НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

До захисту допущено
Завідувач кафедри

_____ **Дмитро ЛАНДЕ**
(підпис)

«19» червня 2025 р.

Дипломна робота
на здобуття ступеня бакалавра
за освітньо-професійною програмою «Системи, технології та математичні методи кібербезпеки»
спеціальності 125 «Кібербезпека»

на тему: «Моделі і методи аналізу вразливостей програмного забезпечення на основі LLM»

Виконав: здобувач вищої освіти **IV** курсу, групи ФБ-12
(шифр групи)

Приходько Юрій Олександрович
(прізвище, ім'я, по батькові) (підпис)

Керівник: К.т.н., доц. Ільїн М.І.
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Рецензент: доцент кафедри ММЗІ, к.т.н
Яковлєв Сергій Володимирович
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім'я, по батькові) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Здобувач вищої освіти _____
(підпис)

Київ - 2025 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 125 «Кібербезпека»
Освітньо-професійна програма «Системи, технології та математичні методи кібербезпеки»

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ Дмитро ЛАНДЕ
(підпис)

«19» червня 2025 р.

ЗАВДАННЯ
на дипломну роботу здобувачу вищої освіти

Приходько Юрію Олександровичу
(прізвище, ім'я, по батькові)

1. Тема роботи: «Моделі і методи аналізу вразливостей програмного забезпечення на основі LLM»,
керівник роботи Ільїн Микола Іванович, кандидат технічних наук,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

   затверджені наказом по університету від 26 травня 2025 р. № 1761-с

2. Термін подання здобувачем вищої освіти роботи 12 червня 2025 р.

3. Вихідні дані до роботи: література з аналізу вразливостей програмного забезпечення, наявні методи пошуку вразливостей та інтеграції інструментів з великими мовними моделями, їх реалізація та технічна документація.

4. Зміст роботи: огляд сучасних методів аналізу вразливостей пошкодження пам'яті, реалізація моделі автоматизації пошуку вразливостей бінарних файлів на основі LLM, експерементальне дослідження запропонованої моделі аналізу.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): «Моделі і методи аналізу вразливостей програмного забезпечення на основі LLM» – презентація.

6. Дата видачі завдання: 02.04.2025

Календарний план

No з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1	Отримання завдання та узгодження теми роботи	02.04.2025	Виконано
2	Вивчення сучасних методів інтеграції інструментів з LLM	02.04.2025 – 11.04.2025	Виконано
3	Огляд наукової літератури по темі	12.04.2025 – 20.04.2025	Виконано
4	Огляд наявних методів пошуку вразливостей	21.04.2025 – 25.04.2025	Виконано
5	Створення та налаштування середовища аналізу	26.04.2025 – 10.05.2025	Виконано
6	Дослідження можливостей моделі та покращення методів аналізу	11.05.2025 – 15.05.2025	Виконано
7	Створення тестових зразків та їх аналіз	16.05.2025 – 21.05.2025	Виконано
8	Визначення ефективності моделі пошуку вразливостей	22.05.2025	Виконано
9	Аналіз отриманих результатів	23.05.2025 – 24.05.2025	Виконано
10	Фінальне оформлення дипломної роботи	25.05.2025 – 07.06.2025	Виконано
11	Передзахист дипломної роботи	12.06.2025	
12	Захист дипломної роботи	19.06.2025	

Здобувач вищої освіти _____ Юрій ПРИХОДЬКО
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ Микола ІЛЬЇН
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Обсяг дипломної роботи 81 сторінка, 22 ілюстрацій, 2 таблиці, 37 джерела.

У роботі визначено сучасні методи аналізу вразливостей програмного забезпечення на основі інтеграції інструментів зворотної розробки з великими мовними моделями (LLM). Створено та програмно реалізовано модель аналізу бінарних файлів з використанням декомпіляторів Binary Ninja, IDA Pro та великих мовних моделей через протокол MCP (Model-Code-Protocol). Програмна реалізація моделі полягає у використанні MCP-серверу, який забезпечує взаємодію між LLM та інструментами декомпіляції, дозволяючи моделям отримувати доступ до функцій аналізу коду, декомпіляції та виявлення вразливостей. За допомогою запропонованої моделі було проаналізовано тестові зразки бінарних файлів на наявність класичних вразливостей пам'яті. Доведено успішність аналізу шляхом виявлення різних типів вразливостей з автоматичною генерацією детального звіту, що включає опис вразливості, приклади експлуатації та рекомендації щодо усунення. Досліджено ефективність використання LLM для задач зворотної розробки та пошуку вразливостей пошкодження пам'яті, включаючи відновлення алгоритмів з бінарного коду та пошук вразливостей у застосунках, бібліотеках і драйверах. Розроблено методику формування ефективних запитів до LLM для максимізації якості аналізу. Запропонована реалізація та отримані результати дослідження стануть корисними в системах автоматизованого аналізу безпеки програмного забезпечення, значно підвищуючи швидкість виявлення вразливостей та знижуючи вимоги до кваліфікації аналітиків. Модель може застосовуватися в підрозділах кібербезпеки для превентивного аналізу програмного забезпечення та виявлення zero-day вразливостей.

Ключові слова: **зворотня розробка, статичний аналіз, аналіз бінарних вразливостей, AI, LLM, MCP**

ABSTRACT

The volume of the thesis is 81 pages, 22 illustrations, 2 table, and 37 sources of literature.

The paper defines modern methods for software vulnerability analysis based on the integration of reverse engineering tools with large language models (LLM). A model for analyzing binary files using Binary Ninja, IDA Pro decompilers and large language models via the MCP (Model-Code-Protocol) protocol has been created and implemented in software. The software implementation of the model consists in using the MCP server, which provides interaction between LLM and decompilation tools, allowing models to access the functions of code analysis, decompilation and vulnerability detection. Using the proposed model, test samples of binary files were analyzed for the presence of classic memory vulnerabilities. The success of the analysis was proven by detecting various types of vulnerabilities with automatic generation of a detailed report, including a description of the vulnerability, examples of exploitation and recommendations for elimination. The effectiveness of using LLM for reverse engineering tasks and memory corruption vulnerabilities, including recovering algorithms from binary code and searching for vulnerabilities in applications, libraries and drivers, was investigated. A methodology for generating effective LLM queries to maximize the quality of analysis has been developed. The proposed implementation and the obtained research results will be useful in automated software security analysis systems, significantly increasing the speed of vulnerability detection and reducing the requirements for analysts' qualifications. The model can be used in cybersecurity departments for preventive software analysis and detection of zero-day vulnerabilities.

Keywords: **reverse engineering, static analysis, binary vulnerability analysis, AI, LLM, MCP**

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AI – Штучний Інтелект, Artificial Intelligence; здатність комп'ютерних систем виконувати завдання, що потребують інтелекту людини (аналіз, навчання, генерація рішень).

LLM – Велика Мовна Модель, Large Language Model; тип моделі машинного навчання, натренованої на великих обсягах тексту, здатної аналізувати та генерувати текстові дані.

MCP – Model-Code-Protocol, протокол взаємодії між мовними моделями (LLM) та інструментами зворотної розробки, наприклад, декомпіляторами.

Зворотна розробка – процес аналізу бінарного програмного забезпечення з метою відновлення логіки його роботи.

Декомпіляція – процес перетворення виконуваного коду у високорівневий псевдокод для полегшення аналізу.

Вразливості пошкодження пам'яті – це клас помилок у програмному забезпеченні, що виникають внаслідок некоректної роботи з оперативною пам'яттю, зокрема при виділенні, записі, читанні або звільненні пам'яті. Такі вразливості можуть призводити до неконтрольованої поведінки програми, витоку даних, порушення цілісності або виконання довільного коду зловмисником.

Промпт інженерія – процес створення та оптимізації запитів до мовних моделей для отримання найбільш релевантних та корисних відповідей. Включає в себе формулювання запитів, налаштування параметрів моделі та аналіз результатів.

ВСТУП

У сучасних умовах швидкого розвитку інформаційних технологій та зростання складності програмного забезпечення кількість вразливостей у бінарних файлах зростає експоненційно. Кожен день з'являються нові загрози безпеці, які можуть призводити до серйозних порушень конфіденційності, цілісності та доступності інформаційних систем. Традиційні методи аналізу бінарних файлів та пошуку вразливостей потребують значних витрат часу та високої кваліфікації фахівців, що робить процес виявлення загроз трудомістким та дорогим. Одним з перспективних підходів до вирішення цієї проблеми є інтеграція інструментів зворотної розробки з великими мовними моделями (LLM). Поєднання можливостей сучасних декомпіляторів з потужністю штучного інтелекту дозволяє значно автоматизувати процес аналізу програмного коду, підвищити швидкість виявлення вразливостей та знизити вимоги до кваліфікації аналітиків. Попри значний прогрес у розвитку як інструментів зворотної розробки, так і технологій штучного інтелекту, тема їх ефективної інтеграції для задач аналізу безпеки залишається недостатньо дослідженою. Особливо актуальним є питання створення уніфікованого підходу до взаємодії LLM з різними декомпіляторами через стандартизовані протоколи.

Актуальність роботи: Розробка та вдосконалення методів автоматизованого аналізу вразливостей програмного забезпечення на основі інтеграції LLM з інструментами зворотної розробки для підвищення ефективності виявлення загроз кібербезпеці та застосування у пошуку недекларованих можливостей.

Мета і завдання дослідження: Вдосконалення методів виявлення та аналізу вразливостей пошкодження пам'яті.

Об'єкти дослідження: Бінарні вразливості в програмному забезпеченні та методи їх виявлення.

Предмет дослідження: Методи інтеграції великих мовних моделей з

інструментами зворотної розробки для автоматизованого аналізу вразливостей.

Методи дослідження:

1. Системний аналіз існуючих підходів до зворотної розробки.
2. Розробка запропонованої моделі.
3. Експериментальне тестування на реальних зразках бінарних файлів.

Наукова новизна одержаних результатів: Запропоновано модель інтеграції інструментів декомпіляції з великими мовними моделями через протокол MCP, що дозволяє автоматизувати процес аналізу бінарних вразливостей та значно знижує трудомісткість виявлення загроз безпеці програмного забезпечення.

Практичне значення одержаних результатів: Досліджена модель інтеграції декомпіляторів з великими мовними моделями через протокол MCP може бути впроваджена в комерційні системи автоматизованого аналізу безпеки програмного забезпечення, забезпечуючи скорочення часу виявлення вразливостей з годин до хвилин та зниження вимог до кваліфікації аналітиків. Практична реалізація забезпечує можливість превентивного виявлення zero-day вразливостей у драйверах та системному програмному забезпеченні, що може запобігти кібератакам типу LOLDrivers та підвищити загальний рівень кібербезпеки організацій.

Апробація результатів роботи: Результати роботи були представлені на XXIII Всеукраїнській науково-практичній конференції студентів, аспірантів та молодих вчених «Теоретичні і прикладні проблеми фізики, математики та інформатики» (14 - 17.05.2025 р., м. Київ, Україна).

1 СУЧАСНІ МЕТОДИ АНАЛІЗУ БІНАРНИХ ВРАЗЛИВОСТЕЙ

Пошук вразливостей пошкодження пам'яті є важливою складовою кібербезпеки, і нерозривно пов'язаний з аналізом компільованого коду та зворотньою розробкою. В свою чершу зворотня розробка програмного забезпечення пройшла довгий шлях розвитку від простих дизасемблерів до комплексних платформ з можливостями статичного та динамічного аналізу. На ранніх етапах аналіз проводився переважно на рівні асемблерного коду, що вимагало глибокого розуміння архітектури процесора та низькорівневих принципів роботи програм. З появою декомпіляторів, стало можливим відновлення високорівневого коду, у вигляді псевдокоду, що значно спростило процес аналізу. Однак, навіть з використанням сучасних інструментів, зворотня розробка, а тим паче пошук бінарних вразливостей в великих проектах, залишається складним завданням, що вимагає значного досвіду та спеціалізованих знань з розуміння алгоритмів, структур даних та принципів роботи програмного забезпечення.

1.1 Традиційні методи пошуку вразливостей

Пошук вразливостей у програмному забезпеченні традиційно здійснюється за допомогою двох основних підходів: статичного та динамічного аналізу. Кожен з цих методів має свої переваги та обмеження, які впливають на ефективність виявлення вразливостей.

1.1.1 Статичний аналіз

Статичний аналіз є одним із найпоширеніших підходів і полягає у виявленні вразливостей у програмному забезпеченні без його виконання. Цей метод базується на дослідженні вихідного коду або декомпільованого пред-

ставлення програми з метою ідентифікації потенційних проблем безпеки через аналіз структури, логіки та використання небезпечних функцій. Основною перевагою статичного аналізу є можливість дослідження всіх можливих шляхів виконання програми без необхідності створення специфічних умов для їх активації. Автоматичні аналізатори здатні виявляти типові патерни вразливостей, такі як переповнення буферів, використання неініціалізованих змінних, витоки пам'яті та інші класичні проблеми безпеки. Сучасні інструменти статичного аналізу [1–3] використовують складні алгоритми для побудови графів потоку управління та потоку даних, що дозволяє відстежувати рух інформації через програму та виявляти місця потенційного некоректного використання. Проте ефективність таких інструментів суттєво знижується при зустрічі з обфускованим кодом і іншими методами захисту навантаження, які навмисно ускладнюють процес аналізу. Додатковою проблемою статичного аналізу є висока кількість хибних спрацювань, особливо при роботі з великими програмними комплексами. Автоматичні сканери часто не враховують контекст виконання та специфічні умови, що призводить до повідомлень про потенційні вразливості в місцях, де вони фактично не можуть бути експлуатовані.

1.1.2 Динамічний аналіз

Динамічний аналіз проводиться під час виконання програми в контрольованому середовищі. Цей підхід дозволяє спостерігати за реальною поведінкою додатка, відстежувати використання ресурсів системи та виявляти аномалії, які можуть свідчити про наявність вразливостей. Основними методами динамічного аналізу є інструментація коду, що дозволяє вставляти додаткові перевірки безпосередньо в програму, та використання спеціалізованих інструментів моніторингу. Такі засоби, як Valgrind [4] та AddressSanitizer [5], здатні виявляти помилки роботи з пам'яттю в реальному часі, включаючи переповнення буферів, використання звільненої пам'яті та витоки ресурсів. Фазинг представляє особливий інтерес серед методів динамічного аналізу. Цей підхід передбачає автоматичну генерацію великої кількості тестових

даних, часто випадкових або мутованих на основі існуючих зразків, з метою викликати непередбачувану поведінку програми. Сучасні фазери, такі як AFL++ [6] та LibFuzzer [7], використовують покриття коду для направленого пошуку нових шляхів виконання та потенційних вразливостей. Проте динамічний аналіз має суттєве обмеження у вигляді неможливості дослідити всі можливі шляхи виконання програми. Вразливості, що знаходяться в рідко виконуваних ділянках коду або потребують специфічних умов для активації, можуть залишитися непоміченими. Крім того, результати динамічного аналізу сильно залежать від якості тестових даних та повноти покриття функціональності програми.

1.2 Недоліки традиційних методів пошуку вразливостей

Традиційні підходи до виявлення вразливостей у бінарних файлах мають ряд фундаментальних обмежень, які знижують їх ефективність у сучасних умовах.

Основною проблемою є залежність від заздалегідь визначених правил та евристик, що робить ці методи малоефективними проти нових типів вразливостей або складних технік обфускації. Статичні аналізатори часто стикаються з проблемою неточного моделювання поведінки програми, особливо в присутності динамічних конструкцій, таких як виклики функцій через покажчики або складні структури даних. Це призводить як до пропуску реальних вразливостей, так і до генерації великої кількості хибних спрацювань, що значно ускладнює роботу аналітиків.

Динамічні методи, у свою чергу, обмежені необхідністю виконання програми та створення відповідних умов для виявлення вразливостей. Багато серйозних проблем безпеки можуть проявлятися лише при специфічних комбінаціях вхідних даних або стану системи, які важко відтворити в лабораторних умовах.

Ще однією значною проблемою є масштабованість традиційних методів. Сучасні програмні комплекси містять мільйони рядків коду, а їх аналіз вимагає значних обчислювальних ресурсів та часу. Ручний аналіз результатів ав-

томатизованих інструментів також потребує високої кваліфікації спеціалістів та значних витрат часу. Традиційні методи також демонструють слабкість у роботі з обфускованим кодом та антидебаг-технологіями. Сучасні зловмисники активно використовують різноманітні техніки приховування логіки програм, що робить традиційний аналіз малоефективним або взагалі неможливим.

1.3 Класифікація та характеристика бінарних вразливостей

Розглянемо основні класи вразливостей на яких зосереджено увагу в дослідженні [8]. Ці вразливості є класичними прикладами проблем з пам'яттю, які можуть призвести до серйозних наслідків, таких як виконання довільного коду, витік даних або відмова в обслуговуванні. Вони часто використовуються в атаках на програмне забезпечення та є основою для розробки нових методів захисту [9].

1.3.1 Переповнення буфера в купі (Heap-based Buffer Overflow)

Переповнення буфера в динамічній пам'яті являє собою одну з найнебезпечніших категорій вразливостей, що виникає при записі даних за межі виділеного буфера в heap-області пам'яті. Особлива небезпека цього типу вразливості полягає в можливості перезапису метаданих менеджера пам'яті, що може призвести до виконання довільного коду з привілеями цільової програми. Heap-менеджери сучасних операційних систем використовують складні структури даних для відстеження виділених блоків пам'яті. Переповнення буфера може пошкодити ці метадані, створивши можливості для контролю над процесом виділення та звільнення пам'яті. Експлуатація таких вразливостей часто базується на детальному розумінні внутрішньої структури heap-менеджера конкретної системи.

```
// vulnerable code example
void process_data(char* input) {
    char* buffer = malloc(64);
```

```
    strcpy(buffer, input); // potential buffer overflow
    process_buffer(buffer);
    free(buffer);
}
```

Складність захисту від heap-based переповнень обумовлена різноманітністю реалізацій менеджерів пам'яті та специфічними особливостями їх роботи в різних операційних системах. Сучасні захисні механізми, такі як guard pages та randomization heap layout, ускладнюють експлуатацію, проте не гарантують повного захисту.

1.3.2 Використання після звільнення (Use-After-Free)

Use-After-Free представляє критичну вразливість, що виникає коли програма продовжує використовувати область пам'яті після її звільнення. Цей тип вразливості особливо поширений у складних програмах з активним динамічним управлінням об'єктами, таких як браузери, JavaScript-двигуни та багатопотокові додатки. Механізм експлуатації UAF базується на тому, що після звільнення пам'яті її вміст може залишатися незмінним протягом певного часу, або ця область може бути повторно виділена для зберігання інших даних. Атакуючий може скористатися цим для контролю над вмістом пам'яті в момент її повторного використання.

```
struct user_data {
    char name[32];
    void (*callback)(void);
};

struct user_data* user = malloc(sizeof(struct user_data));
strcpy(user->name, "example");
user->callback = safe_function;

free(user);
// ... other code ...
user->callback(); // Use after free
```

Особливо небезпечними є ситуації, коли звільнення та повторне викори-

стання відбуваються в межах короткого проміжку часу, що дозволяє атакуючому з високою ймовірністю контролювати вміст звільненої області. Сучасні техніки експлуатації використовують heap spraying для збільшення шансів успішного контролю над повторно виділеною пам'яттю.

1.3.3 Пошкодження стеку (Stack-based Vulnerabilities)

Stack-based вразливості залишаються одними з найкласичніших та найвідоміших загроз, незважаючи на десятиліття розвитку захисних механізмів. Ці вразливості виникають при некоректному управлінні буферами та змінними, розташованими на стеку, що може дозволити перезаписати критичну інформацію та перехопити потік виконання програми. Стек виконання містить критично важливу інформацію для функціонування програми, включаючи локальні змінні, параметри функцій та адреси повернення. Пошкодження стеку може призвести до корупції цієї інформації, створивши можливості для перенаправлення виконання програми або витоку даних.

```
// Stack overflow example
void vulnerable_function(char* input) {
    char buffer[256];
    strcpy(buffer, input); // No size check - overflow
    printf("Processed: %s\n", buffer);
}
```

Переповнення буфера (overflow) відбувається при записі даних за верхню межу виділеного буфера, тоді як недоповнення (underflow) виникає при спробі доступу до даних за нижньою межею масиву або буфера. Обидва типи пошкоджень можуть призвести до читання або перезапису критичних структур стеку. Сучасні операційні системи впроваджують різноманітні захисні механізми, включаючи stack canaries, що розміщуються між локальними змінними та адресою повернення для виявлення корупції стеку. Address Space Layout Randomization (ASLR) ускладнює передбачення адрес для успішної експлуатації. Проте навіть з урахуванням цих захисних заходів, пошкодження стеку залишається актуальною загрозою при наявності додаткових вразливостей для обходу захисту.

1.3.4 Плутанина типів (Type Confusion)

Type Confusion являє собою складну категорію вразливостей, що виникає при некоректному приведенні типів або використанні об'єктів не за їх справжнім призначенням. Ця вразливість особливо поширена в мовах з динамічною типізацією та складних об'єктно-орієнтованих системах, де межі між різними типами даних можуть бути розмитими. Експлуатація type confusion базується на різниці в інтерпретації одної й тієї ж області пам'яті різними типами даних. Атакуючий може маніпулювати програмою таким чином, щоб вона інтерпретувала дані одного типу як дані іншого типу, що може призвести до неправильного доступу до пам'яті.

```
class BaseClass {
public:
    virtual void process() = 0;
    int base_data;
};

class DerivedA : public BaseClass {
public:
    void process() override { /* implementation */ }
    char sensitive_data[64];
};

class DerivedB : public BaseClass {
public:
    void process() override { /* implementation */ }
    void (*function_ptr)();
};

// vulnerable code
BaseClass* obj = create_object(user_input);
DerivedB* confused = static_cast<DerivedB*>(obj); // incorrect cast
confused->function_ptr();
```

Особливо небезпечними є випадки type confusion у системах з підтримкою віртуальних функцій, де некоректне приведення типів може призвести

до виклику неправильних методів або пошкодження таблиць віртуальних функцій.

1.3.5 Переповнення цілих чисел (Integer Overflow/Underflow)

Переповнення цілих чисел часто служить каталізатором для інших типів вразливостей, хоча само по собі може не призводити до безпосереднього компрометації безпеки. Ця вразливість виникає коли результат арифметичної операції перевищує максимальне значення для типу даних або стає меншим за мінімальне значення. Небезпека integer overflow полягає в тому, що некоректні результати обчислень часто використовуються для визначення розмірів буферів, обчислення індексів масивів або контролю циклів. Це може призвести до виділення недостатньої кількості пам'яті або неправильного доступу до даних.

```
size_t calculate_buffer_size(unsigned int count, unsigned int item_size) {
    return count * item_size; // overflow risk
}

void process_items(unsigned int count, unsigned int item_size, char* data) {
    size_t total_size = calculate_buffer_size(count, item_size);
    char* buffer = malloc(total_size); // less memory than needed

    for (unsigned int i = 0; i < count; i++) {
        memcpy(buffer + i * item_size, data + i * item_size, item_size);
    }
}
```

Особливу увагу заслуговують signed integer overflow, що може призводити до неочікуваної поведінки через особливості представлення від'ємних чисел у двійковому доповненні, та wrap-around ефекти в unsigned арифметиці.

1.3.6 Подвійне звільнення пам'яті (Double Free)

Double Free являє собою вразливість, що виникає при спробі звільнити одну й ту ж ділянку пам'яті двічі. Ця помилка може призвести до пошкодження внутрішніх структур даних менеджера пам'яті та створити можливості для експлуатації через контроль над вмістом пам'яті. Heap-менеджери підтримують складні метадані для відстеження стану виділених блоків пам'яті. Подвійне звільнення може пошкодити ці структури, призводячи до непередбачуваної поведінки при подальших операціях з пам'яттю. Атакуючі можуть використовувати це для маніпулювання процесом виділення пам'яті.

```
void cleanup_resources(struct resource* res) {
    if (res->data) {
        free(res->data);
        // no nullification
    }
    if (res->backup_data) {
        free(res->backup_data);
    }
}

void vulnerable_cleanup(struct resource* res) {
    cleanup_resources(res);
    // ... loggic error ...
    if (error_condition) {
        cleanup_resources(res); // freeing already freed memory
    }
}
```

Сучасні захисні механізми, такі як guard malloc та heap integrity checking, можуть виявляти спроби подвійного звільнення, проте не всі системи мають такі захисні заходи активованими за замовчуванням.

1.3.7 Вразливості форматних рядків (Format String)

Format String vulnerabilities виникають коли функції форматованого введення/виведення отримують недовірені дані як формат-рядок. Ця вразливість може дозволити атакуючому читати дані зі стеку, переглядати довільні області пам'яті або навіть записувати значення в пам'ять програми. Функції сімейства printf використовують спеціальні специфікатори формату для інтерпретації аргументів. Коли користувацькі дані передаються як формат-рядок, атакуючий може використовувати ці специфікатори для маніпулювання поведінкою функції.

```
void log_message(char* user_input) {
    printf(user_input); // format string vulnerability
}
```

Специфікатор %n представляє особливу небезпеку, оскільки дозволяє записувати кількість вже виведених символів в змінну за адресою, що передається як аргумент. Атакуючи можуть використовувати цю особливість для запису довільних значень у пам'ять програми.

1.3.8 Помилки типу "на одиницю"(Off-by-One Error)

Off-by-One errors представляють категорію помилок, пов'язаних з неправильним обчисленням індексів або розмірів буферів з різницею в одиницю. Хоча вплив таких помилок може здаватися обмеженим порівняно з повним переповненням буфера, вони можуть бути надзвичайно небезпечними при перезаписі критичних метаданих. Такі помилки часто виникають через плутанину між інклюзивними та ексклюзивними межами масивів, неправильне обчислення довжини рядків або помилки в умовах циклів. Навіть перезапис одного байта може призвести до серйозних наслідків, якщо цей байт містить критично важливу інформацію.

```
void copy_string(char* dest, char* src, size_t dest_size) {
    size_t i;
    for (i = 0; i < dest_size && src[i] != '\0'; i++) {
```

```
        dest[i] = src[i];
    }
    dest[i] = '\0'; // off by one error if src is longer than dest_size
}
```

Особливо небезпечними є off-by-one помилки в контексті stack frames, де перезапис одного байта може призвести до модифікації збережених регістрів або адрес повернення функцій. У heap-контексті такі помилки можуть пошкодити заголовки блоків пам'яті, що використовуються менеджером пам'яті.

1.3.9 Вразливості нульового дня

Вразливості нульового дня представляють найвищий рівень загрози в сучасному ландшафті кібербезпеки. Ці недоліки в програмному забезпеченні характеризуються тим, що вони невідомі розробникам та широкій спільноті безпеки на момент їх виявлення та потенційної експлуатації зловмисниками. Особлива небезпека zero-day вразливостей полягає у відсутності готових засобів захисту або патчів на момент їх використання в атаках. Це створює вікно вразливості, протягом якого системи залишаються беззахисними перед цільовими атаками. Тривалість цього вікна може становити від декількох днів до місяців, залежно від складності виявлення та швидкості реагування розробників. Економічна цінність zero-day вразливостей на чорному ринку може сягати мільйонів доларів, особливо для вразливостей у широко використовуваному програмному забезпеченні або критичній інфраструктурі. Це створює потужні економічні стимули для пошуку та накопичення таких вразливостей зловмисниками та державними акторами. Виявлення zero-day вразливостей традиційними методами є надзвичайно складним завданням, оскільки вони за визначенням не відповідають відомим патернам та сигнатурам. Це підкреслює важливість розвитку нових підходів до аналізу безпеки, здатних виявляти раніше невідомі типи проблем.

1.4 Сучасні методи експлуатації вразливостей

Еволюція захисних механізмів операційних систем та компіляторів призвела до відповідного розвитку методів експлуатації вразливостей. Сучасні техніки атак характеризуються високою складністю, використанням комбінацій різних підходів та здатністю обходити багаторівневі системи захисту. Ці методи вимагають глибокого розуміння архітектури системи та творчого підходу до використання особливостей програмного забезпечення.

1.4.1 Return-Oriented Programming (ROP)

Return-Oriented Programming представляє революційний підхід до експлуатації вразливостей, що дозволяє обійти захист від виконання даних (Data Execution Prevention). Замість впровадження власного шкідливого коду в пам'ять програми, ROP використовує фрагменти існуючого коду, які вже присутні в виконуваних сегментах пам'яті. Основою ROP є концепція "гаджетів коротких послідовностей інструкцій, які закінчуються інструкцією повернення (RET). Атакуючий складає ланцюжок адрес цих гаджетів на стеку, створюючи послідовність виконання, що реалізує потрібну функціональність без порушення політики DEP. Складення ROP-ланцюжка вимагає детального аналізу виконуваного коду для виявлення підходящих гаджетів. Автоматизовані інструменти, такі як ROPgadget та ropper, спрощують процес пошуку та класифікації гаджетів у бінарних файлах. Ефективність ROP також залежить від архітектури процесора та наявності відповідних інструкцій у коді програми. Захист від ROP-атак включає такі механізми, як Control Flow Integrity (CFI), який перевіряє легітимність переходів у коді, та Intel CET (Control Flow Enforcement Technology), що забезпечує апаратну підтримку для виявлення аномальних потоків виконання.

1.4.2 Heap Spraying

Heap Spraying являє собою техніку підготовки динамічної пам'яті для успішної експлуатації вразливостей. Основна ідея полягає в розміщенні великої кількості копій шкідливого коду або контрольованих даних у heap для підвищення ймовірності попадання покажчика на підготовлену область пам'яті. Ця техніка особливо ефективна в середовищах з активним виділенням пам'яті, таких як веб-браузери, де JavaScript може створювати велику кількість об'єктів. Атакуючий може виділити сотні мегабайт пам'яті, заповнених повторюваним патерном, що містить шкідливий код. Ефективність heap spraying залежить від розуміння особливостей роботи менеджера пам'яті цільової системи. Різні heap-менеджери мають різні стратегії виділення пам'яті, що впливає на розташування об'єктів у пам'яті.

1.4.3 Heap Feng Shui

Heap Feng Shui представляє більш точний підхід до маніпулювання heap, що дозволяє атакуючому контролювати точне розташування об'єктів у пам'яті [10]. На відміну від heap spraying, який покладається на ймовірність, feng shui використовує детальне розуміння алгоритмів виділення пам'яті для створення оптимальних умов експлуатації. Ця техніка передбачає послідовність операцій виділення та звільнення пам'яті, спрямованих на досягнення потрібного стану heap. Атакуючий може створити "дірки"у пам'яті певного розміру, які потім будуть заповнені об'єктами з контрольованим вмістом. Успішне застосування heap feng shui вимагає глибокого розуміння внутрішніх алгоритмів менеджера пам'яті, включаючи стратегії coalescing, binning та garbage collection у випадку managed languages.

1.4.4 JIT Spraying

JIT Spraying використовує особливості Just-In-Time компіляції для впровадження шкідливого коду через механізми оптимізації самої програми [11]. Ця техніка особливо ефективна проти веб-браузерів та інших додатків, що використовують динамічну компіляцію скриптів. Основна ідея полягає у створенні коду на високорівневій мові (наприклад, JavaScript), який при JIT-компіляції генерує машинні інструкції, що можуть бути інтерпретовані як шкідливий код. Атакуючий може використовувати особливості кодування констант та арифметичних операцій для генерації потрібних опкодів. JIT-компілятори використовують різні оптимізації, включаючи inline caching та type specialization, які можуть бути використані для точного контролю над генерованим машинним кодом. Розуміння цих механізмів дозволяє атакуючому передбачити структуру компільованого коду.

1.4.5 Обхід ASLR

Address Space Layout Randomization ускладнює експлуатацію за рахунок рандомізації адрес завантаження різних сегментів програми. Проте існують різні техніки для обходу або зменшення ефективності ASLR. Information leaks дозволяють атакуючому отримати інформацію про реальні адреси в пам'яті програми. Навіть витік одної адреси може бути достатнім для обчислення базових адрес інших сегментів через відомі офсети. Partial ASLR bypass використовує той факт, що деякі біти адрес залишаються незмінними через обмеження вирівнювання пам'яті. Це дозволяє проводити brute force атаки з обмеженим простором пошуку.

1.4.6 Обхід Stack Canaries

Stack canaries являють собою випадкові значення, розміщені між локальними змінними та адресою повернення для виявлення переповнення сте-

ку. Проте існують методи обходу цього захисного механізму. Канарки можуть бути витечені через format string вразливості або arbitrary read примітиви. Знаючи значення канарки, атакуючий може відновити її при переповненні буфера. Thread-local canaries та различні реалізації у різних компіляторах створюють додаткові можливості для обходу через розуміння специфіки їх генерації та зберігання.

1.5 Застосування штучного інтелекту в задачах кібербезпеки

З розвитком технологій машинного навчання та, зокрема, великих мовних моделей, з'явилася можливість значно автоматизувати процес аналізу програмного коду [12], підвищивши ефективність роботи фахівців з кібербезпеки. Інтеграція методів штучного інтелекту в галузь кібербезпеки відкриває нові можливості для автоматизації та підвищення ефективності традиційних підходів до аналізу безпеки. Сучасні AI-системи демонструють здатність до аналізу великих обсягів даних, виявлення складних патернів та адаптації до нових типів загроз. Відповідно в суміжних областях кібербезпеки вже успішно застосовуються AI-підходи для вирішення різних задач [13, 14]. Системи виявлення вторгнень (IDS) використовують машинне навчання для ідентифікації аномальної поведінки мережі та потенційних атак. Антивірусні рішення застосовують нейронні мережі для виявлення нових зразків шкідливого програмного забезпечення на основі їх поведінки та характери- стик. Цей досвід може бути адаптований для задач зворотньої розробки та аналізу бінарних файлів.

1.5.1 Великі мовні моделі для аналізу коду

Великі мовні моделі, такі як Claude від Anthropic [15], GPT від OpenAI та інші [16, 17], демонструють значний потенціал для аналізу та розуміння програмного коду. Ці моделі навчені на величезних масивах текстових даних, включаючи програмний код на різних мовах програмування, технічну документацію та обговорення програмістів. Завдяки цьому подібні моделі

представляють революційний підхід і до аналізу програмного коду [18]. Сучасні моделі здатні розуміти не лише синтаксис, але й семантику програм, що відкриває не тільки чудові можливості до аналізу коду, а й навіть його декомпіляції за допомогою LLM [19–21].

Відповідною перевагою LLM є їх здатність працювати з декомпільованим кодом, який часто містить нестандартні конструкції та втрачену семантичну інформацію. Моделі здатні відновлювати логіку програм та ідентифікувати їх призначення навіть за відсутності коментарів та змістовних назв змінних. Здатність LLM до контекстного аналізу дозволяє розглядати вразливості не ізольовано, а в контексті загальної архітектури програми. Це підвищує точність виявлення реальних проблем безпеки та зменшує кількість хибних спрацювань порівняно з традиційними статичними аналізаторами. Можливості генерації природномовних описів коду значно спрощують процес аналізу для спеціалістів різного рівня підготовки. LLM здатні створювати детальну документацію функціональності програм, що особливо цінно при роботі з недокументованим або застарілим кодом.

Серед інших ключових переваг LLM що відкривають можливості для подібних дослдіжень можна виділити:

- Можливість використання сторонніх інструментів та API для інтеграції з існуючими системами аналізу та розробки, що дозволяє створювати гібридні рішення, які поєднують переваги традиційних методів та інноваційних можливостей штучного інтелекту.

- Можлиівсть донавчання та адаптації моделей на основі нових даних, що дозволяє постійно покращувати їх ефективність у виявленні вразливостей.

1.5.2 Переваги підходів з використанням LLM

Використання великих мовних моделей у задачах аналізу безпеки програмного забезпечення демонструє суттєві переваги порівняно з традиційними методами [22]. Здатність до швидкого аналізу великих обсягів коду

дозволяє значно скоротити час, необхідний для комплексного дослідження програмних комплексів. Адаптивність LLM до різних стилів програмування та технік обфускації виявляється особливо цінною при роботі з сучасним програмним забезпеченням. На відміну від статичних правил традиційних аналізаторів, мовні моделі здатні розпізнавати патерни навіть у сильно модифікованому або приховано коді, а також прослідковувати потоки даних та логіку виконання на рівні, який часто недоступний для класичних інструментів [23, 24]. Можливість виявлення складних взаємозв'язків між різними частинами програми дозволяє LLM ідентифікувати вразливості, що виникають через взаємодію компонентів, які можуть бути розташовані в різних файлах або модулях. Такий системний підхід часто недоступний традиційним інструментам аналізу. Здатність до навчання та адаптації означає, що LLM можуть покращувати свою ефективність на основі нових даних та зворотного зв'язку від аналітиків. Це створює можливості для розвитку спеціалізованих моделей, оптимізованих для конкретних типів програмного забезпечення або специфічних завдань безпеки. Інтеграція з існуючими інструментами розробки та аналізу дозволяє створювати гібридні системи, що поєднують переваги традиційних методів з інноваційними можливостями штучного інтелекту. Такий підхід забезпечує плавний перехід до нових технологій без повного відмови від перевірених рішень.

Висновки до розділу 1

У першому розділі було проведено комплексний аналіз сучасного стану проблематики виявлення бінарних вразливостей та перспектив застосування штучного інтелекту для вирішення цих завдань. Розглянуто традиційні підходи до аналізу безпеки програмного забезпечення, їх обмеження та можливості інтеграції з сучасними технологіями машинного навчання. Дослідження традиційних методів пошуку вразливостей виявило критичні недоліки існуючих підходів. Статичний аналіз демонструє слабкість при роботі з обфускованим кодом та генерує значну кількість хибних спрацювань, тоді як динамічний аналіз не здатен охопити всі можливі шляхи виконання програми. Фазинг, незважаючи на свою ефективність, залишається ресурсоємним процесом, що вимагає спеціалізованих знань для налаштування та інтерпретації результатів. Ці обмеження зумовлюють необхідність пошуку альтернативних підходів для підвищення ефективності виявлення вразливостей. Систематизація класичних типів вразливостей пам'яті виявила основні категорії загроз, кожна з яких має специфічні механізми виникнення та експлуатації. Heap-based buffer overflow та use-after-free залишаються найпоширенішими типами вразливостей у сучасному програмному забезпеченні, особливо у браузерах та системах з активним динамічним управлінням пам'яттю. Stack-based overflow, незважаючи на десятиліття розвитку захисних механізмів, продовжує становити значну загрозу при наявності додаткових вразливостей для обходу захисту. Паралельно з виявленням недоліків традиційних методів, дослідження можливостей великих мовних моделей у задачах аналізу безпеки показало значний потенціал для автоматизації процесів виявлення вразливостей. LLM демонструють здатність до розуміння семантики програмного коду, ідентифікації складних патернів та адаптації до різних стилів програмування. Особливо цінною є можливість роботи з декомпільованим кодом та генерації зрозумілих описів функціональності програм. Виявлені обмеження традиційних підходів та революційні можливості LLM створюють передумови для розробки інтегрованого рішення, що поєднує переваги класичних інструментів зворотної інженерії з інтелектуальними

можливостями сучасних мовних моделей. Це обґрунтовує необхідність адаптації та інтеграції LLM з існуючими декомпіляторами для створення більш ефективної системи аналізу бінарних вразливостей.

2 МОДЕЛЬ СИСТЕМИ ПОШУКУ ВРАЗЛИВОСТЕЙ ПОШКОДЖЕННЯ ПАМ'ЯТІ НА ОСНОВІ LLM

Сучасні великі мовні моделі демонструють революційну здатність до використання зовнішніх інструментів через концепцію "tool use"або "function calling що принципово змінює парадигму взаємодії штучного інтелекту з програмними системами. Ця можливість дозволяє LLM не просто генерувати текст, але й активно взаємодіяти з реальними програмними інструментами, виконувати складні багатокрокові операції та приймати рішення на основі отриманих результатів. Моделі навчені розуміти специфікацію API, аналізувати документацію інструментів та формувати правильні запити з відповідними параметрами, що відкриває можливості для створення автономних агентів, здатних виконувати складні технічні завдання. У контексті бінарного аналізу ця здатність є особливо цінною, оскільки дозволяє LLM безпосередньо взаємодіяти з декомпіляторами, дебагерами та іншими спеціалізованими інструментами, перетворюючи їх з пасивних помічників у активних учасників процесу аналізу. Така інтеграція створює синергетичний ефект, де когнітивні здібності LLM поєднуються з технічними можливостями профільних інструментів, що дозволяє автоматизувати складні експертні завдання, які раніше вимагали значного досвіду та ручної роботи фахівців з кібербезпеки.

2.1 Загальна архітектура системи

Розроблена система пошуку вразливостей пошкодження пам'яті представляє собою інтегроване рішення, що поєднує потужності сучасних декомпіляторів з можливостями великих мовних моделей через стандартизований протокол взаємодії. Архітектура системи побудована за принципом модульності, що забезпечує гнучкість у виборі компонентів та можливість масшта-

бування. Клієнт-серверна архітектура дозволяє розділити обов'язки між декомпілятором, який виконує низькорівневий аналіз бінарного коду, та мовною моделлю, яка здійснює високорівневий аналіз та інтерпретацію результатів. Спрощена діаграма процесу роботи системи представлена на рисунку 2.1.

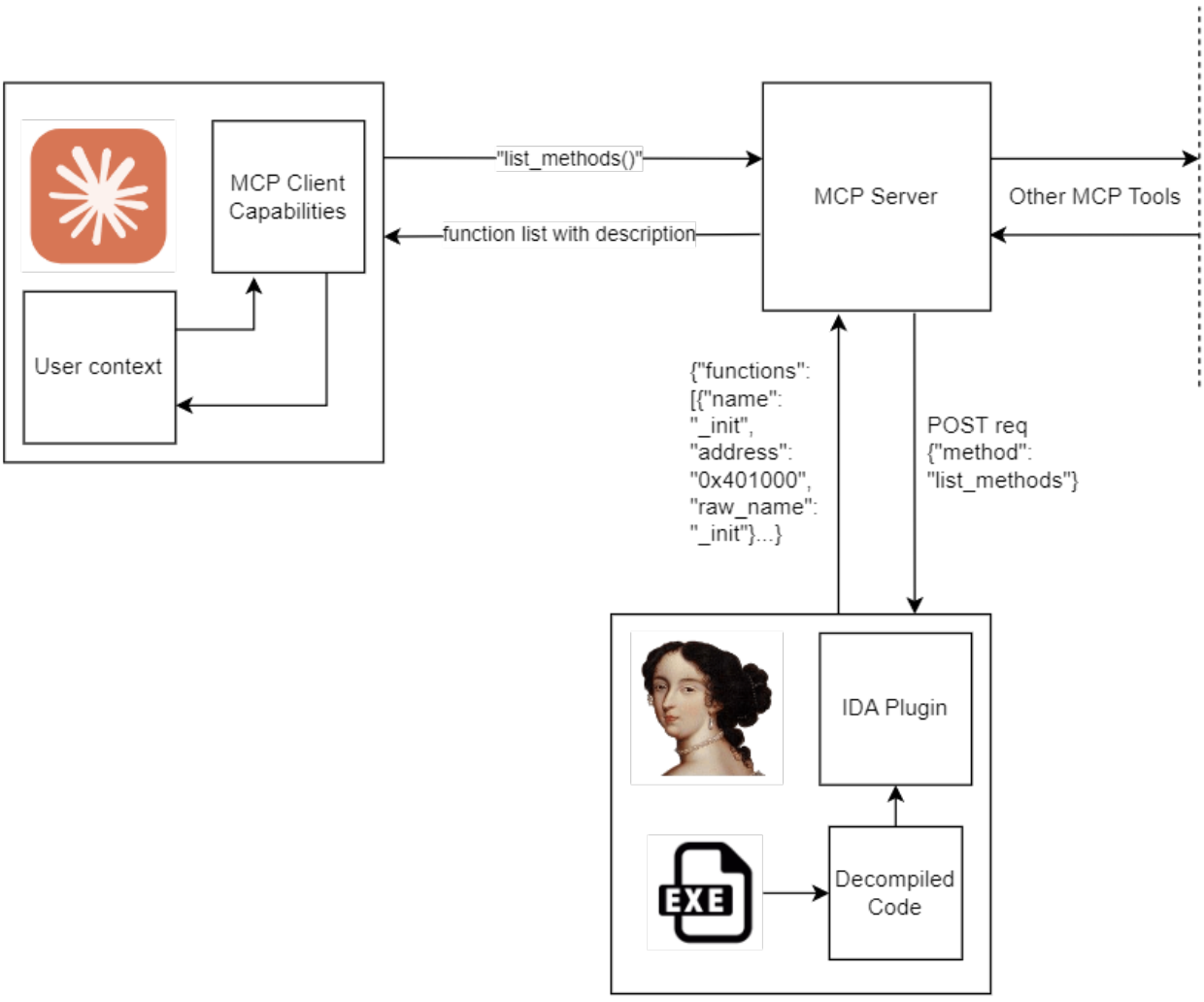


Рисунок 2.1 — Візуалізація процесу роботи LLM з декомпілятором

Основними компонентами системи є декомпілятор IDA Pro [25] з інтегрованим MCP сервером, MCP клієнт Claude, який забезпечує інтерфейс користувача, та відповідна велика мовна модель Claude Sonnet 4.0. Така архітектура дозволяє розділити функціональність між спеціалізованими компонентами, де декомпілятор відповідає за низькорівневий аналіз бінарного коду, MCP протокол забезпечує стандартизовану комунікацію, а LLM вико-

нує високорівневий аналіз та інтерпретацію результатів. Система функціонує за принципом клієнт-серверної взаємодії, де MCP сервер, інтегрований в IDA Pro, надає API для доступу до функцій декомпілятора, а MCP клієнт передає запити від користувача та LLM до сервера. Така архітектура забезпечує ізоляцію між компонентами та дозволяє використовувати різні декомпілятори без зміни логіки роботи з LLM. При створенні користувачем певного контексту, наприклад, прямої вказівки на виклик певної функції під'єднаного інструменту, LLM зможе зрозуміти цей контекст та використати його для формування запиту до MCP сервера з відповідними параметрами. Сервер також надає моделі розуміння типів функцій які вона може викликати, що вони роблять, який формат вхідних та вихідних даних. Ключовою перевагою запропонованої архітектури є можливість збереження контексту аналізу протягом всієї сесії роботи. LLM може зберігати інформацію про попередні кроки аналізу, що дозволяє проводити поглиблене дослідження вразливостей з урахуванням всього контексту програми. Розглянемо окремі компоненти архітектури детальніше.

2.2 Компонент Model-Context-Protocol (MCP)

Model-Context-Protocol [26] представляє собою стандартизований протокол для взаємодії великих мовних моделей з зовнішніми інструментами та системами. Протокол розроблений компанією Anthropic для забезпечення безпечної та ефективної інтеграції AI-агентів з різноманітними програмними інструментами. MCP базується на JSON-RPC 2.0 протоколі та забезпечує двостороннє асинхронне спілкування між клієнтом та сервером. Сучасні моделі що навчені користуватись інструментами вже певний час використовують інструменти MCP загального призначення [27]. Архітектура MCP включає три основні типи повідомлень: запити (requests), відповіді (responses) та сповіщення (notifications). Запити використовуються для виклику функцій на сервері, відповіді містять результати виконання, а сповіщення дозволяють серверу інформувати клієнта про зміни стану або події. Загально потік виконання запиту виглядає наступним чином 2.2.

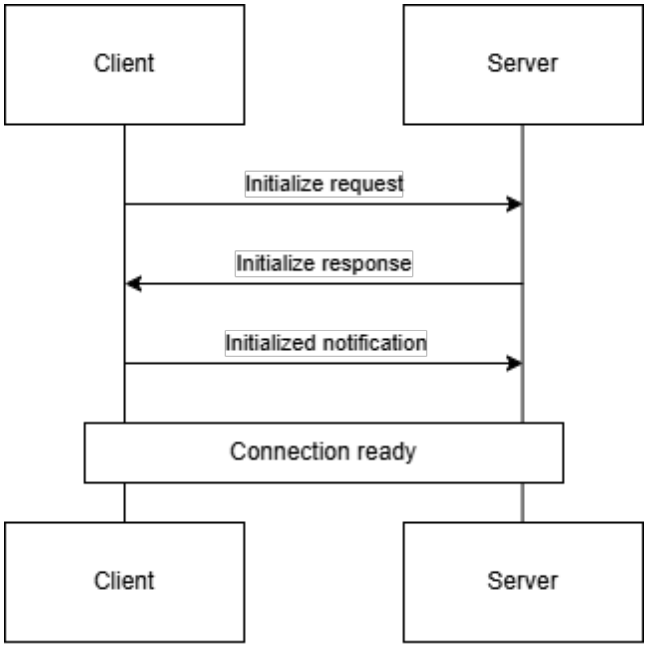


Рисунок 2.2 — Потік виконання запиту в MCP

Протокол підтримує концепцію ресурсів (resources), що дозволяє серверу надавати доступ до різних типів даних через уніфікований інтерфейс. У контексті аналізу бінарного коду ресурсами можуть бути функції, глобальні змінні, рядки, структури даних та інші елементи програми.

MCP також включає механізм discovery, який дозволяє клієнту дізнатися про доступні на сервері функції та ресурси. Це забезпечує динамічну адаптацію клієнта до можливостей конкретного сервера та спрощує інтеграцію нових функцій.

2.3 Компонент MCP сервер

Ця компонента системи працює за межами клієнта і реалізована як окремий сервер з двума модулямии: MCP Bridge та IDA Pro плагін. MCP Bridge забезпечує стандартизований інтерфейс для взаємодії з MCP клієнтом, а IDA Pro плагін реалізує серверну частину, що надає RESTful JSON-RPC API для доступу до функціональності декомпілятора.

2.3.1 Компонент MCP Міст

MCP Bridge функціонує як критично важливий проміжний компонент між клієнтськими додатками та IDA Pro плагіном, забезпечуючи стандартизований інтерфейс взаємодії через Model Context Protocol. Використання такого мосту необхідне для незалежного функціонування плагіна в середині декомпілятора, адже при запуску LLM клієнта, відповідний MCP сервіс вже повинен працювати, але це не гаранує автоматичний запуск плагіну IDA. Відповідно MCP Bridge працює завжди, не очікуючи завантаження плагіну. Це дає можливість до гнучкого управління процесом аналізу застосунків. Міст реалізований як окремий Python сервер, що автоматично генерує та експонує API функції на основі аналізу коду IDA плагіна, створюючи seamless інтеграцію між різнорідними системами.

Функція `make_jsonrpc_request` реалізує надійний механізм проксування запитів до IDA плагіна через HTTP протокол. Функція формує стандартизовані JSON-RPC 2.0 запити з унікальними ідентифікаторами та передає їх до HTTP сервера IDA плагіна. Кожен запит включає назву методу, список параметрів та ідентифікатор для асоціації відповіді з оригінальним запитом. JSON-RPC помилки, що генеруються IDA плагіном, автоматично перетворюються на Python винятки з детальними повідомленнями, включаючи код помилки та додаткові діагностичні дані. Це дозволяє LLM клієнтам отримувати значущу інформацію про проблеми, що виникають під час виконання операцій.

```
def make_jsonrpc_request(method: str, *params):
    """Make a JSON-RPC request to the IDA plugin"""
    global jsonrpc_request_id, ida_host, ida_port
    conn = http.client.HTTPConnection(ida_host, ida_port)
    request = {
        "jsonrpc": "2.0",
        "method": method,
        "params": list(params),
        "id": jsonrpc_request_id,
    }
    jsonrpc_request_id += 1
```

```
try:
    conn.request("POST", "/mcp", json.dumps(request), {
        "Content-Type": "application/json"
    })
    response = conn.getresponse()
    data = json.loads(response.read().decode())

    if "error" in data:
        error = data["error"]
        code = error["code"]
        message = error["message"]
        pretty = f"JSON-RPC error {code}: {message}"
        if "data" in error:
            pretty += "\n" + error["data"]
        raise Exception(pretty)

    result = data["result"]
    # NOTE: LLMs do not respond well to empty responses
    if result is None:
        result = "success"
    return result
except Exception:
    raise
finally:
    conn.close()
```

Міст використовує FastMCP фреймворк для реалізації стандартного Model Context Protocol інтерфейсу. FastMCP забезпечує високорівневі абстракції для створення MCP серверів з мінімальним кодом, автоматично обробляючи деталі протоколу та забезпечуючи сумісність з різними MCP клієнтами. Декоратор `@mcp.tool()` автоматично реєструє згенеровані функції як доступні інструменти для LLM. Система підтримує різні транспортні протоколи, включаючи stdio для безпосередньої інтеграції з терміналом та Server-Sent Events (SSE) для веб-базованих клієнтів. Підтримка SSE особливо корисна для розробки та відладки, оскільки дозволяє використовувати MCP Inspector для моніторингу взаємодії між компонентами у режимі ре-

ального часу. Конфігурація логування оптимізована для роботи з різними клієнтами. Для Cline та інших редакторів коду встановлюється рівень логування ERROR для зменшення шуму у виводі, тоді як для режиму відладки доступний більш детальний рівень `INFO`.

2.3.2 Система безпеки та категоризація функцій

MCP Bridge реалізує дворівневу систему безпеки для контролю доступу до потенційно небезпечних операцій. Функції, позначені декоратором @unsafe у IDA плагіні, автоматично ідентифікуються та вимагають дозволу через параметр командного рядка `-unsafe`. Це запобігає випадковому виконанню операцій, що можуть змінити стан системи або призвести до небажаних побічних ефектів у випадку роботи на хості. Безпечні функції автоматично додаються до списків `autoApprove` та `alwaysAllow` у конфігурації MCP клієнтів, що дозволяє LLM використовувати їх без додаткових підтверджень користувача. Це значно покращує користувацький досвід при роботі з рутинними операціями читання та аналізу, зберігаючи при цьому контроль над критичними операціями модифікації.

Наведемо декілька прикладів категоризації функцій:

Безпечні функції (read-only операції):

- `get_metadata` – отримання метаданих про поточну базу даних
- `list_functions` – список всіх функцій у програмі

Ці функції лише читають інформацію з бази даних IDA Pro, не змінюючи її стан. Автоматично схвалюються в MCP клієнтах та не вимагають додаткових дозволів користувача.

Функції модифікації (write операції):

- `set_comment` – встановлення коментарів до адрес
- `rename_function` – перейменування функцій

Ці функції змінюють базу даних IDA Pro, позначені декоратором `@idawrite` та виконуються в синхронізованому режимі для запобігання корупції даних.

Небезпечні функції (unsafe операції):

- `dbg_start_process` – запуск процесу під відладчиком
- `dbg_run_to` – виконання до певної адреси

2.3.3 Плагін IDA

MCP сервер для IDA Pro представляє собою Python-плагін, що функціонує як HTTP-сервер та надає RESTful JSON-RPC API для доступу до функціональності декомпілятора. Сервер реалізований на основі стандартної бібліотеки Python з використанням модуля http.server та забезпечує безпечну взаємодію з IDA Pro через IDAPython API.

2.3.4 Інтеграція з IDAPython

Ключовою особливістю реалізації є глибока інтеграція з IDAPython API через систему декораторів `@idaread` та `@idawrite`. Ці декоратори забезпечують безпечну багатопоточну взаємодію з IDA Pro шляхом синхронізації всіх операцій з основним потоком програми через механізм `idaapi.execute_sync`. Декоратор `@idaread` використовується для операцій читання з бази даних IDA Pro та забезпечує консистентність даних під час читання. Він виконує функції в режимі `MFF_READ`, що гарантує отримання актуальної інформації без конфліктів з іншими операціями. Декоратор `@idawrite` призначений для операцій модифікації бази даних та виконується в режимі `MFF_WRITE`, забезпечуючи атомарність змін та запобігаючи корупції даних.

2.3.5 Демонстрація розширення функціональності MCP сервера

Надана архітектура дозволяє легко розширюбвати функціональність MCP сервера шляхом додавання нових функцій з відповідними декораторами. Відповідно до можливостей IDA-python користувач може створювати нові методи, які будуть виконувати поставлені задачі, і вони будуть автоматично реєструватись для MCP клієнта.

Розглянемо детальний аналіз однієї з доданих мною функцій – `get_byte_array` як типового прикладу реалізації MCP API:

```
@jsonrpc
@idaread
def get_byte_array(addr: Annotated[int, "Address to start from"],
                   size: Annotated[int, "Size of bytes to get"]) -> str:
    """Gets a byte array of the given size from the given address"""
    data = ida_bytes.get_bytes(addr, size)
    if data is None:
        raise ValueError(f"Could not read {size} bytes from address {hex(addr)}")
    return str(data)
```

В процесі зворотньої розробки та аналізу компільованого коду, модель може створювати та виконувати скрипти мовою JavaScript, що є вбудованим функціоналом агента Claude. Відповідно певну частину процесу можна автоматизувати повністю, надавши змогу моделі самостійно читати дані з пам'яті, інтерпретувати їх та додавати до своїх скриптів.

Написана функція демонструє кілька ключових аспектів архітектури MCP сервера. Декоратор `@jsonrpc` реєструє функцію в глобальному реєстрі RPC методів, роблячи її доступною для віддалених викликів. Декоратор `@idaread` забезпечує безпечне виконання операції читання в контексті основного потоку IDA Pro. Використання `Annotated` типів надає детальну інформацію про параметри функції, включаючи їх призначення та очікувані типи. Ця інформація використовується для автоматичної валідації параметрів та генерації документації API. Система типів забезпечує автоматичну конвертацію вхідних JSON значень до відповідних Python типів.

Ядром функції є виклик `ida_bytes.get_bytes(addr, size)`, який без-

посередньо взаємодіє з IDA Pro API для читання байтів з пам'яті аналізованого бінарного файлу. Функція обробляє випадок, коли читання неможливе (наприклад, через недійсну адресу або розмір), та генерує відповідну помилку з детальним описом проблеми. Гнучкість розширення функціональності Архітектура MCP сервера забезпечує виняткову гнучкість у додаванні нових функцій завдяки використанню Python як мови реалізації та декларативного підходу до реєстрації API методів. Розробники можуть легко розширювати функціональність сервера шляхом створення нових функцій з відповідними декораторами, не змінюючи основний код серверної інфраструктури. Використання Python дозволяє швидко прототипувати та тестувати нові функції, використовуючи всю потужність екосистеми Python, включаючи бібліотеки для обробки даних, машинного навчання та інтеграції з зовнішніми сервісами. Розробники можуть створювати складні аналітичні функції, що поєднують кілька API викликів IDA Pro для отримання комплексної інформації про бінарний код.

2.4 Компонент MCP клієнт

MCP клієнт Claude функціонує як інтерфейс між користувачем, LLM та MCP сервером декомпілятора. Клієнт забезпечує трансляцію природномовних запитів користувача у відповідні виклики функцій MCP API та представлення результатів у зрозумілому для людини форматі. При запуску клієнта відбувається завантаження MCP серверів, доступ до яких вказується в панелі розробника, що показано на зображенні 2.3. В JSON форматі вказано який саме файл запустити аби активувати локальний MCP сервер.

Після успішної реєстрації локального сервера він стане доступним в меню інструментів MCP, як можна побачити на зображенні 2.4. Це меню також дозволяє управління іншими специфічними інструментами, наприклад пошуком інформації в інтернеті, вибір стилей формування відповіді, вибір режимів роботи з LLM.

Безпосередньо в клієнті можна вмикати та вимикати функції MCP сервера, що дозволяє користувачеві контролювати доступні інструменти та функ-

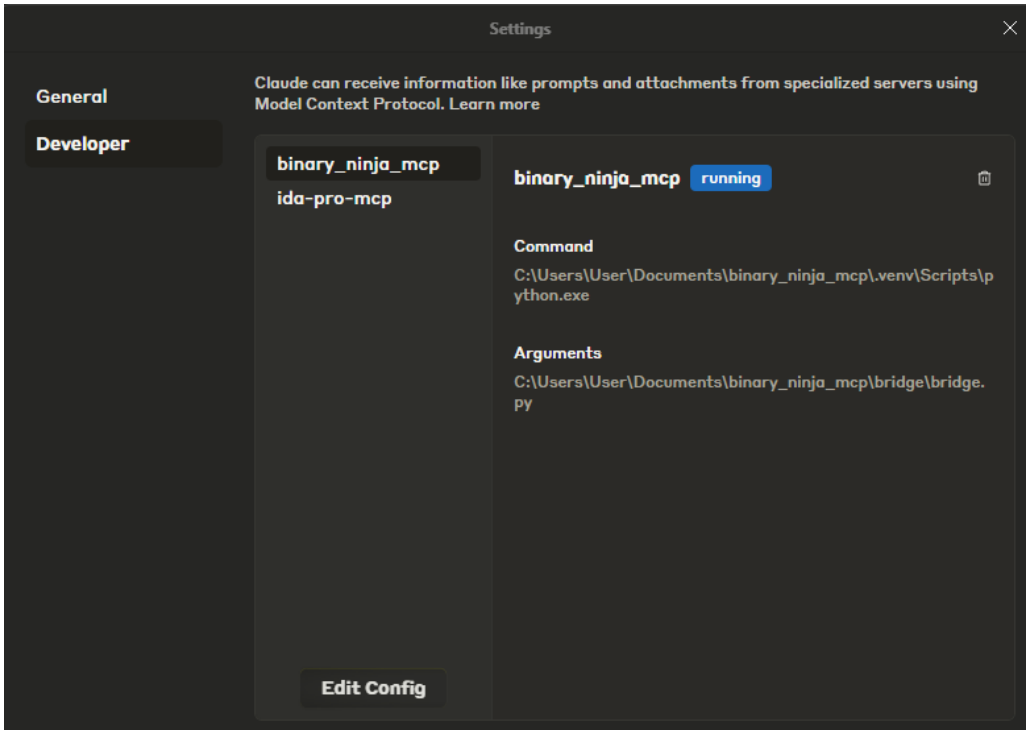


Рисунок 2.3 — додавання локального MCP сервера в панель розробника

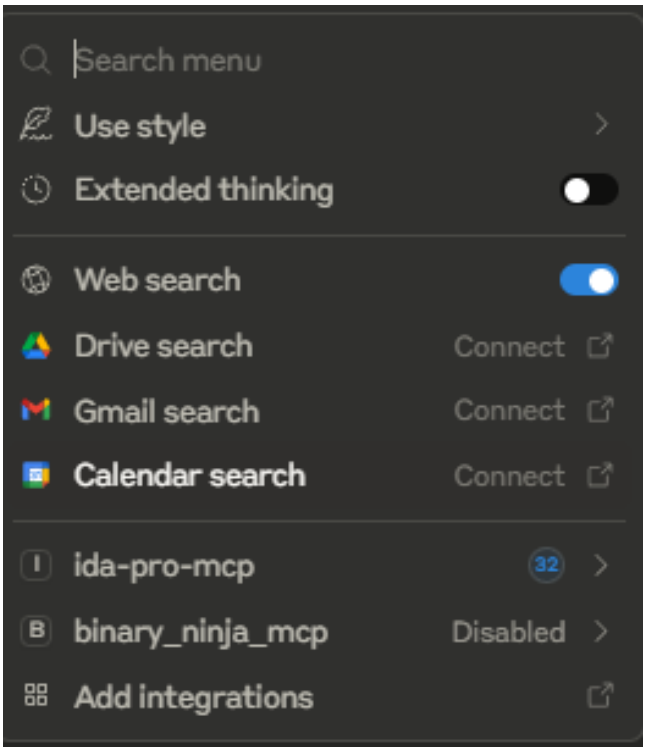


Рисунок 2.4 — Меню інструментів MCP і режимів роботи в клієнті Claude

ції, як це продемонстровано на зображенні 2.5. Після налаштування з'єднання, можемо переконатись в успішній інтграції інструментів аналізу запросивши

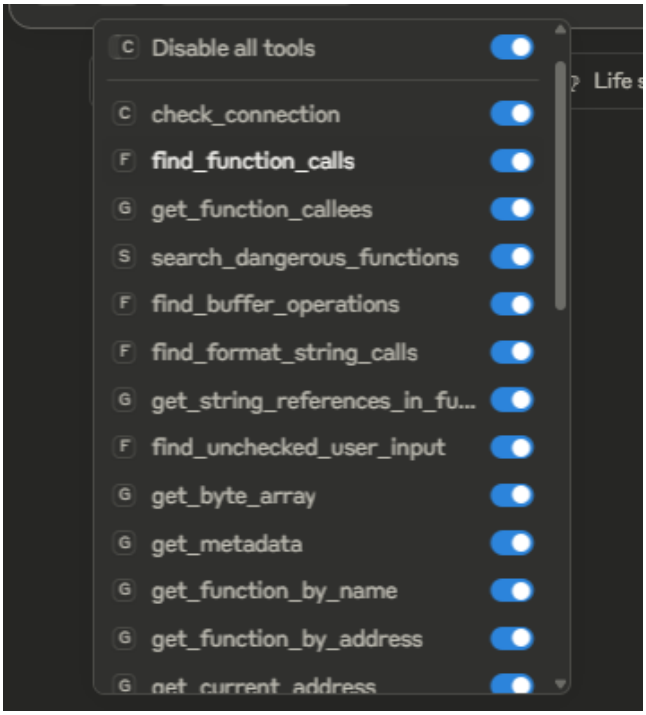


Рисунок 2.5 — Робота з окремими функціями MCP сервера

у ШІ агента Claude Sonnet 4.0 виклик базових функцій, приклад зображено на зображенні 2.6.

Бачимо що модель успішно визначила функції необхідні для задоволення запиту, і викликала їх, про що свідчить історія виконання запитів. В результаті модель опрацювала отримані дані, та сформувала базову відомість про бінарний файл, що аналізується.

2.4.1 Типи LLM клієнтів

Сучасний ландшафт LLM клієнтів включає декілька основних категорій, що відрізняються архітектурою, можливостями та призначенням. Web-базовані клієнти, такі як ChatGPT або Claude Web, забезпечують доступ до моделей через браузерний інтерфейс. Ці рішення характеризуються простотою використання та постійним оновленням можливостей, але мають обмеження у плані інтеграції з локальними інструментами та можуть викликати занепокоєння щодо приватності даних. Desktop-додатки, такі як Claude Desktop, представляють більш потужний клас рішень. Вони поєднують зруч-

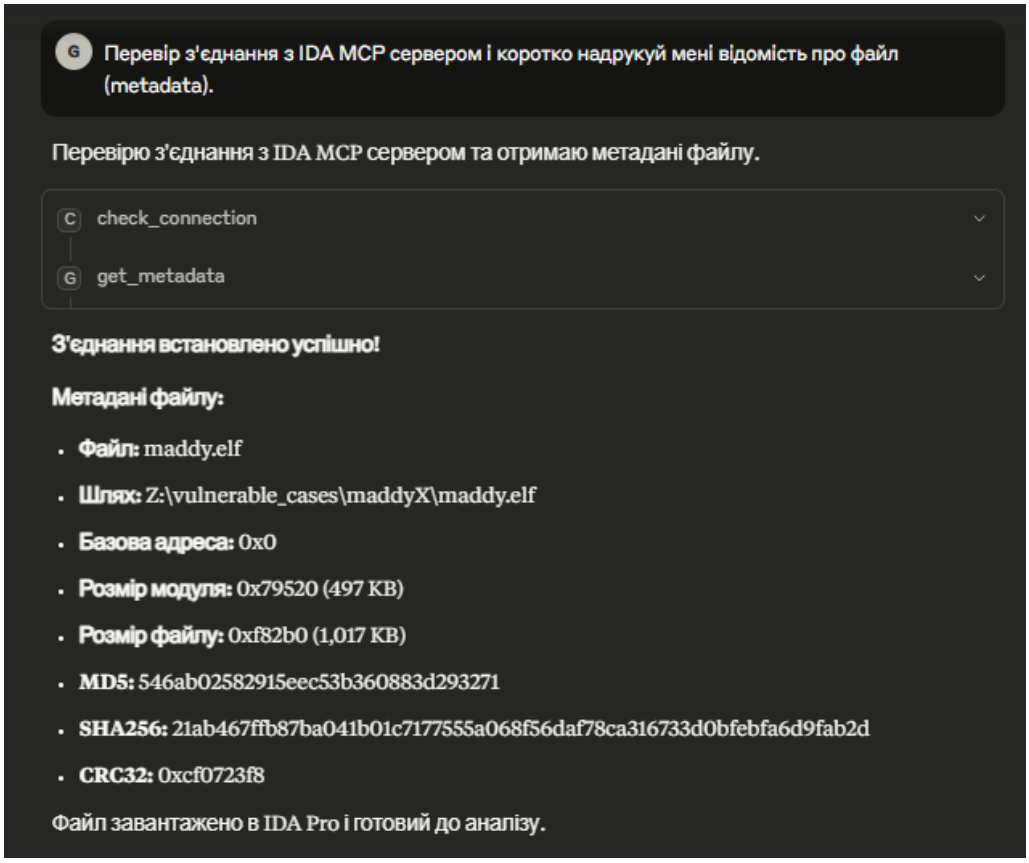


Рисунок 2.6 — Демонстрація використання виклику функцій в клієнті Claude

ність локального інтерфейсу з можливістю глибокої інтеграції з операційною системою та встановленими інструментами. Claude Desktop підтримує MCP протокол, що дозволяє розширювати функціональність через підключення спеціалізованих серверів для роботи з різноманітними інструментами та сервісами. API-клієнти представляють програмні інтерфейси для інтеграції LLM можливостей безпосередньо в існуючі робочі процеси та інструменти. Цей підхід дозволяє розробникам створювати спеціалізовані рішення з точно налаштованою функціональністю. API Claude надає доступ до різних версій моделей, включаючи Claude Opus 4 з більшою кількістю параметрів для найскладніших завдань та Claude Haiku для швидких операцій. Локальні клієнти працюють з моделями, розгорнутими на власній інфраструктурі. Такі рішення, базовані на відкритих моделях типу LLaMA або Mistral, забезпечують повний контроль над даними та можливість кастомізації під специфічні потреби. Однак вони вимагають значних обчислювальних ресурсів та

експертизи для налаштування та підтримки.

Сучасні великі мовні моделі оцінюються за кількома ключовими параметрами, що визначають їх можливості та область застосування. Кількість параметрів залишається одним з основних показників складності моделі. Claude Sonnet 4 з приблизно 200 мільярдами параметрів знаходиться в категорії великих моделей, тоді як Claude Opus 4 може містити до 500 мільярдів параметрів, забезпечуючи вищу якість аналізу складних завдань. Розмір контекстного вікна визначає кількість інформації, яку модель може обробляти одночасно. Claude 4 підтримує контекст до 200,000 токенів, що дозволяє аналізувати великі файли коду або документи без втрати контексту. Це критично важливо для задач зворотної інженерії, де розуміння взаємозв'язків між різними частинами програми є ключовим. Швидкість обробки запитів вимірюється в токенах за секунду та залежить як від архітектури моделі, так і від обчислювальної інфраструктури. Claude Desktop оптимізований для інтерактивного використання, забезпечуючи баланс між якістю відповідей та швидкістю генерації. Спеціалізовані бенчмарки, такі як HumanEval для програмування або MATH для математичних задач, дозволяють порівнювати ефективність різних моделей у специфічних доменах. Важливим аспектом є також підтримка мультимодальності. Сучасні версії Claude можуть обробляти не лише текст, але й зображення, що розширює можливості аналізу графічних інтерфейсів, діаграм та візуалізацій коду. Підтримка різних мов програмування та форматів файлів визначає універсальність застосування моделі у різноманітних проектах та середовищах розробки.

2.4.2 Функціональні можливості

Базовий набір функцій MCP сервера включає операції для отримання метаданих про бінарний файл, списків функцій, глобальних змінних та рядків. Сервер надає можливості декомпіляції функцій, дизасемблювання, отримання cross-references та роботи з коментарями.

Таблиця 2.1 — Повний список функцій MCP API з описом та параметрами

Назва функції	**Параметри**	**Опис функціональності**
get_byte_array	addr: int, size: int	Отримання масиву байтів заданого розміру з вказаної адреси
get_metadata	-	Отримання метаданих про поточну IDB (шлях, MD5, SHA256, розмір)
get_function_by_name	name: str	Пошук функції за її назвою
get_function_by_address	address: str	Отримання інформації про функцію за адресою
list_functions	offset: int, count: int	Отримання списку функцій з пагінацією
list_globals_filter	offset: int, count: int, filter: str	Отримання відфільтрованого списку глобальних змінних
list_globals	offset: int, count: int	Отримання списку всіх глобальних змінних
list_strings_filter	offset: int, count: int, filter: str	Отримання відфільтрованого списку рядків
list_strings	offset: int, count: int	Отримання списку всіх рядків у програмі
decompile_function	address: str	Декомпіляція функції за вказаною адресою
disassemble_function	start_address: str	Дизасемблювання функції за вказаною адресою
get_xrefs_to	address: str	Отримання всіх cross-reference до вказаної адреси
get_xrefs_to_field	struct_name: str, field_name: str	Отримання cross-reference до поля структури
set_comment	address: str, comment: str	Встановлення коментаря для вказаної адреси
rename_local_variable	function_address: str, old_name: str, new_name: str	Перейменування локальної змінної у функції

Продовження таблиці 2.1

Назва функції	**Параметри**	**Опис функціональності**
rename_global_variable	old_name: str, new_name: str	Перейменування глобальної змінної
rename_function	function_address: str, new_name: str	Перейменування функції

2.5 Промпт інженерія

Ефективна взаємодія з LLM у контексті пошуку вразливостей вимагає спеціального підходу до формування запитів. Промпт інженерія для задач бінарного аналізу має свої особливості, пов'язані зі специфікою низькорівневого коду та необхідністю точної інтерпретації результатів.

2.5.1 Принципи створення промптів

Основним принципом створення ефективних промптів є контекстуалізація запиту з урахуванням специфіки аналізованого бінарного файла. Промпт повинен включати інформацію про архітектуру процесора, операційну систему, компілятор та інші релевантні характеристики.

Роглянемо один з запитів, що використовувався для задачі зворотньої розробки.

Task: Analyze a binary file using IDA Pro via the MCP server.

Instructions:
Use IDA decompilation as the primary source of analysis.
Use MCP tools (e.g. convert_number) for assistance —— never perform base conversions manually.
Follow this workflow:

Code Cleanup:
Comment your findings directly in the decompiled code.
Rename functions and variables to meaningful, descriptive names.
Correct types for variables and arguments where applicable (especially arrays and pointers).

If decompilation lacks detail, disassemble functions and add comments.

Restrictions:
Do not brute–force or guess any passwords or keys.
Derive all solutions through static analysis and optionally simple Python scripts.

Output:
Save your findings and methodology in report.md.
When a solution (e.g. password) is found, prompt the user with the result and ask for feedback.
If analysis shows the password cannot be determined statically, clearly document the reasoning in report.md and stop execution.

Промпт має чітку структуру, що включає завдання, інструкції, обмеження та очікуваний результат. Це дозволяє LLM зосередитися на ключових аспектах аналізу та уникнути помилок, пов'язаних з неправильним тлумаченням задачі. Обов'язковим є вказівка на використання MCP інструментів для виконання специфічних операцій, таких як конвертація чисел, що дозволяє уникнути ручних помилок та підвищує ефективність аналізу, так як LLM не має вбудованих функцій для роботи з низькорівневими даними. Також важливим є обмеження щодо використання brute-force методів, що дозволяє зосередитися на статичному аналізі та уникнути непотрібних витрат часу на невірні спроби. Відповідно шляхом використання промпту зменшенна і можливість створювати хибні спрацювання, що може призвести до помилок в аналізі. Також важливою частиною є вказівка на створення звіту про результати аналізу, що дозволяє систематизувати отримані дані та полегшує подальше використання результатів дослідження, а також вказівка на коментування коду, що дозволяє LLM зосередитися на важливих аспектах аналізу та уникнути помилок, пов'язаних з неправильним тлумаченням задачі. Завдяки цьому, людина зможе легко перевірити результат і зрозуміти логіку аналізу, що є критично важливим для задач зворотної розробки.

Розглянемо ще один приклад промпту, що був використаний специфічно для задач аналізу вразливостей бінарного коду:

You task is to analyze a binary file in IDA Pro. You can use the MCP tools to retrieve information.

In general use the following strategy:
– Inspect the decompilation
– If more details are necessary, disassemble the function
– NEVER convert number bases yourself. Use the convert_number MCP tool if needed!
– Create a report.md in which describe general workflow of binary
– Check all functions for presence of known memory corruption vulnerabilities
– If you not sure that vulnerability is present do not describe it to me, avoid false positives
– If you found vulnerability in decompiled code, write report for it, in which describe where, how and why it occurs.
– Also write steps to reproduce the bug, write example of payload that will trigger crash of program with use of this vulnerability
– Provide description on how to patch it
– Do not provide analysis or output until you create complete report
– do not stop until you check all functions
– consider what is vulnerability and what is false positive, report ONLY with high certain.
– Do not write report until you investigate all functions

Цей промпт вже спрямований більше на глибокий аналіз всіх функцій, з нього були виключені етапи, що стосуються документації і коментування, так як процес пошуку займає більше часу, і LLM починає зосереджуватись на коментуванні коду, а не на пошуку вразливостей. Також чітко вказано, в якому форматі має бути створений звіт, що дозволяє LLM зосередитися на важливих аспектах аналізу та уникнути помилок, пов'язаних з неправильним тлумаченням задачі. Відповідно були додані вказівки які допомагають збільшити покриття коду, що аналізується, та зменшити кількість хибних спрацювань.

2.5.2 Створення проектів

Сучасні LLM платформи підтримують концепцію проектів, що дозволяє організовувати роботу навколо конкретних завдань зі збереженням загального контексту між сесіями. Проектний режим забезпечує персистентність інформації, дозволяючи моделі накопичувати знання про специфічні цілі аналізу, використовувані інструменти та попередні результати досліджень. Приклад роботи з проектом можемо побачити на зображенні 2.7.

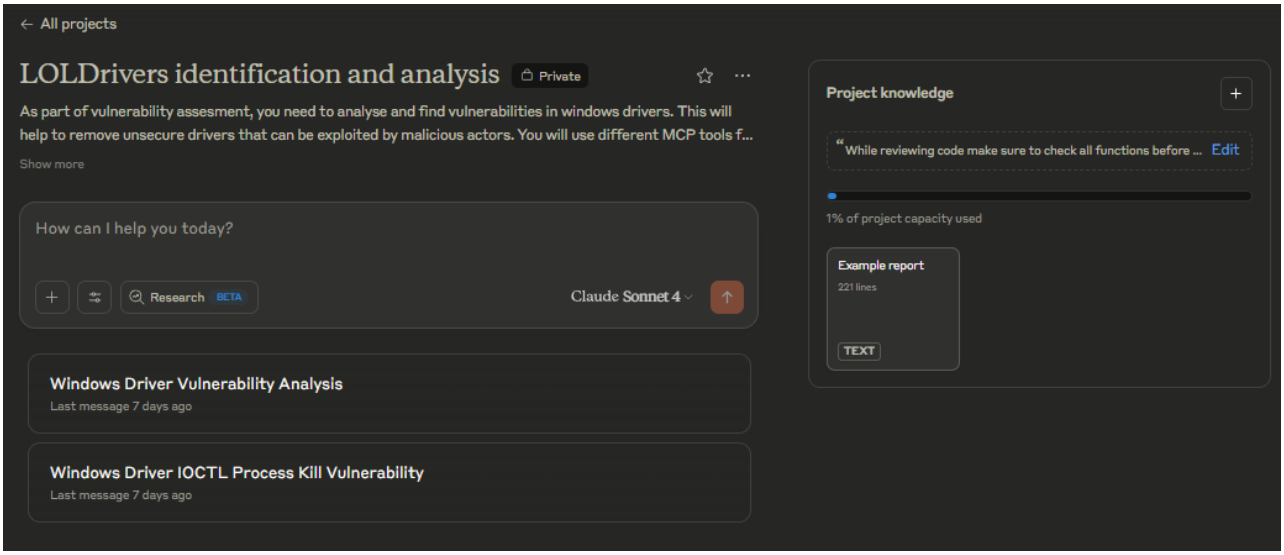


Рисунок 2.7 — Приклад використання проекту в клієнті Claude

У рамках проекту модель може підтримувати детальну інформацію про аналізовані бінарні файли, виявлені вразливості, використовувані методології та результати тестування. Це створює основу для більш ефективної роботи над довгостроковими завданнями, де кожна нова сесія може будуватися на попередніх результатах без необхідності повторного пояснення контексту. Проектний підхід також дозволяє організовувати командну роботу, коли декілька аналітиків можуть працювати з спільним контекстом, додаючи свої спостереження та результати до загальної бази знань проекту. Модель може автоматично синтезувати інформацію з різних джерел та надавати консолідовані звіти про стан дослідження.

2.6 Режими роботи з LLM

Сучасні великі мовні моделі пропонують різноманітні режими роботи, що дозволяють оптимізувати взаємодію залежно від специфіки завдання та вимог до якості результату.

2.6.1 Стильові режими

Адаптивний стильовий режим дозволяє налаштовувати характер та форму відповідей відповідно до аудиторії та контексту використання. Модель може адаптувати рівень технічної деталізації, від простих пояснень для початківців до глибоко технічних аналізів для експертів. Це особливо корисно при створенні документації, навчальних матеріалів або презентацій результатів аналізу для різних стейкхолдерів. У контексті аналізу безпеки програмного забезпечення стильова адаптація може включати формування звітів у вигляді технічних бюлетенів безпеки, академічних публікацій або виконавчих резюме для керівництва. Модель здатна автоматично налаштовувати термінологію, рівень деталізації та структуру викладу відповідно до цільової аудиторії.

2.6.2 Режим Deep Researh

Режим глибокого дослідження представляє найбільш потужний інструмент для комплексного аналізу складних питань. У цьому режимі модель використовує багатоетапний підхід до дослідження, який може тривати від декількох хвилин до години залежно від складності запиту. Процес включає автоматичне планування дослідження, пошук релевантної інформації через доступні інструменти, синтез даних з множини джерел та формування комплексного аналізу. Deep Research особливо ефективний для задач, що вимагають порівняльного аналізу, оцінки трендів у кібербезпеці або дослідження нових типів загроз. Модель може самостійно структурувати дослідження, визначаючи ключові аспекти для аналізу та послідовно опрацьовуючи кожен з них. Результатом є детальний звіт з обґрунтованими висновками та рекомендаціями, що базуються на аналізі актуальної інформації з множини джерел.

2.6.3 Режим Extended Thinking

Режим розширеного мислення активує додаткові механізми рефлексії та самоаналізу у процесі генерації відповідей. Модель явно демонструє свій процес міркування, включаючи розгляд альтернативних підходів, оцінку надійності інформації та ідентифікацію потенційних обмежень або невизначеностей у своєму аналізі. Цей режим забезпечує вищу якість відповідей за рахунок більш тривалого процесу обробки. Extended Thinking режим особливо цінний для критично важливих завдань аналізу безпеки, де точність та повнота аналізу мають першочергове значення. Модель може ідентифікувати потенційні помилки у своєму аналізі, запропонувати альтернативні інтерпретації даних та надати оцінку впевненості у своїх висновках. Це дозволяє аналітикам краще розуміти обмеження результатів та приймати більш обґрунтовані рішення.

2.7 Реалізація за підвищених вимог до конфіденційності

Для організацій, що працюють з конфіденційними бінарними файлами, критично важливо забезпечити захист інформації при використанні LLM. Система підтримує кілька режимів роботи для різних рівнів вимог до конфіденційності. З урахування цих вимог, було реалізовано додаткову модель архітектури, яка використовує локальну модель замість комерційного клієнта Claude. Локальний режим використовує локально розгорнуті відкриті моделі, такі як Llama або Mixtral, що забезпечує повний контроль над даними. Хоча такі моделі можуть бути менш потужними за комерційні аналоги, вони забезпечують повну конфіденційність. На зображенні 2.8 представлено практичну реалізацію системи інтеграції декомпіляторів з LLM через архітектуру n8n workflow. Система дозволяє зібрати конфігурацію з різниї необхідних нам частин власноруч і демонструє автоматизований процес взаємодії між різними компонентами: починаючи від отримання повідомлення користувача, його обробки через AI Agent (Tool Agent), взаємодії з Ollama Chat Model для локального виконання запитів, та завершуючи виконанням команд че-

Рисунок 2.8 — Локальне розгортання архітектури з використанням n8n

рез `MCP EXECUTE` модуль. Workflow показує практичну, локальну, реалізацію концепції, описаної в попередніх розділах роботи, де LLM отримує доступ до функцій декомпілятора через MCP протокол.

Використовувана модель може бути замінена на будь-яку іншу відкриту модель, що підтримує необхідні функції, такі як Llama 3.1 або Mixtral. Тим не менш, будь-яка з відкритих моделей буде набагато менш продуктивною, але має і перевагу, так як певні моделі можуть бути донавчені на специфічних даних, що дозволяє отримати більш точні результати в специфічних задачах, таких як аналіз вразливостей або зворотня розробка.

Також можливі реалізації з гібридними рішеннями, наприклад підключення фазерів до системи з LLM, з генерацією корпусу на основі результатів статичного аналізу проведеного моделлі, що може значно пришвидшити процес фазингу [28].

Висновки до розділу 2

У другому розділі було запропоновано та детально описано модель системи пошуку вразливостей пошкодження пам'яті на основі великих мовних моделей. Розроблена архітектура демонструє інноваційний підхід до інтеграції сучасних декомпіляторів з можливостями штучного інтелекту через стандартизований Model-Context-Protocol. Було реалізовано повнофункціональну систему, що складається з трьох основних компонентів: MCP сервера з інтегрованим плагіном IDA Pro, MCP мосту для забезпечення стандартизованої взаємодії та MCP клієнта Claude для високорівневого аналізу. Архітектура побудована за принципом модульності, що забезпечує гнучкість у виборі компонентів та можливість масштабування. Було до-розроблено комплексний MCP сервер, який містить понад 30 спеціалізованих функцій для роботи з бінарним кодом, включаючи операції декомпіляції, дизасемблювання, отримання метаданих, роботи з cross-references та модифікації структури програми. Система підтримує дворівневу безпеку з автоматичною категоризацією функцій на безпечні, модифікуючі та потенційно небезпечні операції. Були досліджені та імплементовані ключові особливості ефективної взаємодії з LLM у контексті бінарного аналізу, включаючи принципи промпт інженерії, проектний підхід до організації роботи та різні режими функціонування моделей. Особливу увагу приділено розробці методології формування запитів, що забезпечує контекстуалізацію, конкретизацію завдань та мінімізацію хибних спрацювань. Було імплементовано альтернативну архітектуру для роботи в умовах підвищених вимог до конфіденційності з використанням локально розгорнутих відкритих моделей через платформу n8n workflow. Це рішення забезпечує повний контроль над даними при збереженні основної функціональності системи. Розроблена модель демонструє практичну можливість створення автономних AI-агентів для задач зворотної інженерії та аналізу безпеки, що здатні виконувати складні багатокрокові операції з мінімальним втручанням людини. Система забезпечує синергетичний ефект поєднання когнітивних здібностей LLM з технічними можливостями профільних інструментів декомпіляції.

3 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

В дослідженні використаємо модель Claude Sonnet 4.0 в зв'язці з MCP сервером декомпілятора IDA. При невдалих спробах використовувались також різні режими роботи ШІ агента, а також для останніх експериментів було використано модель Claude Opus 4.0 що вийшла в доступ в процесі дослідження. Під кожен тип задач було створено окремий промпт, що дозволяє моделі максимально ефективно виконувати поставлені завдання. Всі експерименти проводились на бінарних файлах з відкритим кодом, а також на реальних прикладах вразливостей з публічних баз даних.

3.1 Застосування до задач зворотньої розробки

Для ефективного пошуку вразливостей модель має показувати високі результати в задачах зворотньої розробки, що включає декомпіляцію бінарних файлів, аналіз їх логіки та структури. Для дослідженн цих можливостей було обрано декілька рівнів складності задач, що підводять до виявлення вразливостей.

3.1.1 Аналіз різних типів бінарних файлів

Компільований код мов різного рівня абстракції може мати різні особливості, що впливають на ефективність аналізу. Було обрано декілька типів бінарних файлів, включаючи:

- Скомпільовані з C
- Скомпільовані з C++ з використанням ООП;
- Скомпільовані з Rust, що має інші підходи до управління пам'яттю;

- Файли позбавлені від символів, що ускладнює аналіз;

Було проведено аналіз кожного типу файлів, щоб визначити, як модель справляється з різними рівнями абстракції та складності. Система впоралась з усіма типами бінарних файлів, продемонструвавши здатність до ефективного використання інструментів декомпіляції та аналізу коду, в результаті роботи модель надала детальний звіт про структуру та логіку виконання програмного коду, дала змістовні коментарі та назви в середині декомпілятора IDA, що дозволило зрозуміти загальну логіку виконання програми.

3.1.2 Відновлення логіки роботи програм

Для тестування адекватності розуміння логіки виконання програмного коду ШІ агентом було обрано класичну задачу зворотньої розробки - відновлення алгоритму перевірки паролів. Було створено бінарний файл, що містить комплексний алгоритм перевірки пароля. Модель повинна була вказати, які саме умови перевіряються, а також відновити структуру коду і безпосередньо пароль що перевіряється. За результатами роботи модель успішно відновила логіку перевірки паролю, що можна побачити на зображенні 3.1.

Агент не тільки зміг самостійно дослідити логіку виконання програми, а й відновити структуру коду, витягти з пам'яті дані за допомогою інструментів IDA, а також відновити пароль за допомогою скрипта мовою JavaScript, що був згенерований та запущений моделлю в ізольованому оточенні.

3.1.3 Виявлення програмних закладок

Основною з основних переваг використання ШІ є його здатність аналізувати великі обсяги даних та виявляти закономірності, що можуть бути неочевидними для людини. Було проведено експеримент з виявлення програмних закладок у бінарних файлах. Модель повинна була ідентифікувати потенційні закладки, що можуть бути використані для віддаленого доступу або збору інформації.

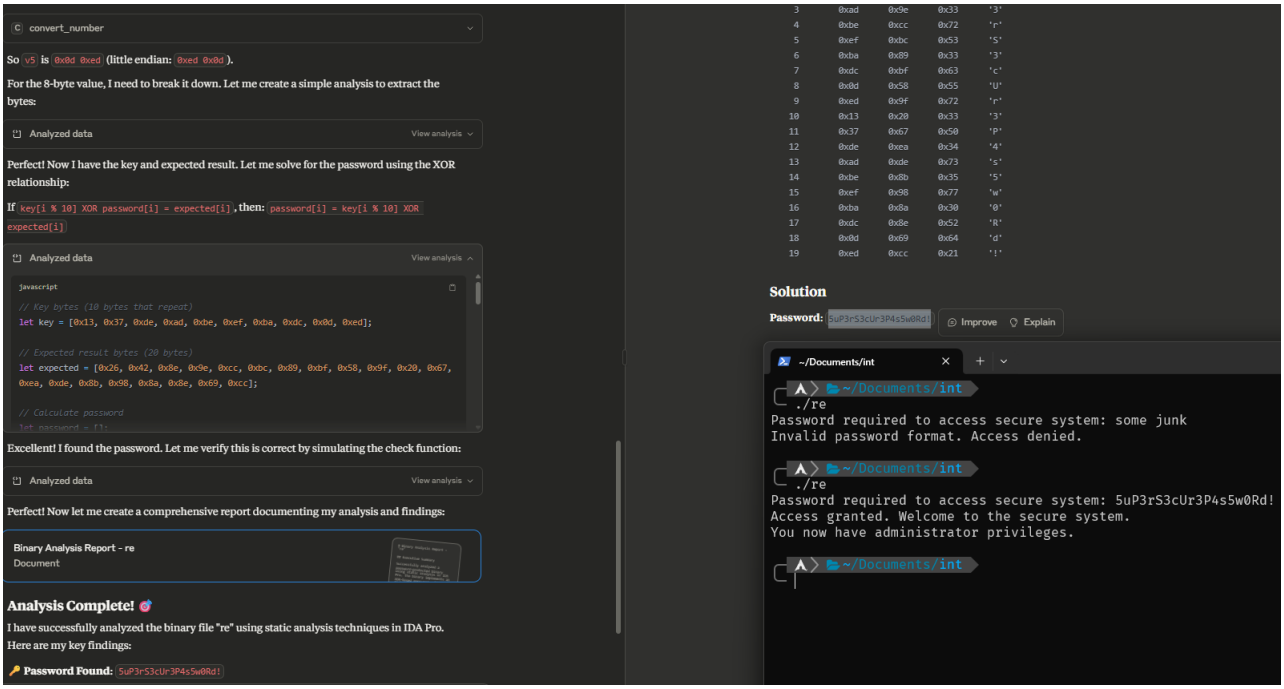


Рисунок 3.1 — Результати аналізу алгоритму перевірки паролю

Для тестування було використано один з попередніх файлів, в який було додано функцію закладку з специфічною умовою виконання, а також вирізано усі символи для поскладнення задачі аналізу. Короткий опис результатів від моделі:

> The analyzed binary complex_c_program_stripped is a complex C program that implements an AVL tree data structure with advanced mathematical computations. CRITICAL FINDING: The binary contains a backdoor that allows unauthorized shell access when specific environment variables are set.

В результаті роботи агент зміг розпізнати програмну закладку серед функцій, а також надати їм змістовні назви, що можна побачити на зображенні 3.2.

3.1.4 Пошук потенційно небезпечних функцій драйверів

LOLDrivers (Living Off The Land Drivers) — це легітимні підписані драйвери, які можуть бути зловживані зловмисниками для виконання шкідливих дій в операційній системі [29, 30]. Ці драйвери мають цифровий підпис від довірених розробників, що дозволяє їм завантажуватися навіть на системах з

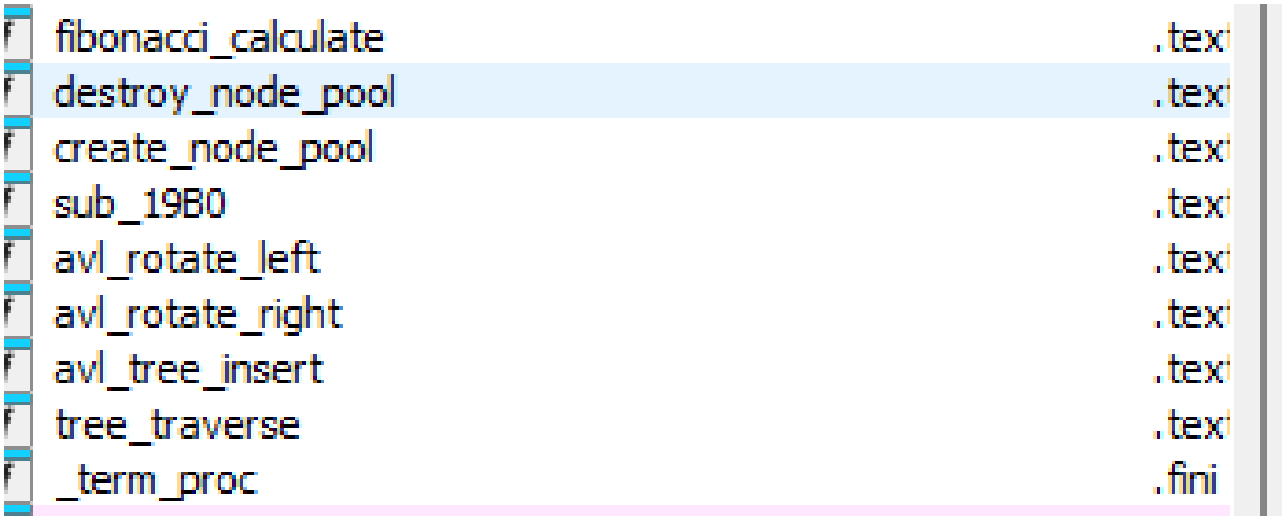


Рисунок 3.2 — Створені агентом коментарі та назви функцій в середині декомпілятора IDA

увімкненою перевіркою підписів драйверів. Офіційно підписані виробниками, такі драйвери можуть надавати доступ до низькорівневих операцій системи і можуть використовуватися для обходу захисних механізмів. Оскільки через легітимність процесів пов'язаних з підписаними драйверами, виявлення такого роду атак є нетривіальною задачею, відповідно такий тип програмного забезпечення має бути найбільш захищеним.

Зловмисники можуть використовувати їх функціонал для завершення захисних процесів (антивірусів, EDR систем), читання та запису в пам'ять, а також для виконання шкідливих дій без підозри з боку користувача. Було проведено експеримент з виявлення таких драйверів у бінарних файлах. Модель повинна була ідентифікувати потенційно небезпечні функції, що можуть бути використані для зловмисних цілей, а саме знайти фрагменти коду що можуть бути використані з метою звершення привілейовних процесів.

Для аналізу був вибраний драйвер що містить відомі вразливості, а саме BdApiUtil.sys. В результаті аналізу модель коректно виявила функції, що можуть бути використані для зловмисних цілей, а також надала рекомендації щодо їх потенційного використання. Файл був відкритиій за допомогою декомпілятора IDA, а потім проаналізований за допомогою моделі Claude Sonnet 4.0, що продемонстровано на зображенні 3.3. Модель була налаштована на пошук специфічних патернів коду, що вказують на потенційно небез-

печні функції.

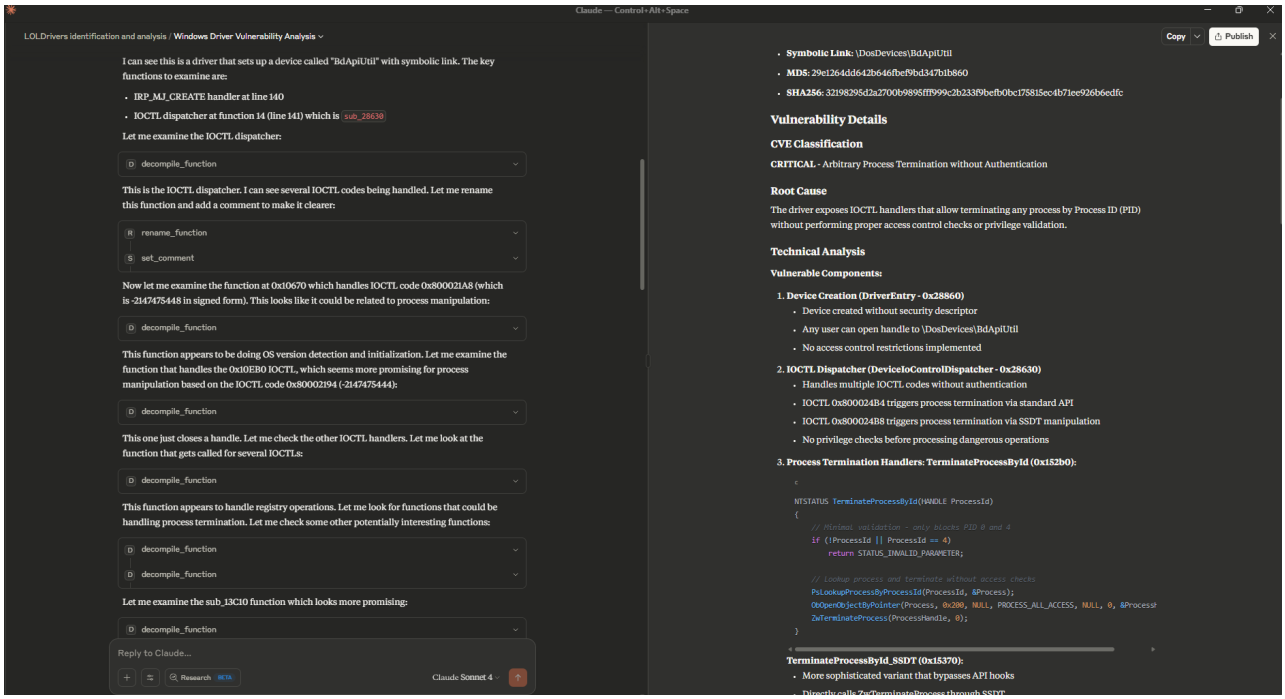


Рисунок 3.3 — Зворотня розробка драйвера BdApiUtil.sys

Цей приклад гарно демонструє здатність моделі до аналізу та розуміння складних бінарних файлів, тому в цій роботі розглянемо цей конкретний приклад більш детально. Завдяки використанню специфічного промпту, модель провела дослідження ефективно, використавши алгоритм за яким працював би дослідник. Спочатку агент зрозумів загальне призначення і роботу програми шляхом аналізу ключових функцій, що видно на зображенні 3.4.

Після чого модель почала шукати специфічні функції і патерни коду, напряму пов'язані з IOCTL. Для цього було використано різні методи, найуспішнімий з який орієнтування за текстовими рядками в середині бінарного файлу. Декомпілятор надає можливість відстежувати де використовується певний текстовий рядок, а також звідки викликаються певні функції. Це дозволяє моделі знаходити зв'язки між різними частинами коду та виявляти потенційно небезпечні виклики функцій, простежуючи загальне виконання програми. В результаті модель змогла знайти місце в програмі яке може бути використане для завершення привілейованих процесів, що можна побачити на зображенні 3.5.

За результатами свого аналізу модель склала повноцінний звіт, що від-

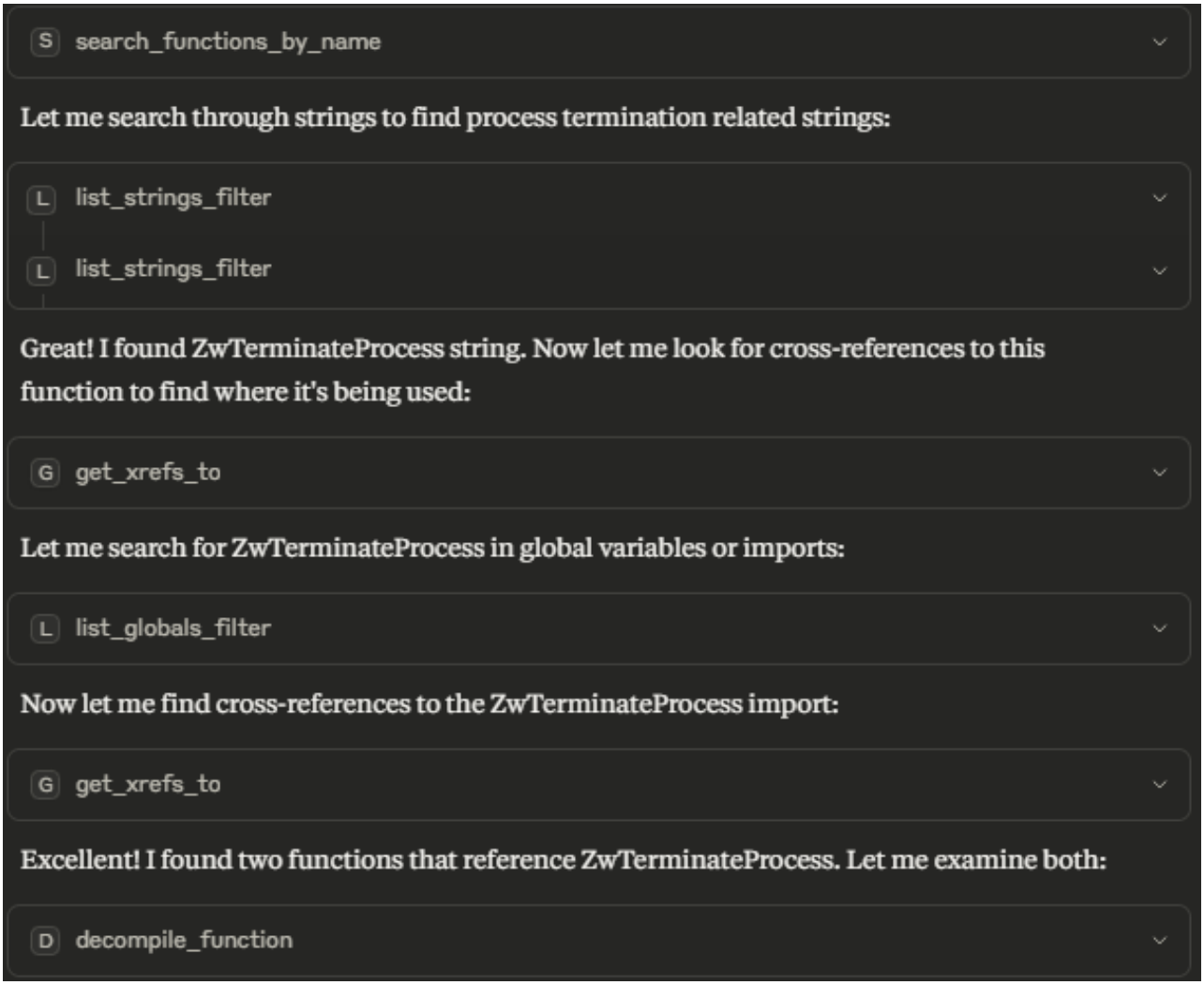


Рисунок 3.4 — Аналіз функцій драйвера BdApiUtil.sys та пошук специфічних патернів

повідає технічному завданню.

Агент надав наступні результати:

Загальний опис та атрибуцію, що містить коротку інформацію та унікальні ідентифікатори драйвера IOCs (Indicators of Compromise), в часності геші. Також було описано та класифіковано знайдену вразливість.

BdApiUtil.sys Driver Security Analysis Report

Executive Summary

The BdApiUtil.sys driver contains a critical privilege escalation vulnerability that allows any user with access to the device to terminate arbitrary processes on the system. This vulnerability completely bypasses Windows process security and can be used to terminate critical system processes, antivirus software, and other security applications.

Driver Overview

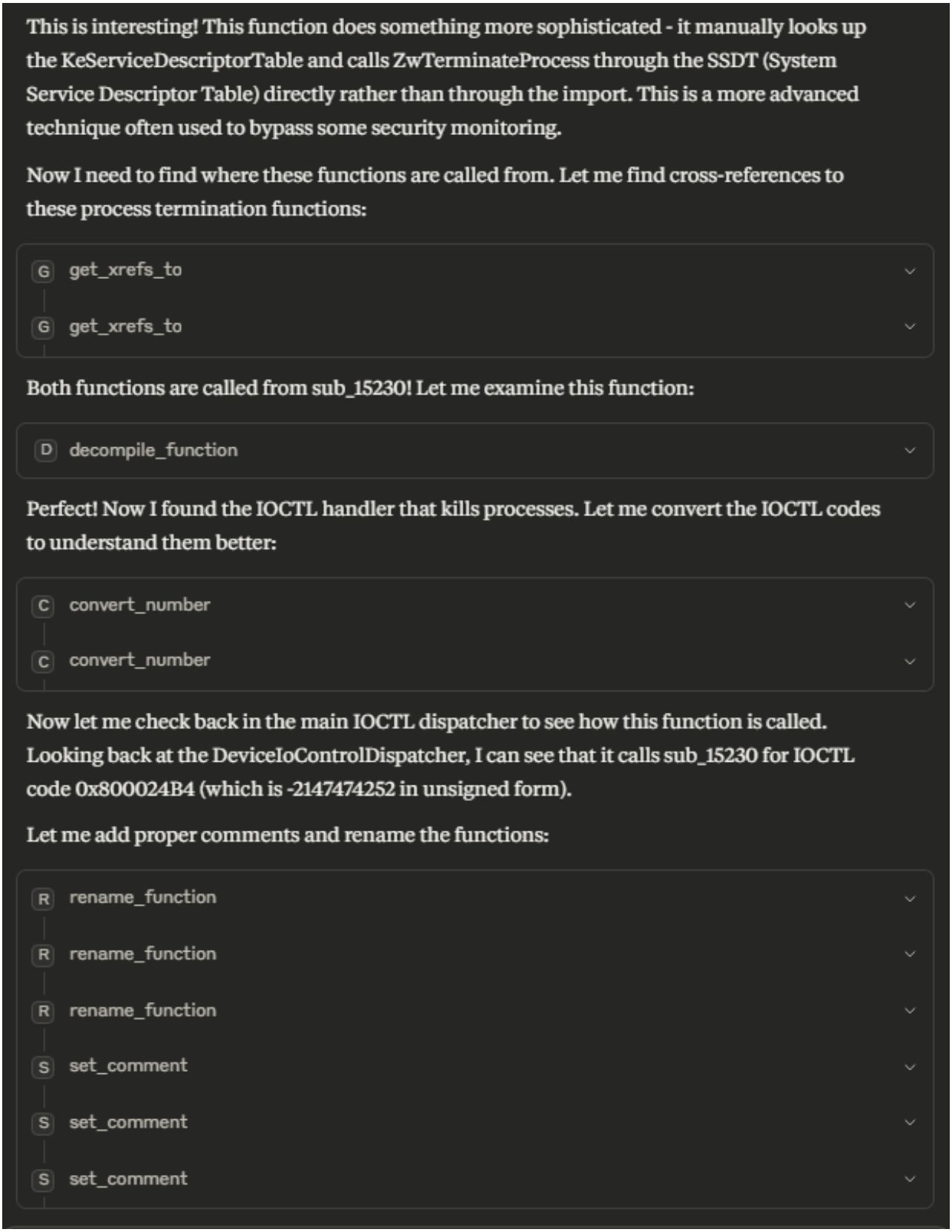


Рисунок 3.5 — Модель встановила місце в програмі яке може бути використане для завершення привілейованих процесів

File Name: BdApiUtil.sys
Device Name: \Device\BdApiUtil
Symbolic Link: \DosDevices\BdApiUtil
MD5: 29e1264dd642b646fbef9bd347b1b860
SHA256: 32198295d2a2700b9895fff999c2b233f9befb0bc175815ec4b71ee926b6edfc

IOCs (Indicators of Compromise)

File: BdApiUtil.sys
Device: \Device\BdApiUtil
Symbolic Link: \DosDevices\BdApiUtil
IOCTL Codes: 0x800024B4, 0x800024B8
Registry Keys: Related to driver installation
Process: Any process opening \.\BdApiUtil device

Vulnerability Details
CVE Classification
CRITICAL – Arbitrary Process Termination without Authentication
Root Cause
The driver exposes IOCTL handlers that allow terminating any process by Process ID (PID) without performing proper access control checks or privilege validation.

Модель також надала детальний технічний аналіз вразливості, що включає шматок вразливого коду, а також пояснення його роботи.

```
NTSTATUS TerminateProcessById(HANDLE ProcessId)
{
    // Minimal validation – only blocks PID 0 and 4
    if (!ProcessId || ProcessId == 4)
        return STATUS_INVALID_PARAMETER;

    // Lookup process and terminate without access checks
    PsLookupProcessByProcessId(ProcessId, &Process);
    ObOpenObjectByPointer(Process, 0x200, NULL, PROCESS_ALL_ACCESS, NULL, 0,
        &ProcessHandle);
    ZwTerminateProcess(ProcessHandle, 0);
}
```

Модель також надала теоретичний аналіз вразливості, включно з опи-

сом та оцінкою ризиків і впливу.

Vulnerability Flow:

User Process opens handle to \\.\BdApiUtil
DeviceIoControl called with IOCTL 0x800024B4 or 0x800024B8
Input Buffer contains 4−byte Process ID
Driver performs minimal validation (only blocks PID 0 and 4)
Process Terminated without further authorization

Security Impact
High Severity Issues:

Arbitrary Process Termination: Can kill any process except System (PID 4)
Security Software Bypass: Can terminate antivirus, EDR, and security monitoring
Critical Process Termination: Can terminate Windows services, LSASS, winlogon, etc.
Denial of Service: Can cause system instability by terminating essential processes
Privilege Escalation: Standard users can impact system−level processes

Attack Scenarios:

Malware Defense Evasion: Terminate security software before deploying payload
System Destabilization: Crash system by terminating critical Windows processes
Credential Harvesting: Terminate LSASS to force re−authentication
Ransomware Operations: Clear security processes before encryption

Також агентом був наданий Proof of Concept (PoC) код, що демонструє вразливість та її використання.

```
#include <windows.h>
#include <stdio.h>

int main() {
    HANDLE hDevice = CreateFile(L"\\\\.\\BdApiUtil",
                                GENERIC_READ | GENERIC_WRITE,
                                0, NULL, OPEN_EXISTING, 0, NULL);

    if (hDevice == INVALID_HANDLE_VALUE) {
        printf("Failed to open device\n");
        return 1;
    }
```

```
    DWORD targetPID = 1234; // Replace with target PID
    DWORD bytesReturned;

    BOOL result = DeviceIoControl(hDevice, 0x800024B4,
                                  &targetPID, sizeof(DWORD),
                                  NULL, 0, &bytesReturned, NULL);

    if (result) {
        printf("Process %d terminated successfully\n", targetPID);
    } else {
        printf("Failed to terminate process: %d\n", GetLastError());
    }

    CloseHandle(hDevice);
    return 0;
}
```

На прикінці репорту, також були надані поради щодо усунненя вразливості, включно з рекомендаціями щодо коду та архітектури драйвера.

Patch Recommendations
Immediate Actions:

Remove Driver: Uninstall BdApiUtil.sys from affected systems
Block Installation: Prevent deployment of vulnerable driver versions
Monitor Usage: Check for exploitation attempts in security logs

Code−Level Fixes:
1. Access Control Implementation:

```
cNTSTATUS CheckCallerPrivileges() {
    SECURITY_SUBJECT_CONTEXT SubjectContext;
    BOOLEAN HasPrivilege = FALSE;

    SeCaptureSubjectContext(&SubjectContext);
    HasPrivilege = SePrivilegeCheck(&SeDebugPrivilege, &SubjectContext, UserMode);
    SeReleaseSubjectContext(&SubjectContext);

    return HasPrivilege ? STATUS_SUCCESS : STATUS_ACCESS_DENIED;
}
```

```
2. Process Protection Lists:
cDWORD ProtectedProcesses[] = {
    4, // System
    0, // Idle
    // Add other critical PIDs
};

BOOLEAN IsProcessProtected(DWORD PID) {
    for (int i = 0; i < ARRAYSIZE(ProtectedProcesses); i++) {
        if (ProtectedProcesses[i] == PID) return TRUE;
    }
    return FALSE;
}
3. Input Validation:
cif (InputBufferLength != sizeof(DWORD)) {
    return STATUS_INVALID_PARAMETER;
}

if (!InputBuffer) {
    return STATUS_INVALID_ADDRESS;
}
4. Device Security:
c// Set restrictive security descriptor on device creation
UNICODE_STRING SecurityDescriptor;
RtlInitUnicodeString(&SecurityDescriptor, L"D:P(A;;GA;;;SY)(A;;GA;;;BA)");
IoCreateDeviceSecure(DriverObject, 0, &DeviceName, FILE_DEVICE_UNKNOWN,
                     0, FALSE, &SecurityDescriptor, NULL, &DeviceObject);
```

Вразливості цього драйверу досліджені людьми, та є загально відомими [31]. Результати аналізу модель повністю співпадають з результатами експертів з кібербезпеки, що підтверджує її здатність до аналізу та розуміння складних бінарних файлів. При тому що модель не була спеціально навчена на цьому драйвері, а лише на основі знань про вразливості та патерни коду, що можуть бути використані для їх виявлення, вона впоралась з визначенням вразливості з короткий проміжок часу, що демонструє її потенціал в задачах зворотньої розробки та аналізу вразливостей.

3.2 Застосування ддя задач аналізу вразливостей у застосунках та бібліотеках

Як вже було зазначено раніше, аналіз вразливостей у бінарному коді є критично важливим компонентом сучасної кібербезпеки, оскільки дозволяє виявляти потенційні загрози безпеки до їх експлуатації зловмисниками. Традиційні методи аналізу, включаючи ручний аудит коду та автоматизовані інструменти типу статичних аналізаторів і фаззерів, часто потребують значних ресурсів та можуть пропускати складні вразливості або генерувати велику кількість хибних спрацювань.

3.2.1 Пошук класичних вразливостей

Виявлення класичних типів вразливостей у бінарному коді є одним з основних завдань сучасної кібербезпеки. Мета цього дослідження полягає у перевірці здатності інтегрованої системи LLM з декомпіляторами автоматично ідентифікувати поширені типи вразливостей, такі як переповнення буферів, використання після звільнення (Use-After-Free), подвійне звільнення пам'яті та формат-стрінг вразливості. Важливість автоматизації цього процесу зумовлена постійним зростанням складності програмного забезпечення та обмеженими ресурсами команд безпеки для проведення ручного аудиту коду. Для тестування потенціалу моделі було обрано декілька класичних вразливостей, що часто зустрічаються у програмному забезпеченні, та були створені відповідні бінарні файли з навмисно закладеними вразливостями. Задача аналізу такий файлів тривіальна, так як ці мета і потік виконання цих файлів навмисно формується навколо вразливості.

В рамках демонстрації дослідження покажемо результати аналізу вразливості Stack-Based buffer overflow, у консольному застосунку `restaurant`. Завантаживши його в декомпілятор IDA, був використаний промпт описаний у попередніх розділах, що дозволяє моделі зосередитись на саме пошуку вразливостей. Результат роботи включно з частиною згенерованого звіту можна побачити на зображенні 3.6.

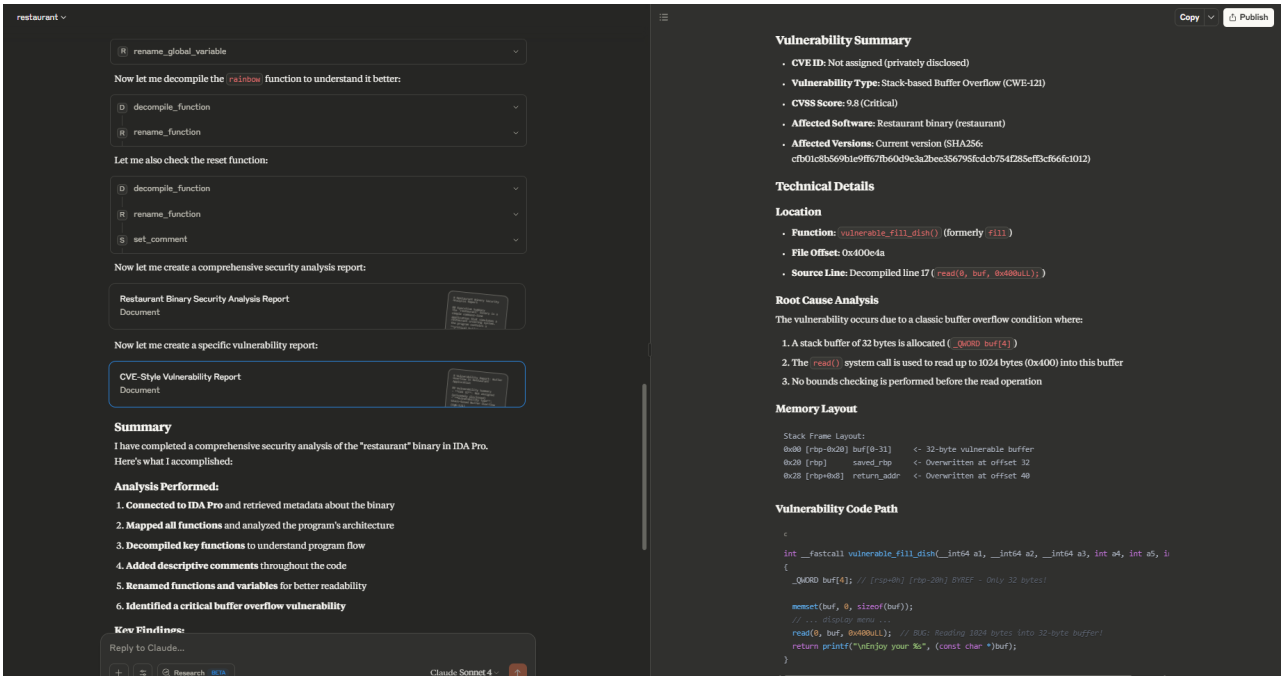


Рисунок 3.6 — Результати аналізу вразливості Stack-Based buffer overflow у консольному застосунку

Модель зосередилась на пошуку патернів коду, що вказують на потенційні вразливості, такі як відсутність перевірок розміру буфера перед записом даних. В результаті аналізу модель виявила, що функція `fill` не перевіряє розмір введених даних, що може призвести до переповнення буфера. При проведені аналізу, модель змінювала назви функцій та додавала коментарі, що можна побачити в середині декомпілятора IDA на зображенні 3.7.

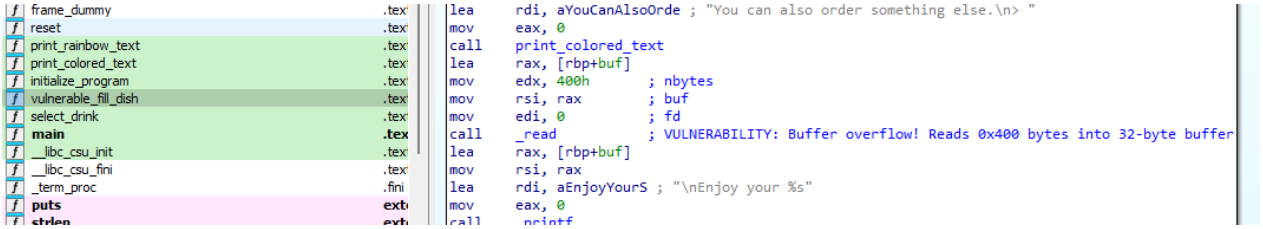


Рисунок 3.7 — Коментарі та назви функцій, що були змінені моделлю під час аналізу

Також модель надала Proof of Concept (PoC) код, що демонструє вразливість та її використання.

Proof of Concept

Simple Crash PoC

```
#!/bin/bash
# Simple crash demonstration
echo –e "1\n$(python3 –c 'print("A"*100)')" | ./restaurant
# Expected: Segmentation fault
```

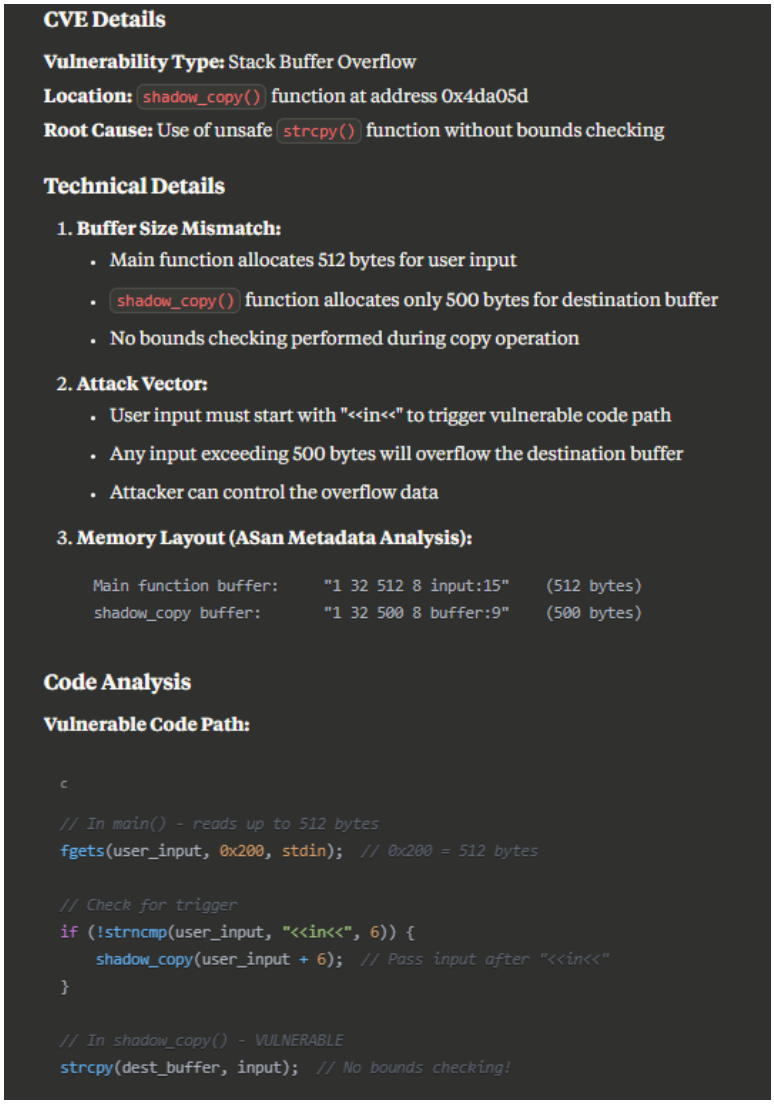
Результат перевірки вразливості можна побачити на зображенні 3.8.

Рисунок 3.8 — Валідація результатів аналізу на прикладі вразливості Stack-Based buffer overflow у консольному застосунку

Також було протестовано ефективність на модельному завданні для фазингу. Застосунок `sa-one` є консольним інструментом з нетривіальним потоком виконання, який був розроблений для демонстрації можливостей фазингу, відповідно при використанні класичних інструментів пошуку вразливостей за допомогою фазингу, наприклад `AFL++`, виявлення вразливостей потребує значних зусиль та часу. В результаті аналізу модель правильно ідентифікувала вразливість, і зробила це значно швидше, ніж традиційні інструменти. Результат аналізу можна побачити на зображенні 3.9.

Так як програма `sa-one` є прикладом для фазингу, вона була скомпільована з `asan` (AddressSanitizer), що дозволяє легше проаналізувати помилки управління пам'яттю, такі як переповнення буфера, використання після звільнення пам'яті та інші що виникли в результаті пошкодження пам'яті при виконані. Як бачимо на зображенні 3.10, запропонований агентом Proof of Concept (PoC) код викликав краш застосунку, що і є кінцеваою метою класичного фазингу.

Як бачимо на тестових прикладах, модель змогла виявити вразливість та надати детальний звіт про її природу, включно з рекомендаціями щодо



CVE Details

Vulnerability Type: Stack Buffer Overflow

Location: `shadow_copy()` function at address 0x4da05d

Root Cause: Use of unsafe `strcpy()` function without bounds checking

Technical Details

1. **Buffer Size Mismatch:**
   - Main function allocates 512 bytes for user input
   - `shadow_copy()` function allocates only 500 bytes for destination buffer
   - No bounds checking performed during copy operation
2. **Attack Vector:**
   - User input must start with "<<in<<" to trigger vulnerable code path
   - Any input exceeding 500 bytes will overflow the destination buffer
   - Attacker can control the overflow data
3. **Memory Layout (ASan Metadata Analysis):**

```
Main function buffer:     "1 32 512 8 input:15"    (512 bytes)
shadow_copy buffer:       "1 32 500 8 buffer:9"    (500 bytes)
```

Code Analysis

Vulnerable Code Path:

```c
// In main() - reads up to 512 bytes
fgets(user_input, 0x200, stdin);  // 0x200 = 512 bytes

// Check for trigger
if (!strncmp(user_input, "<<in<<", 6)) {
    shadow_copy(user_input + 6);  // Pass input after "<<in<<"
}

// In shadow_copy() - VULNERABLE
strcpy(dest_buffer, input);  // No bounds checking!
```

Рисунок 3.9 — Знайдена вразливість в типовому застосунку для фазингу

виправлення, та згенерувати робочий PoC код, що демонструє вразливість. Це свідчить про здатність моделі до аналізу та розуміння складних бінарних файлів, а також її потенціал у задачах виявлення вразливостей.

3.2.2 Тестування на штучних прикладах

Для контрольованої перевірки ефективності системи було створено набір тестових програм з навмисно закладеними вразливостями різних типів. Це дозволяє точно оцінити чутливість системи до конкретних патернів небезпечного коду та визначити межі її можливостей. Тестування на штучних при-

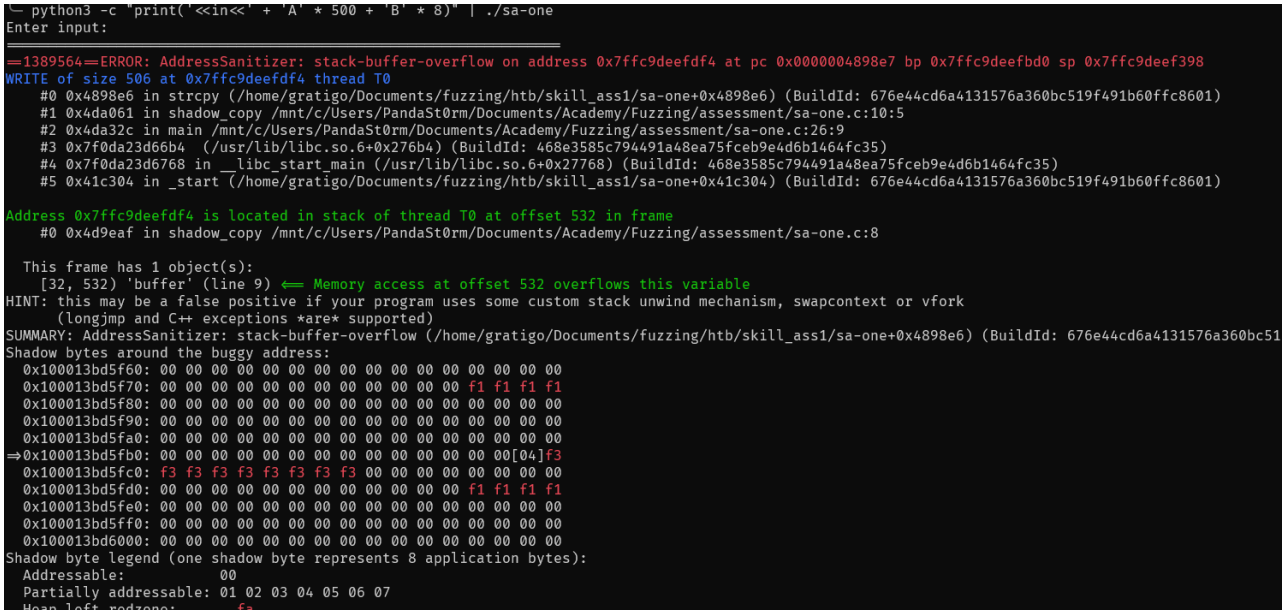


Рисунок 3.10 — Кращ застосунку `sa-one` при використанні PoC коду, що був згенерований моделлю

кладах забезпечує можливість систематичного аналізу поведінки AI-агента в умовах, коли тип та розташування вразливості точно відомі дослідникам, що є критично важливим для валідації методології. Ця секція відрізняється від попередньої тим, що в цьому дослідженні застосунки є легітимними з навмисно внесеною вразливістю до оригінального програмного коду. Відповідно додання вразливостей в існуючі бібліотеки вже є більш складним завданням для аналізу, так як потребує глибокого рощуміння логіки і структури програм, а також розуміння патернів вразливостей.

Було модифіковано та проаналізовано застосунок `maddy` - консольний інструмент для конвертації `markdown` в `HTML`.

Особливість цього аналізу заключається не тільки в тому що пошук відбувається в легітимному застосунку, а й в тому, що застосунок написаний на мові `C++`, з використанням об'єктно-орієнтованого програмування, що ускладнює аналіз, так як вимагає розуміння концепцій класів, спадкування та поліморфізму, особливо коли це стосується вразливостей, пов'язаних з управлінням пам'яттю та доступом до об'єктів. Результати аналізу також успішні і включають в себе детальний звіт про вразливість, що містить опис вразливості, її природу, а також рекомендації щодо виправлення, що можна побачити на зображенні 3.11. Запропонований агентом Proof of Concept (PoC) код ви-

кликав краш застосунку, що і є кінцевою метою класичного фазингу.

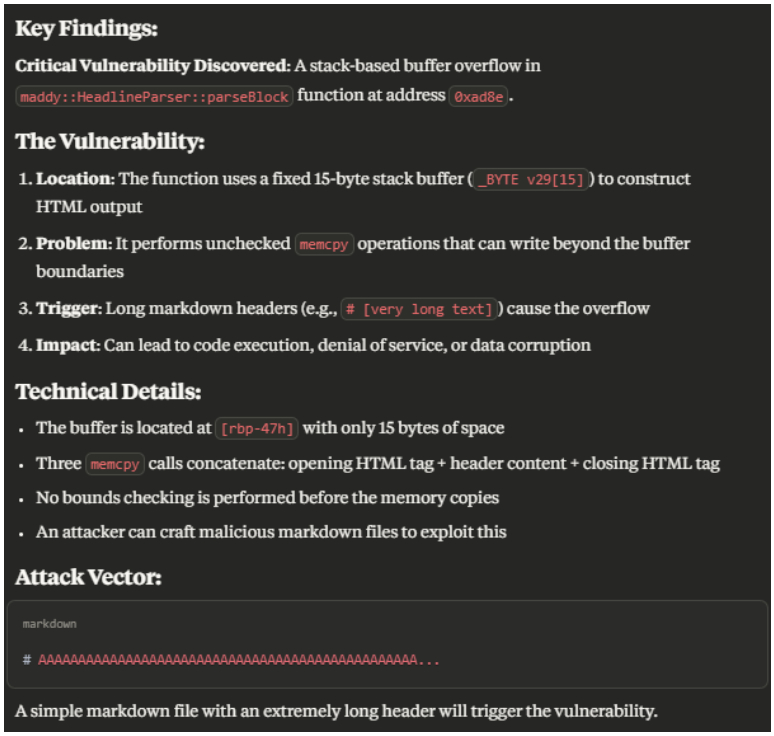


Рисунок 3.11 — Знайдена закладена вразливість в застосунку `maddy`

Було виявлено, що модель здатна ідентифікувати вразливості, пов'язані з управлінням пам'яттю, такі як переповнення буфера, використання після звільнення пам'яті та інші, що виникають в результаті пошкодження пам'яті при виконанні.

3.2.3 Тестування на реальних прикладах

Найбільш важливим та найбільш складним етапом валідації є тестування системи на реальних зразках програмного забезпечення з документованими вразливостями. Для цього використовувались бінарні файли з публічно відомими CVE, що дозволило провести пряме порівняння результатів роботи AI-агента з висновками експертів з кібербезпеки та автоматизованих інструментів типу фаззерів. Таке порівняння є ключовим для оцінки практичної застосовності розробленої методології, оскільки демонструє, чи здатна система виявляти вразливості в умовах реального програмного середовища з його складністю, обфускацією та різноманітними техніками захисту. Успішне

виявлення вразливості в реальних застосунках підтверджує, що система може бути використана для автоматизації процесу аналізу безпеки, зменшуючи навантаження на фахівців з кібербезпеки та підвищуючи ефективність виявлення вразливостей. Було обрано декілька реальних прикладів вразливостей з публічних баз даних, таких як CVE, що дозволило провести пряме порівняння результатів роботи AI-агента з висновками експертів з кібербезпеки та автоматизованих інструментів типу фаззерів.

- `libexif 0.6.14` - це бібліотека для читання та обробки EXIF даних (метаданих) у файлах зображень JPEG, яка широко використовується в Android-застосунках, браузерах, редакторах зображень та інших програмах для отримання інформації про камеру, налаштування зйомки та геолокацію. Аналіз безпеки цієї бібліотеки є критично важливим, оскільки уразливості в ній можуть призводити до віддаленої відмови в обслуговуванні (DoS) або навіть виконання довільного коду через обробку спеціально сформованих EXIF даних, що робить її привабливою ціллю для атакуючих через поширеність використання та автоматичну обробку метаданих при завантаженні зображень.

- `muparser` — це бібліотека з відкритим кодом, яка надає швидкий та гнучкий аналізатор математичних виразів [32]. Вона широко використовується в різних програмах для оцінки математичних виразів та виконання обчислень. Бібліотека дозволяє користувачам визначати власні змінні, функції та оператори, що робить її придатною для широкого спектру математичних обчислень.

При аналізі бібліотеки `libexif` було зкомпільовано її специфічну вразливу версію, та використано бібліотеку при доданні в середину консольного застосунку `exif`. При завантаженні в декомпілятор IDA, та аналізі за допомогою моделі, було ідентифіковано реально існуючу вразливість Integer Overflow в функції `exif_data_load_data_content` що призводить до Heap-based buffer overflow, що можна побачити на зображенні 3.12.

При пошуку вразливості було використано режим `Extended Thinking`, що дозволяє моделі самоаналізувати кожен свій крок, для глибшого розумін-

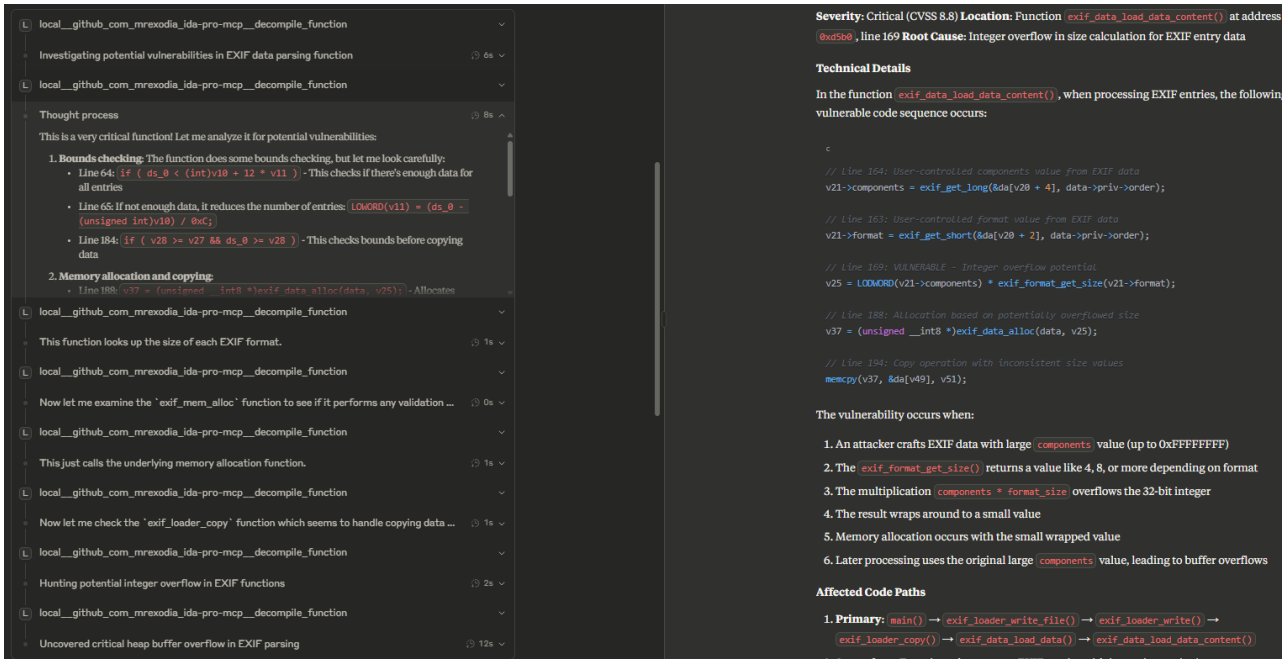


Рисунок 3.12 — Демонстрація роботи моделі в режимі `Extended Thinking` на прикладі бібліотеки `libexif`

ня логіки програми та виявлення потенційних вразливостей. Приклад роботи в цьому режимі можна побачити на зображенні 3.12. Відповідно до звіту, наданого моделлю, знайдена вразливість відповідає `CVE-2020-0198` [33, 34], що є відомою вразливістю в цій бібліотеці. Модель надала детальний звіт про вразливість, включно з описом, рекомендаціями щодо виправлення та Proof of Concept (PoC) кодом, що демонструє вразливість.

При дослідженні вразливостей бібліотеки `muparser` її було скомпільовано та завантажено в декомпілятор IDA, після чого було проведено аналіз за допомогою моделі. В результаті аналізу було виявлено три вразливості, дві з який є хибними спрацюваннями викликаними великою кількістю коду та специфікою декомпіляції IDA, а одна є реальною вразливістю, що була знайдена в функції `mu::ParserBase::ParseCmdCodeBulk()` та призводить до Integer Overflow, що може бути використано для виконання довільного коду. Результат аналізу можна побачити на зображенні 3.13.

Знайдена вразливість відповідає зареєстрованій `OSV-2020-1349` [35–37], що можна підтвердити запустивши вразливу версію бібліотеки з фрейкмворку `arvo`, що показано на зображенні 3.14.

Як бачимо, краш вразливої версії ідентифікує ту саму вразливу функ-

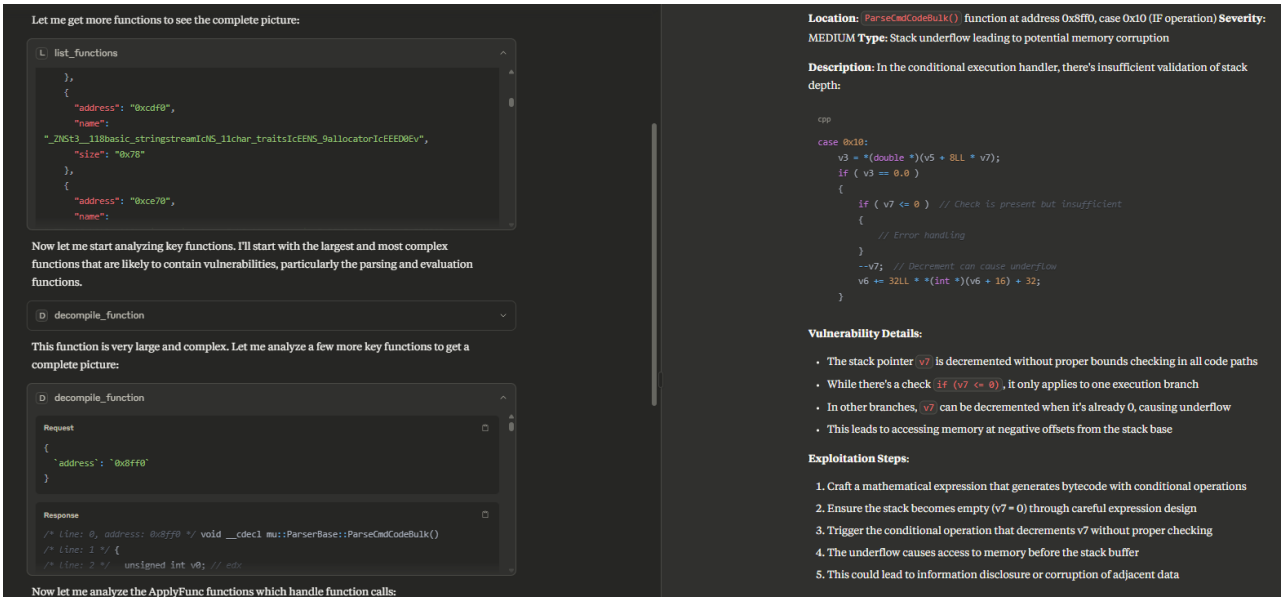


Рисунок 3.13 — Знайдена вразливість в бібліотеці `muparser`

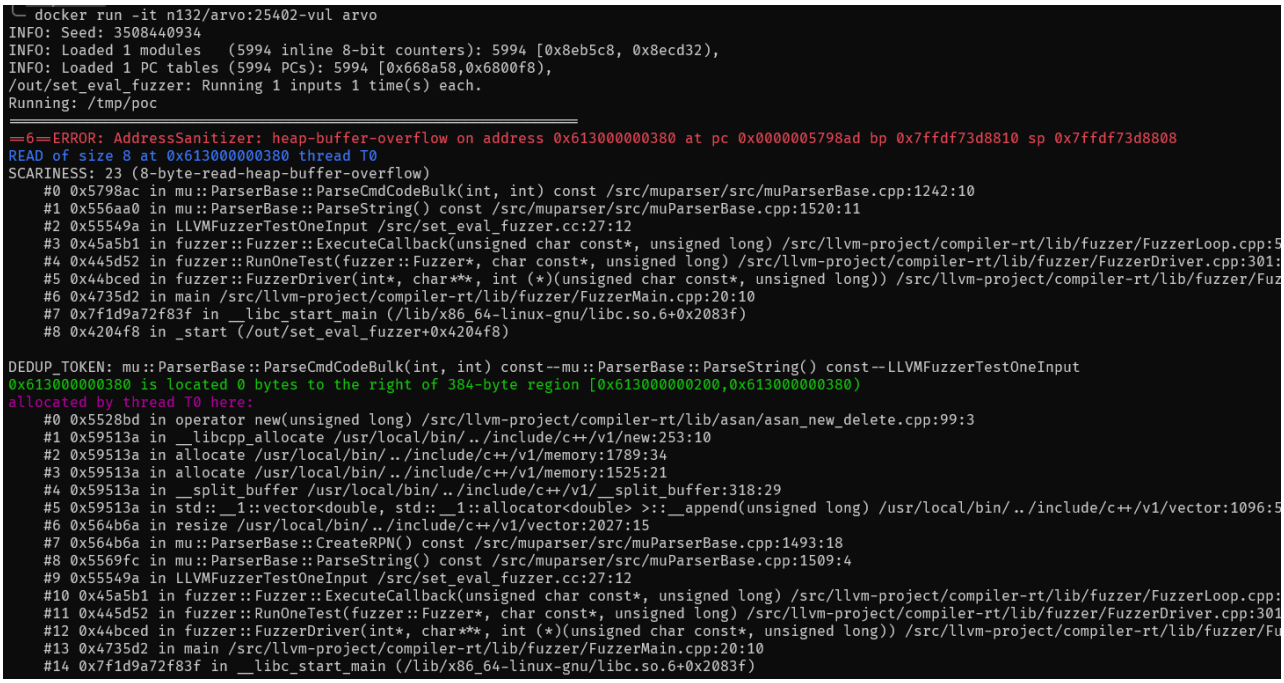


Рисунок 3.14 — Підтвердження деталей знайденої вразлививості за допомогою фреймворку `arvo`

цію що змогла знайти модель завдяки статичному аналізу коду.

3.3 Порівняльний аналіз та оцінка ефективності моделі

Оцінка ефективності запропонованої методології інтеграції LLM з декомпіляторами представляє значні виклики через стохастичну природу як LLM, так і методів фазингу з якими можна порівнювати. На відміну від детерміністичних статичних аналізаторів, результати роботи AI-агентів та фаззерів можуть варіюватися між запусками через різні фактори: температура моделі, випадковість генерації, контекст запиту, а у випадку фазингу – випадковість мутацій та стратегій тестування. Ця варіативність ускладнює пряме порівняння методів та вимагає специфічного підходу до метрик оцінки. Традиційні метрики точності (precision/recall) мають обмежену застосовність, оскільки як AI-агенти, так і фаззери не гарантують виявлення всіх вразливостей в одному запуску, але можуть знаходити різні вразливості при повторних спробах.

3.3.1 Метрики оцінки

Для порівняння введемо наступні метрики оцінювання ефективності:

- Detection Rate - відсоток вразливостей, виявлених хоча б в одному з трьох запусків
- False Positive Rate - кількість помилково виявлених вразливостей на один справжній знахід
- Середній час аналізу - час, необхідний для завершення аналізу одного бінарного файлу

Для об'єктивного порівняння було організовано тестування на контрольованому наборі з 15 бінарних файлів різної складності, що включає як штучні приклади з відомими вразливостями, так і реальні застосунки з документованими CVE. Відповідні приклади було наведено в попередній секції цього розділу.

Порівнювальні підходи включали:

- AFL++ - сучасний фаззер з покращеними стратегіями мутації
- IDA + LLM - інтеграція IDA Pro з Claude через MCP
- Binary Ninja + LLM - інтеграція Binary Ninja з Claude через MCP
- IDA + LLM + Extended Thinking - режим поглибленого аналізу з самоаналізом

3.3.2 Порівняння з традиційними методами

Розглянемо наступне порівняння ефективності виявлення вразливостей на заданих метриках:

Таблиця 3.1 — Порівняння ефективності методів виявлення вразливостей

Метод	**True positive (%)**	**False positive (%)**	**Середній час (хв)**
AFL++	93.3	0.0	40.4
IDA and LLM	80.0	0.16	3.5
Binary Ninja and LLM	66.6	0.3	4.2
IDA and LLM (Extended Thinking)	93.3	0.28	9.7

Порівняння різних підходів до виявлення вразливостей виявило суттєві відмінності у принципах роботи та практичній застосовності кожного методу. Режим Extended Thinking показав найвищу ефективність виявлення серед тестів з LLM, здатний знаходити практично всі закладені вразливості завдяки здатності моделі до самоаналізу та поглибленого дослідження коду. Цей підхід дозволяє AI-агенту переосмислювати свої висновки та досліджувати код під різними кутами, що призводить до більш ретельного аналізу. Стандартний підхід з використанням IDA та LLM демонструє високу ефективність виявлення завдяки якісній декомпіляції, яку забезпечує IDA Pro. Цей інструмент краще справляється з складними конструкціями C++, об'єктно-орієнтованими патернами та створює менше артефактів у псевдокоді порівняно з альтернативами. Binary Ninja у поєднанні з LLM показує

дещо нижчі результати через особливості декомпіляції, що призводить до більшої кількості помилкових спрацювань та пропуску деяких вразливостей. Фазинг демонструє стабільно високі показники виявлення та має унікальну перевагу у вигляді відсутності помилкових спрацювань - кожен краш вказує на реальну проблему у програмі. Однак цей, вже класчний, підхід має суттєві обмеження у швидкості роботи, особливо при аналізі великих проектів, де процес може тривати години або навіть дні. У контексті класифікації типів вразливостей фазинг неспроможний надати детальну інформацію, обмежуючись лише сигналом про наявність проблеми.

Найбільш кардинальна відмінність між LLM-підходами та фазингом проявляється у якості та корисності згенерованих звітів. AI-агенти здатні створювати комплексні звіти, що включають детальний опис знайденої вразливості, точну локацію у коді, готові скрипти для демонстрації експлуатації, оцінку критичності та конкретні рекомендації щодо усунення проблеми. Такі звіти містять також контекстуальну інформацію про умови виникнення вразливості та потенційні наслідки її експлуатації. На противагу цьому, фазинг надає лише краш дамп або сигнал про аварійне завершення програми, залишаючи аналітику завдання самостійно визначати причину проблеми, її тип та способи усунення. Це вимагає додаткових зусиль та часу від спеціалістів з безпеки для інтерпретації результатів та розробки відповідних заходів.

LLM-підходи демонструють оптимальну швидкість аналізу, що робить їх практичними для регулярного скринінгу великої кількості файлів та інтеграції у процеси безперервної інтеграції. Водночас, ці методи не позбавлені недоліків у вигляді можливих помилкових спрацювань та ризику пропуску деяких типів вразливостей, що вимагає додаткової верифікації результатів. У великих програмних проектах фазери потребують значно більше часу для досягнення адекватного покриття коду та виявлення вразливостей, але при достатньому часі виконання демонструють високу надійність результатів. LLM-підходи, навпаки, забезпечують швидкий аналіз але можуть не виявити всі існуючі вразливості за один прохід, особливо ті, що залежать від складних умов виконання або специфічних станів програми.

Висновки до розділу 3

В рамках експериментального дослідження було комплексно протестовано можливості інтеграції великих мовних моделей з декомпіляторами для задач зворотньої розробки та виявлення вразливостей у бінарному коді. Було експериментально підтверджено здатність AI-агентів до ефективного аналізу різних типів бінарних файлів, включаючи скомпільовані програми на C, C++ з використанням ООП, Rust, а також файли без символьної інформації. Система продемонструвала стабільну роботу з усіма типами файлів, забезпечуючи детальний аналіз структури та логіки виконання програм. Було проведено експериментальне дослідження можливостей виявлення програмних закладок та потенційно небезпечних функцій у драйверах LOLDrivers. Зокрема, успішно проаналізований драйвер BdApiUtil.sys з виявленням критичної вразливості довільного завершення процесів, що повністю співпадає з результатами експертного аналізу. Було експериментально встановлено, що AI-агенти здатні виявляти класичні типи вразливостей (переповнення буферів, use-after-free, format string) як у штучних тестових зразках, так і в реальних застосунках з документованими CVE. Успішно ідентифіковані вразливості у бібліотеках libexif (CVE-2020-0198) та muparser (OSV-2020-1349) підтверджують практичну застосовність методології. Були порівняні якісні характеристики різних методів аналізу. LLM-підходи демонструють унікальну перевагу у генерації комплексних звітів з детальним описом вразливостей, готовими PoC експлойтами та рекомендаціями щодо усунення, на відміну від традиційного фазингу, який обмежується лише сигналом про наявність проблеми. Було експериментально доведено, що інтеграція декомпіляторів з великими мовними моделями через MCP протокол забезпечує ефективну автоматизацію процесу аналізу безпеки програмного забезпечення з оптимальним співвідношенням швидкості та точності виявлення вразливостей. Результати підтверджують доцільність практичного впровадження запропонованої методології у процеси кібербезпеки.

ВИСНОВКИ

В роботі був розглянутий процес аналізу вразливостей програмного забезпечення на основі інтеграції великих мовних моделей з інструментами зворотної розробки. Визначено значення протоколу Model-Context-Protocol як стандартизованого інтерфейсу взаємодії між LLM та декомпіляторами для автоматизації процесу виявлення вразливостей. Розглянуті типові методи статичного аналізу бінарного коду на прикладі інтеграції з декомпіляторами IDA Pro та Binary Ninja. Процес аналізу вразливостей бінарного коду визначено комплексним та багатоетапним. Він вимагає від дослідника наявності специфічних знань та навичок, таких як вміння проводити аналіз та зворотну розробку бінарних файлів, розуміння архітектури процесорів та принципів роботи програм, а також досвіду роботи з декомпіляторами та налаштування великих мовних моделей.

Підтверджено актуальність удосконалення наявних моделей шляхом впровадження моделі автоматизованого аналізу вразливостей з використанням штучного інтелекту.

Запропоновано програмну реалізацію моделі автоматизованого процесу аналізу вразливостей програмного забезпечення на основі інтеграції великих мовних моделей з декомпіляторами через протокол MCP з використанням Claude Sonnet 4.0.

Модель містить наступні модулі: MCP-сервера з інтегрованими плагінами для декомпіляторів IDA Pro та Binary Ninja, MCP-мосту для забезпечення стандартизованої взаємодії між компонентами, MCP-клієнта Claude для високорівневого аналізу та інтерпретації результатів, системи промпт інженерії для формування ефективних запитів до LLM, користувацького інтерфейсу для управління процесом аналізу.

Наведені шаблони запитів та налаштувань для типових завдань аналізу вразливостей бінарного коду. Для забезпечення ефективної роботи запропоновані принципи створення промптів з урахуванням специфіки низькорівне-

вого коду та необхідності точної інтерпретації результатів. Експериментально підтверджено, що програмна реалізація запропонованої моделі інтеграції декомпіляторів з великими мовними моделями через протокол MCP успішно виявляє вразливості пошкодження пам'яті, включаючи stack-based buffer overflow, heap-based buffer overflow, use-after-free та integer overflow вразливості в тестових та реальних зразках програмного забезпечення. Було проведене експериментальне дослідження з визначення ефективності різних режимів роботи системи. Після вимірювання показників для різних конфігурацій було визначено, що режим Extended Thinking демонструє найвищу ефективність виявлення (93.3%) при середньому часі аналізу 9.7 хвилини. Стандартний режим IDA + LLM забезпечує оптимальне співвідношення швидкості (3.5 хвилини) та ефективності (80%) для регулярного використання. Розкид результатів виявився відносно незначним, що пояснюється стабільністю роботи LLM при правильному формуванні промптів. Запропоновано використовувати режим Extended Thinking для критично важливих завдань аналізу, враховуючи отримані результати співвідношення часу аналізу та якості виявлення вразливостей. Помічено, що система генерує комплексні звіти з детальним описом знайдених вразливостей, готовими PoC експлойтами та рекомендаціями щодо усунення, що значно підвищує практичну цінність результатів порівняно з традиційними методами статичного аналізу та фазінгу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Vector 35. Binary Ninja Reverse Engineering Platform. — Режим доступу: https://binary.ninja/.

2. National Security Agency. Ghidra Software Reverse Engineering Suite. — 2019. — Режим доступу: https://ghidra-sre.org/.

3. Radare2 Community. Radare2: Unix-like reverse engineering framework and command-line toolset. — Режим доступу: https://rada.re/n/.

4. Nethercote Nicholas, Seward Julian. Valgrind: A Suite of Tools for Debugging and Profiling Programs. — Режим доступу: https://valgrind.org/.

5. AddressSanitizer: A fast address sanity checker / Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitriy Vyukov // USENIX Annual Technical Conference. — 2012.

6. Fioraldi Andrea, Maier Dominik, Eißfeller Heiko, Cremona Marc. AFL++: The AFL++ Fuzzer. — 2020. — Режим доступу: https://aflplus.plus/.

7. LLVM Project. LibFuzzer: a library for coverage-guided fuzz testing. — Режим доступу: https://llvm.org/docs/LibFuzzer.html.

8. RunSafe Security. Memory Safety Vulnerabilities: The Most Common Binary Security Issues. — 2024. — Режим доступу: https://runsafesecurity.com/blog/memory-safety-vulnerabilities/.

9. One Aleph. Smashing the stack for fun and profit // Phrack magazine. — 1996. — Т. 7, № 49.

10. Sotirov Alexander. Heap Feng Shui in JavaScript. — Black Hat Europe. — 2007.

11. Blazakis Dion. Interpreter Exploitation: Pointer Inference and JIT Spraying. — Black Hat DC. — 2010.

12. Zhou Xin, Zhang Ting, Lo David. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead // ACM Transactions on Software Engineering and Methodology. — 2024. — arXiv:2404.02525.

13. When LLMs meet cybersecurity: a systematic literature review / Jian Zhang, Hao Bu, Hui Wen та ін. // Cybersecurity. — 2025. — Т. 8, № 1. — Режим доступу: https://cybersecurity.springeropen.com/articles/10.1186/s42400-025-00361-w.

14. Tmylla, contributors. Awesome-LLM4Cybersecurity: An overview of LLMs for cybersecurity. — 2024. — Режим доступу: https://github.com/tmylla/Awesome-LLM4Cybersecurity.

15. Anthropic. Claude Desktop Application. — 2024. — Режим доступу: https://claude.ai/download.

16. Cursor Team. Cursor: The AI-first Code Editor. — 2024. — Режим доступу: https://cursor.com.

17. Cline Team. Cline: AI Assistant for Developers. — 2024. — Режим доступу: https://cline.bot.

18. Sultana Sadaf, Afreen Shaila, Eisty Nasir U. Code Vulnerability Detection: A Comparative Analysis of Emerging Large Language Models. — 2024. — arXiv:2409.10490.

19. Tan Hanzhuo та ін. LLM4Decompile: Reverse Engineering: Decompiling Binary Code with Large Language Models. — 2024. — Режим доступу: https://github.com/albertan017/LLM4Decompile.

20. Hosseini Iman, Dolan-Gavitt Brendan. Beyond the C: Retargetable Decompilation using Neural Machine Translation // NDSS Binary Analysis

Research Workshop. — 2023. — Режим доступу: https://arxiv.org/abs/2212.08950.

21. Semantics-Recovering Decompilation through Neural Machine Translation / Ruohang Liang, Ying Cao, Ping Hu, Kai Chen // Inscrypt. — 2024. — Режим доступу: https://arxiv.org/abs/2112.15491.

22. Zhou Xin, Zhang Ting, Lo David. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions // Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results. — 2024. — arXiv:2401.15468.

23. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach / Haonan Li, Yu Hao, Yizhuo Zhai, Zhiyun Qian // International Conference on Software Engineering (ICSE). — 2024. — Режим доступу: https://dl.acm.org/doi/10.1145/3649828.

24. LLM-Powered Static Binary Taint Analysis / Puzhuo Liu, Yaowen Zheng, Kangkang Zenn та ін. // ACM Transactions on Software Engineering and Methodology. — 2024. — arXiv:2310.08275.

25. Hex-Rays. IDA Pro: The Interactive Disassembler. — Режим доступу: https://hex-rays.com/ida-pro/.

26. Anthropic. Model Context Protocol: Introduction. — 2024. — Режим доступу: https://modelcontextprotocol.io/introduction.

27. Anthropic. Model Context Protocol Servers. — Режим доступу: https://github.com/modelcontextprotocol/servers.

28. MTFuzz: fuzzing with a multi-task neural network / Dongdong She, Rahul Krishna, Lu Yan та ін. // Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. — 2020. — arXiv:2005.12392.

29. LOLDrivers Project. Living Off The Land Drivers Database. — 2024. — Режим доступу: https://www.loldrivers.io/.

30. MagicSword.io. LOLDrivers: Curated list of Windows drivers used by adversaries. — 2024. — Режим доступу: https://github.com/magicsword-io/LOLDrivers.

31. LOLDrivers Community. LOLDrivers Issue #204: Driver Analysis Discussion. — 2024. — Режим доступу: https://github.com/magicsword-io/LOLDrivers/issues/204.

32. Beltoforion. muParser: Fast Math Parser Library - README. — 2024. — Режим доступу: https://github.com/beltoforion/muparser/blob/master/README.md.

33. MITRE Corporation. CVE-2020-0198: libexif Integer Overflow. — 2020. — Режим доступу: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-0198.

34. Red Hat Inc. Red Hat Bugzilla: CVE-2020-0198 - libexif Integer Overflow. — 2020. — Режим доступу: https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2020-0198.

35. ARVO Project. ARVO-Meta: muparser Vulnerability Case 25402. — 2020. — Режим доступу: https://github.com/n132/ARVO-Meta/blob/main/meta/25402.json.

36. Vulert. Vulert Vulnerability Database: OSS-Fuzz muparser-11427. — 2024. — Режим доступу: https://vulert.com/vuln-db/oss-fuzz-muparser-11427.

37. Google OSV Database. OSV-2020-1349: muparser Integer Overflow. — 2020. — Режим доступу: https://osv.dev/vulnerability/OSV-2020-1349.