A

ML Project Report

Source Code Analyzer

Submitted in partial fulfillment of the requirements for the MCA degree

By

Rohan Das
23MCAN0218

Under the Guidance of

Dr. Anooja A

Academic Year
2024-2025

Table of Contents

Page Left Blank Intentionally

1. Introduction

1.1 Background

With the rapid advancement of digital technologies and the increasing dependence on software applications, security vulnerabilities in source code have become a critical concern for organizations and developers alike. Modern software applications are composed of thousands — sometimes millions — of lines of code, often developed by distributed teams or integrated from third-party components. In such large and heterogeneous codebases, even a minor oversight can introduce severe security flaws that attackers can exploit.

Traditional software development life cycles (SDLC) often failed to integrate security at early stages, leading to vulnerabilities being discovered only during production or post-deployment phases. This delayed detection increases the cost and complexity of remediation. Consequently, **Secure Software Development Life Cycle (SSDLC)** and **DevSecOps** practices have emerged, emphasizing the need for early detection and resolution of security risks during development itself.

However, manual code reviews are time-intensive and demand deep expertise in both programming and security. Even automated static analyzers, while efficient, often lack the semantic understanding of code, leading to false positives or missing contextual vulnerabilities. To bridge this gap, machine learning (ML) and artificial intelligence (AI) techniques are increasingly being used to augment traditional analysis methods. These approaches offer the ability to learn patterns from large datasets, enabling them to detect complex and context-sensitive vulnerabilities that might be invisible to conventional tools.

This project arises from the intersection of these needs — combining **pattern-based static analysis** with **machine learning-based detection** to build an effective, lightweight, and user-friendly tool for vulnerability analysis in source code.

1.2. Overview

The **Source Code Vulnerability Analyzer** is a hybrid analysis tool designed to detect vulnerabilities in source code written in multiple programming languages, including **Python, PHP, JavaScript, and Java**. This tool implements a two-pronged detection mechanism:

1. **Rule-based static analysis**: The tool uses regular expressions and keyword-matching techniques to identify suspicious function calls and unsafe coding practices, such as use of eval(), system(), unescaped user inputs, hardcoded credentials, and insecure cryptographic functions.

2. **Machine learning-based analysis**: A pre-trained transformer-based model (like BERT or CodeBERT) is used to analyze lines or blocks of code for more complex vulnerability patterns. This ML model is trained on labeled secure/insecure code snippets, allowing it to generalize and detect vulnerabilities beyond simple regex patterns.

The tool operates through a **command-line interface** with graphical file dialog support for ease of file selection. After analysis, results are displayed in a color-coded format in the terminal for readability and saved into structured result files for auditing or further review.

The goal of this project is not to replace existing tools but to present a lightweight, flexible, and extendable platform that combines traditional and modern techniques to help developers and security professionals identify insecure code during the development phase.

1.3. Problem Statement

In the modern software development landscape, ensuring the security of applications is a complex challenge. Developers are under constant pressure to deliver functional software quickly, which often leads to overlooked security best practices. Moreover, not all developers are trained in identifying security flaws. This creates a dangerous situation where applications are deployed with undetected vulnerabilities, leaving them exposed to threats such as injection attacks, remote code execution, insecure deserialization, and privilege escalation.

While static analysis tools exist, they often suffer from the following limitations:

- **Limited scope**: Many are restricted to specific programming languages.
- **False positives/negatives**: Pattern-matching without context can misidentify safe code as dangerous and vice versa.
- **Complex setup**: Industrial-grade tools like SonarQube may require server environments, database setup, and continuous integration configuration.
- **Inability to detect novel vulnerabilities**: Rule-based systems can only detect known patterns.

There is a growing need for a lightweight, extendable, and ML-enhanced source code analyzer that developers can easily integrate into their workflow. Such a tool should not only catch common coding mistakes but also flag potentially insecure logic based on learned patterns.

The **Source Code Vulnerability Analyzer** aims to address these gaps by delivering a hybrid detection system that balances speed, accuracy, and usability.

1.4. Objectives

The main objectives of this project are as follows:

1. **Develop a multi-language vulnerability analyzer** capable of scanning source code in Python, PHP, JavaScript, and Java.

2. **Design a static analysis engine** using rule-based logic (via regex and keyword checks) to detect insecure coding practices.

3. **Integrate a machine learning-based classifier** trained to identify insecure code snippets beyond simple syntactic rules.

4. **Provide a simple user interface** using a command-line menu and file dialog for accessibility.

5. **Ensure results are interactive and persistent** by displaying them in the terminal and saving them into a structured results directory.

1.5. Scope

The project focuses on **static analysis of source code** and does not execute or sandbox any scripts. The supported languages include **Python, PHP, JavaScript, and Java**, as they are commonly used in web and enterprise application development. The tool is designed to be **operating system agnostic**, running on both Linux and Windows environments where Python 3 is available.

The scope covers:

- Analysis of individual source code files selected by the user.

- Identification of high-risk functions and constructs (e.g., eval(), os.system(), direct SQL queries).

- ML-based classification of lines that may not match known patterns but still represent insecure logic.

- Terminal-based interaction and reporting without any GUI dependency.

Out of scope:

- Real-time monitoring of running code

- Dynamic (runtime) analysis or execution of uploaded files

- Deep code dependency analysis (e.g., full call graph construction)

1.6 Methodology

The system follows a **hybrid analysis methodology** combining both static rule-based techniques and ML classification.

- **Step 1: Language Selection & File Upload**
  The user launches menu.py and selects a language. A graphical file dialog appears to choose the source file.

- **Step 2: Pattern-Based Analysis**
  A predefined set of regular expressions is used to match vulnerable code patterns depending on the selected language.

- **Step 3: ML-Based Classification**
  Each code line is passed through a transformer model (e.g., BERT) trained on labeled datasets of secure and insecure code snippets.

- **Step 4: Output Generation**
  Results are color-coded in the terminal using colorama and saved to a timestamped file under the results/ folder.

This layered approach ensures robust vulnerability detection even for code that deviates from known patterns or structures.

2. Literature Review

2.1. Existing Tools

Numerous tools exist for source code analysis, each with its strengths and limitations. Among the most popular is **SonarQube**, an open-source platform that inspects code quality and security across multiple

languages. While powerful, SonarQube typically requires server setup and integration, which might not suit individual developers or students. Moreover, its rule base is often conservative, with limited AI capabilities.

Bandit is another well-known Python-specific static analyzer. It works by scanning Python source code and applying AST (Abstract Syntax Tree) analysis to identify common vulnerabilities like insecure subprocess usage or hardcoded passwords. While lightweight and fast, Bandit does not support multi-language analysis and lacks contextual ML intelligence.

Semgrep is a modern, open-source static analysis engine that supports multiple languages and uses a simplified query syntax for rule creation. It's flexible and fast but still rule-dependent. While Semgrep offers some pattern-matching beyond basic regex, it is not trained to detect context-based vulnerabilities that don't match expected patterns.

Other commercial tools such as **Checkmarx**, **Veracode**, and **Fortify** offer robust enterprise-level code scanning but are often closed-source, expensive, and require advanced setup. This highlights the need for a simple yet intelligent tool that combines static analysis with ML.

2.2. Related Research Papers / ML-Based Tools

Recent advancements in ML for code understanding have inspired tools like **CodeBERT**, **GraphCodeBERT**, and **DeepCode**. These models use transformer-based architectures trained on massive datasets of source code to perform tasks like summarization, completion, and classification. While primarily used for enhancing developer productivity, their utility in security auditing is increasingly being explored.

In the paper *"Learning to Represent Programs with Graphs" (Allamanis et al., ICLR)*, the authors demonstrate how graph neural networks (GNNs) can model code semantics better than traditional token-based approaches. Similarly, *VulDeePecker* introduces a deep learning-based vulnerability detection framework that uses code gadgets and BiLSTMs to detect security bugs.

ML-based tools like **ML4Sec**, **BabelFish**, and **SAVI** show promising results in identifying insecure patterns in code by leveraging contextual understanding. However, many of these tools are still research-oriented or require significant compute resources to operate efficiently, which is why integrating light ML models into lightweight tools (like this project) bridges the gap between academia and usability.

2.3. Limitations of Current Systems

Despite their usefulness, existing tools have several critical limitations. Most static analyzers are:

- **Language-specific**, requiring different tools for different codebases.
- **Rule-dependent**, meaning they can only catch what they are explicitly told to.
- **Hard to configure**, especially for non-security experts or students.
- **Limited in context**, failing to understand how a piece of code interacts with other components or variables.
- **Prone to high false positives**, leading developers to ignore alerts.

ML-based tools, on the other hand, are often:

- **Resource-intensive**, requiring GPU or large RAM for model inference.
- **Difficult to interpret**, as black-box models don't always provide understandable reasoning.
- **Not well integrated into pipelines**, lacking the simplicity of static scanners.

These limitations underscore the importance of a hybrid tool that is both **lightweight and intelligent**, combining the speed and predictability of static analysis with the adaptability and contextual awareness of ML.

2.4. Justification for This Project

This project aims to address the gap between traditional static analyzers and modern intelligent detection systems. By using a **hybrid architecture**, this analyzer combines the best of both worlds—speed and accuracy of rule-based analysis and contextual depth of machine learning models.

Supporting multiple languages in a single tool simplifies the workflow for developers working in heterogeneous environments. Furthermore, the lightweight nature of the tool ensures that it can run on most systems without requiring GPU acceleration or high-end infrastructure.

This project also contributes to the growing field of **secure coding tools for education and research**, as it is open-source, easy to extend, and well-documented. Its modular architecture allows future integration

with CI/CD pipelines, GitHub Actions, and VSCode extensions, making it scalable beyond academic or personal use.

By democratizing access to intelligent code security tools, this project empowers a broader audience to write secure software and proactively manage security debt in their codebases.

3. System Requirements

3.1 Hardware Requirements

To ensure seamless execution and analysis, the Source Code Vulnerability Analyzer project requires a system configuration that can handle machine learning model inference, GUI-based file selection, and real-time code scanning. The hardware requirements are moderately demanding due to the use of libraries such as PyTorch and Transformers.

Minimum Requirements:

- **Processor**: Intel Core i3 or AMD Ryzen 3 (dual-core)
- **RAM**: 4 GB
- **Storage**: At least 500 MB of free disk space
- **Display**: 1024x768 resolution or higher (to support GUI file dialogs)

Recommended Requirements:

- **Processor**: Intel Core i5/i7 or AMD Ryzen 5/7 (quad-core or higher)
- **RAM**: 8 GB or more (for faster ML model processing and handling larger code files)
- **Storage**: SSD with 2 GB free space (to support dependencies and result files)
- **Graphics**: Integrated GPU is sufficient, though a discrete GPU can slightly improve performance when working with large models

A better hardware setup leads to faster vulnerability scanning, especially when processing large codebases or multiple languages simultaneously.

3.2. Software Requirements

The software requirements are centered around supporting the Python environment and necessary libraries. Since the tool includes a graphical interface (using tkinter), command-line interaction, and model-based prediction systems, compatibility with modern operating systems and Python runtimes is essential.

Operating System:

- Linux (Ubuntu, Debian, Kali, Arch)
- Windows 10 or later (Home, Pro, Enterprise)
- macOS 10.14 (Mojave) or later

Other Software:

- Python 3.8 or above
- pip (Python package manager)
- Git (for cloning and updating repositories)
- IDE or text editor (e.g., VSCode, PyCharm, or Jupyter Notebook)

The system must support Python GUI applications, meaning it should have tkinter pre-installed or installable via system package managers.

3.3. Python Dependencies

The core functionality of the project relies on several third-party Python libraries, most of which are specified in a requirements.txt file included in the project folder. These libraries serve different purposes — from machine learning and model inference to GUI handling and color-coded terminal outputs.

Dependencies in requirements.txt:

numpy
torch
transformers
scikit-learn
colorama

Explanation:

- **numpy**: For numerical operations and array handling.
- **torch**: Core ML library (PyTorch) used to load and run the pre-trained models (.pkl files).
- **transformers**: Used for leveraging advanced language models for code understanding (e.g., BERT, CodeBERT).
- **scikit-learn**: Assists in preprocessing and supporting ML model structures (especially classification tasks).
- **colorama**: Enables colored output in the terminal, enhancing the readability of scan results.

All these dependencies are lightweight and can be installed with:

pip install -r requirements.txt

This file ensures that the development and production environments remain consistent and dependency management is simplified.

3.4. Platform Compatibility

One of the strong design goals of the Source Code Vulnerability Analyzer is its **cross-platform compatibility**. The tool is built in pure Python and only relies on standard Python GUI libraries (tkinter) and cross-platform ML packages (torch, transformers), making it runnable on all major operating systems.

Supported Platforms:

- **Linux**: Fully supported. Ideal for developers using Ubuntu, Kali, or other Debian-based distributions. Linux offers superior performance for Python and machine learning workloads.

- **Windows**: Fully supported. The tool runs via the Command Prompt or PowerShell, and tkinter dialogs are native on Windows systems.

- **macOS**: Supported with minimal configuration. Users may need to install tkinter via Homebrew or ensure Python from the official website includes GUI support.

Because it avoids OS-specific bindings or GUI frameworks like PyQt, the tool maintains portability and ease of use across environments. Additionally, Jupyter Notebook files for each language (python_02.ipynb, php_02.ipynb, etc.) allow execution and retraining in Google Colab, which is OS-independent.

4. System Design & Architecture

4.1. High-level Design

The **high-level design** of the Source Code Vulnerability Analyzer system emphasizes a modular and flexible structure that allows for easy integration of new programming languages, machine learning models, and additional features. The design also aims to be scalable and maintainable for future updates or extensions.

The system is structured into **four primary modules**: File I/O and Selection, Pattern-Matching Engine, ML Model Pipeline, and Output and Reporting. Each of these modules works in tandem to perform specific tasks, from reading source code files to reporting vulnerabilities detected by the machine learning models.

At its core, the system uses a **client-server architecture**. The client interface (built with **tkinter** for GUI and CLI) allows users to upload code snippets, select the language for scanning, and display results. Meanwhile, the backend (comprising ML pipelines and pattern-matching engines) processes the uploaded code, analyzes it for vulnerabilities, and returns results to the user.

The system is designed to handle **multiple programming languages**, enabling the analysis of Python, Java, JavaScript, and PHP code. Each language is managed in its own module, allowing for easy customization and updates.

4.2. Modules

The Source Code Vulnerability Analyzer system is divided into the following modules:

4.2.1. File I/O and Selection

The **File I/O and Selection** module is the first point of interaction for the user. It allows the user to upload source code files that need to be analyzed for vulnerabilities. The module leverages the **tkinter** library to create a **file dialog box** for selecting files based on their extension, which is matched to the programming language (Python, Java, JavaScript, or PHP).

Functionality:

- **File Upload**: Users can upload files from their local system using the **file dialog box**. Once a file is selected, the system reads the content of the file and forwards it to the analysis pipeline.
- **Input Validation**: This module ensures that only files with valid extensions (e.g., .py, .java, .js, .php) are accepted.
- **Error Handling**: In case of incorrect file types or errors during the file reading process, the module displays appropriate error messages.

The file upload mechanism forms the basis for interacting with the rest of the system and triggers the core analysis process.

4.2.2. Pattern-Matching Engine

The **Pattern-Matching Engine** is responsible for identifying potential vulnerabilities by searching for predefined code patterns that are commonly associated with security flaws. This engine scans the uploaded source code to detect insecure coding practices that can lead to vulnerabilities.

Functionality:

- **Static Code Analysis**: The engine uses regular expressions or pattern-matching techniques to find hardcoded passwords, improper data validation, unsafe function calls, and other potential vulnerabilities within the code.
- **Heuristic Analysis**: In addition to pattern-matching, the engine may also include heuristic methods for detecting suspicious code constructs that are not explicitly defined in a pattern library but may indicate potential security issues.
- **Modular Patterns**: The patterns are organized by language and can be easily updated to incorporate new security threats or to improve detection capabilities.

This module performs an essential role by providing a quick preliminary scan before the more sophisticated machine learning models are applied.

4.2.3. ML Model Pipeline

The **ML Model Pipeline** is the core of the Source Code Vulnerability Analyzer. It uses machine learning models trained on code samples to predict and classify vulnerabilities in the uploaded source code. These models were developed using **Python-based frameworks** such as **PyTorch** and **Transformers**. Each programming language (Python, Java, JavaScript, PHP) has its own corresponding model trained on specific data related to that language.

Functionality:

- **Preprocessing**: The uploaded code is preprocessed into a format suitable for feeding into the ML model (e.g., tokenizing the source code, vectorizing variables, and removing unnecessary noise).
- **Model Inference**: Once the code is preprocessed, it is passed to the trained model, which outputs a prediction regarding the security of the code. The models are capable of detecting a variety of vulnerabilities, such as buffer overflows, SQL injection, and cross-site scripting (XSS).
- **Post-Processing**: The output of the ML models is analyzed and filtered for false positives. The model also provides additional context about the type of vulnerability detected (e.g., SQL injection, insecure dependency).
- **Model Training**: The pipeline also supports the future enhancement of models by enabling the addition of new training datasets to improve accuracy and coverage.

By leveraging machine learning, this module provides deep insight into the code's security posture, going beyond simple pattern matching and recognizing more complex vulnerabilities.

4.2.4. Output and Reporting

The **Output and Reporting** module is responsible for displaying the results of the vulnerability analysis in a user-friendly manner. It is also tasked with generating a comprehensive report that outlines the vulnerabilities detected and their respective severity levels.

Functionality:

- **Display Results**: After code analysis, the results are presented in a **color-coded terminal output** (using the **colorama** library) or **GUI popups**. The vulnerabilities are marked with their types and locations in the code.
- **Detailed Report**: A detailed report is generated, including the following:
    - **Vulnerable Code Snippets**: The lines of code containing potential security issues.

    - **Severity Levels**: The severity of each vulnerability (e.g., critical, high, medium, low).
    - **Suggested Fixes**: Recommendations or best practices for fixing the vulnerabilities.
- **Save to File**: The report is saved in a text file within the **results folder** on the user's local machine for future reference.

This module ensures that the output from the analysis is not only informative but also actionable, providing the user with both detailed information and suggestions for remediation.

4.3. Architecture

The **Architecture** represents the entire system and how the modules interact. It is a high-level overview that provides clarity on the flow of data and processes. The key components include:

1. **User Interface (Tkinter)**: The user interacts with the system via a GUI for file upload and analysis.
2. **File I/O Module**: Accepts and processes the uploaded code.
3. **Pattern-Matching Engine**: Performs a preliminary scan for known vulnerabilities.
4. **Machine Learning Pipeline**: Analyzes the code using a trained model and predicts vulnerabilities.
5. **Output & Reporting**: Displays results and generates detailed reports.
6. **Results Folder**: Stores the generated reports.

4.4. Workflow

The **Workflow** depicts the step-by-step process that occurs from when the user uploads the code to when the results are displayed and saved. The following steps occur in sequence:

1. **File Upload**: The user selects a file to upload.

2. **File Validation**: The system checks if the file extension is valid.
3. **Pattern-Matching**: The code is scanned for common vulnerabilities using predefined patterns.
4. **Machine Learning Analysis**: The code is passed through the ML model pipeline for deeper analysis.
5. **Results Generation**: The results of the analysis are displayed, and a report is created.
6. **Save Results**: The report is saved in the **results folder** for future reference.

5. Implementation Details

5.1. menu.py

The **menu.py** file serves as the **entry point** for interacting with the Source Code Vulnerability Analyzer. This module is designed to handle user inputs, initiate the analysis workflow, and display the results in both GUI and terminal formats. The main function of this file is to allow seamless interaction between the user and the backend system, guiding them through the process of selecting files, running analysis, and receiving output.

Structure:

- **Imports**: The script imports libraries like tkinter for the GUI, and other core libraries such as os, sys, pathlib, and shutil for file manipulation and system management.
- **Main Menu Functionality**:

  The main function in menu.py presents the user with an option to either run the code vulnerability scan via a GUI or CLI. Based on the user's choice, the script will either open a **file dialog** to select source code or read from the terminal input.

- **User Input Handling**:

  The system provides a **menu-driven interface** (either GUI or CLI) to guide the user through various steps. The menu might offer options like:

  - **Start Scan**: Triggers the vulnerability scanning process.
  - **Select File**: Opens a file dialog for the user to select the code file.
  - **Exit**: Exits the program.

- **Integration with Other Modules**:

  menu.py orchestrates the interaction with the **Regex Pattern Matching Engine**, the **ML Model**, and the **Rule-based Engine** by calling their respective functions based on the user's selection. This ensures the code is processed step-by-step, from file upload to results display.

Code Snippet:

```
#!/usr/bin/env python
# coding: utf-8

import sys
import os
import tkinter as tk
from tkinter import filedialog
from tkinter import messagebox
from colorama import init, Fore, Style
import platform

# Initialize colorama
init(autoreset=True)

# Add the languages directory to the Python path
sys.path.append(os.path.join(os.path.dirname(__file__), 'languages'))

def get_analyze_code_function(language):
    try:
        if language == 'python':
            from python import analyze_code
        elif language == 'php':
            from php import analyze_code
        elif language == 'javascript':
            from javascript import analyze_code
        elif language == 'java':
            from java import analyze_code
        else:
            raise ValueError("Unsupported language")
        return analyze_code
    except ImportError as e:
        print(f"{Fore.RED}Import error: {e}")
        raise
    except Exception as e:
        print(f"{Fore.RED}Error: {e}")
        raise

def save_results(file_path, results):
    # Ensure the results directory exists
    results_dir = os.path.join(os.path.dirname(__file__), 'results')
    os.makedirs(results_dir, exist_ok=True)

    # Create a results file with the same name as the uploaded file
    base_name = os.path.basename(file_path)
    result_file_name = f"{base_name.replace('.', '_')}_results.txt"
    result_file_path = os.path.join(results_dir, result_file_name)

    with open(result_file_path, 'w') as result_file:
        result_file.write(results)
```

```
    return result_file_path

def upload_and_analyze(language):
    root = tk.Tk()
    root.withdraw()  # Hide the root window

    file_types = {
        'python': [("Python Files", "*.py")],
        'php': [("PHP Files", "*.php")],
        'javascript': [("JavaScript Files", "*.js")],
        'java': [("Java Files", "*.java")]
    }

    file_path = filedialog.askopenfilename(
        title=f"Select a {language.capitalize()} Script",
        filetypes=file_types[language]
    )

    if file_path:
        try:
            # Read the contents of the uploaded file
            with open(file_path, 'r') as file:
                code = file.read()

            # Get the appropriate analyze_code function
            analyze_code = get_analyze_code_function(language)

            # Analyze the code snippet
            unsafe_lines, prediction = analyze_code(code)



            # Prepare results
                result_message = f"\n\n{Fore.YELLOW}[!] Unsafe Lines of Code with Vulnerability Types:
{Style.RESET_ALL}\n\n"
            if not unsafe_lines:
                result_message += f"[!]\n{Fore.GREEN} No vulnerabilities found.{Style.RESET_ALL}\n"
            else:
                for line, vuln_type in unsafe_lines:
                    result_message += f"{Fore.RED}{line} - {vuln_type}{Style.RESET_ALL}\n"

            # Save results to file
            result_file_path = save_results(file_path, result_message)

            # Display results
            print("\n")
            print(result_message)
            print(f"\n{Fore.CYAN}Results have been saved to {result_file_path}{Style.RESET_ALL}")

            input(f"\n{Fore.BLUE}[*] Press Enter to continue...{Style.RESET_ALL}\n")
```

```
        except Exception as e:
            messagebox.showerror("Error", f"An error occurred while reading the file: {e}")

def main():
    while True:
        if platform.system() == 'Windows':
            os.system('cls')  # Command for Windows
        else:
            os.system('clear')  # Command for Unix-like systems


        print(f"{Fore.GREEN}*****************************************{Style.RESET_ALL}")
        print(f"{Fore.MAGENTA}     Source Code Vulnerability Scanner  {Style.RESET_ALL}")
        print(f"{Fore.GREEN}*****************************************{Style.RESET_ALL}\n")
        print(f"{Fore.YELLOW}[1] Upload and Analyze Python Script{Style.RESET_ALL}")
        print(f"{Fore.YELLOW}[2] Upload and Analyze PHP Script{Style.RESET_ALL}")
        print(f"{Fore.YELLOW}[3] Upload and Analyze JavaScript Script{Style.RESET_ALL}")
        print(f"{Fore.YELLOW}[4] Upload and Analyze Java Script{Style.RESET_ALL}")
        print(f"\n{Fore.RED}[5] Exit{Style.RESET_ALL}")
        print(f"\n{Fore.GREEN}*****************************************{Style.RESET_ALL}")

        choice = input(f"\n{Fore.BLUE}[+] Enter your choice (1-5): {Style.RESET_ALL}")

        if choice == '5':
            print(f"\n{Fore.RED}[-] Exiting the program...{Style.RESET_ALL}")
            break

        language_map = {
            '1': 'python',
            '2': 'php',
            '3': 'javascript',
            '4': 'java'
        }

        if choice in language_map:
            upload_and_analyze(language_map[choice])
        else:
                print(f"\n{Fore.RED}[!] Invalid choice. Please enter a number between 1 and 5.
{Style.RESET_ALL}\n")
            input(f"\n{Fore.BLUE}[*] Press Enter to continue...{Style.RESET_ALL}\n")

if __name__ == "__main__":
    main()
```

Explanation:

- The code prompts the user with a menu, guiding them to either start the analysis or exit.

- open_file_dialog() opens a file dialog window to allow file selection.
- The selected file is passed to the run_analysis() function for further processing.

You need to place your actual function calls for vulnerability scanning where the run_analysis() placeholder exists.

5.2. Regex Pattern Matching Engine

The **Regex Pattern Matching Engine** is a crucial component of the Source Code Vulnerability Analyzer. It acts as the first layer of security analysis, scanning for well-known, hardcoded patterns, and vulnerabilities in the source code. This engine uses regular expressions (regex) to match common patterns related to security flaws like **SQL injection**, **cross-site scripting (XSS)**, and **buffer overflows**.

Working of the Regex Engine:

- **Pattern Library**: The engine is built with a collection of regular expressions that match insecure code patterns. Each pattern corresponds to a specific vulnerability, such as SQL injection, insecure file access, etc.
- **Pattern Matching**: Once the source code is uploaded, the regex engine scans it for any pattern matches. If a pattern match is found, it flags the line or section of the code that matches.
- **False Positives Handling**: The engine incorporates a filtering mechanism to reduce false positives by matching only relevant and commonly exploited patterns.

Code Snippet:

```
import re
import os
import torch
from transformers import RobertaTokenizer, RobertaModel
import pickle

# Function to clean JavaScript code
def clean_code(code):
    code = re.sub(r'\/\*[\s\S]*?\*\/', '', code)  # Remove multiline comments
    code = re.sub(r'\/\/.*', '', code)  # Remove single line comments
    code = re.sub(r'#.*', '', code)  # Remove python comments
    code = re.sub(r'\s*\n\s*', '\n', code)  # Remove extra whitespace around newlines
    return code.strip()

# Function to label JavaScript code (0 for safe, 1 for unsafe)
def label_code(snippet):
    unsafe_patterns = {
        'eval': 'Code Injection',                    # Executes code from a string
        'innerHTML': 'Cross-Site Scripting (XSS)',       # Inserts HTML content into an element
        'outerHTML': 'Cross-Site Scripting (XSS)',       # Similar to innerHTML but for the element itself
        'document.write': 'Cross-Site Scripting (XSS)',  # Writes directly to the HTML document
        'setTimeout': 'Potential Code Injection',        # Executes code after a delay
        'setInterval': 'Potential Code Injection',       # Repeatedly executes code at intervals
        'Function': 'Code Injection',                # Creates new functions from strings
        'location.href': 'Open Redirect',              # Changes the URL of the current page
        'document.location': 'Open Redirect',            # Similar to location.href
        'XMLHttpRequest': 'Sensitive Data Exposure',     # Handles HTTP requests, can be misused
        '<script>': 'Cross-Site Scripting (XSS)',        # Directly injects JavaScript code
         'document.body.innerHTML': 'Cross-Site Scripting (XSS)'  # Sets the HTML content of the body
element
    }


    unsafe_lines = []
    for line in snippet.split('\n'):
        for pattern, vuln_type in unsafe_patterns.items():
            if pattern in line:
                unsafe_lines.append((line.strip(), vuln_type))
    return unsafe_lines
```

Explanation:

- unsafe_patterns: A dictionary of regex patterns that represent different vulnerabilities.

5.3. ML Model (Transformer, Dataset, Training Steps)

The **Machine Learning (ML) Model** is the backbone of your Source Code Vulnerability Analyzer, providing deep analysis of the code based on **transformer models** like **CodeBERT**. This model processes the code by analyzing syntactic and semantic structures and predicting possible vulnerabilities.

Model Choice:

- **CodeBERT**: A transformer model pre-trained on source code across multiple languages. It's designed to handle code-related tasks such as vulnerability detection, code completion, and more.

Dataset:

- **Dataset for Training**: You can use publicly available datasets like **CodeXGLUE**, which includes code from multiple programming languages, or use custom datasets consisting of labeled examples of secure and insecure code.

Training Steps:

- **Preprocessing**: The code is tokenized using the tokenizer from the **Transformers** library. Each code sample is converted into tokens that represent different code structures.
- **Model Training**: The model is fine-tuned on the dataset. The training step involves adjusting the weights of the transformer network based on the loss function, which compares the model's predictions against the true labels.

Training Example in Google Colab:

```
# Load pre-trained model and tokenizer
tokenizer = RobertaTokenizer.from_pretrained('microsoft/codebert-base')
model = RobertaModel.from_pretrained('microsoft/codebert-base')
```

```
# Load existing preprocessed dataset
with open('javascript.json', 'r') as f:
    data = json.load(f)

# Tokenize and get embeddings for the code snippets
inputs = tokenizer([d['code'] for d in data], padding=True, truncation=True, return_tensors='pt')
with torch.no_grad():
    outputs = model(**inputs)
embeddings = outputs.last_hidden_state.mean(dim=1).numpy()

# Create labels
labels = np.array([d['label'] for d in data])

# Train a simple classifier
clf = RandomForestClassifier()
clf.fit(embeddings, labels)

# Testing the classifier with a new code snippet
test_code = """
    var userInput = '{"name": "John", "age": 30}';
    var userObj;
    try {
        userObj = JSON.parse(userInput);  // Safely parse JSON without using eval
    } catch (e) {
        console.error("Invalid JSON input");
    }

    var userInput = "alert('Hello, world!')";
    eval(userInput);  // Unsafe
    var safe = 1243;
    document.getElementById('output').innerHTML = userInput;  // Unsafe
    document.write(userInput);  // Unsafe

"""
cleaned_test_code = clean_code(test_code)
unsafe_lines = label_code(cleaned_test_code)
# unsafe_lines = label_code(test_code)

print("Unsafe Lines of Code with Vulnerability Types:\n")
for line, vuln_type in unsafe_lines:
    print(f"{line} - {vuln_type}")

test_inputs = tokenizer([cleaned_test_code], padding=True, truncation=True, return_tensors='pt')
with torch.no_grad():
    test_outputs = model(**test_inputs)
test_embeddings = test_outputs.last_hidden_state.mean(dim=1).numpy()
prediction = clf.predict(test_embeddings)
# print("\nPrediction:", prediction)
```

```
!pip install gradio

import gradio as gr

# Function to detect unsafe code
def detect_unsafe_code(code):
    # Clean and label the provided code
    cleaned_code = clean_code(code)
    unsafe_lines = label_code(cleaned_code)

    # Format the output
    formatted_output = "\n".join([f"{line[0]} - {line[1]}" for line in unsafe_lines])
    return formatted_output

# Create and launch the Gradio interface
iface = gr.Interface(
    fn=detect_unsafe_code,
    inputs="text",
    outputs="text",
    title="JavaScript Code Safety Analyzer",
    description="Enter your JavaScript code to detect unsafe lines."
)

iface.launch()
```

Explanation:

- The code first loads the **CodeBERT** model and tokenizer.
- It tokenizes the dataset and proceeds with a basic training loop where the model is fine-tuned for 3 epochs.

5.4. Rule-based Engine

The **Rule-based Engine** complements the pattern-matching and ML-based analysis by providing an additional layer of checks for vulnerabilities based on predefined rules. These rules are typically **heuristic** and can catch patterns that may not have been flagged by either the regex engine or the ML model.

Working of the Rule-based Engine:

- **Custom Rules**: Users or developers can define custom rules related to specific security practices.
- **Checks for Vulnerabilities**: The engine scans the code and applies these rules to detect vulnerabilities, such as missing security headers, improper use of cryptographic functions, and unsafe data validation.

5.5. File Structure

The **file structure** of the Source Code Vulnerability Analyzer is designed to organize different components for ease of development and maintainability.

/project_root
- **/datasets** # Training dataset
  - java.json
  - javascript.json
  - php.json
  - python.json
- **/google colab files**
  - java_02.ipynb
  - javascript_02.ipynb
  - php_02.ipynb
  - python_02.ipynb
- **/languages**
  - java.py
  - javascript.py
  - php.py
  - python.py
- **/models**
  - java_model.pkl # Trained ML model

javascript_model.pkl
php_model.pkl
python_model.pkl
/training scripts
java_training.py
javascript_training.py
php_training.py
python_training.py
/results
report.txt # Output of vulnerability scan

5.6. UI /UX (File Dialog and Terminal)

The project provides two types of user interfaces:

- **Graphical File Dialog**:
    - Implemented using tkinter, allowing users to select files easily.
    - Provides a more intuitive way to scan files, especially for users uncomfortable with the command line.
- **Terminal Interface**:
    - Suitable for power users and automated systems.
    - Offers quick input and output without GUI overhead.

Both interfaces are seamlessly integrated into the menu.py module, offering flexibility based on user preference.

5.7. Google Colab Files and Pickle Files

The following files were developed, tested, and saved using **Google Colab**:

- **Model Training Colab**:

java_02.ipynb
javascript_02.ipynb
php_02.ipynb
python_02.ipynb

- Trains a transformer model on the code vulnerability dataset.

- **Saved Files**:
    - **java_model.pkl:** Pickle file containing the trained model.
    - **java.json :** Training and testing dataset used during training.

6. Vulnerabilities Detected

6.1. Introduction

In the realm of software development, security vulnerabilities pose significant risks. This chapter explores common vulnerabilities across multiple programming languages, illustrating how they manifest in code and how detection tools identify them. By understanding these vulnerabilities, developers can write more secure code and utilize tools effectively to mitigate potential threats.

6.2. Command Injection

Explanation:
Command injection occurs when an application constructs and executes system commands using unsanitized user input. Attackers can manipulate inputs to execute arbitrary commands, potentially compromising the system.

Insecure Code Example (Java):

```
String userInput = request.getParameter("cmd");
Runtime.getRuntime().exec(userInput);
```

Secure Code Example (Java):

```
String userInput = request.getParameter("cmd");
ProcessBuilder pb = new ProcessBuilder("sh", "-c", userInput);

pb.start();
```

Detection Mechanism:
Tools scan for patterns like Runtime.getRuntime().exec or ProcessBuilder combined with user input. They flag instances where user input is directly passed to system command execution functions.

6.3. SQL Injection

Explanation:
SQL injection involves inserting malicious SQL statements into an entry field, allowing attackers to manipulate the database.

Insecure Code Example (Python):

```
user_id = input("Enter user ID:")
cursor.execute("SELECT * FROM users WHERE id = " + user_id)
```

Secure Code Example (Python):

```
user_id = input("Enter user ID:")
cursor.execute("SELECT * FROM users WHERE id = ?", (user_id,))
```

Detection Mechanism:
Detection tools look for string concatenation in SQL queries, especially where user input is involved. They identify patterns where parameters are not properly sanitized or parameterized.

6.4. Cross-Site Scripting (XSS)

Explanation:
XSS attacks inject malicious scripts into web pages viewed by other users, potentially stealing session cookies or defacing websites.

Insecure Code Example (JavaScript):

```
document.getElementById("output").innerHTML = location.hash;
```

Secure Code Example (JavaScript):

```
document.getElementById("output").textContent = location.hash;
```

Detection Mechanism:
Tools analyze the use of functions like innerHTML, document.write, or eval with user-controlled input. They flag instances where user input is rendered without proper encoding or sanitization.

6.5. Insecure Deserialization

Explanation:
Insecure deserialization occurs when untrusted data is used to abuse the logic of an application, inflict denial-of-service attacks, or execute arbitrary code upon deserialization.

Insecure Code Example (Java):

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.ser"));
Object obj = in.readObject();
```

Secure Code Example (Java):

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.ser"));
in.setObjectInputFilter(info -> {

    if (info.serialClass() == AllowedClass.class) {

        return ObjectInputFilter.Status.ALLOWED;

    }

    return ObjectInputFilter.Status.REJECTED;

});

Object obj = in.readObject();
```

Detection Mechanism:
Detection tools look for usage of deserialization functions like readObject without proper validation or filtering. They flag instances where deserialization is performed on untrusted sources.

6.6. Insecure Randomness

Explanation:
Using predictable random number generators can lead to vulnerabilities, especially in security-critical contexts like token generation.

Insecure Code Example (Python):

```
import random
token = str(random.random())
```

Secure Code Example (Python):

```
import secrets
token = secrets.token_hex(16)
```

Detection Mechanism:
Tools identify the use of non-cryptographically secure random functions like random.random() in contexts requiring secure randomness. They recommend using secure alternatives like secrets or os.urandom.

6.7. Hardcoded Credentials

Explanation:
Embedding credentials directly into source code can lead to unauthorized access if the code is exposed.

Insecure Code Example (Java):

```
String password = "P@ssw0rd!";
```

Secure Code Example (Java):

```
String password = System.getenv("DB_PASSWORD");
```

Detection Mechanism:
Tools scan for patterns resembling credentials, such as strings assigned to variables named password, secret, or token. They also check for usage of functions like System.getenv to encourage secure practices.

6.8. File Inclusion Vulnerabilities

Explanation:s
Improper handling of file paths can allow attackers to include unintended files, leading to information disclosure or code execution.

Insecure Code Example (PHP):

```
include($_GET['page']);
```

Secure Code Example (PHP):

```
$allowed_pages = ['home', 'about', 'contact'];
$page = in_array($_GET['page'], $allowed_pages) ? $_GET['page'] : 'home';

include($page . '.php');
```

Detection Mechanism:
Detection tools look for dynamic file inclusion functions like include or require with user-controlled input. They flag instances where input is not validated against a whitelist.

6.9. Buffer Overflow

Explanation:
Buffer overflows occur when data exceeds the buffer's storage capacity, potentially overwriting adjacent memory and leading to code execution.

Insecure Code Example (C):

```
char buffer[10];
strcpy(buffer, input);
```

Secure Code Example (C):

```
char buffer[10];
strncpy(buffer, input, sizeof(buffer) - 1);

buffer[sizeof(buffer) - 1] = '\0';
```

Detection Mechanism:
Tools analyze functions like strcpy, gets, or sprintf without bounds checking. They recommend safer alternatives that limit the amount of data copied.

6.10. Improper Error Handling

Explanation:
Displaying detailed error messages can reveal sensitive information about the application's structure or database.

Insecure Code Example (Java):

```
catch (Exception e) {
   e.printStackTrace();

}
```

Secure Code Example (Java):

```
catch (Exception e) {
   logger.error("An error occurred.");

}
```

Detection Mechanism:
Tools detect the use of functions like printStackTrace or displaying exception messages directly to users. They advise logging errors securely without exposing internal details.

6.11. XML External Entity (XXE) Injection

Explanation:
XXE attacks exploit vulnerabilities in XML parsers to access internal files or execute remote requests.

Insecure Code Example (Java):

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();

Document doc = db.parse(new File("data.xml"));
```

Secure Code Example (Java):

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);

DocumentBuilder db = dbf.newDocumentBuilder();

Document doc = db.parse(new File("data.xml"));
```

Detection Mechanism:
Detection tools analyze XML parsing configurations, checking for features that allow external entity processing. They flag instances where such features are enabled without proper safeguards.

7. Challenges Faced

7.1. Machine Learning (ML) Integration Challenges

7.1.1 Model Selection and Performance Bottlenecks

One of the first hurdles faced in integrating ML into a security detection system was identifying a suitable model that could accurately classify code snippets as vulnerable or safe. Pretrained models often required fine-tuning, while building models from scratch posed challenges related to data volume, preprocessing, and training time. Ensuring the model didn't overfit on training data while still generalizing well to unseen snippets was a delicate balance.

7.1.2. Dataset Limitations

Most datasets for vulnerability detection are either labeled by static rules or scraped from open-source platforms. However, inconsistencies in labeling, lack of examples for certain edge-case vulnerabilities (e.g., insecure randomness), and language bias (e.g., dominance of Python or Java samples) made training difficult. In some cases, manual curation was necessary to ensure quality.

7.1.3. Feature Extraction Difficulties

Converting raw code into numerical formats suitable for ML models (using vectorizers like TF-IDF, AST-based embeddings, or code2vec) often lost context-sensitive information. Capturing semantics—such as understanding the use of system-level calls or credential handling—was not straightforward and required custom tokenization strategies.

7.1.4. Real-Time Performance Concerns

Embedding the ML model into a real-time vulnerability scanner required it to make fast predictions on short code snippets. However, larger models like BERT or CodeBERT were slow and required GPU acceleration. Striking the right trade-off between model complexity and execution speed was a constant challenge.

7.2. Regex Accuracy

7.2.1 Crafting Precise Patterns

Regular expressions were initially used to flag risky code patterns—like detecting hardcoded credentials, SQL injection strings, or unsafe system calls. However, making regex rules that were neither too permissive nor too restrictive was an ongoing struggle. For instance, matching password\s*=\s*["'].*["'] often caught legitimate logging variables as well.

7.2.2. Handling Code Variants

Different developers write the same logic in different ways, and regex patterns struggled to catch all syntactic variants. For example, SQL queries could be concatenated over multiple lines or constructed via helper functions—making it hard for simple pattern-matching to detect vulnerabilities accurately.

7.2.3. False Positives and False Negatives

Regex often led to high false positives, flagging benign strings as dangerous. Conversely, if regex was too strict, it missed real vulnerabilities. Tuning expressions to balance sensitivity and specificity required extensive testing, which grew more difficult as the number of supported languages increased.

7.2.4 Context-Awareness Limitations

Regex is inherently limited in its ability to understand the context of code. It cannot distinguish between a hardcoded string used in logging and one used for authentication. This limitation prompted the use of hybrid techniques combining regex with lightweight syntax trees or token-based ML models.

7.3. File Handling Challenges

7.3.1. File Format Detection

Since the tool was expected to support various languages (e.g., .java, .py, .php, .cpp), a robust system to auto-detect file types was needed. File extension-based detection was not always reliable, especially when analyzing uploaded code snippets without extensions.

7.3.2. Large Files and Memory Optimization

Some real-world source code files exceeded 10,000 lines, causing memory issues during parsing or feature extraction. Techniques such as chunking, lazy loading, and streaming parsers were implemented to prevent crashes and optimize memory usage.

7.3.3. Encoding and Special Characters

Encountering files with non-standard encodings (like UTF-16, Latin-1) or non-ASCII characters occasionally led to errors in preprocessing. Some legacy codebases included characters that broke the tokenizer or parser modules.

7.3.4. Temporary File Management and Security

The tool creates temporary files for pre-processing, intermediate results, and logs. Ensuring these files were properly cleaned up and didn’t contain sensitive data became essential to prevent accidental data leakage or disk bloating over time.

7.3.5. File Permission and OS Compatibility

Handling file read/write operations across Windows, Linux, and macOS posed platform-specific challenges. Some environments restricted file access via permissions or antivirus tools, leading to unexpected runtime errors. These bugs had to be handled with cross-platform exception handling routines.

7.4. Multi-Language Support

7.4.1. Syntax Variations Across Languages

Each programming language has its own syntax, libraries, and ways of performing similar tasks. For instance, executing system commands in Python (os.system), Java (Runtime.exec), and C (system()) all differ syntactically. Creating a generalized parser that accurately recognized vulnerabilities across languages required significant design effort.

7.4.2. AST Generation Complexity

While Abstract Syntax Trees (ASTs) helped in understanding code structure, generating them across multiple languages introduced complications. Some languages like JavaScript and PHP needed third-party parsers, which were either slow or inconsistent. AST structures were also not standardized—Java's AST differs greatly from Python's.

7.4.3. Balancing Language-Specific and Universal Rules

The tool had to walk a tightrope between generic rules that applied to all languages (like regex for hardcoded passwords) and highly specific rules (like detection of unsafe C functions such as gets()). Managing these rule sets while avoiding redundancy and conflicts required extensive testing and rule prioritization.

7.4.4. Dependency Management and Parser Setup

Many languages required external libraries or compilers to parse code (e.g., Java's ANTLR, TypeScript's TSC parser). Setting up and maintaining these dependencies across different platforms added to project complexity and increased installation times.

7.4.5. Testing and Validation

Each supported language required its own set of vulnerable and safe test cases to validate the detection engine. Creating these test sets manually was time-consuming, especially for less common or verbose languages like Ruby or C++. Furthermore, certain edge-case behaviors (like PHP's include and eval) required dynamic execution to test properly.

8. Future Enhancements

8.1. Integration with CI/CD Pipelines

A critical future step is enabling **seamless integration with CI/CD pipelines** such as Jenkins, GitLab CI, GitHub Actions, and CircleCI. Security needs to be shifted left in the development lifecycle, meaning vulnerabilities should be caught early—during development or integration, not after deployment.

The tool can be packaged as a command-line utility or Docker container and embedded within CI workflows to automatically scan new commits or pull requests for vulnerabilities. This integration will:

- Reduce human error by automating the scanning process.
- Help developers receive real-time feedback during development.

- Improve security compliance and auditability.

In future versions, webhook support and report generation for CI dashboards can also be implemented to alert developers in case of high-severity vulnerabilities.

8.2. GUI Version

Currently, the tool operates in a terminal or script-based environment, which may not be user-friendly for all users, particularly non-technical stakeholders. Creating a **Graphical User Interface (GUI)** will vastly improve accessibility and user experience.

The GUI can offer:

- Drag-and-drop functionality for uploading files.
- Real-time code scanning with syntax highlighting.
- Dashboard visualization of vulnerability statistics.
- Export options for PDF/HTML reports.

The GUI could be built using frameworks such as **PyQt**, **Electron**, or **Tkinter**, and could optionally be packaged as a desktop application or web app.

8.3. Expanded Language Support

While the current version supports a core set of languages (e.g., Python, Java, C, JavaScript), real-world applications are often developed in a wide variety of languages. Expanding support to include languages like:

- **Ruby**
- **Go**
- **Swift**
- **Kotlin**

- **TypeScript**
- **Shell (Bash) scripts**
  would significantly increase the tool's value and applicability across industries.

This will require:

- Designing new regex rules and parsers per language.
- Leveraging AST generation tools specific to those languages.
- Curating new datasets for ML training in less-represented languages.

8.4. Better ML Training Dataset

Although the current ML model performs reasonably well, its accuracy is inherently limited by the quality and diversity of the training data. The next major step would be to:

- Curate a **larger and more balanced dataset** that includes a wide array of secure and insecure code patterns across different languages.
- Add **real-world code samples** from open-source projects (with proper license compliance).
- Introduce **edge-case vulnerabilities**, like insecure encryption, SSRF, insecure deserialization, etc., which are often missed by existing datasets.

Incorporating **transfer learning** using pretrained models like **CodeBERT**, **GraphCodeBERT**, or **CodeT5** will also be explored to enhance performance, especially in multi-language support and code semantics understanding.

9. Conclusion

This project aimed to develop a hybrid vulnerability detection system capable of identifying security flaws in source code using both **machine learning** and **static analysis** methods (primarily regex). The tool successfully detects various classes of vulnerabilities, including hardcoded secrets, unsafe function calls, and insecure database handling across multiple programming languages.

9.1. Summary of Achievements

- **Machine Learning Integration**: Built and trained an ML model capable of classifying code snippets based on vulnerability.
- **Regex-Based Detection Engine**: Created a rule-based system to detect common vulnerabilities using handcrafted patterns.
- **Multi-language Support**: Designed the framework to support scanning code written in Python, Java, JavaScript, and C/C++.
- **Code Parsing and Tokenization**: Implemented techniques for language-specific tokenization and partial AST parsing.
- **Reporting System**: Generated structured results and vulnerability classifications for end-user consumption.

10. References

10.1. Tools and Frameworks Used

- **Python 3.11**
- **Scikit-learn** – for ML model training.
- **NLTK / spaCy** – for tokenization and preprocessing.
- **Pandas & NumPy** – for data handling and transformation.
- **Regex (re module)** – for pattern matching and static rule enforcement.
- **AST (Abstract Syntax Tree)** – for structural code analysis in Python.
- **Tkinter / Streamlit (planned GUI)**
- **Visual Studio Code** – primary development environment.

10.2. Research Papers and References

- Learning to Represent Programs with Graphs
- CodeBERT: A Pre-Trained Model for Programming and Natural Languages
- Survey on Static and Dynamic Analysis for Vulnerability Detection
- Deep Learning Based Vulnerability Detection: Are We There Yet?
- OWASP Top 10 – https://owasp.org/www-project-top-ten/

10.3. GitHub Repositories Referenced

- CodeQL by GitHub Security Lab
- Semgrep: Lightweight static analysis
- SonarQube Rules
- Bandit – Python security linter
- CodeBERT Pre-trained Model

10.4. Documentation and Articles

- Python re documentation: https://docs.python.org/3/library/re.html
- Scikit-learn documentation: https://scikit-learn.org/stable/
- AST Module docs: https://docs.python.org/3/library/ast.html
- GitHub REST API: https://docs.github.com/en/rest
- OWASP Cheat Sheets: https://cheatsheetseries.owasp.org/

*****