Politechnika Wrocławska
Wydział informatyki i Telekomunikacji

Kierunek: **Cyberbezpieczeństwo (CBE)**
Specjalność: **Bezpieczeństwo danych (CBD)**

PRACA DYPLOMOWA
INŻYNIERSKA

Zastosowanie dużych modeli językowych do wykrywania i naprawiania błędów bezpieczeństwa i podatności w kodzie aplikacji webowych

Patryk Fidler

Opiekun pracy
Dr. hab. inż. Maciej Piasecki

Słowa kluczowe: modele językowe, Sztuczna Inteligencja, statyczna analiza kodu

WROCŁAW 2024

STRESZCZENIE

Praca inżynierska zatytułowana "Zastosowanie dużych modeli językowych do wykrywania i naprawiania błędów bezpieczeństwa i podatności w kodzie aplikacji webowych" koncentruje się na zastosowaniu zaawansowanych modeli językowych, takich jak GPT-3.5, GPT-4, a w razie możliwości Falcon, do automatycznego wykrywania i naprawiania błędów bezpieczeństwa w kodzie oprogramowania i aplikacji webowych.

Motywacja tej pracy wynika z rosnącej roli dużych modeli językowych (LLM) w różnych dziedzinach, w tym w cyberbezpieczeństwie. W kontekście tych działań, badana jest możliwość wykorzystania tych modeli do wykrywania i naprawiania podatności takich jak XSS, SQL Injection, CSRF, Buffer Overflow i tym podobne.

Punktem wyjścia dla pracy dyplomowej jest artykuł napisany w 2021 roku "Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs?" - Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, Brendan Dolan-Gavitt (https://arxiv.org/pdf/2112.02125v1.pdf). Autorzy podkreślają znaczący potencjał tych modeli, a niniejsza praca ma na celu kontynuację tych badań i poszerzenie ich zakresu.

Planowane działania obejmują przygotowanie zbioru danych zawierającego podatne bazy kodu źródłowego, testowanie zdolności detekcji błędów przez modele językowe OpenAI, oraz porównanie tych wyników z istniejącymi rozwiązaniami, oferowanymi przez firmę Snyk.

Szczególny nacisk zostanie położony na wykorzystanie technik soft-prompting i in-context learning, które mogą pomóc w udoskonaleniu detekcji i wyników, nawet przy ograniczonych zasobach. Praca przewiduje również implementację autonomicznego agenta AI zdolnego do analizy kodu, wykonania testów bezpieczeństwa i podejmowania decyzji na podstawie wyników tych testów i kontekstu.

Opcjonalnie, badane będą możliwości detekcji błędów przez otwarte modele językowe, a w dalszej perspektywie, możliwości specjalizacji modeli w zakresie cyberbezpieczeństwa za pomocą fine-tuning'u. Wszystkie te działania mają na celu nie tylko zrozumienie, ale także poprawienie możliwości LLM w kontekście cyberbezpieczeństwa.

Głównym celem pracy jest zwiększenie świadomości na temat potencjału dużych

modeli językowych w cyberbezpieczeństwie oraz proponowanie praktycznych rozwiązań, które mogą pomóc programistom w tworzeniu bardziej bezpiecznych aplikacji.

ABSTRACT

The engineering thesis titled "Application of Large Language Models for Detecting and Fixing Security Bugs and Vulnerabilities in Web Application Code" focuses on the utilization of advanced language models, such as GPT-3.5, GPT-4, and if possible, Falcon, for automated detection and rectification of security bugs in the code of software and web applications.

The motivation for this work stems from the growing role of Large Language Models (LLMs) in various fields, including cybersecurity. In the context of these efforts, the possibility of using these models to detect and fix vulnerabilities such as XSS, SQL Injection, CSRF, Buffer Overflow, and the like is being investigated.

The starting point for this dissertation is the 2021 article "Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs?" by Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, Brendan Dolan-Gavitt (https://arxiv.org/pdf/2112.02125v1.pdf). The authors emphasize the significant potential of these models, and this work aims to continue this research and broaden its scope.

Planned activities include preparing a dataset containing vulnerable source code databases, testing the error detection capabilities of OpenAI language models, and comparing these results with existing solutions offered by Snyk.

Particular emphasis will be placed on the use of soft-prompting and in-context learning techniques, which can help improve detection and results, even with limited resources. The work also envisages the implementation of an autonomous AI agent capable of analyzing code, performing security tests, and making decisions based on the results of these tests and context.

Optionally, the possibilities of error detection by open language models will be explored, and in the longer term, the possibilities of specializing models in the field of cybersecurity through fine-tuning. All these actions aim not only to understand but also to improve the capabilities of LLMs in the context of cybersecurity.

The main goal of the work is to increase awareness of the potential of large language models in cybersecurity and to propose practical solutions that can help developers create more secure applications.

SPIS TREŚCI

WPROWADZENIE

Niniejsza praca inżynierska nosi tytuł "Zastosowanie dużych modeli językowych do wykrywania i naprawiania błędów bezpieczeństwa i podatności w kodzie aplikacji webowych". Celem tej pracy jest zbadanie, w jaki sposób zaawansowane modele językowe, takie jak GPT-3.5, GPT-4 oraz modele otwarto-źródłowe - Mistral 7B, Falcon-7B-instruct, mogą być wykorzystane do automatycznego wykrywania i naprawiania błędów bezpieczeństwa w kodzie oprogramowania i aplikacji webowych. W tym celu zostanie opracowane i zaimplementowane narzędzie do statycznej analizy kodu, które będzie wykorzystywać modele językowe do generowania kodu i naprawy błędów. Narzędzie to zostanie przetestowane i porównane z innymi rozwiązaniami, takimi jak Snyk, które oferują podobne funkcjonalności. W pracy zostaną przedstawione wyniki badań, które mają na celu odpowiedzieć na pytanie, czy modele językowe mogą być wykorzystane do tego celu, oraz jak skuteczne są one w porównaniu z innymi rozwiązaniami. W ramach pracy zostaną również zbadane ograniczenia i wyzwania związane z wykorzystaniem tych technologii w kontekście cyberbezpieczeństwa.

PYTANIA BADAWCZE

W ramach pracy stawiam następujące pytania badawcze:

1. Czy duże modele językowe mogą być wykorzystane do wykrywania i naprawiania błędów bezpieczeństwa w kodzie aplikacji webowych?
2. Jak skuteczne są te modele w porównaniu z innymi rozwiązaniami?
3. Czy komercyjne modele językowe znacznie różnią się od otwartych modeli?
4. Jakie są ograniczenia i wyzwania związane z wykorzystaniem tych technologii w kontekście cyberbezpieczeństwa?

HIPOTEZY

Hipotezy pracy to:

1. Duże modele językowe, dzięki swojej zdolności do analizy i generowania kodu, mogą skutecznie identyfikować i naprawiać błędy bezpieczeństwa w kodzie źródłowym.
2. Mimo obiecującego potencjału, modele te mogą napotykać ograniczenia, szczególnie w bardziej złożonych i specyficznych scenariuszach związanych z cyberbezpieczeństwem.

UZASADNIENIE TYTUŁU

Tytuł pracy został dobrany tak, aby odzwierciedlał główny obszar zainteresowania badawczego, jakim jest wykorzystanie nowoczesnych technologii językowych w celu poprawy bezpieczeństwa aplikacji webowych. W kontekście rosnącej zależności od cyfrowych rozwiązań, temat ten zyskuje na znaczeniu, oferując nowe perspektywy i podejścia do zagadnień bezpieczeństwa. Tytuł można skrócić do **"Zastosowanie dużych modeli językowych w statycznej analizie kodu"**, ponieważ tak nazywa się problem odnajdywania i korekcji błędów w kodzie źródłowym. Korpus badawczy pracy został rozszerzony o projekty open-source aplikacji natywnych i desktopowych oraz wycinki błędnego kodu i poprawnego kodu.

OMÓWIENIE LITERATURY NAUKOWEJ I STOPNIA JEJ PRZYDATNOŚCI

Podstawę teoretyczną pracy stanowi literatura naukowa skupiająca się na dużych modelach językowych oraz ich zastosowaniu w cyberbezpieczeństwie. Szczególną uwagę poświęcono artykułowi "Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs?", który posłużył jako punkt wyjścia dla badań. Praca ta ma na celu kontynuację i poszerzenie zakresu tych badań, wykorzystując literaturę naukową jako fundament do eksploracji nowych możliwości w zakresie analizy i naprawy błędów w kodzie. Różnica między pracą a literaturą naukową polega na tym, że praca skupia się na praktycznym zastosowaniu modeli językowych w celu wykrywania i naprawiania błędów bezpieczeństwa w kodzie, podczas gdy literatura naukowa skupia się na badaniu możliwości Sztucznej Inteligencji w tym zakresie.

CEL PRACY

Głównym celem pracy jest zbadanie możliwości wykorzystania dużych modeli językowych do wykrywania i naprawiania błędów bezpieczeństwa i podatności w kodzie źródłowym aplikacji webowych. W tym kontekście można wyróżnić następujące cele pośrednie:

— Opracowanie praktycznego rozwiązania do statycznej analizy kodu dla aplikacji webowych oraz lokalnych.
— Badanie skuteczności dużych modeli językowych w wykrywaniu podatności i luk bezpieczeństwa.

ZAKRES PRACY

Zakres pracy obejmuje:

— Analizę istniejącej literatury i badań, w szczególności artykułu ’Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs?’.
— Projektowanie i implementacja narzędzia do statycznej analizy kodu opartego na modelach OpenAI.
— Przygotowanie zbioru danych z kodem zawierającym potencjalne podatności.
— Testowanie i porównanie skuteczności z innymi rozwiązaniami, np. oferowanymi przez firmę Snyk.
— Analiza wyników i formułowanie wniosków.

1. ANALIZA ISTNIEJĄCEJ LITERATURY ORAZ DOTYCHCZASOWYCH BADAŃ

1.1. CAN OPENAI CODEX AND OTHER LARGE LANGUAGE MODELS HELP US FIX SECURITY BUGS?

1.1.1. Metodologia

W badaniu "Czy OpenAI Codex i inne duże modele językowe mogą pomóc nam naprawić błędy bezpieczeństwa?", autorzy skupili się na wykorzystaniu dużych modeli językowych (LLM) do naprawy podatności w kodzie w sposób zero-shot. Badanie koncentrowało się na projektowaniu monitów skłaniających LLM do generowania poprawionych wersji niebezpiecznego kodu. Przeprowadzono eksperymenty na szeroką skalę, obejmujące różne komercyjne modele LLM oraz lokalnie wytrenowany model.

1.1.2. Wyniki

Wyniki wykazały, że LLM mogą skutecznie naprawić 100% syntetycznie wygenerowanych scenariuszy oraz 58% podatności w historycznych błędach rzeczywistych projektów open-source. Odkryto, że różne sposoby formułowania informacji kluczowych w monitach wpływają na wyniki generowane przez modele. Zauważono, że wyższe temperatury generowania kodu przynoszą lepsze wyniki dla niektórych typów podatności, ale gorsze dla innych.

1.2. EXAMINING ZERO-SHOT VULNERABILITY REPAIR WITH LARGE LANGUAGE MODELS

W artykule "Examining Zero-Shot Vulnerability Repair with Large Language Models", autorzy kontynuowali badanie możliwości wykorzystania LLM do naprawy podatności w kodzie, koncentrując się na wyzwaniach związanych z generowaniem funkcjonalnie poprawnego kodu w rzeczywistych scenariuszach. Badanie to rozszerzało wcześniejsze prace, biorąc pod uwagę bardziej złożone przypadki użycia LLM.

Podstawowe pytania badawcze były następujące:

1. Czy LLM mogą generować bezpieczny i funkcjonalny kod do naprawy podatności?
2. Czy zmiana kontekstu w komentarzach wpływa na zdolność LLM do sugerowania poprawek?

3. Jakie są wyzwania przy używaniu LLM do naprawy podatności w rzeczywistym świecie?
4. Jak niezawodne są LLM w generowaniu napraw?

Eksperymenty potwierdziły, że choć LLM wykazują potencjał, ich zdolność do generowania funkcjonalnych napraw w rzeczywistych warunkach jest ograniczona. Wyzwania związane z inżynierią promptów i ograniczenia modeli wskazują na potrzebę dalszych badań i rozwoju w tej dziedzinie.

1.3. RÓŻNICE MIĘDZY OBECNĄ PRACĄ A ISTNIEJĄCĄ LITERATURĄ

W przeciwieństwie do dotychczasowych badań skoncentrowanych głównie na teoretycznym potencjale dużych modeli językowych (LLM) w kontekście zero-shot, niniejsza praca dyplomowa podejmuje kroki w kierunku praktycznego zastosowania tych technologii. Główną różnicą jest tutaj zastosowanie metod takich jak Retrieval Augmented Generation (RAG) oraz in-context learning, co przesuwa nasze podejście w stronę kontekstu few-shot.

— **Zastosowanie Metod RAG i In-context Learning:** W odróżnieniu od tradycyjnych podejść zero-shot, które polegają na generowaniu odpowiedzi bez uprzedniego dostosowania modelu do specyficznego zadania, moja praca wykorzystuje RAG i uczenie się w kontekście, aby lepiej dostosować modele do konkretnych scenariuszy związanych z bezpieczeństwem kodu. Te metody pozwalają na bardziej precyzyjną analizę i naprawę błędów w kodzie.
— **Praktyczne Zastosowanie Modeli Językowych:** Podczas gdy większość istniejących badań skupia się na badaniu możliwości SI w teorii, ta praca koncentruje się na praktycznym zastosowaniu modeli językowych do wykrywania i naprawiania błędów bezpieczeństwa w kodzie. Przez to podejście, praca ta dostarcza bezpośrednich, aplikatywnych rozwiązań, które mogą być wykorzystane w rzeczywistych środowiskach programistycznych.

Takie podejście pozwala nie tylko na zrozumienie teoretycznego potencjału LLM, ale także na ocenę ich praktycznej przydatności w realnych scenariuszach związanych z cyberbezpieczeństwem. Znacząco poszerza to zakres badań w dziedzinie wykorzystania sztucznej inteligencji do poprawy bezpieczeństwa aplikacji, dostarczając nowych perspektyw i rozwiązań.

2. METODYKA ROZWIĄZANIA

W niniejszej pracy dyplomowej zastosowano szereg metod i środków, aby zbadać i ocenić potencjał dużych modeli językowych w kontekście wykrywania i naprawiania błędów bezpieczeństwa w kodzie źródłowym aplikacji.

Metody:

— Algorytmy fragmentowania (własny kod)
— In-context learning (uczenie się w kontekście)
— Retrieval Augmented Generation - RAG (napisany własny kod, dostępny poprzez OpenAI Assistant API)
— Analiza porównawcza
— Programowanie obiektowe oraz funkcyjne

Środki:

— Modele językowe GPT-3.5, GPT-4
— Ollama - środowisko kontenerowe dla modeli językowych, łatwe w użyciu, otwartoźródłowe modele językowe
— OpenAI Assistant API
— Zbiór danych z kodem zawierającym podatności
— Statyczne testy podatności, np. CodeQL
— Istniejące rozwiązania komercyjne, np. Snyk
— Python 3.12
— Komputer osobisty

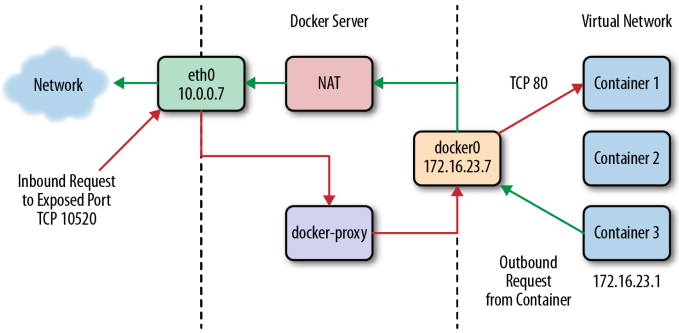
Metody i środki te zostały wybrane, aby zapewnić efektywne i wszechstronne podejście do analizy i naprawy kodu. Generacja wspomagana odzyskiwaniem danych (RAG ang. Retrieval Augmented Generation) oraz uczenie się w kontekście (in-context learning) umożliwiają efektywną analizę i generowanie kodu. Z kolei analiza porównawcza pozwala na ocenę skuteczności różnych modeli i podejść. Wykorzystanie modeli językowych GPT-3.5 i GPT-4, środowiska Ollama, oraz innych narzędzi i zasobów, zapewnia solidną bazę do przeprowadzenia kompleksowych testów i analiz.

2.1. RYSUNKI

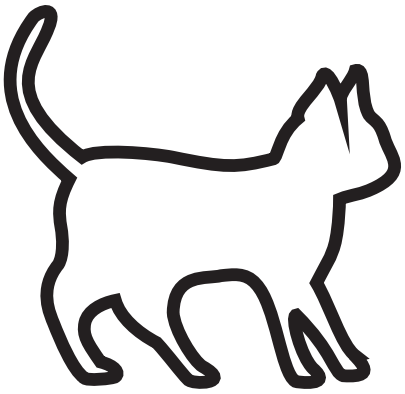
Na rysunku 2.1 …

2.1.1. Dwa rysunki obok siebie

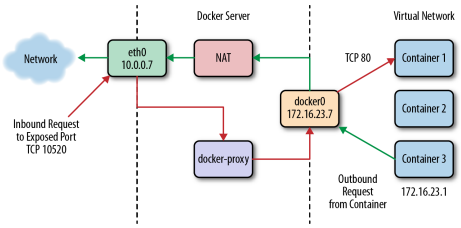
Na rysunkach 2.2 i 2.3 …



Rys. 2.1: Sieć dokera



Rys. 2.2: Lewy rysunek



Rys. 2.3: Prawy rysunek

2.2. TABELE

W tabeli 2.1 …

2.2.1. Równania

$$\sum_{i=1}^{\infty} a_i \tag{2.1}$$

W równaniu 2.1 …

2.3. LISTINGI

Na listingu 2.1 …

Tabela 2.1: Tytuł tabeli (patrz dodatek A)

Pierwszy	Drugi	Trzeci
Pierwszy	Drugi	Trzeci

```c
int main()
{
   int a=2*3;
   printf("**Ala ma kota\n**");
   while(!I2C_CheckEvent(I2C1, I2C_EVENT_MASTER_MODE_SELECT)); /* EV5 */
   return 0;
}
```

Listing 2.1: Język C

3. PROJEKT ORAZ IMPLEMENTACJA ROZWIĄZANIA

3.1. WSTĘP

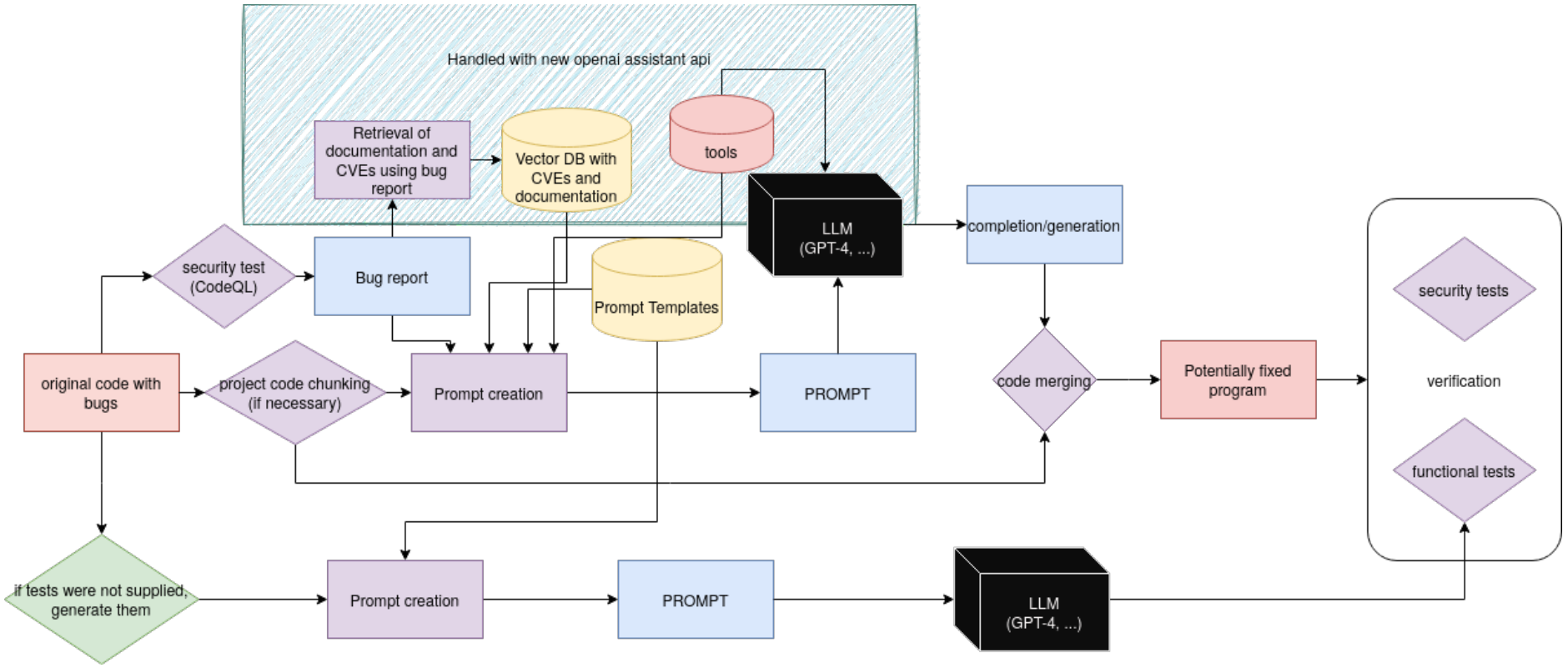
W ramach pracy inżynierskiej opracowano kompleksowe narzędzie ‘gptester‘, które wykorzystuje zaawansowane modele językowe do analizy statycznej kodu. Narzędzie to wykorzystuje przede wszystkim model GPT-4 do generowania raportów na temat jakości kodu oraz proponowania poprawek, ze szczególnym uwzględnieniem bezpieczeństwa kodu.

3.2. ARCHITEKTURA SYSTEMU

3.2.1. Ogólny opis

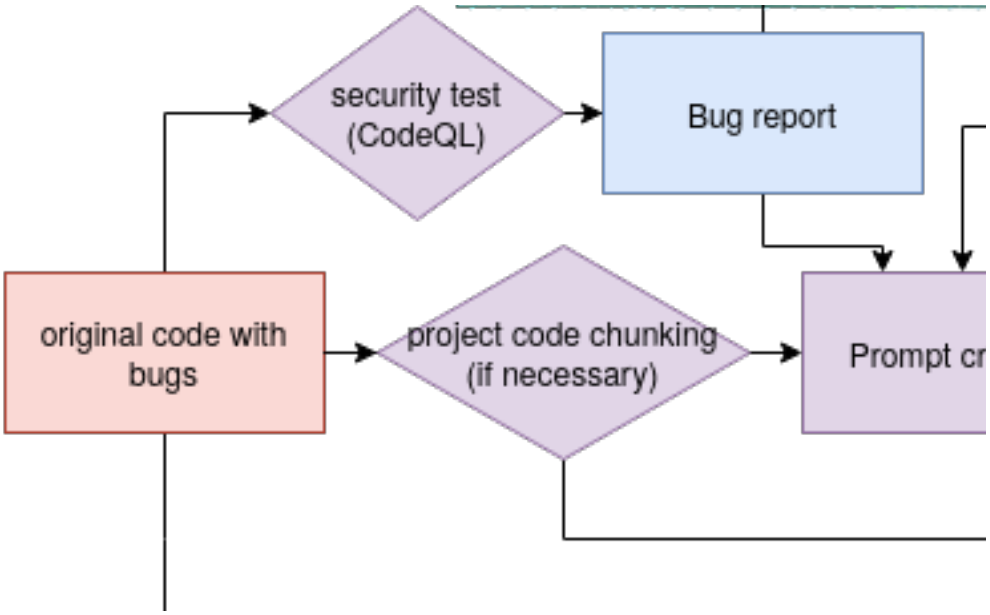
‘gptester‘ jest programem napisanym w języku Python, wykorzystującym model GPT-4 (lub GPT-3.5-turbo) dostarczony przez OpenAI. Jest zaprojektowany do uruchamiania z linii poleceń, a wyniki jego pracy są zapisywane w pliku formatu markdown oraz do osobnego katalogu z plikami wynikowymi - poprawionymi. W przyszłości planowane jest wprowadzanie poprawek do bazy kodu za pomocą git patch.

3.2.2. Schemat blokowy



Rys. 3.1: Schemat blokowy działania aplikacji ‘gptester‘

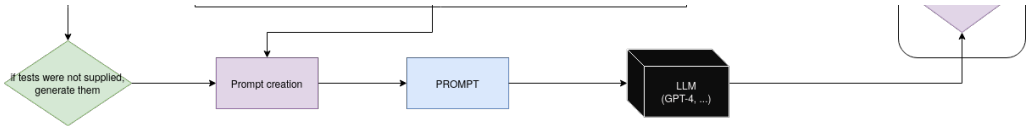
3.2.2.1. Dane wejściowe i wstępne przetwarzanie



Rys. 3.2: dane wejściowe w schemacie blokowym

Proces rozpoczyna się od ‘oryginalnego kodu z błędami‘. Następnie jeśli użytkownik dostarczył wystarczająco danych dla CodeQL zostanie ono wykorzystane do wstępnej analizy, a dane wynikowe zostaną wykorzystane do wzbogacenia monitów(promptów). W przeciwnym wypadku monit(prompt) zostanie skreowany na podstawie danych posiadanych.

3.2.2.2. Generacja testów funkcjonalnych

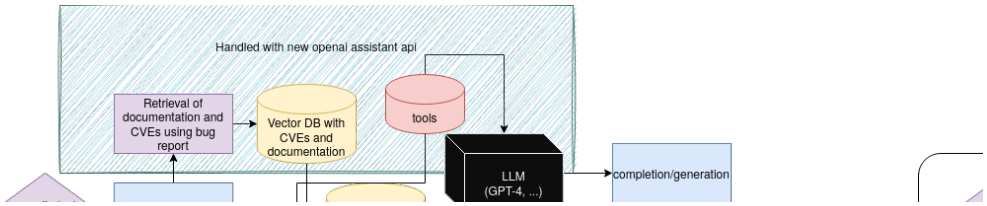


Rys. 3.3: Część schematu opisująca proces generatora testów funkcjonalnych

Jeżeli testy funkcjonalne dla naszego kodu nie zostały zapewnione, zostaną one wygenerowane za pomocą osobnego agenta.

3.2.3. Wzbogacanie monitów(promptów)

Dla uzyskania jak najlepszych wyników oraz zapewnienia najnowszej dostępnej wiedzy na temat podatności wykorzystywane zostaje RAG, aby wzbogacić monit o dodatkowe informacje. Są to informacje otrzymane z CodeQL, które zostają użyte do semantycznego wyszukania powiązanych wpisów w bazie danych CVE. Jeżeli CodeQL nie został użyty, monit zostaje wzbogacony o informacje z bazy CVE, które zostały wyszukane semantycznie w wektorowej bazie wiedzy. W tej chwili o użyciu dnaych z bazy wiedzy



Rys. 3.4: Część schematu opisująca proces RAG

decyduje wybrany model OpenAI, dzięki nowym możliwością API. Nowe możliwości API jak własne narzędzia dla LLM, code interpreter (interpreter kodu) oraz semantic search (semantyczne wyszukiwanie) zostały wprowadzone 06.11.2023r. Sprawia to, że niniejszy własny kod do wzbogacania monitów jest niepotrzebny, ale w przyszłości może zostać użyty do wzbogacania monitów o dodatkowe informacje dla modeli otwartoźródłowych.

```
def get_embedding(text, model="text-embedding-ada-002"): # alternatively
↪  use code-embedding-ada
    text = str(text)
    text = text.replace("\n", " ")
    if len(text) != 0:
        return openai.Embedding.create(input=[text],
        ↪  model=model)["data"][0]["embedding"]
    return [0.0] * 1536
```

Listing 3.1: Kod tworzący reprezentację wektorową tekstu za pomocą API OpenAI, domyślnie ‘text-embedding-ada-002’, (models.py)

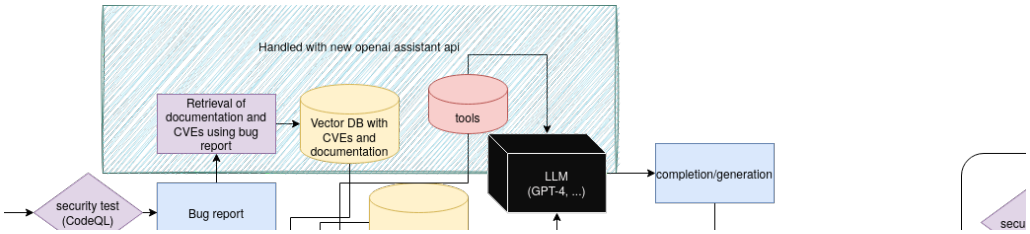
```
    def relevance_for(self, query: str) -> float:
        embedding = get_embedding(query)
        task = get_embedding(self.name)
        score = cosine(task, embedding)
        return score
```

Listing 3.2: Kod porównujący semantyczną odległość (models.py)

Przedstawione zostały rzeczywiste skrawki kodu znajdujące się w projekcie, natomiast w testach oraz podczas działania na modelach komeryjnych OpenAI używane są funkcje dostępne za pomocą API.

3.2.3.1. Tworzenie monitów i interakcja z LLM

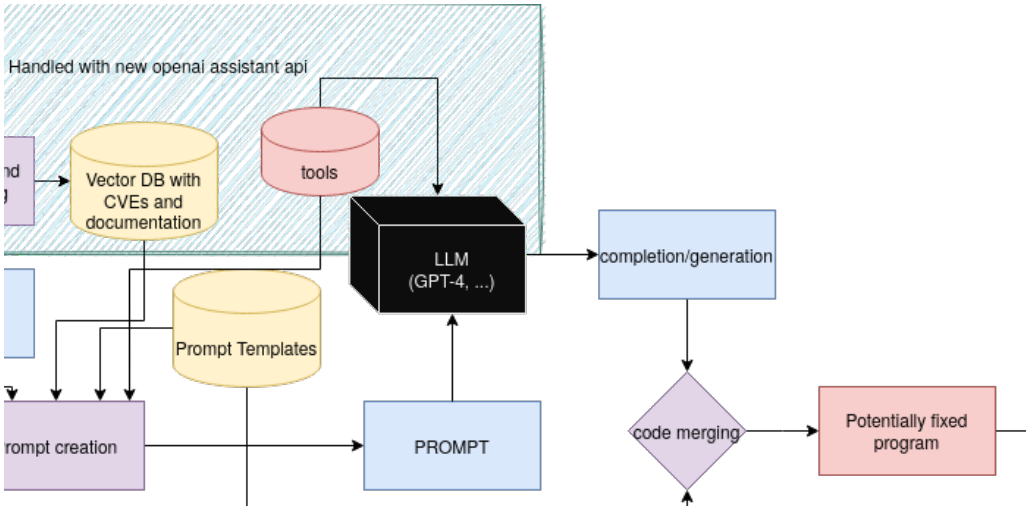
Dla każdej części bazy kodu opdowiednio mieszczącej się w oknie kontekstu dla modeli, ‘tworzony jest prompt (monit)‘, który jest następnie przetwarzany przez ‘duże



Rys. 3.5: Część schematu opisująca proces RAG

modele językowe (GPT-4, ...)‘. W tym celu wykorzywane są ‘szablony monitów‘ oraz semantycznie wyszukane skrawki z ‘bazy danych CVE‘. Tak spreparowane zapytanie jest następnie zadane modelowi językowemu, który identyfikuje podatności oraz proponuje poprawki.

3.2.3.2. Generowanie kodu i scalanie

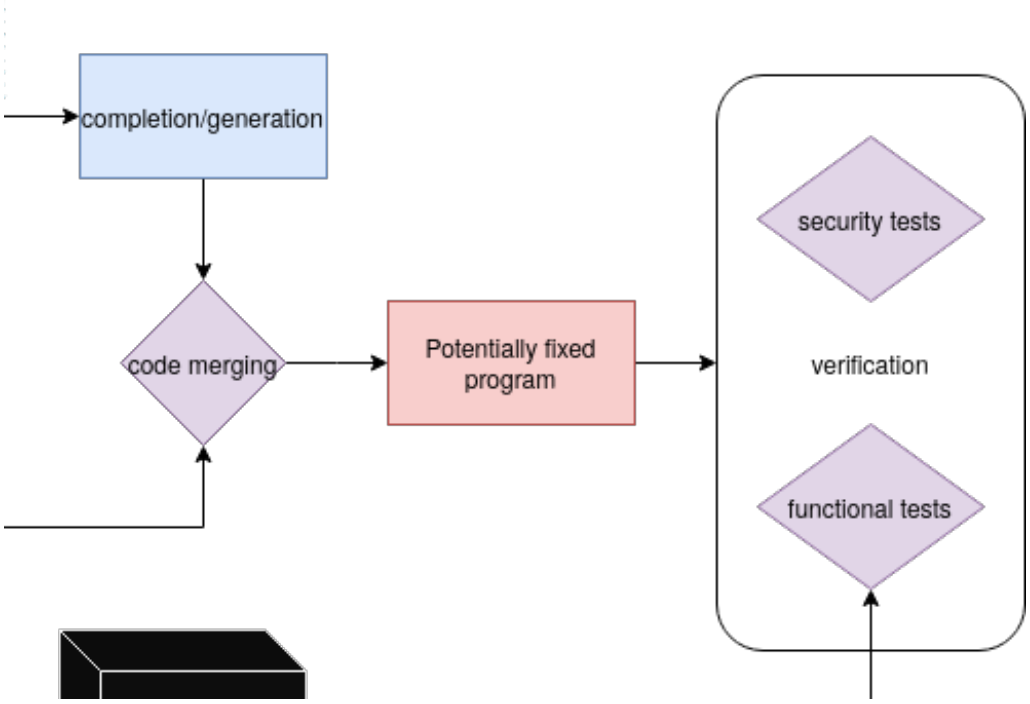


Rys. 3.6: Czarna skrzynka - LLM (Large Language Model)

LLM generuje ‘komplecje/generacje kodu’, które są następnie ‘łączone‘ w potencjalnie ‘naprawiony program‘. Proces ten wykorzystuje również ‘narzędzia‘(code interpreter, knowledge retrieval, file writing, git patch, ...), aby ułatwić obróbkę wyników oraz wzbogacić generację.

3.2.3.3. Testy i weryfikacja

Naprawiony kod jest poddawany ‘testom bezpieczeństwa‘ oraz ‘funkcjonalnym testom‘ w celu zapewnienia, że poprawki nie wprowadzają nowych błędów i że aplikacja nadal działa zgodnie z oczekiwaniami. Testy bezpieczeństwa są przeprowadzane ręcznie dla potwierdzenia poprawności działania, natomiast testy funkcjonalne są wykonywane automatycznie.



Rys. 3.7: Czarna skrzynka - LLM (Large Language Model)

3.2.3.4. Dokumentacja i raportowanie

Wyniki pracy ‘gptester‘ są dokumentowane w ‘raporcie o błędach‘ i zapisywane w plikach markdown oraz w katalogu z ‘poprawionymi plikami wynikowymi‘. W przyszłości ‘poprawki będą wprowadzane do bazy kodu za pomocą git patch‘.

3.2.3.5. Podsumowanie

Diagram blokowy przedstawia kompleksowy proces analizy i naprawy kodu, który jest silnie zależny od danych wejściowych (kod źródłowy i testy bezpieczeństwa), zaawansowanych algorytmów przetwarzania (duże modele językowe) oraz dokładności w generowaniu poprawek kodu i ich weryfikacji. Cały proces jest automatyzowany z możliwością ręcznej interwencji na etapie generowania monitów i weryfikacji kodu wynikowego.

3.3. IMPLEMENTACJA ORAZ UŻYCIE

3.3.1. Środowisko programistyczne i wymagania

Projekt ‘gptester‘ został opracowany w środowisku programistycznym Python, z wykorzystaniem modelu GPT-4 dostarczonego przez OpenAI. Proces konfiguracji środowiska rozpoczyna się od przygotowania odpowiedniego środowiska Pythona i zainstalowania niezbędnych zależności.

Wymagania wstępne:

— Python w wersji 3.x – Język programowania wykorzystany do napisania ‘gptester‘.

— Dostęp do internetu – Niezbędny do pobrania zależności i interakcji z modelem GPT-4 przez API OpenAI.

Instalacja zależności:

```
pip install -r requirements.txt
```

Plik ‘requirements.txt‘ zawiera wszystkie niezbędne biblioteki Pythona wymagane do działania ‘gptester‘. Instalacja zależności jest prosta i może być wykonana w terminalu lub wirtualnym środowisku Pythona, co jest zalecane w celu uniknięcia konfliktów z istniejącymi pakietami.

3.3.2. Funkcje programu

Program ‘gptester‘ został zaprojektowany jako wszechstronny narzędzie do analizy statycznej kodu, wykorzystując zaawansowane modele językowe do wykrywania i naprawiania błędów bezpieczeństwa w kodzie. Kluczowe funkcje programu są dostępne za pomocą różnorodnych argumentów linii poleceń, umożliwiając szeroką konfigurację i dostosowanie do specyicznych potrzeb analizy.

— **-h, –help**: Wyświetla pomoc programu, zawierającą informacje o dostępnych opcjach i ich krótki opis. Jest to przydatne dla użytkowników, którzy chcą szybko zrozumieć, jak korzystać z programu.
— **-v, –verbose**: Aktywuje tryb szczegółowych informacji. W tym trybie, ‘gptester‘ wyświetla dodatkowe informacje na temat każdego etapu przetwarzania, co jest przydatne do debugowania i analizy szczegółów wykonania.
— **-m MODEL, –model MODEL**: Umożliwia wybór modelu językowego używanego do analizy kodu. Domyślnie ustawiony na "gpt-4-1106-preview", ale użytkownik może wybrać inny model, jeśli jest dostępny i lepiej odpowiada wymaganiom projektu.
— **-o OUTPUT, –output OUTPUT**: Określa ścieżkę i nazwę pliku, do którego będą zapisane wyniki analizy. Domyślnie, raport jest zapisywany w pliku markdown w folderze "raports" z nazwą opartą na nazwie analizowanego folderu i znaczniku czasowym. Użytkownik może dostosować tę lokalizację według własnych preferencji.
— **-t TESTS, –tests TESTS**: Pozwala na podanie ścieżki do testów funkcjonalnych, które mają zostać wykonane na analizowanym projekcie. Ta funkcja jest szczególnie przydatna w środowiskach, gdzie istnieje potrzeba zintegrowanego podejścia do testowania i analizy kodu.
— **-c, –codeql**: Włącza integrację z CodeQL, zaawansowanym narzędziem do analizy kodu. Użytkownik musi mieć zainstalowane CodeQL-CLI, aby skorzystać z tej funkcji. Jest to szczególnie przydatne w wykrywaniu bardziej złożonych problemów w kodzie, które mogą umknąć prostym analizom.

— **–command COMMAND**: Umożliwia określenie polecenia budującego projekt, co jest niezbędne dla prawidłowej integracji z CodeQL w przypadku, gdy projekt nie zawiera pliku cmake lub podobnego w katalogu głównym. Domyślnie ustawione na "make".
— **–language LANGUAGE**: Pozwala na określenie języka programowania projektu do analizy w CodeQL. Domyślnie ustawione na "cpp", ale można dostosować do innych języków wspieranych przez CodeQL, co rozszerza możliwości analizy na różnorodne środowiska programistyczne.

Przykład użycia z pełną konfiguracją:

```
./main.py /ścieżka/do/projektu --verbose --model "gpt-4-1106-preview"
↪ --output "moj_raport.md" --tests "/ścieżka/do/testów" --codeql
↪ --command "cmake" --language "java"
```

W powyższym przykładzie, gptester analizuje kod znajdujący się w podanej ścieżce, z włączonym trybem szczegółowych informacji, korzystając z modelu GPT-4, zapisując wyniki do określonego pliku raportu, wykonując testy funkcjonalne, integrując z CodeQL, używając polecenia cmake do budowy projektu w języku Java.

3.4. INTEGRACJA Z CODEQL

Integracja ‘gptester‘ z CodeQL znacznie rozszerza jego funkcjonalność analizy statycznej kodu. CodeQL, opracowany przez GitHub, to zaawansowane narzędzie do semantycznej analizy kodu, które umożliwia wykrywanie złożonych podatności i błędów bezpieczeństwa.

Główne cechy integracji z CodeQL:

— **Zaawansowana Analiza Bezpieczeństwa**: CodeQL przekształca kod źródłowy w zapytywalną formę, co pozwala na przeprowadzenie głębokich analiz w poszukiwaniu subtelnych luk bezpieczeństwa.
— **Wsparcie Dla Wielu Języków**: Obsługa różnych języków programowania przez CodeQL, takich jak C++, Java, Python, co jest wykorzystywane przez ‘gptester‘ do analizy różnorodnych projektów.
— **Konfiguracja Procesu Budowania**: Możliwość dostosowania polecenia budowania projektu za pomocą opcji `-command`, niezbędna w przypadku braku pliku konfiguracyjnego jak cmake w katalogu głównym.
— **Elastyczność Analizy**: Użytkownik może wybrać między szybkimi analizami a bardziej dogłębnymi badaniami, co umożliwia dostosowanie procesu do konkretnych wymagań projektu.
— **Automatyzacja Wykrywania Podatności**: CodeQL automatyzuje proces wykrywania podatności, zwiększając skuteczność i efektywność analizy bezpieczeństwa kodu.

Integracja z CodeQL czyni ‘gptester‘ narzędziem nie tylko do wykrywania błędów syntaktycznych i strukturalnych, ale także do efektywnego identyfikowania subtelniejszych problemów bezpieczeństwa, które mogą umknąć podczas standardowych analiz.

3.5. PRZYPADKI UŻYCIA

Przypadki użycia programu ‘gptester‘ demonstrują jego wszechstronność i elastyczność w różnych scenariuszach analizy kodu. Poniżej przedstawiono kilka przykładowych scenariuszy, które ilustrują, jak ‘gptester‘ może być wykorzystywany do osiągnięcia różnych celów w procesie analizy statycznej kodu.

3.5.0.1. Podstawowa Analiza Kodu

W najprostszym przypadku, ‘gptester‘ może być używany do przeprowadzenia podstawowej analizy kodu w określonym katalogu. Przykład użycia:

```
./main.py /ścieżka/do/projektu
```

W tym scenariuszu, ‘gptester‘ przeanalizuje kod w podanym katalogu, używając domyślnych ustawień dla modelu językowego i zapisze raport w standardowej lokalizacji.

3.5.0.2. Analiza z Wysokim Poziomem Detali

Dla bardziej szczegółowych informacji, użytkownik może włączyć tryb verbose, co pozwoli na śledzenie szczegółowych informacji o każdym etapie analizy. Przykład użycia:

```
./main.py --verbose /ścieżka/do/projektu
```

W tym przypadku, ‘gptester‘ dostarczy rozszerzone informacje o przeprowadzanej analizie, co jest szczególnie przydatne do debugowania i zrozumienia procesu analizy.

3.5.0.3. Analiza z Integracją CodeQL

Dla zaawansowanej analizy bezpieczeństwa, ‘gptester‘ może być używany w połączeniu z CodeQL. Przykład użycia:

```
./main.py --codeql --language java /ścieżka/do/projektu
```

W tym scenariuszu, ‘gptester‘ wykorzystuje CodeQL do przeprowadzenia głębokiej analizy kodu w języku Java, co pozwala na wykrycie bardziej subtelnych i złożonych problemów bezpieczeństwa.

3.5.0.4. Analiza Projektu z Konkretnym Modelem Językowym

Użytkownik może również wybrać specyficzny model językowy dla analizy. Przykład użycia:

```
./main.py --model "gpt-4-1106-preview" /ścieżka/do/projektu
```

Taki wybór modelu może być użyteczny, gdy użytkownik chce skorzystać z określonych cech lub możliwości danego modelu, które mogą być lepiej dostosowane do charakterystyki analizowanego kodu. W przyszłości wybór zostanie poszerzony o modele otwartoźródłowe nieograniczone narzuconymi limitami OpenAI.

3.5.0.5. Kompleksowa Analiza z Wykorzystaniem Testów Funkcjonalnych

gptester pozwala na integrację testów funkcjonalnych, co może być użyteczne do bardziej kompleksowej analizy projektu. Przykład użycia:

```
./main.py --tests "/ścieżka/do/testów" /ścieżka/do/projektu
```

W tym przypadku, oprócz standardowej analizy kodu, gptester wykonuje również testy funkcjonalne, co pozwala na lepsze zrozumienie, jak zmiany w kodzie wpływają na działanie aplikacji.

3.5.0.6. Zaawansowana Konfiguracja dla Specyficznych Wymagań

Dla projektów o specyficznych wymaganiach, gptester oferuje szeroki zakres opcji konfiguracyjnych, pozwalając na dostosowanie analizy do indywidualnych potrzeb. Przykład użycia:

```
./main.py --verbose --model "gpt-4-1106-preview" --output "moj_raport.md" --tests
```

Tutaj, gptester jest konfigurowany do pracy z konkretnym modelem językowym, zapisuje raport w określonym pliku, wykonuje testy funkcjonalne, integruje z CodeQL, korzysta z określonego polecenia budowania projektu i jest dostosowany do analizy kodu w języku Java.

```
Przykład użycia podstawowego

cd gptester
python main.py -h
Przykład użycia z opcją verbose

./main.py --verbose
Przykład użycia z integracją CodeQL

./main.py --codeql --command "make" --language "cpp"
```

Listing 3.3: Przykłady użycia programu gptester

3.6. ROZWÓJ I PLANY NA PRZYSZŁOŚĆ

Sekcja ta skupia się na omówieniu obecnego stanu projektu ‘gptester‘ oraz planowanych rozszerzeń i ulepszeń, które mają zostać wprowadzone w przyszłości. Planowane działania są zgodne z informacjami zawartymi w sekcji "In development" w README.md.

3.6.1. Obecne osiągnięcia

Projekt ‘gptester‘ osiągnął już kilka kluczowych kamieni milowych w swoim rozwoju:

— **Podstawowa funkcjonalność**: Program już teraz oferuje podstawowe funkcje analizy statycznej kodu, umożliwiając identyfikację typowych błędów i podatności.
— **Wykorzystanie technik RAG oraz in-context learning**: ‘gptester‘ wykorzystuje zaawansowane techniki generacji wspomaganej odzyskiwaniem danych oraz uczenia się w kontekście, co pozwala na lepsze dostosowanie modeli językowych do specyficznych zadań.
— **Integracja z CodeQL**: Znaczącym osiągnięciem jest wdrożenie integracji z CodeQL, co znacznie rozszerza możliwości analizy kodu, szczególnie w zakresie wykrywania złożonych błędów bezpieczeństwa.

Te osiągnięcia stanowią solidną podstawę dla dalszego rozwoju i rozbudowy ‘gptester‘, kładąc nacisk na wydajność, dokładność i wszechstronność narzędzia.

3.6.2. Planowane rozszerzenia

W ramach dalszego rozwoju, projekt ‘gptester‘ ma w planach kilka istotnych rozszerzeń i ulepszeń:

— **Aktualizacja kodu za pomocą funkcji git i plików patch**: Rozwój funkcjonalności, która pozwoli na automatyczne wprowadzanie poprawek do kodu źródłowego na podstawie wygenerowanych plików git i patch. To ulepszenie ułatwi proces naprawy kodu, umożliwiając automatyczne aplikowanie poprawek oraz interaktywne wybieranie elementów z obu wersji.
— **Dodanie więcej testów oraz automatyzacja testów bezpieczeństwa**: Rozbudowa zestawu testów funkcjonalnych, jednostkowych i bezpieczeństwa, co pozwoli na lepsze sprawdzanie niezawodności i efektywności ‘gptester‘. Automatyzacja testów pozwoli na skuteczniejsze wykrywanie błędów oraz ułatwi badania.
— **Obsługa więcej języków programowania dla CodeQL**: Rozszerzenie integracji z CodeQL o więcej języków programowania, co zwiększy użyteczność ‘gptester‘ w różnorodnych projektach programistycznych. Planowane jest dodanie wsparcia dla popularnych języków takich jak JavaScript, Python czy Ruby.

Te planowane rozszerzenia mają na celu nie tylko ulepszenie obecnych funkcjonalności gptester, ale również wprowadzenie nowych możliwości, które uczynią narzędzie jeszcze bardziej wszechstronnym i przydatnym w różnych kontekstach analizy kodu.

3.7. PODSUMOWANIE

W niniejszym rozdziale przedstawiono szczegółowy opis projektu ‘gptester‘, jego obecne możliwości oraz plany rozwojowe. ‘gptester‘, jako zaawansowane narzędzie do analizy statycznej kodu, wykorzystujące model GPT-4 od OpenAI, stanowi znaczący krok naprzód w dziedzinie automatyzacji i poprawy jakości kodu źródłowego.

Podstawowe osiągnięcia:

— Rozwój podstawowych funkcjonalności analizy statycznej, umożliwiających efektywne wykrywanie i naprawianie błędów w kodzie.
— Integracja z CodeQL, dzięki której ‘gptester‘ zyskuje zdolność do przeprowadzania bardziej zaawansowanych analiz bezpieczeństwa.
— Elastyczność w obsłudze różnorodnych scenariuszy użytkowania poprzez konfigurowalne argumenty linii poleceń.

Plany rozwojowe:

— Rozbudowa funkcjonalności aktualizacji kodu źródłowego za pomocą plików git i patch, co uprości proces wprowadzania poprawek.
— Dodanie wsparcia dla dodatkowych języków programowania w integracji z CodeQL, co rozszerzy zakres zastosowania ‘gptester‘.
— Automatyzacja testów.

Podsumowując, ‘gptester‘ już teraz stanowi potężne narzędzie do analizy i poprawy kodu źródłowego, a planowane rozbudowy i ulepszenia sprawią, że będzie ono jeszcze bardziej wszechstronne i skuteczne. Projekt ten pokazuje, jak technologie AI i narzędzia do automatycznej analizy kodu mogą przyczynić się do poprawy jakości oprogramowania oraz efektywności procesów programistycznych.

4. ZBIORY DANYCH I ICH PRZYGOTOWANIE

W tym rozdziale skupimy się na omówieniu zbiorów danych wykorzystanych w projekcie ‘gptester‘, a także na procesie ich przygotowania i przetwarzania. Zbiory danych odgrywają kluczową rolę w procesie analizy statycznej kodu, pozwalając na dokładne testowanie i kalibrację narzędzia.

4.1. PRZEGLĄD WYKORZYSTANYCH ZBIORÓW DANYCH

Opisujemy tutaj źródła danych, ich charakterystykę oraz znaczenie dla projektu. Może to obejmować otwarte zbiory danych, repozytoria kodu, bazy danych podatności itp.

— **snoopysecurity/Vulnerable-Code-Snippets**: Repozytorium w serwisie Github zawierające zbiór fragmentów kodu zawierających luki w Internecie. Fragmenty pobrane z różnych wpisów na blogach, książek, zasobów itp. Zbiór w głównej mierze używany do testowania implementacji. Niektóre fragmenty kodu zawierają wskazówki w nazwach/komentarzach. Ewentualne naruszenie praw autorskich niezamierzone. `https://github.com/snoopysecurity/Vulnerable-Code-Snippets`
— **DiverseVul**: `https://arxiv.org/abs/2304.00409` Opis kolejnego zbioru danych, jego charakterystyka i zastosowanie w kontekście ‘gptester‘.
— **CVEfixes**: `https://github.com/secureIT-project/CVEfixes`

4.2. PROCES PRZYGOTOWANIA DANYCH

Tutaj opisujemy, jak zbiory danych zostały przygotowane do użycia w projekcie. To obejmuje procesy takie jak czyszczenie danych, transformacja, ewentualne wzbogacanie oraz metody ich selekcji.

4.2.1. Czyszczenie i normalizacja danych

Opisujemy, jakie kroki zostały podjęte, aby dane były spójne i wolne od błędów, które mogłyby wpłynąć na wyniki analizy.

4.2.2. Transformacja danych

Omawiamy wszelkie przekształcenia danych, które były konieczne, takie jak konwersja formatów, zmiana struktury danych itp.

4.2.3. Metody selekcji i wzbogacania danych

Opisujemy, jak wybrano dane do analizy i czy były one w jakiś sposób wzbogacane (np. poprzez dodanie etykiet, użycie dodatkowych źródeł danych).

4.3. WYZWANIA I OGRANICZENIA

W tej sekcji omawiamy wyzwania, które napotkano podczas pracy z danymi, oraz ograniczenia zbiorów danych, które mogą wpłynąć na wyniki projektu.

4.3.1. Problemy z jakością danych

Omawiamy wszelkie napotkane problemy z jakością danych i jak wpłynęły one na proces analizy.

4.3.2. Ograniczenia zbiorów danych

Opisujemy ograniczenia zbiorów danych, takie jak ich rozmiar, zakres, reprezentatywność itp., i jak mogą one wpłynąć na ogólne wnioski projektu.

4.4. PODSUMOWANIE

Podsumowujemy, jak przygotowanie i analiza zbiorów danych wpłynęła na projekt ‘gptester‘ i jakie wnioski można z tego wyciągnąć.

5. BADANIA EKSPERYMENTALNE - WYNIKI I WNIOSKI

W tym rozdziale omawiamy wyniki i wnioski z przeprowadzonych badań eksperymentalnych. Wszelkie wyniki powinny być opisane w sposób zrozumiały i czytelny, a także powinny być poparte odpowiednimi wykresami i tabelami. Wnioski powinny być wyciągnięte na podstawie wyników i powinny być zgodne z celami projektu.

5.1. BADANIA NA ZBIORZE *SNOOPYSECURITY/VULNERABLE-CODE-SNIPPETS*

PODSUMOWANIE

Curabitur tellus magna, porttitor a, commodo a, commodo in, tortor. Donec interdum. Praesent scelerisque. Maecenas posuere sodales odio. Vivamus metus lacus, varius quis, imperdiet quis, rhoncus a, turpis. Etiam ligula arcu, elementum a, venenatis quis, sollicitudin sed, metus. Donec nunc pede, tincidunt in, venenatis vitae, faucibus vel, nibh. Pellentesque wisi. Nullam malesuada. Morbi ut tellus ut pede tincidunt porta. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam congue neque id dolor.

BIBLIOGRAFIA

SPIS RYSUNKÓW

SPIS LISTINGÓW

SPIS TABEL

Dodatki

A. DODATEK 1

Nulla ac nisl. Nullam urna nulla, ullamcorper in, interdum sit amet, gravida ut, risus. Aenean ac enim. In luctus. Phasellus eu quam vitae turpis viverra pellentesque. Duis feugiat felis ut enim. Phasellus pharetra, sem id porttitor sodales, magna nunc aliquet nibh, nec blandit nisl mauris at pede. Suspendisse risus risus, lobortis eget, semper at, imperdiet sit amet, quam. Quisque scelerisque dapibus nibh. Nam enim. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nunc ut metus. Ut metus justo, auctor at, ultrices eu, sagittis ut, purus. Aliquam aliquam.