сервера, отримуючи доступ до особистої інформації користувачів. Вона вплинула на велику кількість веб-сайтів та сервісів. Згідно [1] за попередніми даними в мережі ще залишаються вразливими близько 600,000 сервісів, які все ще використовують стару версію бібліотеки OpenSSL.
2. У 2015 році згідно [4] Британський телекомунікаційний оператор TalkTalk став жертвою атаки, в результаті якої було викрадено особисті дані більше ніж 150,000 користувачів, включаючи банківські реквізити. Атака стала можливою через помилки в програмному коді, що відповідав за взаємодію з базою даних, де і зберігалась викрадена інформація.
3. У 2016 році DDoS-атака на провайдера DNS Dyn призвела до того, що багато великих веб-сайтів стали недоступними для користувачів у США. Атака була здійснена за допомогою великої кількості заражених IoT-пристроїв. Атака не була результатом конкретної "помилки в коді" у

традиційному розумінні цього терміну. Проте, багато пристроїв, які були використані в атаці, мали слабкі паролі за замовчуванням або інші вразливості безпеки, які дозволили зловмисникам захопити на ними контроль.

4. У 2017 році шкідливе програмне забезпечення WannaCry використовувало вразливість в операційній системі Windows для шифрування файлів на комп'ютері жертви, вимагаючи викуп за їх розшифровку. Він поширився по всьому світу, заражаючи сотні тисяч комп'ютерів.

Загалом варіантів вразливостей, що призводять до проблем з безпекою даних, є дуже багато. Згідно статистики, зібраної на порталі Stanista в [7], можна побачити, що в 2022 році було виявлено понад 25000 нових загальних інформаційно-технологічних вразливостей (CVE) в усьому світі, що стало найвищою щорічною цифрою.

Рисунок 1.1 - Динаміка росту кількості вразливостей, пов’язаних з інформаційною безпекою з 2009 року по квітень 2023 року.

Прогноз на 2023 рік в [2] вказує, що буде більше 1900 нових CVE на місяць, включаючи 270 високого рівня та 155 вразливостей критичного рівня. Інше джерело в [9] також прогнозувало подібну цифру в 1900 середніх щомісячних критичних CVE на 2023 рік, що вказує на збільшення на 13% у порівнянні з 2022 роком.

1.3 Задача забезпечення інформаційної безпеки програмних систем

Забезпечення інформаційної безпеки на етапі створення програмної системи є складним завданням, що вимагає глибокого розуміння не лише основ програмування, але й потенційних загроз та вразливостей. Однією з основних проблем є те, що на етапі розробки не завжди очевидно, які конкретні рішення можуть призвести до проблем з безпекою в майбутньому.

Розробники, зосереджені на виконанні функціональних вимог до програмної системи, можуть не враховувати всі можливі сценарії використання або зловживання їхнього коду. Наприклад, розробник може реалізувати систему, яка неправильно обробляє дані, введені користувачами, що призводить до можливості SQL Injection. Така вразливість дозволяє зловмисникам виконувати небажані SQL-запити, що може призвести до несанкціонованого доступу до бази даних. Згідно з дослідженням, опублікованим на Wordfence [8], SQL Injection входить до списку найбільш поширених вразливостей в додатках на базі WordPress.

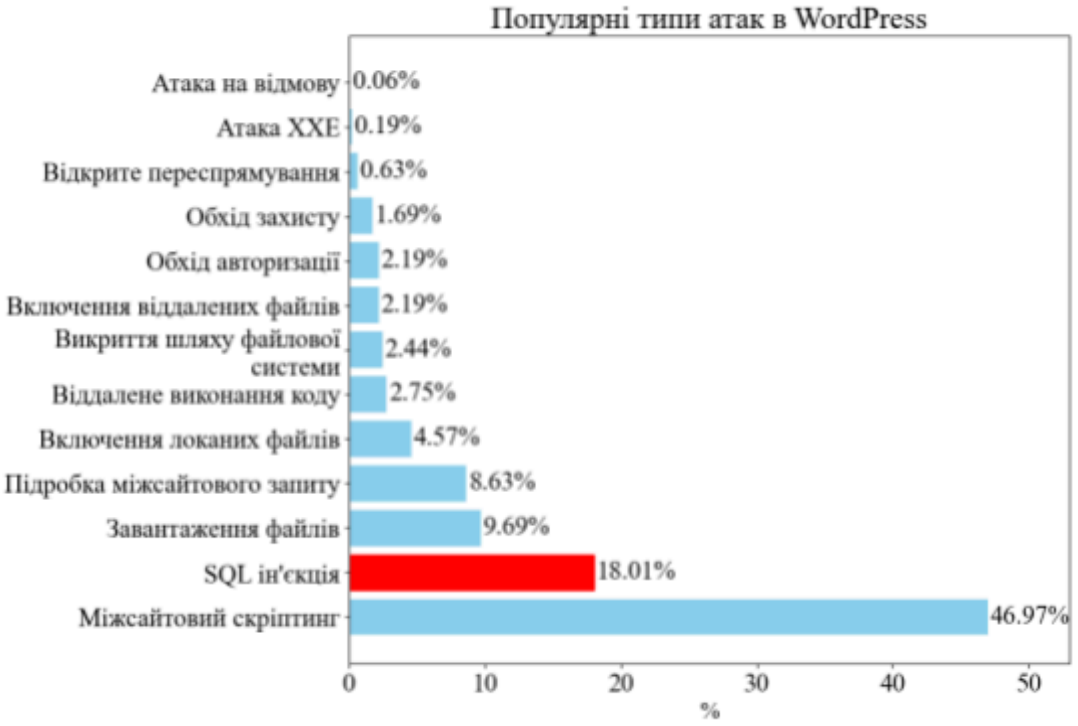

Рисунок 1.2 - Популярні типи атак в додатках на базі WordPress.

Інший поширений приклад - це використання застарілих методів шифрування. Застарілі методи можуть бути вже зламані, і їх використання може призвести до витоку конфіденційної інформації. Це небезпечно, оскільки зловмисники можуть легко розшифрувати захищені дані, використовуючи відомі їм методи.

Також, існує ризик реалізації системи, яка помилково логує конфіденційні дані користувачів. Така помилка може призвести до витоку важливої інформації, якщо логи потраплять до рук зловмисника або ж в публічний доступ.

Ще однією поширеною проблемою є залишення в коді засобів для відлагодження, таких як breakpoint. Під час розробки програми, розробники часто використовують breakpoint для зупинки виконання коду в певних місцях, щоб детально вивчити стан програми та змінних. Однак, якщо такі засоби відлагодження залишаються в коді після його випуску, це може стати серйозною вразливістю. Зловмисники можуть використовувати ці breakpoint для отримання контролю над виконанням програми, вивчення внутрішньої структури застосунку

та отримання доступу до конфіденційних даних. Тому дуже важливо проводити ретельну перевірку коду перед його випуском, щоб упевнитися, що всі засоби для відлагодження були видалені.

У такий спосіб, ключовою є не лише розробка програмного системи, але й його постійний моніторинг та аналіз з метою виявлення та усунення потенційних загроз. Щоб ефективно забезпечувати інформаційну безпеку, розробники повинні бути в курсі актуальних тенденцій у цій сфері, систематично оновлюючи свої знання та вдосконалюючи навички. Але критично важливо знати та застосовувати відповідні інструменти для аналізу та захисту коду.

1.4 Огляд та аналіз існуючих засобів та методів вирішення задачі забезпечення інформаційної безпеки

Завдання забезпечення інформаційної безпеки програмних систем не є новим. Протягом років були розроблені численні методи та інструменти, спрямовані на виявлення та усунення потенційних загроз. Далі ми розглянемо існуючі інструменти забезпечення інформаційної безпеки, представлені в [5] та [6]. Ці інструменти варіюються відповідно до етапу розробки та розгортання системи, на якому вони застосовуються. Розглянемо їй вити а також переваги та недоліки кожного з них:

1. Bug Bounty програми. Це ініціативи, які заохочують незалежних дослідників знаходити та повідомляти про уразливості в обмін на винагороду. Компанії створюють платформи, де вони публікують свої програмні системи для тестування, а дослідники шукають уразливості, намагаючись "зламати" систему. Зазвичай використовується у випадках, коли система пройшла всі попередні автоматичні та мануальні методи тестування.

   Переваги:

- залучення широкої спільноти експертів;
- виявлення реальних загроз, які можуть бути пропущені внутрішніми командами.

Недоліки:

- відсутність контролю над якістю тестування;
- потенційний ризик витоку інформації.

2. Penetration Testing - це процес імітації атаки на систему з метою виявлення вразливостей. Експерти з інформаційної безпеки використовують різні методики та інструменти, щоб спробувати "проникнути" в систему, використовуючи відомі уразливості.

   Переваги:

   - реалістична оцінка захищеності системи;
   - виявлення слабких місць, які можуть бути пропущені автоматичними інструментами.

   Недоліки:

   - висока вартість;
   - може впливати на роботу системи під час тестування.

3. DAST (Dynamic Application Security Testing) - метод тестування, який аналізує програмну систему в режимі реального часу під час його виконання, спрямований на виявлення вразливостей, які можуть бути експлуатовані під час його роботи. Він не зосереджується на внутрішньому коді програмної системи, а шукає потенційні слабкі місця в його роботі. Наприклад, у веб-застосунку є форма входу з полями для вводу логіну та паролю. DAST може автоматично спробувати різні комбінації введення, щоб перевірити, чи можливо обійти процес аутентифікації. Якщо DAST виявить, що певний введений текст може викликати непередбачувану реакцію системи, це може вказувати на наявність вразливості. Зараз існує багато відомих реалізацій даного методу на ринку. Ось кілька популярних інструментів:

- OWASP ZAP - вільний відкритий інструмент, розроблений спільнотою OWASP, який часто використовується для виявлення вразливостей в веб-застосунках;
- Burp Suite - платний інструмент, який використовується для аналізу безпеки веб-застосунків і включає в себе ряд функцій, таких як сканер, проксі-сервер, повторник та інші;
- AppScan від IBM - рішення для автоматичного тестування безпеки, яке виявляє уразливості в веб-застосунках і службах.

Переваги:

- може виявити уразливості, які пропущені статичними аналізаторами, такими як ті, що пов'язані з конфігурацією сервера, сесіями користувачів або бізнес-логікою;
- велика кількість наявних інструментів.

Недоліки:

- може пропустити вразливості, які знаходяться в коді, але не використовуються або недоступні через веб-інтерфейс;
- висока частота помилкового спрацювання.

4. SAST (Static Application Security Testing) - статичний аналіз коду, що дозволяє аналізувати код програми на наявність потенційних вразливостей ще до її запуску. Такі інструменти автоматично перевіряють код на відповідність стандартам безпеки та виявляють типові помилки, які можуть призвести до вразливостей.

   Переваги:

   - раннє виявлення вразливостей;
   - зменшення вартості виправлення помилок завдяки їх ранньому виявленню;
   - аналіз всіх функцій програмної, включаючи навіть ті, які тимчасово вимкнути або недоступні через інтерфейс роботи.

   Недоліки:

- може не виявити деяких вразливостей, які з'являються лише при взаємодії компонентів системи;
- невелика кількість наявних інструментів, які розробляються для кожної мови програмування окремо.

Таблиця 1.1 - Порівняння методів забезпечення інформаційної безпеки.

	SAST	DAST	Penetration testing	Bug Bounty
Середовище тестування	Код програми	Тестове	Pre production	Production
Переваги	Дешеве та раннє тестування	Велика кількість наявних інструментів	Виявлення вразливостей, пропущені автоматичним и методами тестування	Реальне тестування всієї системи
Недоліки	Неможливість виявити вразливості, що проявляються тільки при виконанні програми	Висока частота помилкового спрацювання	Висока вартість	Ризики витоку інформації

Отже, як ми бачимо, жоден інструмент не є універсальним рішенням, і кожен має свої обмеження. Однак, поєднуючи їх можливості, можна досягти значно вищого рівня захисту. Тому для ефективного забезпечення інформаційної безпеки необхідний комплексний підхід, який об'єднує в собі різні методики та технології.

Висновок до розділу 1

У першому розділі роботи розглянуто важливість інформаційної безпеки в контексті сучасних програмних систем. Визначено ключові поняття, обговорено актуальність забезпечення інформаційної безпеки та описано основні задачі, які стоять перед фахівцями в цій галузі. Проаналізовано існуючі засоби та методи вирішення задач забезпечення безпеки, з акцентом на їх переваги та недоліки, а також на необхідність комплексного підходу до забезпечення безпеки.

Розглядаючи сучасні виклики, ми відзначаємо постійне зростання кількості кібератак та різноманітності їх форм. Це вимагає ретельного аналізу поточного стану безпеки, оцінки потенційних ризиків та розробки ефективних стратегій реагування. Велику роль відіграє розуміння різних видів загроз, від витоку даних до складних кібератак, та розробка відповідних захисних механізмів.

У цьому контексті особливу увагу приділено використанню статичного аналізу коду як одного з ключових інструментів забезпечення інформаційної безпеки. Статичний аналіз дозволяє виявляти вразливості на ранніх етапах розробки, забезпечуючи можливість їх своєчасного усунення. Цей підхід є невід'ємною частиною розробки надійних та безпечних програмних продуктів.

Розглянувши різноманітні аспекти інформаційної безпеки, можна зробити висновок, що сучасний світ вимагає від фахівців у сфері кібербезпеки не тільки глибоких технічних знань, але й здатності прогнозувати та адаптуватися до постійно змінюваного ландшафту загроз. Такий комплексний підхід до забезпечення безпеки є ключем до створення надійних та захищених програмних рішень.

2 ЗАДАЧА ПІДВИЩЕННЯ ІНФОРМАЦІЙНОЇ БЕЗПЕКИ ПРОГРАМНИХ СИСТЕМ ПІД ЧАС РОЗРОБКИ НА МОВІ ПРОГРАМУВАННЯ CRYSTAL

В даному розділі ми розглянемо сучасну мову програмування Crystal, її ключові характеристики та особливості а також розглянемо, як на сьогоднішній день забезпечується інформаційна безпека в програмних системах, розроблених на Crystal. Далі ми розглянемо, як працює компілятор даної мови програмування та як це може бути використано для підвищення інформаційної безпеки.

2.1 Аналіз вразливостей в ПС, написаних на мові програмування Crystal

Crystal, будучи відносно новою мовою програмування, вже зарекомендувала себе завдяки своїм унікальним особливостям. Однією з важливих характеристик є її синтаксис, який схожий на Ruby, роблячи перехід для розробників знайомих з Ruby більш плавним і зручним. Однак, не дивлячись на схожість синтаксису, Crystal розвивається як незалежна мова з власними особливостями.

Crystal використовує статичну типізацію, яка забезпечує безпеку типів і водночас зберігає високий рівень гнучкості у розробці. Це досягається завдяки тому, що мова не вимагає явного вказування типів, знижуючи кількість шаблонного коду і зберігаючи читабельність.

Також значною перевагою Crystal є можливість інтеграції з кодом мови C. Це відкриває доступ до широкого спектру існуючих бібліотек і інструментів C, розширюючи можливості Crystal та підвищуючи її практичність і універсальність. Ці характеристики роблять Crystal особливо привабливою для використання у різноманітних областях програмування, від веб-розробки до системного програмування.

Crystal є відносно новою мовою програмування, яка, не дивлячись на свою молодість, вже здобула популярність завдяки своїм унікальним особливостям. На перший погляд, Crystal дуже схожий на іншу мову програмування - Ruby з точки зору синтаксису, що робить Crystal зручним для розробників, знайомих з Ruby. Однак, на відміну від Ruby, Crystal є статично типізованою мовою і компілюється в машинний код, що дозволяє досягти високої продуктивності виконання. Згідно [3] Crystal може вирішувати широкий спектр задач, від веб-розробки до системного програмування. Вже зараз існує дуже багато програмних систем, написаних на мові програмування Crystal, таких як клієнти майже всіх відомих баз даних, поштові клієнти, різні фреймворки та утиліти для розробки web-застосунків, інші мови програмування та навіть системи для машинного навчання.

Маючи такий широкий спектр використання даної мови, логічним постає питання аналізу вразливостей в програмних системах, написаних на Crystal. На жаль, кількість таких інструментів наразі обмежена через новизну мови. Традиційні методи, такі як DAST, Penetration testing та bug bounty програми, можуть бути застосовані для додатків, розроблених на Crystal, але ці методи, як ми вже обговорювали, мають свої обмеження. А особливо важливим є ще й використання статичного аналізу коду, який може виявити потенційні вразливості ще на етапі розробки програмних систем.

То як же зараз забезпечується інформаційна безпека в застосунках, розроблених на мові програмування Crystal? На превеликий жаль, інформаційна безпека в таких застосунках часто обмежена через відсутність спеціалізованих інструментів для аналізу вразливостей. Це серйозний ризик, який ставить під загрозу не лише окремі застосунки, але й майже всю екосистему Crystal. Навіть майбутнє самої мови може бути під загрозою, якщо не буде розроблено ефективних засобів для забезпечення її безпеки.

Основою для забезпечення інформаційної безпеки в будь-якій мові програмування є наявність інструментів для статичного аналізу коду. Такий

інструмент може виявляти потенційні вразливості на ранніх етапах розробки, що дозволяє вирішувати проблеми ще до того, як вони стануть реальною загрозою. Для мови Crystal такий інструмент стає не просто бажаним, але й критично важливим для гарантування інформаційної безпеки в майбутніх застосунках.

2.2 Особливості розробки статичного аналізатора коду

Будь-яка сучасна компільована мова програмування, пропонує ряд унікальних можливостей та викликів для аналізу безпеки. Компіляція коду включає в себе кілька ключових етапів, які потенційно можна використовувати для підвищення якості коду та його безпеки.

Компілятор - це програма, яка перетворює вхідний код, що написаний розробником, в семантично еквівалентний код в інших формат (наприклад, bytecode), що може виконуватись машиною. Розглянемо високорівневу архітектуру компілятора для мови програмування Crystal. Компілятор Crystal починає свою роботу з парсингу вихідного коду. На цьому етапі він розбиває код на окремі лексеми, що дозволяє визначити основні структурні елементи програми. Після цього відбувається синтаксичний аналіз, результатом якого є абстрактне синтаксичне дерево (AST), яке відображає ієрархічну структуру коду. AST дозволяє проводити глибокий аналіз коду, виявляючи зв'язки між різними частинами програми. Далі компілятор виконує семантичний аналіз, що забезпечує правильність взаємодії між різними частинами коду.

Після семантичного аналізу, компілятор Crystal виконує фазу оптимізації коду. Цей етап важливий, оскільки він покращує ефективність та продуктивність готового коду, не змінюючи його функціональність. Оптимізація може включати в себе різні техніки, такі як видалення недосяжного коду, згортання констант та оптимізація функцій. Ці оптимізації сприяють зменшенню розміру та часу

виконання програми, а також забезпечують додатковий рівень перевірки коду на наявність помилок.

Після цього компілятор Crystal переходить до фази генерації коду, де він перетворює оптимізоване AST в машинний код або в проміжне представлення, яке може виконувати цільова система. Ця фаза є критичною, оскільки саме тут вирішується, як код буде виконуватися на апаратному рівні. Коректне представлення та оптимізація на цьому етапі можуть значно вплинути на продуктивність та безпеку кінцевої програми.

Останнім, але не менш важливим етапом компіляції є зв'язування та завантаження. На цьому етапі компілятор об'єднує всі компоненти програми, включаючи бібліотеки та модулі, в один виконуваний файл. Правильне управління залежностями та ресурсами на цьому етапі є ключовим для забезпечення безпеки та стабільності програми. Це також відкриває можливості для статичного аналізу залежностей та забезпечення безпеки на рівні модулів та компонентів.

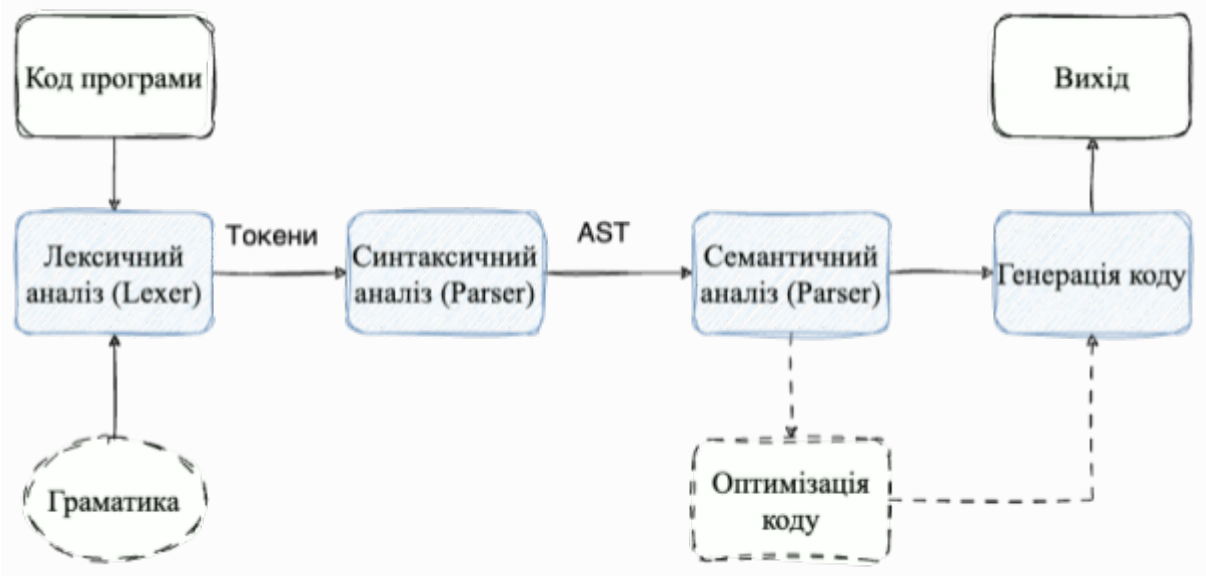

Рисунок 2.1 - Як працює компілятор мови програмування Crystal

На щастя, мова програмування Crystal надає доступ до своїх внутрішніх механізмів, що дозволяє легко побудувати AST дерево для будь-якої програми. Це

відкриває можливості для створення потужних інструментів сильно спрощує нашу задачу.

Для початку розглянемо як же працює статичний аналізатор. Цей інструмент, насправді, є досить складним у своїй структурі, але його можна розбити на чотири основних модулі, кожен з яких відповідає за певний етап обробки коду.

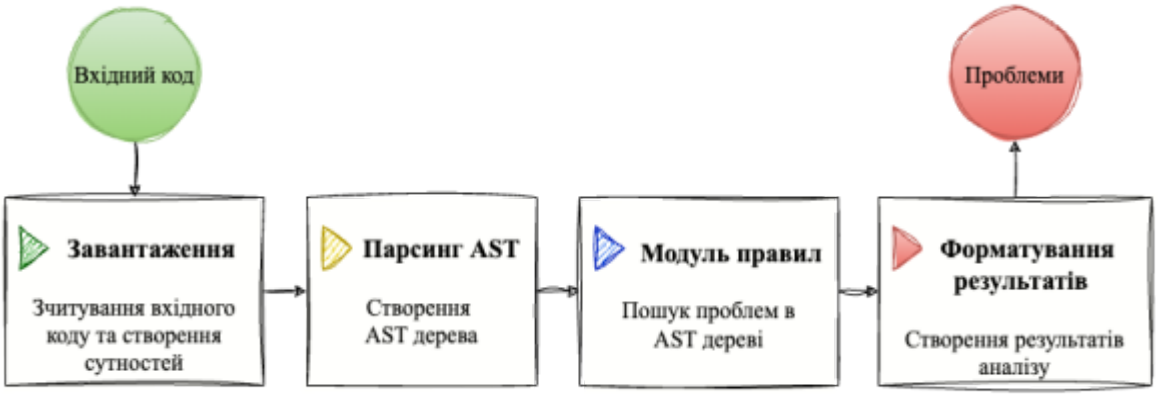

Рисунок 2.2 - Основні модулі статичного аналізатора

1. Завантаження вхідного коду. На цьому етапі аналізатор отримує доступ до вхідного коду програми, яку потрібно проаналізувати. Зазвичай це просто файл із написаним кодом програми.
2. Парсинг та створення AST. Після отримання вхідного коду, аналізатор перетворює його на абстрактне синтаксичне дерево (AST). Це дерево відображає структурну ієрархію коду, роблячи його зрозумілим для машини і дозволяючи проводити глибокий аналіз.
3. Модуль правил. За допомогою цього модуля аналізатор перевіряє AST на наявність певних патернів або конструкцій, які можуть вказувати на потенційні вразливості або проблеми в коді. Ці правила базуються на відомих вразливостях, стандартах кодування та кращих практиках безпеки.
4. Форматування результатів. Після завершення аналізу, аналізатор представляє результати у зручному для користувача форматі. Це може бути

текстовий звіт, графічний інтерфейс або навіть інтеграція з іншими системами.

Отже, наша задача зводиться до побудови статичного аналізатора, використовуючи внутрішні механізми компілятора Crystal для створення AST дерева. Ми також реалізуємо інші ключові модулі статичного аналізатора, зокрема модуль правил, який буде відповідальний за виявлення потенційних вразливостей та інших проблем в коді, базуючись на відомих патернах та стандартах безпеки. Оскільки існує багато правил, що відображають різноманітні аспекти безпеки, в даній роботі ми фокусуємось на розробці правил, що зможуть гарантувати:

1. Безпечну обробку та зберігання інформації. Аналізатор зможе ідентифікувати потенційні вразливості, пов'язані з ненадійним зберіганням або обробкою даних.
2. Контроль шифрування даних. Ми зможемо виявляти ненадійні або застарілі методи шифрування та вразливості в програмного коді.
3. Захист від витоку інформації при логуванні. Аналізатор зможе виявляти потенційні витоки інформації через системи логування.
4. Безпечну обробка даних при серіалізації та десеріалізації даних. Ми зможемо ідентифікувати вразливості, пов'язані з ненадійною серіалізацією або десеріалізацією даних.

Висновок до розділу 2

Даний розділ досліджує вразливості в програмних системах, написаних на мові Crystal, акцентуючи на важливості розробки статичного аналізатора коду. Актуальність цієї задачі особливо зростає у світлі постійного розвитку кіберзагроз та необхідності ефективного виявлення та запобігання вразливостей у

програмному коді. Через детальний аналіз процесів компіляції та особливостей мови Crystal, підкреслюється важливість глибокого розуміння внутрішньої логіки програмних систем для ефективного забезпечення безпеки. Особлива увага приділена структурі та компонентам статичного аналізатора коду, який вважається ключовим інструментом у виявленні потенційних вразливостей на ранніх стадіях розробки. Висвітлено різні методи та підходи до статичного аналізу, підкреслюючи їх значення в процесі забезпечення інформаційної безпеки.

Розгляд статичного аналізу коду як інструменту захисту від вразливостей набуває особливої актуальності в сучасному програмуванні. Своєчасне виявлення слабких місць у коді може значно знизити ризики безпеки, пов'язані з експлуатацією вразливостей зловмисниками. Аналізатор, здатний ефективно сканувати код на предмет вразливостей, є важливим елементом в стратегії кібербезпеки будь-якої організації, що розробляє або використовує програмне забезпечення.

3 МЕТОДИ ПІДВИЩЕННЯ ІНФОРМАЦІЙНОЇ БЕЗПЕКИ ШЛЯХОМ СТАТИЧНОГО АНАЛІЗУ КОДУ

Статичний аналіз коду – це метод виявлення помилок та вразливостей у програмному коді, який виконується без запуску самої програми. На прикладі мови програмування Crystal, статичний аналіз може включати перевірку типів, виявлення недосяжного коду, аналіз потоку даних та інші перевірки, що виявляють помилки або вразливості. Наприклад, аналізатор може ідентифікувати випадки, коли змінна не ініціалізована перед використанням, або коли існує потенційна SQL-ін'єкція при роботі з базою даних.

Особливість статичного аналізу в тому, що він заснований на аналізі коду без його виконання. Це означає, що аналіз може виконуватися автоматично, як частина процесу розробки, без необхідності втручання з боку розробника. Це забезпечує ефективне виявлення помилок та вразливостей на ранніх стадіях розробки, значно підвищуючи якість та безпеку кінцевого продукту.

В даному розділі ми розглянемо алгоритми та метод статичного аналізу для забезпечення інформаційної безпеки в програмних системах, написаних на мові програмування Crystal. Для цього спочатку розглянемо наявні інструменти для ефективної роботи з вхідним кодом, що і дозволить нам ефективно реалізувати правила для статичного аналізу.

3.1 Ефективний парсинг коду

Для виконання статичного аналізу необхідно вміти ефективно працювати зі вхідним програмним кодом. Більшість мов програмування, включаючи Crystal, мають інструменти для представлення програмного коду у вигляді AST дерева. AST, або ж абстрактне синтаксичне дерево - це деревоподібна структура, що відображає синтаксичну структуру коду. Кожен вузол в AST представляє окремий

елемент програми, дозволяючи детально аналізувати її складові. Вузлом дерева може бути будь-яка конструкція мови програмування: клас, функція, виклик функції, змінна, присвоєння, операція множення або додавання і тд.

Представимо простий фрагмент коду, написаний на мові програмування Crystal, за допомогою AST дерева: a = abs(b * c) + 1. Даний код виконує операцію присвоєння у змінну a результату додавання виклику функції та константи 1. Кожен вузол AST для цього виразу буде описаний наступним чином:

1. Вузол, який представляє операцію присвоєння a = abs(b * c) + 1.
2. Вузол, що представляє змінну a.
3. Вузол, що представляє операцію додавання abs(b * c) + 1.
4. Вузол виклику функції abs(b * c).
5. Вузол, що представляє константу 1.
6. Вузол, який представляє операцію множення b * c.
7. Вузол, що представляє змінну b.
8. Вузол, що представляє змінну c.

У цьому виразі, вузол присвоєння визначає змінну a та її значення, яке вираховується як результат додавання вузла функції abs(...) та константи 1. Вузол функції abs(...) отримує як аргумент результат множення b * c, що представлено окремим вузлом множення. Візуальне представлення такого AST дерева можна побачити на Рисунку 3.1.

Як бачимо, наш код може бути представлено за допомогою дерева, вузли якого представляють конструкції мови, а листки - це найдрібніші синтаксичні елементи мови (змінні, константи, рядки), які вже не можна представити кількома вузлами в дереві.

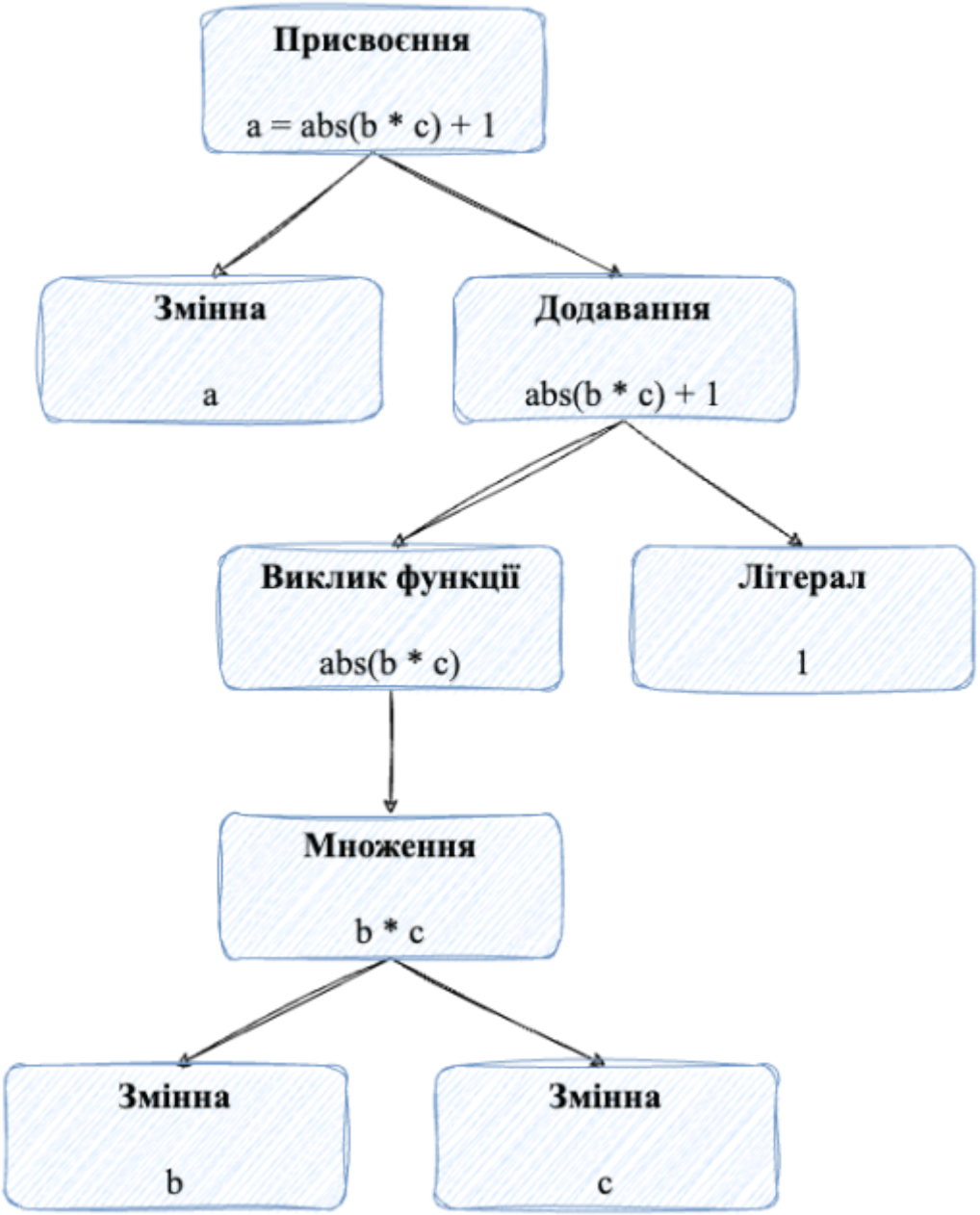

Рисунок 3.1 - Представлення коду у вигляді AST дерева

Кожен вузол AST дерева має свої властивості та містить релевантну інформацію. Наприклад, вузол з типом змінна (Var) містить в собі інформацію про ім’я змінної. Вузол з типом оператор (BinaryOp) містить інформацію про вузол зліва від оператора та вузол, що справа від оператора. А вузол виклику функції

містить інформацію про назву функції, об’єкт, на якому дана функція викликана та аргументи для виклику. Таке представлення може здатися дещо складним, бо має рекурсивну структуру. Але на практиці AST являє собою дуже потужний та гнучкий інструмент, оскільки дозволяє проаналізувати кожен вузол дерева та отримати інформацію про його наслідників.

Мова програмування Crystal має вбудований механізм для представлення AST дерева. У вихідному коді Crystal можна знайти інформацію про всі вузли [10], які представляють синтаксис даної мови програмування. Наприклад, оператор розгалуження (if) в Crystal представлений наступним чином:

```
754       class If < ASTNode
755         property cond : ASTNode
756         property then : ASTNode
757         property else : ASTNode
758         property? ternary : Bool
759
760         # The location of the `else` keyword if present.
761         property else_location : Location?
762
763         def initialize(@cond, a_then = nil, a_else = nil, @ternary = false)
764           @then = Expressions.from a_then
765           @else = Expressions.from a_else
766         end
767
768         def accept_children(visitor)
769           @cond.accept visitor
770           @then.accept visitor
771           @else.accept visitor
772         end
773
774         def clone_without_location
775           If.new(@cond.clone, @then.clone, @else.clone, @ternary)
776         end
777
778         def_equals_and_hash @cond, @then, @else
779       end
```

Рисунок 3.2 - Приклад представлення AST вузла в Crystal

На рисунку 3.2 можна побачити, що оператор розгалуження має кілька важливих властивостей:

1. cond - вузол, що представляє умову оператора розгалуження.
2. then - вузол, що представляє then гілку оператора розгалуження.
3. else - вузол, що представляє else гілку оператор розгалуження.

Також можна побачити, що даний AST вузол, як і всі інші містять інформацію про свою позицію в вхідному файлі. Дана інформація може бути корисною для пошуку відповідного коду даному AST вузлу у вхідному коді.

Отже, вхідний код можна представити за допомогою AST дерева, але для статичного аналізу нам також потрібно мати можливість ефективно робити обхід даного дерева та мати можливість швидко знаходити потрібні AST вузли.

3.2 Обхід AST дерева

Для обходу AST дерева в мові програмування Crystal використовується паттерн проектування Відвідувач (Visitor). Саме він дозволяє компілятору та іншим програмам виконувати різні алгоритми на одній і тій же структурі даних без конфліктів.

Цей патерн включає дві функції: accept та visit. Спочатку вузол у структурі AST "приймає" відвідувача, а потім передає його кожному зі своїх дочірніх вузлів рекурсивно. Це досить простий, але потужний механізм для обходу всього абстрактного синтаксичного дерева [11].

Використаємо попередній приклад AST дерева для демонстрації роботи патерну Visitor. Для цього потрібно створити AST дерево нашого прикладу коду та передати корінь дерева в accept метод нашого патерну Visitor.

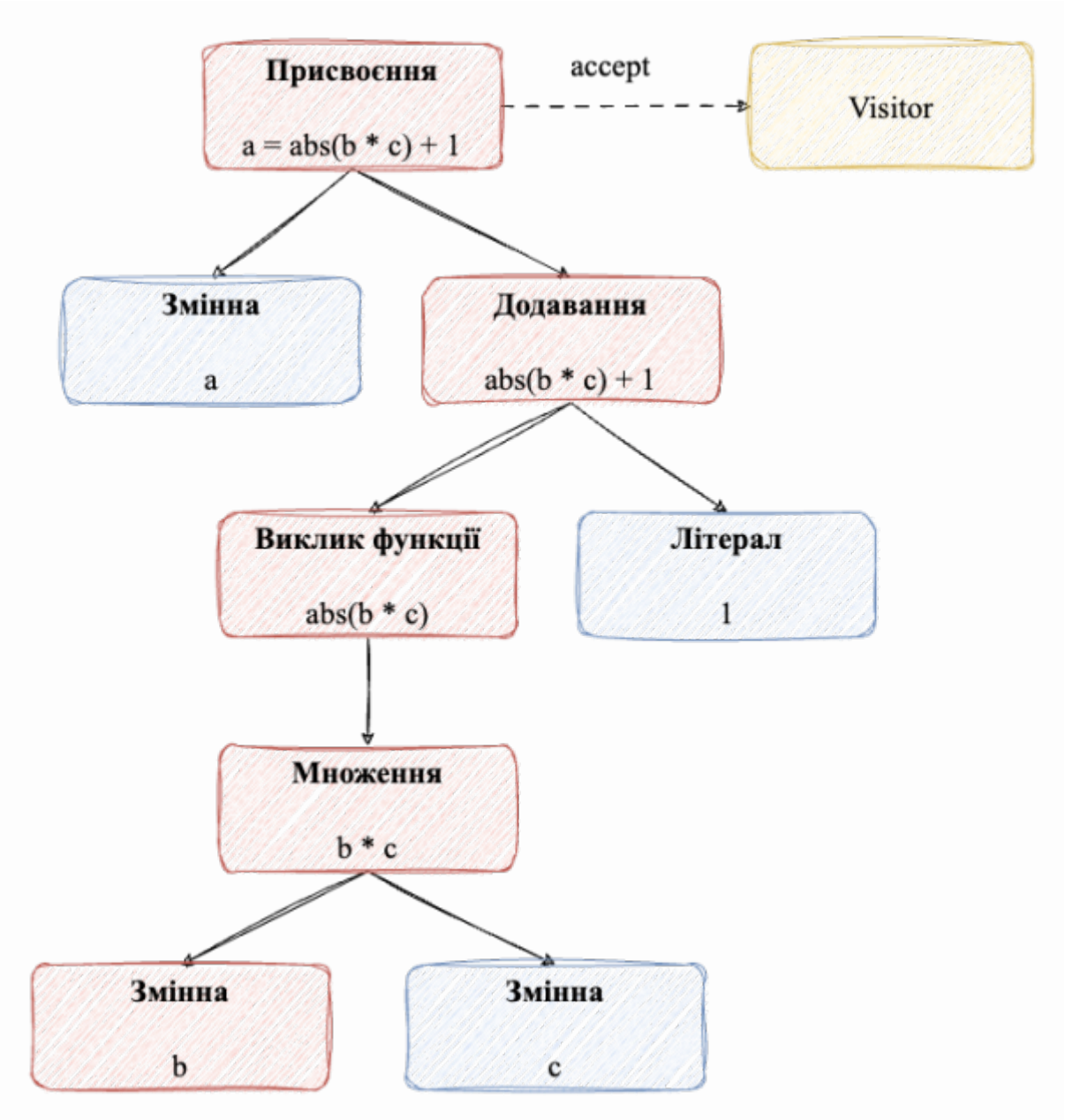

Рисунок 3.3 - Обхід AST дерева за допомогою патерну Visitor

Visitor починає свою роботу з кореня переданого дерева та рухається рекурсивно до наступних його вузлів. Якщо один із вузлів не має наслідників, то Visitor повертається до попереднього вузла і проходить всі його наслідники. Процес триває до моменту, поки Visitor не здійснить повний обхід дерева. На рисунку 3.3 показано частковий прохід дерева. Червоним відображаються вузли, які паттерн вже пройшов, а синім, які ще залишились.

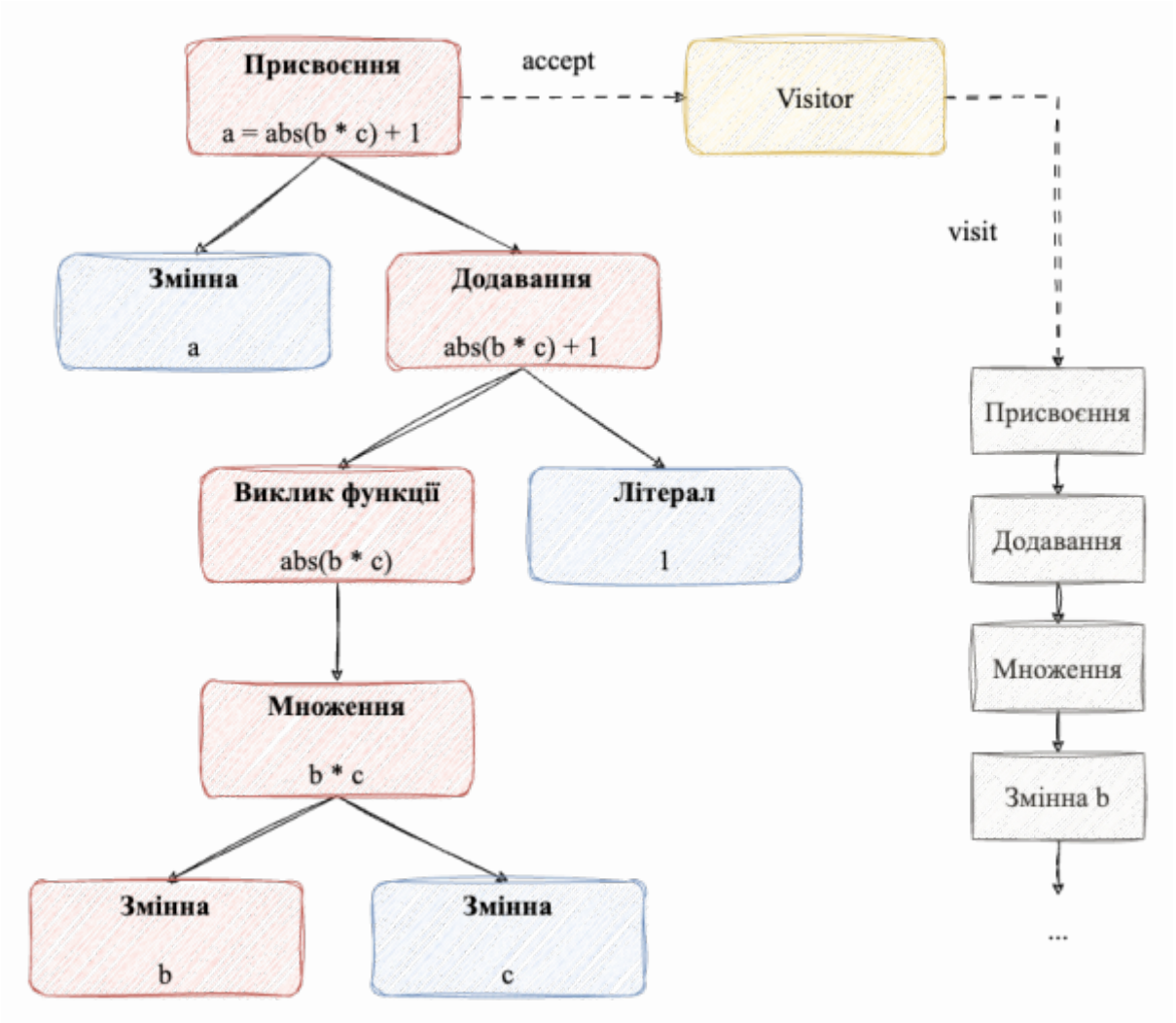

Рисунок 3.4 - Частковий прохід AST дерева за допомогою патерну Visitor

Також з даної діаграми можна побачити порядок, за яким Visitor проходить дерево та передає відвідані вузли в функцію visit. Порядок проходу може відрізнятись та залежить від реалізації конкретного патерну Visitor. Кожен клас вузлів AST може мати свою власну реалізацію методу accept, який дозволяє контролювати, як саме наслідники цього вузла будуть оброблятися. Це дозволяє реалізувати складні механізми обходу, при яких деякі вузли можуть бути проігноровані або оброблені особливим чином.

Загалом, даний підхід є досить ефективним, оскільки дозволяє зробити повний або частковий обхід дерева. Можна створити стільки завгодно сутностей Visitor та обходити дерево в той чи інший спосіб. Механізм також дозволяє ігнорувати частину дерева та сконцентруватись тільки на вузлах із певних типом.

Наприклад, можна створити Visitor, який буде обходити тільки вузли для виклику функції, тоді всі інші вузли просто будуть ігнорувати.

Один з прикладів використання патерну Visitor - це клас MainVisitor у Crystal, який відповідає за основну обробку програми. Цей клас відвідує різні вузли AST та виконує різноманітні операції, такі як визначення змінних, присвоєння типів та розширення літералів масивів. Важливою функцією MainVisitor є присвоєння типів кожному значенню у програмі. Наприклад, при відвідуванні вузла NumberLiteral, Crystal визначає його тип як Int32. Цей механізм типізації є частиною процесу виведення типів (type inference), який дозволяє Crystal автоматично визначати типи змінних, що робить код більш чистим та зрозумілим.

У компіляторі Crystal вузли AST, такі як Assign, обробляються з використанням виведення типів. Компілятор рекурсивно обробляє дочірні вузли та визначає їх типи. Це дозволяє точно визначати типи структур даних, таких як масиви, і відповідно присвоювати типи змінним.

Загалом, використання патерну Visitor у Crystal демонструє гнучкий підхід до обходу та аналізу AST, який є фундаментальним для ефективного статичного аналізу коду.

3.3 Безпечна обробка та зберігання інформації

Проблема обробки даних в програмному забезпеченні є критичною для інформаційної безпеки. Це стосується як захисту даних від несанкціонованого доступу, так і їх цілісності та конфіденційності. Коли дані зберігаються і обробляються ненадійно, вони стають вразливими до атак ззовні, наприклад, через SQL-ін'єкції, атаки на приватність або втручання в процес обробки даних.

Ненадійна обробка даних може включати в себе неправильну обробку вхідних даних, що призводить до вразливостей, таких як SQL-ін'єкції, коли

зловмисник може вставити шкідливий код у запити до бази даних. Недоліки в зберіганні даних можуть включати недостатню шифрацію або використання слабких алгоритмів шифрування, що робить збережену інформацію вразливою до витоку або компрометації.

Ефективні методи захисту від таких проблем передбачають ретельний аналіз вхідних даних, забезпечення безпечного зберігання даних та застосування сучасних методів шифрування для забезпечення конфіденційності.

Наведемо приклад використання статичного аналізу для того, щоб ідентифікувати потенційні вразливості з обробкою даних в програмних системах, написаних на Crystal, при роботі з базами даних. SQL Injection є типом атаки, де зловмисник може виконувати шкідливі SQL-запити в електронній базі даних через вхідні дані користувача в програмних системах. Ось невеликий приклад програмного коду на мові програмування Crystal, який вразливий до SQL Injection через неправильну обробку вхідних даних:

```
1  db = SQLite3::Database.new "test.db"
2
3  get "/users" do
4    user_id = params["user_id"]
5
6    # Небезпечний код: використання вхідних даних
7    # користувача без санітізації або параметризації
8    results = db.execute("SELECT * FROM users WHERE id = #{user_id}")
9
10   # Вивід результатів
11   results.to_s
12 end
```

Рисунок 3.5 - Вразливість SQL Injection в програмному коді

У цьому коді значення змінної user_id береться безпосередньо з параметрів запиту користувача та вставляється в SQL-запит без будь-якої обробки або екранування. Це означає, що користувач може ввести будь-який SQL-код, який буде виконаний базою даних, що в свою чергу може призвести до витоку інформації, оскільки при правильно підібраній умові, сформований запит вибере всіх користувачів з таблиці в електронній базі даних.

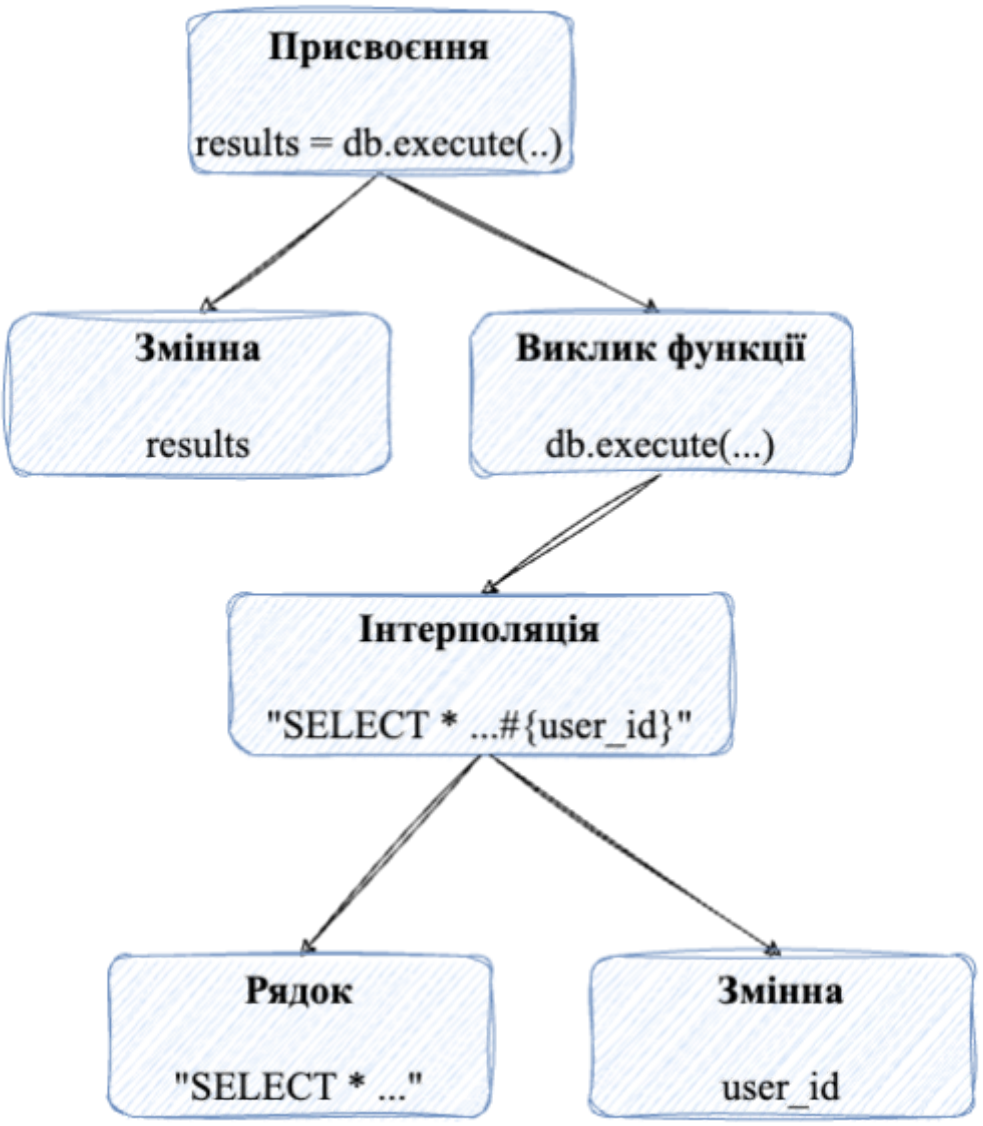

Рисунок 3.6 - AST дерево програмного коду із вразливістю SQL Injection

Для того, щоб ідентифікувати даний небезпечний код за допомогою статичного аналізу потрібно виконати кілька операцій:

1. Проаналізувати потік даних у додатку, відслідковуючи вхідні дані від користувача до місць, де вони можуть бути використані в небезпечний спосіб, наприклад, у SQL-запитах.
2. Знайти використання методів, які можуть виконувати SQL-запити з даними, отриманими від користувача. В нашому випадку це метод execute.
3. Перевірити, чи вхідні дані користувача обробляються перед використанням у SQL-запитах. Якщо дані вставляються безпосередньо, то очевидно, що ми знайшли вразливість.

Використовуючи синтаксичне дерево AST та метод його обходу, що описаний вище, ми можемо з легкістю знайти відповідні вузли дерева, що можуть призвести до вразливості SQL Injection. Наприклад, вразливий код із нашого прикладу може бути представлено за допомогою синтаксичного дерева на рисунку 3.6.

Використовуючи структуру AST ми можемо написати правило для нашого статичного аналізатора, згідно якого ми можемо знайти шукати патерни, що і будуть сигналізувати про вразливість в нашому коді. Наприклад, алгоритм виявлення SQL injection для нашого випадку, використовуючи AST та його обхід, може бути реалізований наступним чином:

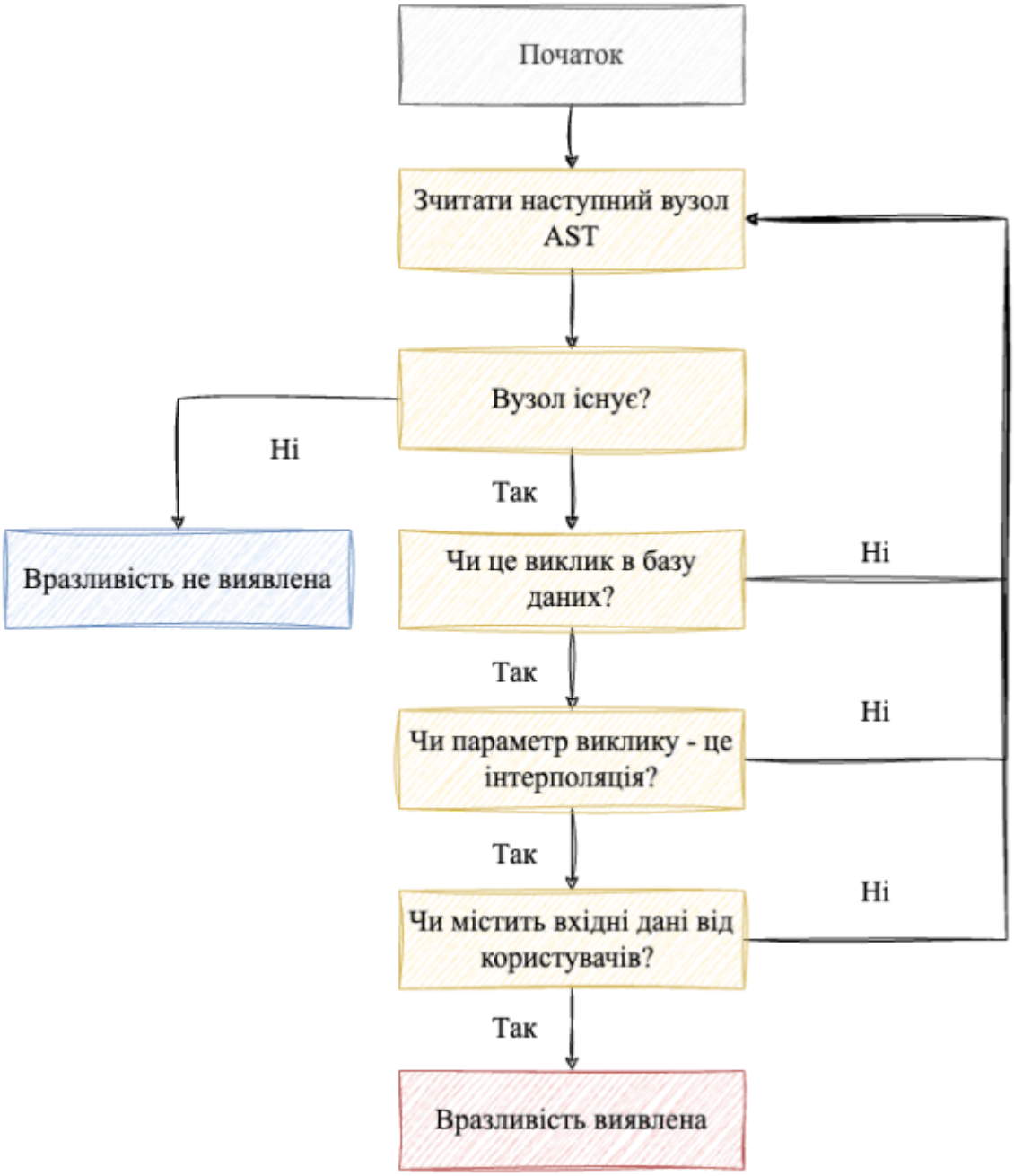

Рисунок 3.7 - Алгоритм роботи правила для пошуку SQL Injection

Як видно з рисунку 3.7, для виявлення SQL Injection вразливості, нам потрібно рухатись по синтаксичному дереві до моменту, поки ми не перевіримо кожен його вузол. Якщо ми знаходимо вузол, який представляє собою виклик в базу даних та його аргумент - це інтерполяція, що містить параметр від користовувача, то ми сигналізуємо про вразливість. Якщо ж ми перебрали всі

вузли синтаксичного дерева і не знайшли такі, що відповідають описаним правилам, то можемо сказати що даної вразливості не виявлено.

В прикладі ми розглянули один з базових випадків вразливості SQL Injection. Однак варто відзначити, що на практиці існує широкий спектр різновидів цієї вразливості, кожен з яких має свої особливості та вимагає індивідуального підходу у статичному аналізі. Для ефективного виявлення та запобігання таких вразливостей, необхідно розробляти різні специфічні правила статичного аналізу, адаптовані до конкретних сценаріїв. Наведене правило є прикладом загального підходу, який може бути модифікований та розширений для покриття більш широкого діапазону потенційних вразливостей SQL Injection.

3.4 Контроль шифрування даних

Шифрування даних є важливим аспектом забезпечення безпеки інформації. Існують різні методи шифрування, які можна класифікувати як сучасні та застарілі. Серед сучасних методів шифрування використовуються такі алгоритми, як AES (Advanced Encryption Standard) та RSA. Застарілі методи, такі як DES (Data Encryption Standard) та RC4, вважаються менш безпечними через виявлені в них вразливості.

В контексті мови програмування Crystal, для роботи з шифруванням використовується модуль Digest [12]. Цей модуль включає такі алгоритми хешування, як MD5, SHA1, SHA256 та SHA512. Хоча MD5 та SHA1 зараз вважаються застарілими через потенційні вразливості, SHA256 та SHA512 залишаються сильними та рекомендованими для використання.

Отже, наш статичний аналізатор може виконувати два завдання:

1. Виявляти застарілі алгоритми шифрування, а саме попереджати про використання MD5 та SHA1, звертаючи увагу на ризики безпеки, пов'язані з цими алгоритмами.
2. Перевіряти правильність використання сучасних алгоритмів, таких як SHA256 та SHA512, щоб переконатися в тому, що шифрування виконується належним чином.

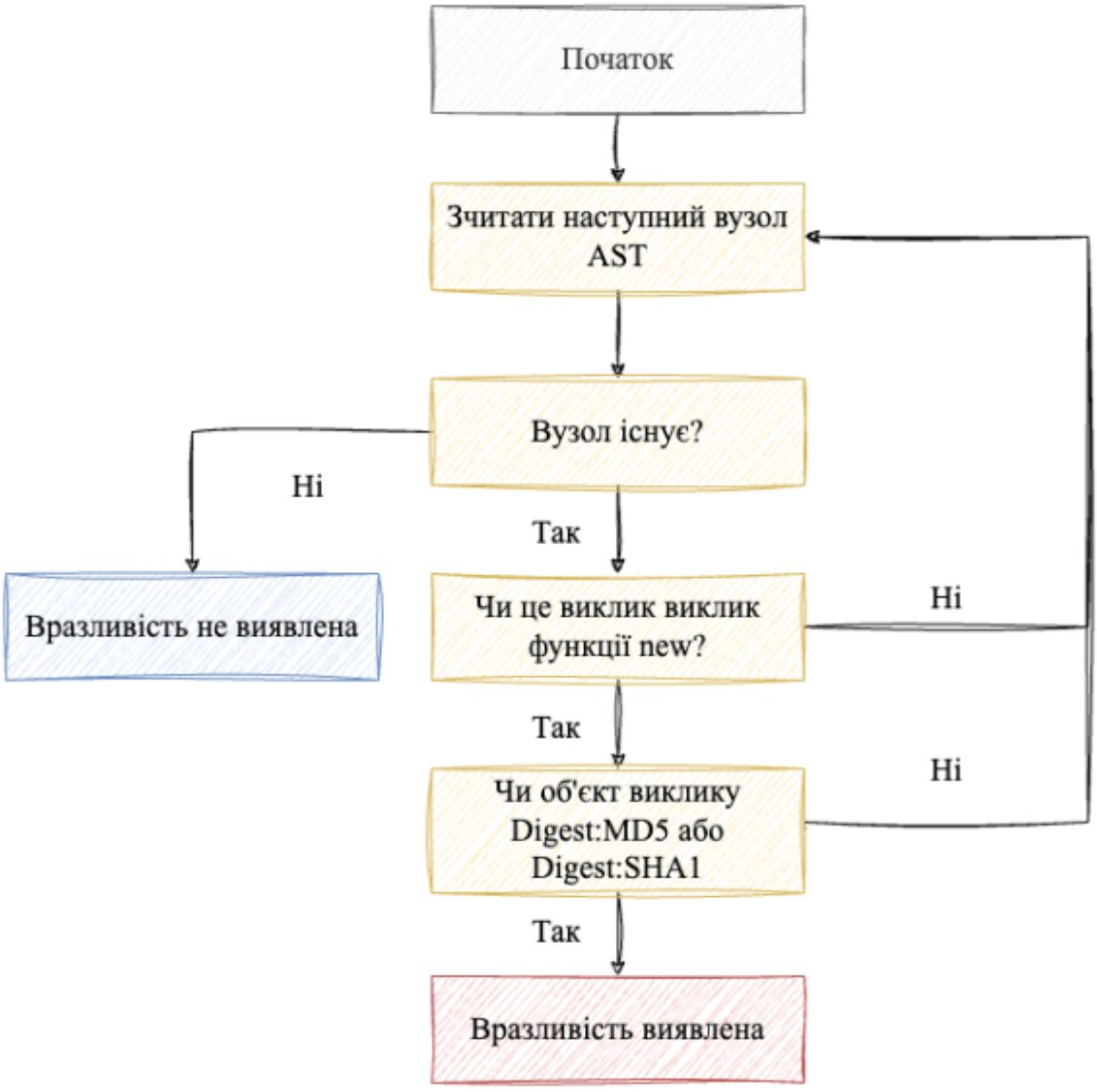

Рисунок 3.8 - Алгоритм роботи правила для виявлення застарілих методів шифрування.

Для того, щоб виявляти застарілі алгоритми шифрування, потрібно знаходити відповідні виклики створення об’єктів Digest::MD5 або Digest::SHA1. Це досить легко зробити використовуючи AST дерево та можливість його обходу. Для цього достатньо створити відповідний Visitor паттерн, який буде знаходити тільки виклики функції створення об’єктів на відповідних класах для застарілих методів шифрування.

Алгоритм виявлення вразливості можна побачити на рисунку 3.8. Даний метод виявлення є простим і в той же час досить ефективний, оскільки мінімізує кількість вузлів в AST дереві, які потрібно обійти для знаходження потрібних ознак вразливості.

Використання AST дерева для гарантування правильності використання сучасних методів шифрування є ефективним підходом, хоча й потребує складнішого алгоритму аналізу. Такий алгоритм повинен зосереджуватися на виявленні випадків, коли хеш-функції, як-то SHA256 або SHA512, використовуються без додавання солі. Відсутність солі в шифруванні може значно знизити безпеку, збільшуючи вразливість до атак на основі таблиць. Крім того, алгоритм має перевіряти правильність використання ключових методів із бібліотеки Digest, зокрема методів update та final, які є вирішальними для точного обчислення хешів.

На рисунку 3.9 представлено алгоритм для виявлення потенційних вразливостей у використанні методів шифрування. Зі схеми видно, що алгоритм зосереджений на перевірці декількох ключових аспектів шифрування за допомогою бібліотеки Digest у мові програмування Crystal. Зокрема, перевіряється, чи методи update та final використовуються в правильному порядку та чи до процесу шифрування додана сіль. Це забезпечує високий рівень стійкості шифрування. Важливим є також перевірка, що сіль є випадковою величиною, що підвищує безпеку шифрування та гарантує, що зашифровані дані не можна буде легко розшифрувати за допомогою стандартних методів.

Отже, використання сучасних методів шифрування та виявлення застарілих технік є критичними для забезпечення інформаційної безпеки в програмних системах. У контексті мови програмування Crystal, аналізатори, засновані на AST, мають величезний потенціал для виявлення вразливостей шифрування.

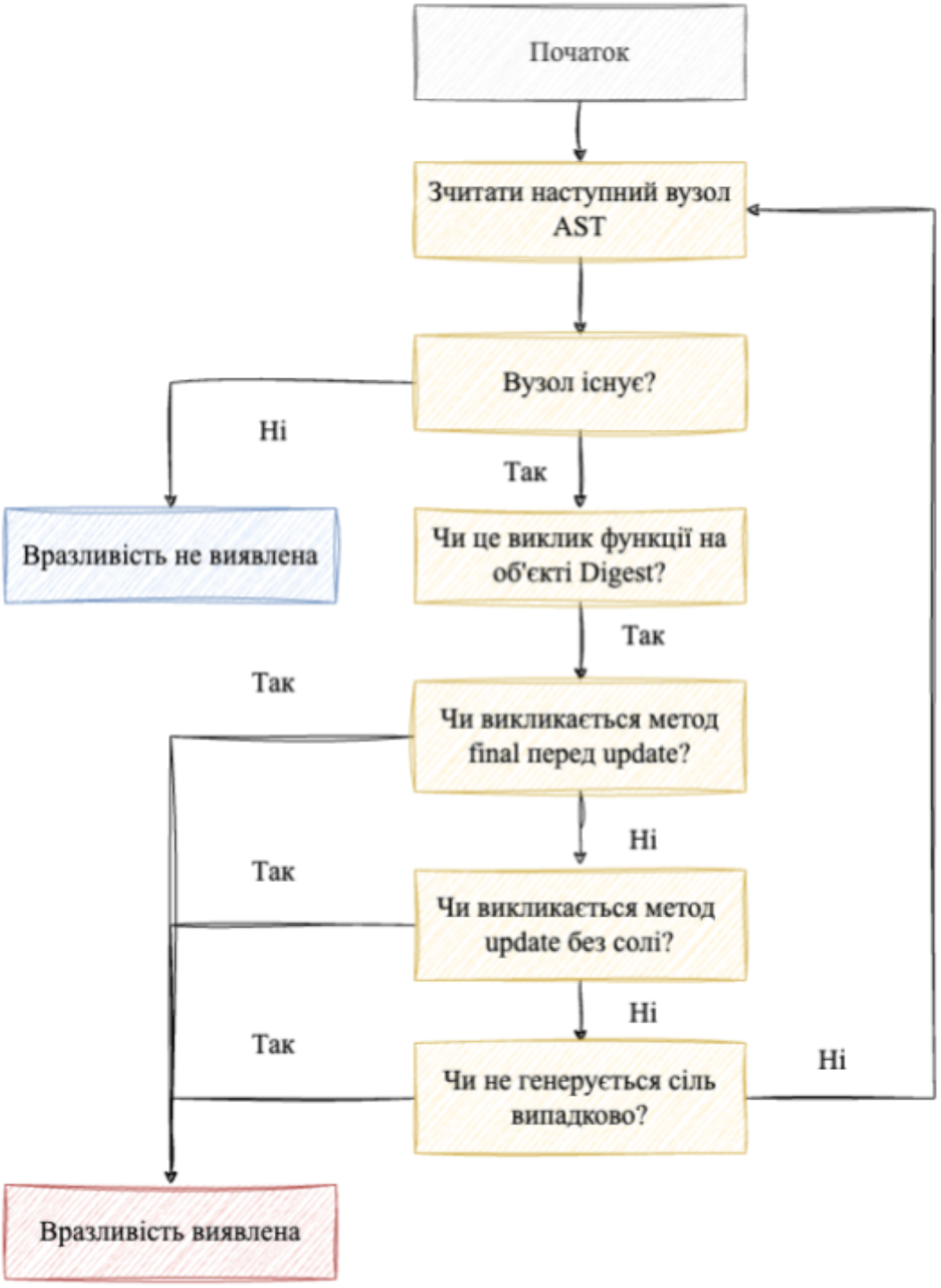

Рисунок 3.9 - Алгоритм роботи правила для вразливостей у використанні сучасних методів шифрування.

Алгоритми, представлені на рисунках 3.8 та 3.9, демонструють, як AST може бути використаний для ефективного виявлення потенційних вразливостей у шифруванні. Через можливість обходу AST та виявлення конкретних вузлів або шаблонів, стає можливим систематично виявляти та виправляти недоліки у реалізації шифрування. Такий підхід дозволяє розробникам бути впевненими у тому, що використовувані методи шифрування відповідають сучасним стандартам безпеки та ефективно захищають конфіденційні дані від потенційних загроз.

3.5 Захист від витоку інформації при логуванні

Проблема витоку інформації при логуванні виникає, коли конфіденційна інформація, така як паролі, токени доступу або особисті дані користувачів, випадково або неналежно заноситься в логи. Це може статися внаслідок неправильного використання функцій логування або через недоліки у процесі обробки даних. Наприклад:

```
1  # Приклад небезпечного логування в Crystal
2  user_email = "user@example.com"
3  user_password = "my_secure_password"
4
5  # Небезпечне логування, яке може призвести до витоку інформації
6  logger.debug "Logging in user: #{user_email} with password: #{user_password}"
```

Рисунок 3.10 - Приклад коду з небезпечним логуванням.

У цьому прикладі, інформація про електронну пошту та пароль користувача відкрито логується, що створює ризик витоку цієї конфіденційної інформації.

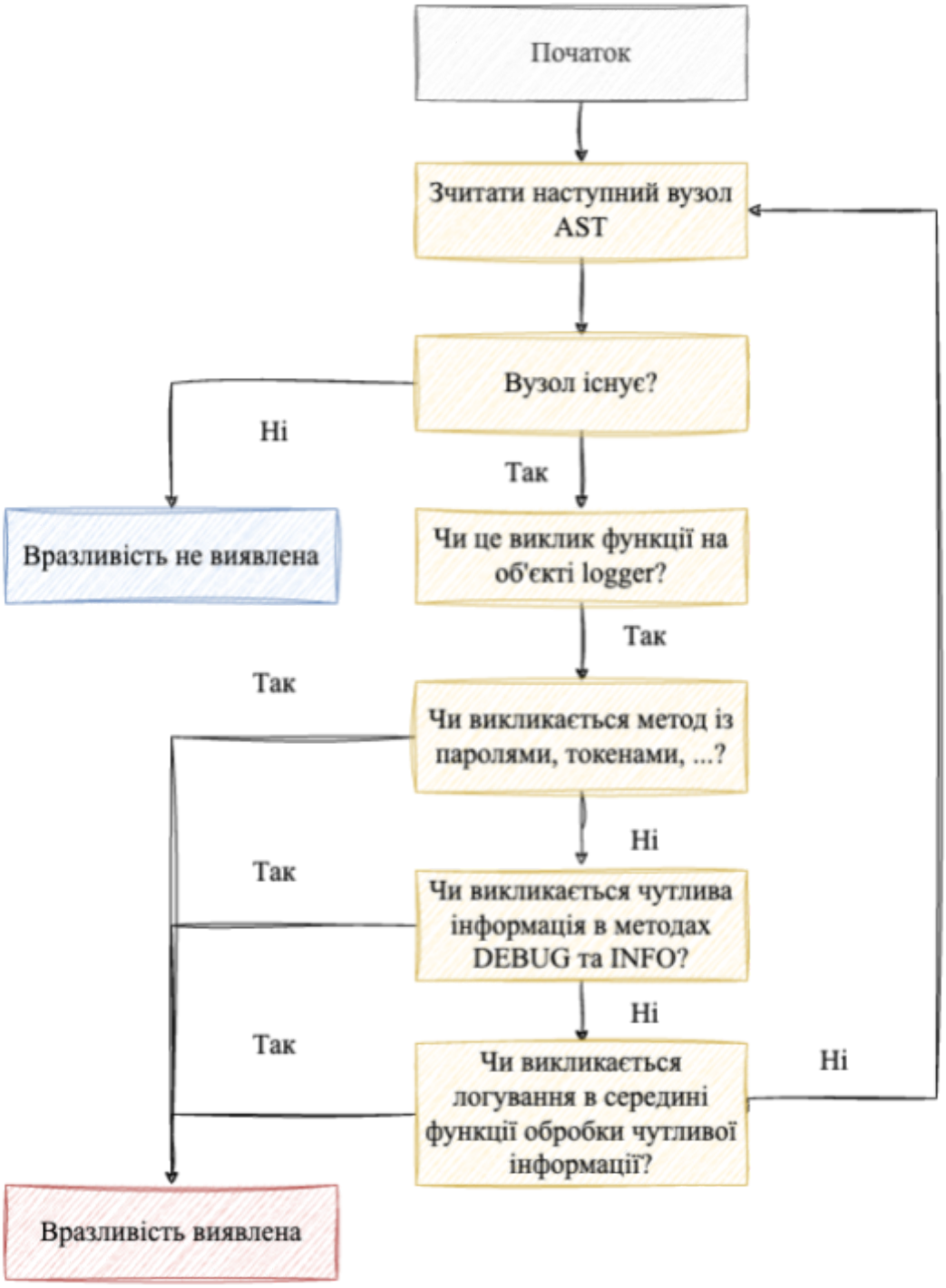

Рисунок 3.11 - Алгоритм виявлення небезпечного логування.

Статичний аналізатор може допомогти виявити та запобігти таким випадкам витоку інформації, виконуючи наступні завдання:

1. Виявлення чутливої інформації в логах. Аналізатор може сканувати код на предмет використання змінних або виразів, що містять чутливу інформацію, в контексті викликів функцій логування.
2. Перевірка рівня логування. Аналізатор може також перевіряти, чи логування використовується на відповідному рівні. Наприклад, чутлива інформація не повинна з'являтися в логах на рівні DEBUG або INFO.
3. Аналіз контексту логування. Перевіряти контекст, в якому виконується логування. Аналізатор може ідентифікувати потенційно небезпечні практики, такі як логування всередині функцій обробки паролів або особистих даних.

Враховуючи важливість захисту конфіденційної інформації, використання алгоритму, заснованого на AST та його обході, є ефективним методом виявлення потенційних вразливостей у логуванні. Цей підхід дозволяє детально аналізувати програмний код на предмет небезпечного використання функцій логування, що може призвести до витоку чутливої інформації. Завдяки точному визначенню та відстеженню потоків даних всередині AST, можливо ідентифікувати випадки, коли особисті дані, паролі або інші чутливі елементи потрапляють у логи.

Подальше вдосконалення алгоритму обходу AST дозволяє створювати більш гнучкі та адаптивні рішення для забезпечення безпеки програмного коду. Це не тільки сприяє попередженню непрямих витоків інформації, але й підвищує якість та надійність програмних продуктів загалом. Використання таких аналітичних інструментів є невід'ємною частиною сучасної розробки програмного забезпечення, направленої на забезпечення високих стандартів безпеки та приватності.

3.6 Безпечна обробка даних при серіалізації та десеріалізації

При серіалізації та десеріалізації даних існує ризик витоку інформації та інших вразливостей безпеки, особливо коли ці процеси використовуються для обробки чутливих або конфіденційних даних. Основні проблеми включають:

1. Витоки Даних. Чутливі дані можуть бути випадково включені в серіалізований об'єкт, роблячи їх доступними під час передачі або зберігання.

2. Небезпечна десеріалізація. Вразливості пов'язані з десеріалізацією можуть дозволити зловмисникам виконувати шкідливий код або маніпулювати об'єктами в пам'яті.

На рисунку 3.12 можна побачити приклад коду на мові програмування Crystal, що включає небезпечну серіалізацію даних. Як бачимо, емейл та пароль помилково серіалізуються у формат JSON. Зазвичай серіалізовані дані потім відправляються у інші системи, чи зберігаються в базу даних. Як результат така операція часто призводить до витоку даних, оскільки розробник часто забуває виключити чутливу до витоку інформацію.

```
1  require "json"
2
3  class User
4    JSON.mapping(
5      email: String,
6      password: String
7    )
8  end
9
10 user = User.from_json(%({"email": "user@example.com", "password": "my_secure_password"}))
11 puts user.to_json
```

Рисунок 3.12 - Приклад коду з небезпечною серіалізацією даних.

В даному випадку наш статичний аналізатор може допомогти ідентифікувати небезпечні ділянки коду, в яких відбувається серіалізація важливих даних та запобігти їх витоку. Для цього потрібно реалізувати наступні кроки:

1. Виявлення чутливих даних у серіалізації. Аналізатор може шукати випадки, коли чутливі дані, такі як паролі або персональні ідентифікатори, включаються в серіалізовані об'єкти.
2. Перевірка методів десеріалізації. Аналізувати використання методів десеріалізації, щоб виявити потенційно небезпечні практики, які можуть дозволити виконання шкідливого коду.
3. Аналіз правил серіалізації/десеріалізації. Перевіряти, чи встановлені належні правила та обмеження для процесів серіалізації та десеріалізації, зокрема, перевіряти наявність заходів безпеки для обробки чутливих даних.

Мова програмування Crystal має два відомих та важливих методи серіалізації та десереалізації, а саме JSON [13] та YAML [14]. Для виконання серіалізації використовують метод to_json та to_yaml, на перевірці яких ми і можемо сконцентруватись. Також ще існують метод build та dump, які наш аналізатор теж буде перевіряти, оскільки і вони можуть бути використані для серіалізації даних.

На рисунку 3.13 можна побачили блок схему для алгоритму виявлення небезпечної серіаліації, що може призвести до витоку даних. Як бачимо, для цього ми знову можемо застосувати AST дерево та зробити його обхід. Це дозволяє з знайти відповідні вузли в дереві, що відповідають викликам функції серіалізації, таких як to_json, to_yaml, build та dump. Далі алгоритм перевіряє чи відбуваються дані виклики функції із чутливими даними, наприклад, паролями, токенами або іншою інформацією, які важливо не поширювати.

Також варто зауважити, що даний алгоритм є досить швидким, оскільки працює тільки з потрібними вузлами в AST дереві, ігноруючи всі інші. Тобто, для виявлення вразливості нам не потрібно робити повний обхід дерева, а ми можемо сконцентруватись тільки на вузлах, що відповідають за виклики потрібних функцій.

Рисунок 3.13 - Алгоритм виявлення небезпечної серіалізації даних.

Такий підхід дозволяє виявити та запобігти потенційним вразливостям, пов'язаним з неналежною обробкою даних під час серіалізації, забезпечуючи вищий рівень захисту конфіденційної інформації.

Вище ми розглянули алгоритм для виявлення та запобігання вразливостям при серіалізації даних. Для того щоб запобігти небезпечній десеріалізації за допомогою статичного аналізу можна, потрібно виконати наступні кроки:

1. Ідентифікувати потенційно небезпечні функцій десеріалізації. Статичний аналізатор може шукати використання функцій десеріалізації в коді, особливо тих, що відомі своїми вразливостями. Це включає функції, які приймають довільний код або об'єкти від зовнішніх джерел, без належних перевірок або валідації.
2. Перевіряти вхідні дані для десеріалізації. Аналізатор може визначити, чи вхідні дані для десеріалізації проходять достатню валідацію та санітізацію. Наявність перевірок може зменшити ризик виконання шкідливого коду через десеріалізацію.
3. Забезпечувати використання безпечних практик десеріалізації. Переконатися, що в коді використовуються безпечні методи десеріалізації. Це може включати використання спеціальних бібліотек або функцій, які забезпечують додатковий рівень безпеки.
4. Обмежити доступ до класів під час десеріалізації. Перевіряти, чи обмежується доступ до певних класів під час процесу десеріалізації, щоб запобігти створенню та використанню потенційно шкідливих об'єктів.
5. Аналіз контексту використання десеріалізації. Аналізувати контекст, у якому використовується десеріалізація, наприклад, у сценаріях, де десеріалізовані дані отримуються з ненадійних джерел.

Імплементуючи ці підходи в статичному аналізі, можна значно знизити ризик вразливостей, пов'язаних з небезпечною десеріалізацією, та підвищити загальну безпеку програмного продукту.

Висновок до розділу 3

У даному розділі аналізується застосування AST дерева для ефективного статичного аналізу коду. Цей підхід дозволяє глибоко зануритися в структуру та логіку програмного коду, виявляючи потенційні вразливості та недоліки. Через систематичний обхід AST дерева, ми отримуємо змогу виявляти слабкі місця в програмному забезпеченні, включаючи витоки інформації при логуванні, недоліки в процесах шифрування та безпечної обробки даних.

Застосовуючи AST дерева та їх обхід, ми змогли представити та реалізувати алгоритми та правила для виявлення та усунення потенційних вразливостей, включаючи витоки інформації при зберіганні даних, логуванні, недоліки в шифруванні даних, а також проблеми, пов'язані з безпечною обробкою даних під час серіалізації та десеріалізації в програмних системах на мові програмування Crystal.

Отже, було запропоновано новий метод статичного аналізу коду, що базується на використанні AST дерев для програмних систем на мові Crystal. Даний підхід має на меті підвищити швидкість виявлення вразливостей та покращити загальну ефективність процесу забезпечення інформаційної безпеки ще на етапі розробки програмного забезпечення. Він дозволяє гнучко адаптувати процес аналізу під різні вимоги та потреби, відкриваючи нові можливості для захисту програмного забезпечення від сучасних кіберзагроз. Також, даний підхід може бути адаптований і використаний для описання правил для знаходження інших вразливостей. Впровадження цих алгоритмів в практику розробки програмного забезпечення може значно підвищити його безпеку, забезпечуючи більш надійний захист від сучасних кіберзагроз.

4 АНАЛІЗ ЕФЕКТИВНОСТІ РОЗРОБЛЕНИХ АЛГОРИТМІВ СТАТИЧНОГО АНАЛІЗУ

4.1 Практичне застосування статичного аналізатора

Результатом даного дослідження є реалізація статичного аналізатору Ameba [15] для мови програмування Crystal. Ameba створений для того, щоб підвищити рівень інформаційної безпеки в програмних системах на мові Crystal, пропонуючи унікальні можливості для аналізу коду та виявлення вразливостей. Даний статичний аналізатор активно використовує внутрішні механізми Crystal, такі як абстрактне синтаксичне дерево (AST) та паттерн Visitor, що дозволяє детально аналізувати програмний код і виявляти вразливості з високою точністю. Ameba ефективно використовує ці механізми та реалізує алгоритми, описані у розділі 3, що є критично важливим для ідентифікації складних вразливостей.

Особливістю створеного аналізатора є його здатність робити детальні звіти про виявлені помилки та вразливості. Ці звіти не тільки допомагають розробникам швидко ідентифікувати та виправляти помилки, але й надають цінну інформацію для покращення процесів розробки програмних систем. На рисунку 4.1 можна побачити приклад звіту, що містить інформацію про знайдену вразливість SQL Injection при виконанні статичного аналізу на етапі розробки програмного забезпечення.

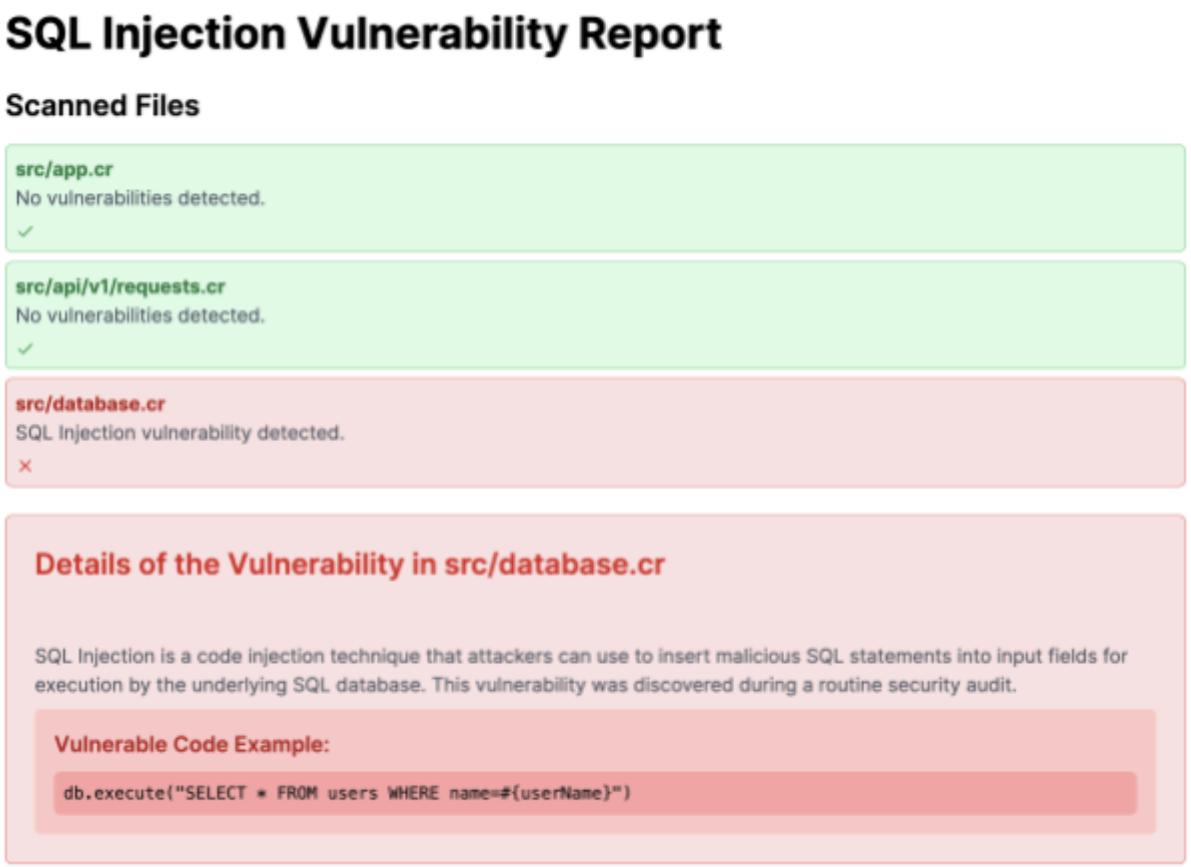

Рисунок 4.1 - Приклад звіту про знайдену вразливість SQL Injection.

Ще однією з ключових особливостей Ameba є її здатність інтегруватися з різними системами та інструментами, що використовуються для статичного аналізу, такими як Codacy [16] та ReviewDog [17]. Це забезпечує широкі можливості для впровадження Ameba в різні процеси розробки та розширює її застосування за межі вузько спеціалізованих сценаріїв. Інтеграція з такими системами спрощує процес впровадження інструменту для його використання під час розробки програмного забезпечення.

Варто також зазначити, що наразі Ameba користується значною популярністю серед розробників Crystal, оскільки це єдиний інструмент такого роду для даної мови програмування. На рисунку 4.2 можна побачити частоту скачувань інструменту для проведення аналізу в період між 27/11/2023 та 10/12/2023. Зеленим відображається загальна кількість скачувань, а синім - кількість унікальних скачувань. Можна побачити, що цифри варіюються між 100

та 1000 скачувань в день, що є досить великою кількістю. Використання аналізатора відображає ефективність інструменту і потребу спільноти мови програмування Crystal у надійних та ефективних методах для забезпечення безпеки коду.

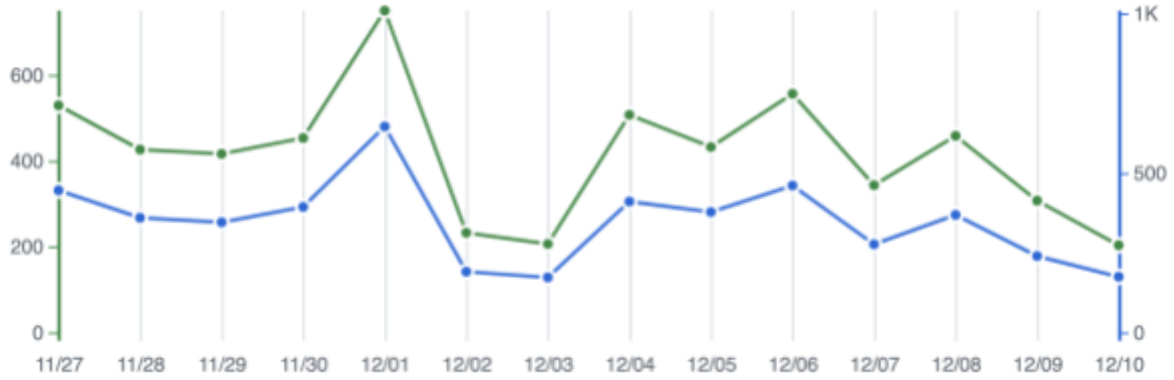

Рисунок 4.2 - Частота скачувань розробленого аналізатора для проведення аналізу.

Отже, Ameba не тільки відіграє важливу роль у підвищенні безпеки розробки на Crystal. Можливість інтеграції з іншими системами та широка підтримка з боку спільноти робить Ameba цінним інструментом для будь-якого розробника, який прагне забезпечити високий рівень безпеки своїх програмних продуктів.

4.2 Розробка та проведення опитування для оцінки ефективності

У рамках цього дослідження ми провели опитування з метою оцінки ефективності статичного аналізатора, розробленого для мови програмування Crystal, та його впливу на забезпечення інформаційної безпеки програмного коду. Опитування спрямоване на отримання об'єктивних думок та оцінок від експертів та фахівців, які працюють із мовою програмування Crystal та використовують наш статичний аналізатор в процесі розробки програмних систем.

Цей опитувальник створено з метою оцінити ефективність розробленого статичного аналізатора коду, спрямованого на виявлення вразливостей, пов'язаних з інформаційною безпекою. Практичне застосування таких інструментів вимагає постійного перегляду та оцінки їхньої роботи, з метою вдосконалення та забезпечення високої якості результатів. Через опитування ми плануємо зібрати відгуки користувачів та експертів, які використовували аналізатор в реальних умовах розробки.

Ключовою метою цього опитувальника є визначення, наскільки добре аналізатор впорався із задачами виявлення різноманітних типів вразливостей. Відповіді допоможуть виявити потенційні слабкі місця в алгоритмах аналізатора, оцінити його здатність виявляти різні типи вразливостей, включаючи SQL ін'єкції, проблеми з шифруванням даних, витоки інформації при логуванні та серіалізації/десеріалізації.

Окрім технічної ефективності, важливими аспектами є також помилкове виявлення вразливостей (false positives) та невиявлення реальних вразливостей (false negatives). Це дозволить зрозуміти баланс між точністю та надійністю аналізатора. Відповіді на ці питання допоможуть виявити ключові аспекти, які потребують поліпшення, та покращити загальну якість роботи аналізатора.

Наш опитувальник складається із 7 питань, кожне з яких потрібно оцінити за шкалою від 1 до 10. В даному випадку 10 - означає дуже добре, а 1 - дуже погано. Опитувальник виглядає наступним чином:

1. Загальна ефективність. Як би ви оцінили загальну ефективність аналізатора у виявленні вразливостей в коді, пов’язаних з інформаційною безпекою?
2. Виявлення SQL ін'єкцій. Наскільки ефективним є статичний аналізатор у виявленні SQL ін'єкцій? Чи зумів аналізатор виявити всі відомі випадки SQL ін’єкцій в вашому коді?
3. Виявлення проблем із шифруванням. Як ви оцінюєте точність роботи аналізатора, спрямованого на виявлення проблем із шифруванням даних?

4. Виявлення витоків інформації при логуванні. Наскільки добре аналізатор виявляє потенційні витоки інформації під час логування?
5. Виявлення витоків інформації при серіалізації/десеріалізації. Наскільки добре аналізатор виявляє потенційні витоки інформації при серіалізації та десеріалізації даних?
6. Помилкове виявлення вразливостей. Чи виявляв аналізатор вразливості, які насправді не є такими (false positives)? 1 - таких випадків дуже багато, 10 - таких випадків майже не існує.
7. Помилкове не виявлення вразливостей. Чи існували випадки, коли аналізатор не виявляв реальні вразливості в коді (false negatives), але мав би згідно документації? 1 - таких випадків дуже багато, 10 - таких випадків майже не існує.

Збір та аналіз відгуків є важливою частиною процесу розвитку та удосконалення статичного аналізатора. Інформація, отримана з цього опитування, допоможе в налагодженні більш ефективного та надійного інструменту для забезпечення інформаційної безпеки програмного забезпечення.

4.3 Метод експертних оцінок для аналізу результатів опитування

Поширивши опитувальник та отримавши певні відповіді, ми можемо сфокусуватись на аналізі результатів опитування, проведеного серед користувачів нашого статичного аналізатора коду. Опитування було спрямоване на оцінку різних аспектів ефективності аналізатора, включаючи його здатність виявляти різноманітні типи вразливостей. Мета аналізу - отримати зворотний зв'язок від реальних користувачів, що дозволить нам краще зрозуміти, як аналізатор виконує свої функції в практичних умовах.

Аналіз результатів опитування виконується за допомогою методу експертних оцінок. Цей метод дає нам змогу систематизувати та кількісно оцінити

відгуки, отримані від користувачів. Експертні оцінки базуються на агрегації відповідей користувачів, що дозволяє нам визначити загальні тенденції та висновки про ефективність аналізатора.

Цей підхід допоможе нам ідентифікувати ключові сильні та слабкі сторони аналізатора, а також надасть цінні вказівки на те, в яких напрямках необхідно розвивати та вдосконалювати продукт. Результати цього аналізу будуть служити фундаментом для наступних етапів розвитку та вдосконалення аналізатора.

Даний аналіз надає нам безпосередній зворотний зв'язок від тих, хто використовує наш продукт. Ця інформація є ключовою для забезпечення того, щоб наш аналізатор залишався не тільки технічно передовим, але й максимально відповідав потребам користувачів. Результати опитування серед п’яти користувачів можна побачити в таблиці 4.1.

Таблиця 4.1 - Метод експертних оцінок для аналізу ефективності розробленого статичного аналізатору.

№	Питання	1	2	3	4	5	Середнє	Нормоване
1	Загальна ефективність	7	8	6	7	7	7,00	0,16
2	Виявлення SQL injection	8	9	7	8	9	8,20	0,19
3	Виявлення проблем з шифруванням	6	5	7	6	7	6,20	0,14
4	Виявлення витоків інформації при логуванні	5	4	6	5	6	5,20	0,12
5	Виявлення витоків інформації при серіалізації/десеріалізації	4	5	3	4	5	4,20	0,10
6	Помилкове виявлення вразливостей	7	6	8	7	8	7,20	0,17
7	Помилкове не виявлення вразливостей	4	6	6	7	5	5,60	0,13
Сума середніх значень							43,60	1,00

На основі проведеного опитування та методу експертних оцінок можна зробити наступні висновки:

1. Загальна ефективність. Загальна ефективність аналізатора оцінюється як досить висока з середнім балом 7,00. Це свідчить про те, що користувачі в цілому задоволені роботою аналізатора у виявленні вразливостей.
2. Виявлення SQL Injection. Найвищий середній бал, 8,20, вказує на високу ефективність аналізатора у виявленні SQL ін'єкцій. Це підкреслює, що цей аспект аналізатора працює дуже добре.
3. Виявлення проблем з шифруванням: Середній бал 6,20 у цій категорії свідчить про задовільну ефективність аналізатора у виявленні проблем з шифруванням, хоча ця область може потребувати покращення.
4. Виявлення витоків інформації при логуванні та серіалізації/десеріалізації: Нижчі середні бали (5,20 та 4,20 відповідно) вказують на те, що ці аспекти є слабшими сторонами аналізатора і потребують додаткової уваги та вдосконалення.
5. Помилкове виявлення вразливостей: Середній бал 7,20 вказує на добру здатність аналізатора мінімізувати випадки помилкового виявлення вразливостей.
6. Помилкове не виявлення вразливостей: З оцінкою 5,60 ця категорія показує, що існує простір для покращення в аспекті виявлення реальних вразливостей, які аналізатор міг упустити.

У цілому, результати опитування свідчать про досить високу ефективність розробленого аналізатора, хоча в деяких областях є потенціал для подальшого розвитку та вдосконалення. Очевидно, що виявлення деяких вразливостей, таких як витоки інформації при логуванні та серіалізації, можна покращити, але це в свою чергу може збільшити помилкові виявлення вразливостей. Тому тут важливо дотримуватись певного компромісу, оскільки будь-який статичний аналіз завжди

перебуває на межі між високою ефективністю та високим коефіцієнтом помилкового виявлення вразливостей або ж низькою ефективністю та низьким коефіцієнтом помилкового виявлення.

Висновки до розділу 4

В даному розділі ми сфокусуватися на аналізі ефективності розробленого аналізатора Ameba. Цей аналізатор, створений як результат наших досліджень, виявився ефективним інструментом для забезпечення інформаційної безпеки програмних систем на мові Crystal. Використовуючи внутрішні механізми Crystal, такі як AST та паттерн Visitor, Ameba демонструє високу ефективність у виявленні вразливостей, одночасно забезпечуючи детальні звіти, що допомагають розробникам у локалізації та виправленні помилок.

Практичне застосування Ameba підтверджує її значимість та популярність серед спільноти розробників Crystal. Її унікальність як єдиного аналізатора для Crystal, а також здатність інтегруватися з іншими системами, такими як Codacy та ReviewDog, робить її важливим інструментом в сучасному процесі розробки.

Опитування та метод експертних оцінок, використані для аналізу ефективності Ameba, дали змогу отримати детальну інформацію від користувачів, виявивши ключові сильні та слабкі сторони аналізатора. В цілому, результати показують, що Ameba ефективно виконує свої функції, пропонуючи глибокий аналіз коду, що є важливим для підвищення якості та безпеки програмного забезпечення.

Цей розділ також підкреслює важливість безперервного оновлення та вдосконалення аналізатора, реагуючи на потреби розробників та виклики у сфері програмування.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було проведено глибоке дослідження з питань забезпечення інформаційної безпеки в програмних системах. Основною метою роботи було виявлення можливостей попереднього статичного аналізу коду як ефективного інструменту для підвищення рівня безпеки програмних продуктів.

Дослідження акцентувало увагу на сучасних викликах у сфері інформаційної безпеки, зокрема на зростаючій кількості кіберзагроз. Аналіз різних методів і підходів до забезпечення безпеки виявив, що статичний аналіз коду є однією з ключових компонент комплексного підходу до захисту інформаційних систем. Цей метод дозволяє виявити потенційні вразливості на ранніх етапах розробки програмного забезпечення, що істотно підвищує ефективність процесу забезпечення інформаційної безпеки.

В рамках роботи було розглянуто специфіку застосування статичного аналізу для мови програмування Crystal. Оцінюючи переваги та обмеження цього підходу, було показано, що впровадження статичного аналізу може значно підвищити рівень захисту програмних продуктів, однак вимагає врахування специфіки конкретної мови програмування та контексту застосування.

Враховуючи отримані результати, можна зробити висновок, що забезпечення інформаційної безпеки в сучасних програмних системах вимагає комплексного підходу, який включає різноманітні техніки та технології. Статичний аналіз коду є ефективним інструментом в цьому процесі, але його застосування має бути інтегрованим з іншими методами захисту. Такий підхід дозволяє досягти більш високого рівня безпеки та адаптивності до змінюваних умов кіберпростору.

Загалом, дипломна робота надає цінний внесок у розуміння важливості та ефективності статичного аналізу коду як частини комплексного забезпечення інформаційної безпеки в програмних системах. Робота демонструє, що впровадження таких технологій не лише підвищує безпеку, але й сприяє розвитку більш надійних та ефективних програмних рішень. Рекомендації, викладені в

роботі, можуть бути корисними для спеціалістів у сфері кібербезпеки та розробників програмного забезпечення, що прагнуть досягти вищого рівня захисту інформації та даних.

Тезиси до даної роботи були опубліковані на міжнародному молодіжному форумі «Радіоелектроніка та молодь у XXI столітті» [18].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. “600000 servers vulnerable to Heartbleed.” IT Security Guru. URL: https://www.itsecurityguru.org/2014/04/10/600000-servers-vulnerable-heartbleed (дата звернення 10.10.2023).
2. Barker, Ian. “Number of new Common Vulnerabilities and Exposures (CVEs) expected to increase in 2023.”. URL: https://betanews.com/2023/02/01/number-of-new-cves-expected-to-increase-2023/ (дата звернення 04.11.2023).
3. Dietrich, George, and Guilherme Bernal. Crystal Programming: A Project-Based Introduction to Building Efficient, Safe, and Readable Web and CLI Applications, 2022. 212 с.
4. Gibbs, Samuel. “TalkTalk criticised for poor security and handling of hack attack.” The Guardian. URL: https://www.theguardian.com/technology/2015/oct/23/talktalk-criticised-for-poor-security-and-handling-of-hack-attack (дата звернення 21.09.2023).
5. Hsu, Tony Hsiang-Chih. Practical Security Automation and Testing: Tools and Techniques for Automated Security Scanning and Testing in DevSecOps, 2019. 128 с.
6. Weidman, Georgia. Penetration Testing: A Hands-On Introduction to Hacking, 2014. 354 с.
7. Petrosyan, Ani. Number of common IT security vulnerabilities and exposures (CVEs) worldwide from 2009 to 2023 YTD. URL: https://www.statista.com/statistics/500755/worldwide-common-vulnerabilities-and-exposures/ (дата звернення 01.11.2023).
8. “Understanding How SQL Injection Attacks Work.” Wordfence. URL: https://www.wordfence.com/learn/how-to-prevent-sql-injection-attacks/ (дата звернення 01.11.2023).

9. Venkat, Apurva. “Vulnerabilities and exposures to rise to 1900 a month in 2023: Coalition.”. URL: https://www.csoonline.com/article/574485/vulnerabilities-and-exposures-to-rise-to-1900-a-month-in-2023-coalition.html (дата звернення 01.11.2023).
10. AST для мови програмування Crystal. URL: https://github.com/crystal-lang/crystal/blob/2d8ff9cb94a211c5beef1217f8b7b804cc3b8644/src/compiler/crystal/syntax/ast.cr (дата звернення 09.12.2023).
11. Visiting an Abstract Syntax Tree, Pat Shaughnessy. URL: https://patshaughnessy.net/2022/1/22/visiting-an-abstract-syntax-tree (дата звернення 09.12.2023).
12. Модуль Digest в стандартній бібліотеці мови програмування Crystal, URL: https://crystal-lang.org/api/1.10.1/Digest.html (дата звернення 09.12.2023).
13. Модуль JSON в стандартній бібліотеці мови програмування Crystal. URL: https://crystal-lang.org/api/1.10.1/JSON.html (дата звернення 09.12.2023).
14. Модуль YAML в стандартній бібліотеці мови програмування Crystal. URL: https://crystal-lang.org/api/1.10.1/YAML.html (дата звернення 09.12.2023).
15. Ameba - статичний аналізатор коду для мови програмування Crystal. URL: https://github.com/crystal-ameba/ameba (дата звернення 09.12.2023).
16. Linting Crystal on Codacy. URL: https://medium.com/@veelenga/linting-crystal-on-codacy-4c19aa8216df (дата звернення 09.12.2023).
17. A code review dog who keeps your codebase healthy. URL: https://medium.com/@haya14busa/reviewdog-a-code-review-dog-who-keeps-your-codebase-healthy-d957c471938b#.8xctbaw5u (дата звернення 21.12.2023).
18. Elenhaupt V. V. Examining the influence of static application security testing on the overall security of web applications // 27-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті». Зб. матеріалів форуму. Т. 3. – Харків: ХНУРЕ. 2023. – С. 253-254.