Enhancing Software Security with AI: Detecting Vulnerabilities in High-Level Programming Languages

KEVIN DAHLÉN
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Enhancing Software Security with AI: Detecting Vulnerabilities in High-Level Programming Languages

Kevin Dahlén
kevin.dahlen.8683@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisors: Christian Gehrmann & Syafiq Al Atiiq

Examiner: Thomas Johansson

November 28, 2024

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

With the rapid growth of software dependencies in critical systems, detecting vulnerabilities early in the development cycle has become a necessity. While extensive research exists on vulnerability detection in low-level languages like C/C++, there is a significant gap in addressing vulnerabilities in higher-level languages, including JavaScript, Python, and PHP. This thesis explores the feasibility of using fine-tuned large language models (LLMs) to detect vulnerabilities in these higher-level languages, leveraging data-driven approaches to bridge the existing research gap.

In this thesis we work with fine-tuning LLMs on a curated dataset derived from CVEFixes, implementing the model in an API format to facilitate integration into CI/CD pipelines. Key performance metrics, including accuracy, precision, and F1 score, reveal that the model achieved varying effectiveness across different languages, with strong results in JavaScript and Java but weaker results in PHP, highlighting the nuanced challenges of vulnerability detection in diverse programming contexts. To demonstrate real-world applicability, the models proposed in this thesis were deployed through a user-friendly web interface, with API accessibility allowing seamless integration for developers.

In our discussion, we address the ethical and security implications of using AI-driven vulnerability detection, including potential misuse by malicious actors and over-reliance on automated findings. The findings suggest that while LLMs are promising for certain languages, further refinement is needed to improve accuracy and reliability across diverse high-level languages. Future work should explore hybrid models that combine traditional and AI-based detection to mitigate current limitations and enhance practical use in software security, and CWE-specific training for higher-level languages.

Popular Science Summary

Can Artificial Intelligence Help Stop Software Security Breaches?

Our reliance on digital devices and online platforms grows every day, and with it, the need for secure software. This thesis explores how AI could automate the detection of software vulnerabilities, with a focus on the languages behind today's most common apps. Could a specialized AI tool help developers quickly find and fix weak spots in code before hackers can exploit them?

Across industries, from healthcare to finance, software security is crucial. Yet every day, news breaks of hackers who have exploited vulnerabilities in widely-used applications. While some vulnerabilities are found by cyber-security experts before they cause damage, many remain undetected and can result in data breaches, system crashes, or worse. Detecting these weaknesses manually, or even with traditional software tools, can be difficult, time-consuming, and may miss new or complex vulnerabilities that AI could easily spot.

This thesis work aims to bridge that gap using artificial intelligence. The approach fine-tuned an advanced AI model, a large language model (LLM), on data specifically related to common software vulnerabilities. By "training" it on a wide range of real-world examples, this LLM can now recognize patterns of code that might indicate a security flaw — particularly in the high-level languages that power popular web and mobile applications.

Why AI in Security? The Potential for Broader, Faster Vulnerability Detection

AI models like LLMs have proven to be effective in understanding and generating human-like text, which makes them well-suited to understanding the structure and flow of programming code. In this thesis, an LLM is specially trained to detect vulnerabilities in languages such as JavaScript, Python, and Java. This is especially relevant because of the research gap in this subject. While traditional tools for vulnerability detection have made a huge impact on writing secure code, there is a lack of easily accessible, and automated vulnerability detection systems for developers to take advantage of.

Not only does this AI approach promise to detect vulnerabilities faster, but it can also provide a valuable extra line of defense for developers. Integrating this model into developers' workflow allows it to scan code in real-time, quickly flagging potential vulnerabilities, even if they occur in more complex or obscure areas of the code. This could mean that the AI scans code as developers write it, detecting vulnerabilities before deployment.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's world, everything runs on software—our phones, cars, and even our fridges. As societies become more dependent on these digital systems, the stakes get higher when something goes wrong. A single vulnerability in a piece of software could allow malicious attackers to exploit the entire system, potentially leading to intrusions, denial of service, or even massive data breaches. For example, in 2014, a vulnerability later called *Heartbleed*[1] was revealed, allowing attackers to access secrets from an estimated 24-55% of popular HTTPS protected websites [21]. It is a growing concern, and unfortunately, one that is becoming increasingly common.

As this concern grew, researchers started investigating ways of preventing vulnerabilities. One of the first ideas, from 1976, was to have developers manually inspect the code, in an attempt to find bugs or other weaknesses [22]. Then, in 1990, dynamically running the program and feeding certain methods random data as a means of testing was developed [43], later known as *Fuzzing*. Even more recently, in 1995, tools for inspecting code without running it to find bugs and weaknesses, known as static analysis, started gaining traction [68].

Traditionally, people manually look through code or use these tools to try and spot these vulnerabilities. AI models, especially Large Language Models (LLMs) that can read, contextualize, and understand code, has potential to automate the process of identifying these weak spots before attackers can get to them.

Most research so far has been hyper-focused on C and C++, languages that are notorious for being difficult to secure. What about other languages that power modern web apps, machine learning projects, and mobile apps — such as Python, JavaScript, and Java? Despite the possibility of these languages containing just as risk-imposing security issues, there is a significant gap in research aimed at detecting weaknesses in these high-level languages.

This thesis aims to develop AI tools for detecting vulnerabilities in source code. Despite prior research efforts, existing models have struggled to accurately predict code vulnerabilities. The thesis work is supported by Gehrmann Trusted ICT, which is where the thesis is performed. The work is based on existing vulnerable code datasets and aims to extend the current research in cyber-security and AI.

[1] `https://heartbleed.com/`

1.1 Motivation

With the growing complexity of software systems and digital infrastructure becoming more crucial, security vulnerabilities in source code have become a critical issue. Attack vectors continue to evolve, and even minor weaknesses in code can lead to severe security breaches, threatening user privacy and financial stability. As the volume of code expands, manual review processes become increasingly unsustainable. This creates an urgent need for scalable, automated approaches to identify vulnerabilities in software.

Traditional static and dynamic analysis tools, such as Checkmarx[2] and Acunetix[3] to name a few, have provided significant advancements in detecting vulnerabilities but often struggle with scalability and time costs [9, 63]. Machine learning (ML) approaches, specifically the emergence of LLMs, present new possibilities for automating vulnerability detection by leveraging vast amounts of code data to learn patterns associated with insecure coding practices [71]. Models such as GPT [8] and BERT [17], trained on massive text corpora, have shown promising capabilities in tasks such as natural language understanding, code generation, and even bug detection. However, fine-tuning these models specifically for the task of vulnerability detection presents unique challenges.

The motivation for this thesis stems from the need to explore how fine-tuning LLMs can be optimized for identifying vulnerabilities in languages that are less frequently targeted by existing research in LLMs. Given the rapid advancement in both programming languages and security threats, exploring data-driven approaches using LLMs can bridge the gap between theoretical security practices and real-world applications.

1.2 Thesis Purpose

This thesis explores the potential of fine-tuning LLMs for detecting security vulnerabilities in non-C/C++ programming languages. While much of the existing research focuses on the vulnerabilities associated with C and C++, higher-level languages that are widely used in modern development environments also pose significant risks. Python, JavaScript, and Java, among others, have been increasingly utilized in various domains, from web development to machine learning, making them attractive targets for attackers.

1.2.1 Model Fine-tuning

The first goal of this thesis is to leverage the capabilities of LLMs to analyze code for common security vulnerabilities in these higher-level languages. This includes preparing datasets of vulnerable and non-vulnerable code samples, fine-tuning pre-trained LLMs for the task, and evaluating the model's performance on unseen data. By identifying vulnerabilities early in the development cycle, the

[2] https://checkmarx.com/
[3] https://www.acunetix.com/

fine-tuned models could provide a useful tool to developers, enhancing software security without significantly increasing development time.

1.2.2 Model Usage

As a last step, a website and a pipeline for Continuous Integration (CI) / Continuous Deployment (CD) will be built to further enhance the usability of the models. For this, an API needs to be created and users should be able to authenticate themselves to control the use of the models. The API should be able to accept code functions, and make a prediction on whether the given code function is vulnerable or not. The reasoning behind providing code functions to the model, as opposed to an entire code base, is due to technical limitations in how much data can be processed by an LLM. On the other hand, it is not feasible to make a prediction on vulnerability based on a single line of code either, as the model should have enough context to make an accurate prediction. Therefore, keeping it at a code function level provides a good balance between technical limits while giving enough context.

1.2.3 Research Questions

The purpose of the thesis can be summarized into the following research questions:

- **RQ1:** How effective are AI models in detecting vulnerabilities in high-level programming languages?
  - How do the results compare to traditional vulnerability detection methods regarding speed, accuracy, and ease of use?
- **RQ2:** What are the challenges and benefits of integrating AI vulnerability detection models into continuous integration / continuous deployment pipelines in GitHub and GitLab?

1.3 Outline

The remainder of this thesis is structured as follows:

- **Chapter 2: Background** provides an overview of the relevant research and concepts in software vulnerability detection, machine learning, and large language models. It includes a discussion on traditional static and dynamic analysis techniques and the recent trends in machine learning-based approaches to code analysis.
- **Chapter 3: Data Description** covers the data used when training the models, providing information about what data is available, how they were filtered, and what alterations were made.
- **Chapter 3: Methodology** describes the model architecture and fine-tuning strategy used in this research. It also discusses the hyperparameters and training process, as well as the frameworks and tools used for building the API, website, and CI/CD integrations.

- **Chapter 4: Experiments and Results** presents the metrics used for measuring model performance, and the results obtained from the fine-tuned models, including comparisons with baseline techniques, as well as the system structure of the API and CI/CD integrations.
- **Chapter 5: Discussion** provides an interpretation of the experimental results and reflects on the challenges and limitations of using LLMs for vulnerability detection in non-C/C++ languages.
- **Chapter 6: Conclusion and Future Work** summarizes the findings of this research and suggests potential future directions for improving vulnerability detection using LLMs.

Chapter 2
Background

To bridge the foundational concepts and methodologies of vulnerability detection with its practical application, this section provides an overview of the tools and techniques used to uncover software security flaws.

2.1 Software Vulnerability Detection

Vulnerability detection is a preventive measure taken to mitigate the security risks of a software solution, aiming to identify potential security flaws in code that attackers can exploit.

According to Chu et. al., the three most common methods for vulnerability detection are fuzz testing, symbolic execution, and formal verification [15]. These approaches are generally not easily executed, with efficient fuzz testing requiring both time and in-depth knowledge of the software that is being fuzzed [6], symbolic execution requiring large amounts of memory, and sometimes not being feasible for large code projects [4], and formal verification being an advanced, time-consuming and costly process [37].

A graph of different approaches to analyzing programs can be seen in figure 2.1 [16], depicting the relations between static/dynamic-ness, and how much manual work is required to use the different tools. Some of these approaches will be further discussed.

2.1.1 Dynamic Analysis

Dynamic Analysis, also known as Dynamic Application Security Testing (DAST), is a set of techniques for examining programs during runtime to identify potential vulnerabilities [46]. Since there is an overlap between functionality/bug testing and security testing, some of these tools and techniques are well-known in the developer industry, such as unit-testing [35], code-coverage analysis with gcov[1], and memory error detectors such as AddressSanitizer [61] and Valgrind [47].

[1] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

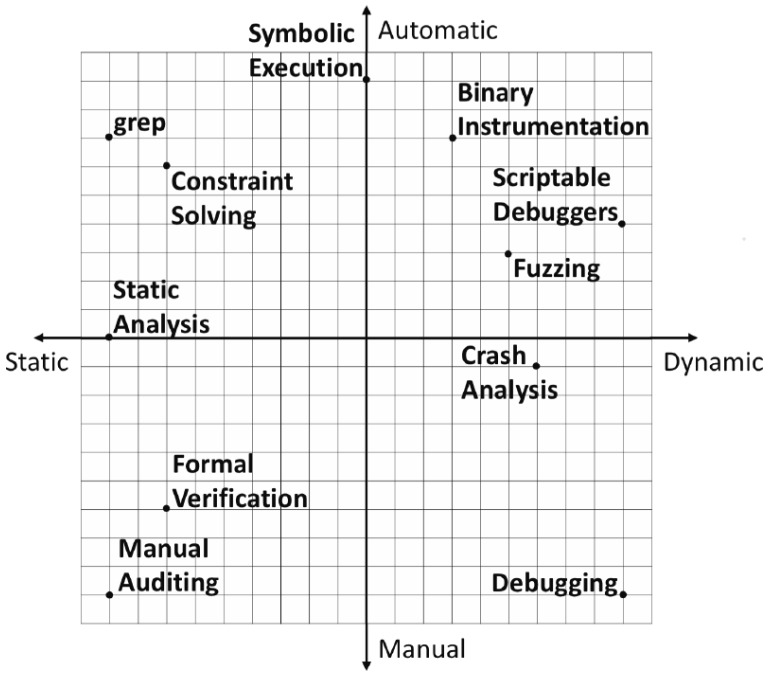

Figure 2.1: Static/dynamic automatic/manual program analysis tools [16].

Fuzz Testing

Fuzz testing, also known as fuzzing, is the concept of repeatedly sending machine-generated input into a function, to validate that no memory issues arise regardless of the input [7]. The fuzzing engine, typically AFL[2] or libFuzzer[3], will continuously monitor the branch coverage for each generated input, and based on the code coverage, mutate the input in an attempt to further increase the branch coverage. If a new input has triggered more edge cases in the code, it is saved to a corpus that the engine uses in future fuzzing sessions. If a crash happens, the input will be saved along with the crash information. This procedure is depicted in figure 2.2.

Despite being an effective and sound technique for finding security vulnerabilities, fuzzing has its challenges. Fuzzing is a time-consuming process, with the idea being that one lets a fuzzer run for as long as possible to find all edge cases in the program. Since a fuzzer is sending semi-randomized data, it may never find a certain bug if it is complex enough. Additionally, the developer creating the fuzzing harnesses requires an understanding of the solution as a whole.

[2]https://github.com/google/AFL
[3]https://llvm.org/docs/LibFuzzer.html

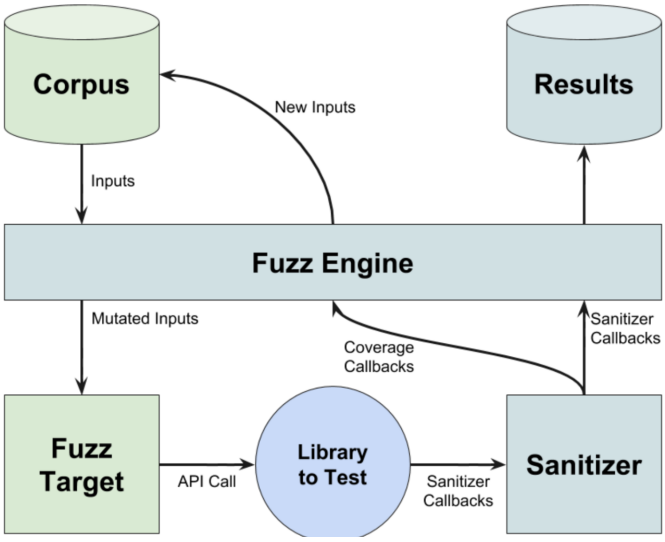

Figure 2.2: Fuzz testing

2.1.2 Static Analysis

Static analysis is an umbrella term for several code analysis methods, such as data flow analysis, control flow analysis, pattern matching for common vulnerabilities, and Abstract Syntax Tree (AST) analysis. All these methods share the fact that the code is never executed, making them categorizable under **static** analysis.

Data Flow Analysis

The key concept of data flow analysis is to analyze a program's Control-Flow Graph (CFG) to determine which values certain variables may have. By analyzing the flow of data, potential vulnerabilities may be discovered, such as uninitialized variables (*reaching definitions* [64]) and tainted user input which potentially could lead to injection attacks.

Control Flow Analysis

By using control flow analysis, possible execution paths are modeled, making it possible to detect issues such as infinite loops, unreachable code, or improper exception handling [48], which could lead to security vulnerabilities.

2.2 LLM-based Vulnerability Detection

LLMs are scalable and capable of analyzing large codebases with relatively fast results. They also simplify the vulnerability detection process, since they do not require any advanced setup from the user's point of view, making it more accessible

for developers. As more developers adopt this approach, more secure code could be published, reducing the risk of attacks.

2.2.1 Data-Driven Approaches to Vulnerability Detection

A data-driven approach to vulnerability detection using LLMs relies heavily on the availability of correctly labeled datasets of code. Issues such as imbalanced data, where secure code significantly outnumbers insecure samples, and the labels' correctness, must be considered when building a robust model [56, 10]. For example, if 99% of the data is non-vulnerable code, with only 1% being vulnerable, the model would be biased into classifying all code as non-vulnerable, as it gets a high accuracy on its predictions. While handling the class imbalance, maintaining enough data is important. This imposes several challenges for the data to be used.

This thesis explores the feasibility of leveraging LLMs for vulnerability detection in non-C/C++ languages by fine-tuning them on relevant datasets and evaluating their effectiveness in real-world applications.

Vulnerability Classification Systems

To standardize vulnerability identification and classification, several frameworks exist, primarily CVEs [70] and CWEs [4]. These frameworks provide structures for identifying and categorizing vulnerabilities across software types and development environments, aiding in detection, prevention, and remediation.

CVEs provide a universal reference for publicly known vulnerabilities. The CVE system, managed by MITRE, assigns unique identifiers to vulnerabilities across products and software, enabling precise tracking. Each CVE entry describes a specific vulnerability, including its impact, affected versions, and patch information, facilitating rapid identification of risks within code bases.

CWEs catalog vulnerability types based on underlying weaknesses that lead to security flaws. Unlike CVEs, which document individual instances, CWEs focus on patterns and categories of vulnerabilities, such as CWE-89 (SQL Injection) and CWE-787 (Out-of-bounds Write). This classification system is widely recognized and critical in identifying and grouping vulnerabilities for targeted remediation strategies.

In addition to CVEs and CWEs, other systems contribute to vulnerability management. The National Vulnerability Database (NVD)[5] provides detailed metadata for CVEs, including severity ratings (using CVSS scores[6]) and impact metrics. Moreover, the Open Web Application Security Project (OWASP)[7] lists top vulnerabilities by prevalence and risk, offering resources for mitigating web application vulnerabilities.

Automated vulnerability detection tools often reference these frameworks, matching identified weaknesses against CVE or CWE listings. This alignment

[4] `https://cwe.mitre.org/about/index.html`
[5] `https://nvd.nist.gov/`
[6] `https://nvd.nist.gov/vuln-metrics/cvss`
[7] `https://owasp.org/www-project-top-ten/`

allows developers to address risks according to severity and to comply with industry standards for software security.

2.3 The C/C++ Dilemma

To the best of our knowledge, there is no published research where a large language model has been trained to achieve statistically good results in code vulnerability detection. The majority of research in this field is based around C and C++. Inspecting the available vulnerable code data indicates some possible causes. One major issue is transitive vulnerabilities [44], e.g., a function may be classified as vulnerable in a dataset if the function calls another vulnerable function. While calling a vulnerable function is a security issue, it is not reasonable to claim that the function responsible for making the call is vulnerable - fixing the actual vulnerability will result in the calling function being non-vulnerable, without requiring any modification.

Additionally, several CWEs are more frequent in lower-level languages such as C and C++ due to their direct memory management and lack of built-in protection. These vulnerabilities may be mitigated, or even non-existent, in higher-level languages. Some examples include CWE-416 Use After Free and CWE-476 Double Free since automated garbage collectors are often used, or CWE-787 Out-of-bounds write because of built-in checks during runtime that throw exceptions.

For this thesis, C/C++ will not be investigated due to previous rigorous examination by multiple researchers. Instead, this thesis will be limited to examining JavaScript, PHP, Java, Python, and Go, since large data exists covering these languages, further described in following sections.

2.4 Previous Research

The concept of utilizing AI in secure coding, and in cyber-security more broadly, is currently undergoing rigorous examination by researchers. This section aims to shed light on the current research situation in the field.

2.4.1 Data-gathering

Gathering data for machine learning purposes is difficult, as machine learning relies heavily on correct and sound data. The most common way to gather vulnerable code is to scrape data from a CVE database, such as NVD, and identify the git change that fixed the vulnerability. This approach was taken both by Bhandari et.al., published as CVEFixes [5], and Fan et.al., published as Big-Vul [23].

Chakraborty et.al. proposed a different data-gathering methodology, where they selected the Linux Debian Kernel and Chromium projects, and scraped security issues and their fixes according to their GitHub issues and their Bugzilla repository respectively, released as a dataset ReVeal [12].

Nikitopoulos et.al. combined the methodologies of ReVeal and CVEFixes/Big-Vul, both scraping NVD and using issue-scraping, releasing the data under the CrossVul dataset [50].

2.4.2 Dataset Curation

While CVEFixes contains mostly accurate data (since it scrapes CVEs and their supposed fixes for them through git history), it does not have as much data as Big-Vul. Big-Vul, however, contains duplicated code and is not well curated, since it uses multiple sources for its data. An analysis of datasets performed by Chen et.al. by randomly sampling code and manually investigating the correctness of a label showed that out of the mentioned datasets, CVEFixes had the highest correctness for the vulnerability label [14].

2.4.3 Machine Learning for Vulnerability Detection

Machine Learning (ML) in the context of vulnerability detection is a relatively new field, currently being scrutinized by researchers. A study where API requests were classified as malicious or not using various ML models, was conducted by Piyush Ranjan [55]. Ranjan's study compared some approaches to ML and their efficiency in correctly detecting malicious requests. The models compared were Decision Tree[8], Random Forest[9] [57], Support Vector Machine (SVM)[10] [62], and Long Short-Term Memory (LSTM) [33]. The study found LSTM to be the most efficient ML approach for detecting malicious requests, correctly classifying 95.6% of requests.

Another study of using ML in vulnerability detection was performed by Welearegai et. al. [67], where multiple models were tried to classify requests to various Raspberry Pi[11] devices as either benign or under attack and act upon the classification accordingly. They achieved an accuracy of 75% for online monitoring of attacks.

In a study performed by Ma et. al. [41], they analyzed the AST of JavaScript programs to detect malicious code using Random Forest [57], SVM [62], XGBoost [13], and Multilayer Perceptron (MLP) [1]. By utilizing backpropagation through the AST nodes, they marked code segments as tainted or not tainted, indicating whether a value sent to a function had been tainted (e.g. modified) by the function, and then training the different model architectures to get a comparison of how the different models performed on detecting vulnerabilities. Using this methodology, they achieved an accuracy on detecting malicious code of 85.60% using MLP architecture, and an accuracy of 96.40% using XGBoost.

2.4.4 Fine-tuning

Fine-tuning is a process in machine learning where a pre-trained model, initially trained on a large and general dataset, is adapted for a specific task through additional training on a more specialized dataset. This approach leverages the generalized knowledge embedded in the model from pre-training and customizes it for narrower applications. In the context of LLMs, fine-tuning involves adjusting

[8] `https://www.ibm.com/topics/decision-trees`
[9] `https://www.ibm.com/topics/random-forest`
[10] `https://www.ibm.com/topics/support-vector-machine`
[11] `https://www.raspberrypi.com/`

the model's parameters to improve performance on a target task, such as vulnerability detection in code, by learning from domain-specific patterns present in labeled data.

In recent years, fine-tuning large language models for vulnerability detection has shown varying results, especially for identifying security risks in low-level languages like C and C++. Fine-tuning allows LLMs, such as those based on transformer architectures, to adapt to specific security contexts. Studies by Ding et al. with the PrimeVul dataset [18], for instance, demonstrated the effectiveness of such models in identifying code vulnerabilities by leveraging fine-tuning on curated datasets of vulnerable and non-vulnerable code snippets. PrimeVul is one of the most recent, better-performing research outputs in this field, compared to other previously mentioned studies.

Notable fine-tuning attempts have focused on classifying functions as either vulnerable or non-vulnerable. However, other variations of vulnerability classification have also been attempted. Research by Atiiq et al. [2] has highlighted that fine-tuning LLMs specifically on vulnerability types, as classified by CWE identifiers, significantly improves detection accuracy and allows the models to recognize vulnerabilities with specific attributes, like SQL injection, index out-of-bounds, or cross-site scripting. Despite these advances, fine-tuning models for universal vulnerability detection across multiple CWEs and languages remains challenging due to the variations in syntax, security issues, and data availability for different languages.

LLM Vulnerability Detection in Multiple Languages

In a study by Dozono et. al. [19], they identified the research gap in non-C/C++ vulnerability detection using LLMs. Using a technique known as *prompt priming*[12], which involves setting the tone, style, or structure for the LLMs responses, combined with a regular prompt asking if the code is vulnerable or not, they queried GPT [8] models, CodeLlama [58] models, and Gemini 1.5 Pro[13] on the languages Python, C, C++, JavaScript, and Java. They experimented with different prompts and achieved comparatively good results, with the arguably best prompt combination being `‘‘You are an AI binary vulnerability classifier that identifies whether the provided code is vulnerable or not vulnerable. You should respond with either only 'vulnerable' or 'not vulnerable'. + ‘‘Classify the following code in vulnerable or not vulnerable. Output either only 'vulnerable' or 'not vulnerable'’’`. They chose not to perform fine-tuning due to the requirement of substantial computational resources but mentioned that fine-tuned models typically outperform other machine-learning approaches. After experimenting, they achieved code vulnerability detection accuracies between 47% (C) to 88% (JavaScript).

Bae et. al. [3] attempted to classify C++, Java, and Python code using GPT and Claude [8, 20]. Specifically, they used GPT-3.5 Turbo, GPT-4 Turbo, GPT-4o, and Claude-3.5 Sonnet to classify the code and detect whether it contained

[12] `https://learnprompting.org/docs/basics/priming_prompt`
[13] `https://deepmind.google/technologies/gemini/pro/`

a vulnerability in a given CWE class. They evaluated three types of prompting; concise prompts, tip setting prompts, and step-by-step prompts. They define concise prompting as:

> "Please evaluate if the above code has a CWE-*** vulnerability. Does this code have a CWE-*** vulnerability? (Yes/No). Please rate your confidence in this answer on a scale from 0 to 100, where 100 means very confident and 0 means not confident at all."

In tip setting prompts, they prepend the prompt with "I'm going to tip $500k for a better solution.", and in step-by-step prompts, they append "Let's think step by step". Using this methodology, they achieved a maximum accuracy of 90.22% using GPT-4o and step-by-step prompting, and a minimum accuracy of 54.95% using GPT-3.5 Turbo with concise prompting. On average, Claude-3.5 Sonnet performed the best across methodologies, while GPT-4o with step-by-step prompting was the most promising.

Chapter 3

Data Description

This chapter outlines the process of collecting, analyzing, and preparing data for training and evaluation. It provides details about the dataset selection criteria, exploratory data analysis (EDA), and preprocessing steps to ensure the data's suitability for machine learning tasks.

3.1 Dataset Collection

This section describes the gathering, analysis, and preprocessing of data. The initial datasets consisted of Big-Vul [23], CVEFixes [5], CrossVul [50], ReVeal [12], and PrimeVul [18]. After filtering out those focusing solely on C and C++, only CVEFixes and CrossVul contained non-C/C++ data. However, upon closer inspection, the CrossVul dataset revealed numerous empty files and '404: Not Found' errors. This, in combination with Nikitopoulos et.al.'s findings regarding CrossVul's label correctness in contrast to CVEFixes [50], leads to the choice of only using CVEFixes. CVEFixes is a comprehensive dataset, and combining it with the CrossVul dataset may lead to data duplication, given the nature of CVEFixes's data-gathering methodology.

3.2 Exploratory Data Analysis (EDA)

The CVEFixes dataset is provided as an SQL database with an ER diagram depicted in figure 3.1. By using SQL query (A.1) and the Python library Pandas [51], the data is stored in a data frame. There are 16 123 occurrences of method changes without an associated programming language, 14 651 changes in C, 5 409 changes in C++, and a single occurrence with the programming language *None*, all of which are filtered out. This data is visualized in figure 3.2.

A comparison between the number of vulnerable vs. non-vulnerable code snippets for the ten most common languages from figure 3.2 is shown in figure 3.3. These figures highlight which languages need data balancing.

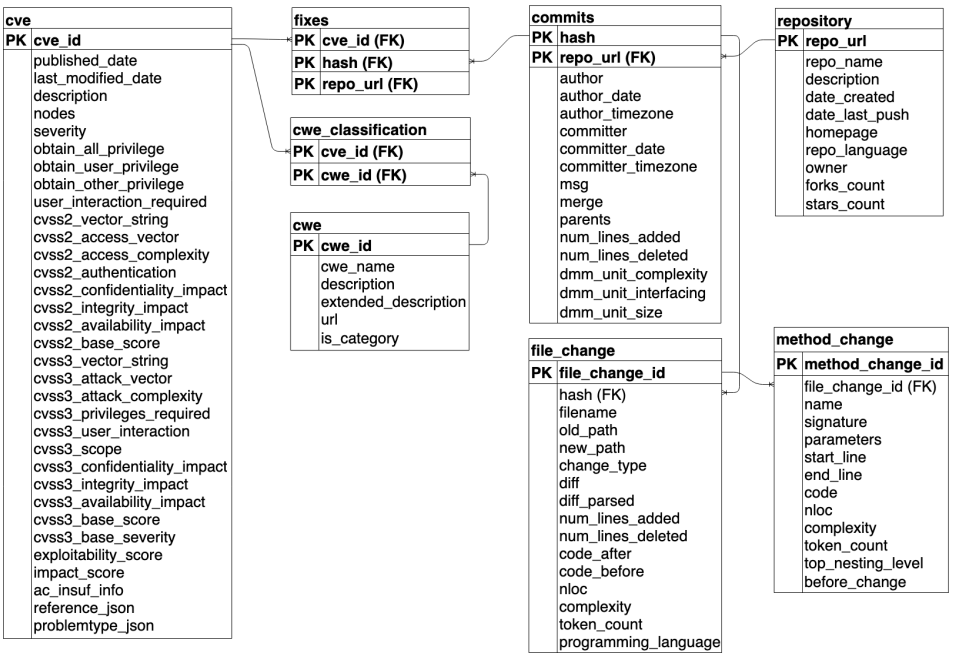

Figure 3.1: The entity-relation diagram from CVEFixes dataset

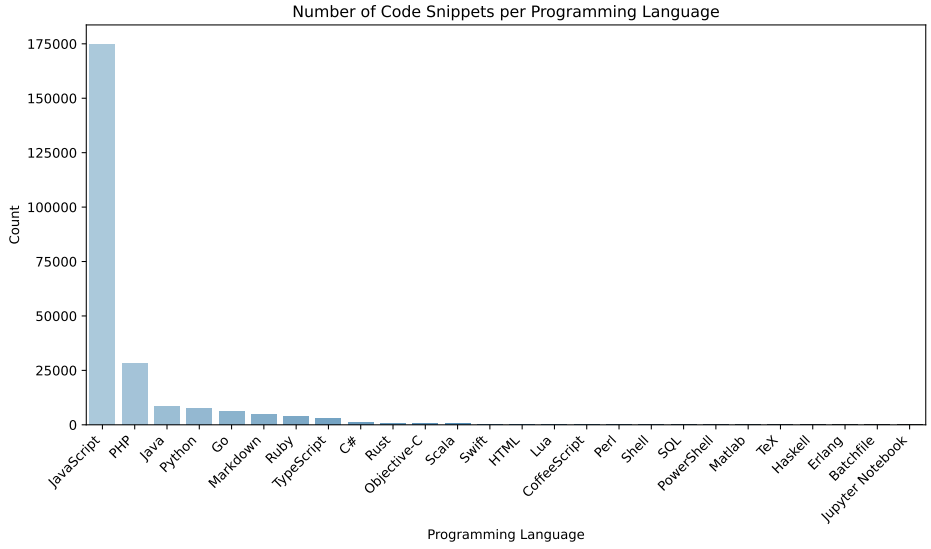

Figure 3.2: Number of Code Snippets per Programming Language

3.3 Data Preprocessing

The common data preprocessing includes formatting the data for each language to a JSON file [54] with one column being the code function, and another column

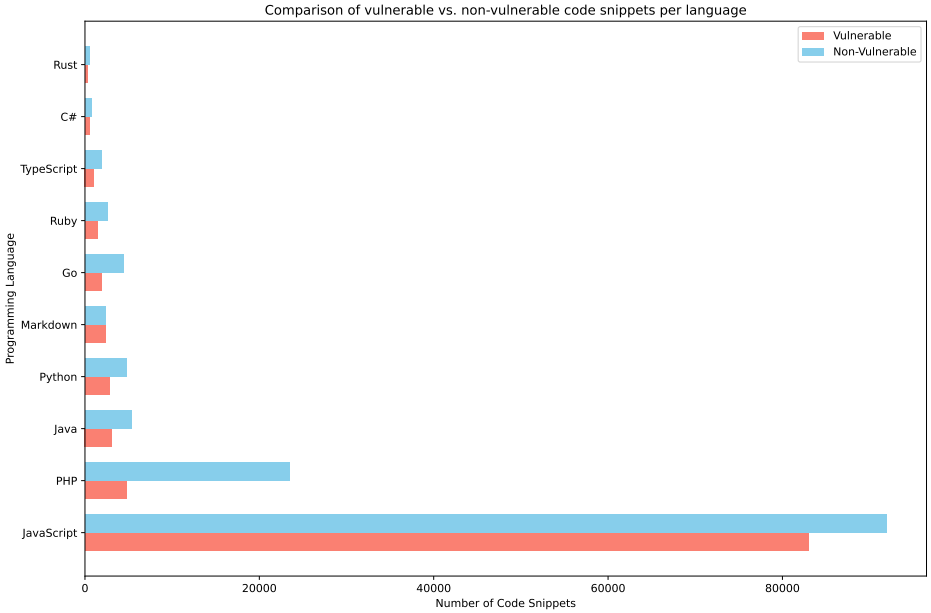

Figure 3.3: Vulnerable vs. non-vulnerable for the ten most frequent languages.

being whether the code is vulnerable or not, using the integer 1 as a label for vulnerability, and 0 otherwise. Duplicated code entries are made unique. The data is then split into 80% training data, 10% evaluation data, and 10% test data.

An issue in machine learning is data leakage. Data leakage is when the model is trained on data that is later used for testing and evaluation. This is mitigated by removing duplicated code entries, leaving the main issue to be the concept of time traveling, as described in Ding et.al.'s paper [18]. Time traveling applies in this case since there is a possibility that historically written code did not have the same bugs as modern code. Therefore, data is split on the date the bug was committed to the public repository, which is gathered from the CVEFixes dataset.

Chapter 4
Methodology

This chapter outlines the methodologies employed to fine-tune and evaluate a large pre-trained language model for the task of code vulnerability detection. It details the fine-tuning process, hyperparameter selection, model architecture, and the tools used. Additionally, it describes the implementation of an API, a continuous integration/continuous deployment (CI/CD) pipeline, and a web-based demonstration for practical application.

4.1 Model Fine-tuning and Hyperparameter Selection

This section describes how the fine-tuning was performed, and which parameters were chosen. The model fine-tuning was conducted using PyTorch [52] and HuggingFace's Transformers libraries [69], with the DeepSeek-Coder 1.3B [31] pre-trained model as the base. The fine-tuning involved adapting the pre-trained model for a binary classification task, e.g. classifying code functions with one of two labels: vulnerable or non-vulnerable. The training was run on four NVIDIA A100 Graphical Processing Units (GPUs)[1] with 80 GB of memory per GPU device.

4.1.1 Motivation of HuggingFace

The HuggingFace[2] ecosystem has rapidly become a go-to platform for working with LLMs due to its comprehensive library of pre-trained models, ease of use of APIs, thorough documentation, and robust developer tools. Its extensive support for transformers, and state-of-the-art models for Natural Language Processing (NLP) tasks, makes it highly suitable for the purpose of this thesis.

By using HuggingFace libraries, this thesis benefits from a state-of-the-art, flexible, and easy-to-use toolset designed for modern machine-learning workflows, making it an ideal choice for the thesis.

4.1.2 Hyperparameters

When fine-tuning a model, some parameters can be set to obtain better results. The used parameters are described here.

[1] `https://www.nvidia.com/en-us/data-center/a100/`
[2] `https://huggingface.co/`

Learning Rate

The learning rate, also known as step size, is a tuning parameter that determines the step size at each iteration, concerning minimizing the loss function. The value decides how large corrections should be made between each iteration in an epoch. In general machine learning algorithms, setting the learning rate to a small value will result in the model converging to a minimum loss slowly, while setting it too large may instead lead to failure in converging towards a minimum loss [45].

A learning rate of 2×10^{-5} was chosen to ensure gradual and stable updates during fine-tuning. Fine-tuning pre-trained models requires careful optimization because these models have already learned generalized patterns from large-scale datasets. A small learning rate prevents significant changes to the pre-trained weights, which could lead to catastrophic forgetting of previously learned information [42]. A small value is commonly used in fine-tuning transformer-based models (such as BERT, GPT, etc.) and is effective for making subtle adjustments without destabilizing the model's performance. Using a value that is too small will find an error minimum, at the cost of time required to fine-tune the model [53]. The selected value was sufficient to prevent catastrophic inference and maintain a high level of learning per epoch.

Number of Epochs

An epoch is a single pass through the entire dataset, first training, followed by evaluation [45]. The number of epochs was set to 10 to provide sufficient opportunity for the model to adapt to the specific task, without risking overfitting, e.g. "[...] creating a model that matches (memorizes) the training set so closely that the model fails to make correct predictions on new data. An overfit model is analogous to an invention that performs well in the lab but is worthless in the real world" [27]. Fine-tuning generally requires fewer epochs compared to training a model from scratch, as the model has already been exposed to a large amount of general-purpose code data. Ten epochs provide a balance between underfitting and overfitting, allowing the model to converge appropriately for most tasks without over-adapting to the fine-tuning data.

Batch Size

The batch size represents the number of samples used in one iteration[3]. A small batch size has the advantage of capturing more details in the training data, resulting in higher accuracy, in contrast to using gradients to generalize the data. This comes at the cost of time needed to train the models [34]. Additionally, the technical limitations of GPU memory when using a larger batch size for fine-tuning a model of this size were ruled out.

[3] `https://www.sabrepc.com/blog/Deep-Learning-and-AI/Epochs-Batch-Size-Iterations`

Seed

The seed[4] is a value used for randomness. Using the same seed should always produce the same results, despite any randomness involved. In this training, the seed value was arbitrarily set to 42 to ensure reproducibility. The seed controls the randomness involved in processes such as data shuffling and weight initialization, ensuring that experiments can be replicated under the same conditions. The choice of 42 is arbitrary but widely recognized in the machine learning community as a standard seed value, allowing for consistent results across multiple training runs.

Summary

The following hyperparameters were chosen for fine-tuning the models:

```
hyperparameters = {
    "learning_rate": 2e-5,
    "num_epochs": 10,
    "train_batch_size": 1,
    "eval_batch_size": 1,
    "seed": 42,
}
```

4.1.3 Fine-tuning

In this subsection, the fine-tuning methodology will be described.

Tokenization

A tokenizer is responsible for segmenting the text into a vocabulary, and making it suitable for machine learning by assigning arbitrary (unique) integer indices to the vocabulary entries. DeepSeek Coder uses HuggingFace's Tokenizer[5] to implement the Bytelevel-BPE [65] algorithm [31]. DeepSeek-Coder was chosen due to its high rating on the EvalPlus Leaderboard[6], in second place with GPT-4-Turbo being in first place, combined with the fact that it is free and open-source.

The tokenization of the textual data was done using a pre-trained tokenizer from DeepSeek-Coder, with the special token [PAD] added for padding.

Texts were truncated to a maximum length of 4020 tokens to ensure consistency in the input size, and the tokenization function applied truncation and padding for each input example. After tokenization, the dataset was reformatted into PyTorch tensors[7] to enable efficient processing during model training and evaluation.

[4] https://en.wikipedia.org/wiki/Random_seed
[5] https://huggingface.co/docs/transformers/en/main_classes/tokenizer
[6] https://evalplus.github.io/leaderboard.html
[7] https://pytorch.org/docs/stable/tensors.html

Model Architecture

The model architecture used is the `AutoModelForSequenceClassification`[8] from the HuggingFace library. This library is a configuration wrapper, which allows for multiple model architectures, with the shared element of the models being sequence classification types, e.g. taking a sequence of text and classifying it into a category, such as vulnerable or non-vulnerable code. The architecture is initialized with the same checkpoint as the tokenizer and adapted for a binary classification task by setting the `num_labels` parameter to 2. The model's embedding layer was resized to accommodate the tokenizer's vocabulary, such as the additional padding token.

Training

Training and evaluation were carried out using data loaders created via PyTorch's `DataLoader`[9], a library for wrapping PyTorch Datasets with an iterator for ease of access to samples, compatible with HuggingFace, with batch sizes of 1 for both training and evaluation to handle memory limitations and potentially improve the generalization of the model on smaller datasets.

The training loop was implemented using the `Accelerate` library from HuggingFace, which enables and optimizes model training across various hardware configurations, including multi-GPU and distributed environments [29]. The main components of the training procedure can be described as:

- **Data Preparation:** The function begins by preparing the data loaders for both training and evaluation datasets. The `set_seed` function was invoked to ensure reproducible results, and the optimizer used was AdamW, a variant of Adam that incorporates weight decay to mitigate overfitting.
- **Learning Rate Schedule:** A linear learning rate scheduler with warm-up steps was employed. The warm-up period consists of 1000 steps, during which the learning rate gradually increases to the specified maximum, after which it decays linearly. This helps stabilize the model's learning early in the training process and prevents abrupt changes in the gradient updates, which can lead to convergence issues.
- **Training Loop:** The model was trained over 10 epochs, as specified by the hyperparameters. For each epoch, the model alternates between training and evaluation modes. During training, the model's predictions and losses were calculated using the tokenized inputs. The optimizer was updated at each step, and gradient accumulation was managed by the Accelerator library to ensure proper synchronization across devices in a distributed setting.
- **Evaluation Loop:** After each epoch, the model was evaluated on the validation set. Predictions from the model were collected and evaluated using several key metrics, further discussed in section 4.2 and 5.1.

[8] `https://huggingface.co/transformers/v3.0.2/model_doc/auto.html#automodelforsequenceclassification`

[9] `https://pytorch.org/tutorials/beginner/basics/data_tutorial.html`

- **Model Saving:** For each epoch, the model weights and tokenizer were saved to the disk, enabling later analysis or further fine-tuning.

4.2 Evaluation

This section outlines the evaluation methodology used to assess the performance of the fine-tuned model. The evaluation measured the model's ability to classify code correctly. Two types of evaluation were performed on the models: one between each epoch using the evaluation data set and one after the model had finished training utilizing the test set to ensure that the model did not learn from the test evaluation.

The model was evaluated using a held-out validation set, consisting of domain-specific data points not included in the training data. This dataset acts as unseen real-world data and helps estimate the model's generalization capabilities. The use of a validation set ensures that the model is not merely memorizing the training data but can generalize its learned patterns to new inputs [36].

4.3 API Building

An API is developed using Flask [28], a lightweight Python web framework, to provide an interface for users to submit code functions for vulnerability detection. The API operates through different endpoints that process incoming requests, verify user authentication, and return results from model inference.

4.3.1 Framework and Libraries

Flask was chosen for its simplicity and flexibility in handling web requests, managing routing, and handling other aspects of web development. Several additional libraries and sub-libraries of Flask were integrated, such as Flask-JWT-Extended for token-based authentication to secure API endpoints, Flask-Bcrypt for password hashing, and SQLAlchemy for facilitating database interaction, token, and user data management.

4.3.2 Serving the Models

The core functionality of the API revolves around the AI models. These models were initialized in the backend using the HuggingFace Transformers pipeline [69] for text classification. The API requires a valid access token in the request headers to ensure that only authenticated users can access the vulnerability detection models. The token's validity is confirmed by looking up the corresponding user in the database.

The primary API endpoint for vulnerability detection is the `/predict` route, which accepts POST requests containing code functions. This is sent to the endpoint as JSON formatted text. The structure of the request is expected to contain function names grouped by programming language, with each function name pointing to the function code. This allows the backend to predict the vulnerability of

each function, and return the predictions grouped by language and function name for the user to analyze.

4.3.3 User Handling

Endpoints were implemented to manage user authentication and session control within the API.

Registration

Users may register through the `/register` endpoint. This endpoint accepts POST requests containing the user's username, email, and password. The password is hashed using Flask-Bcrypt before being stored in the database to ensure security. The user is also assigned a UUID (Unique User ID) token, which is used for authentication purposes when making API requests to the `/predict` endpoint.

Login

The `/login` endpoint allows users to log in by sending a POST request containing their username and plaintext password. Since the backend is storing the hash of a password, any login attempt will securely transmit the plaintext password through HTTPS, and the hash of the plaintext password can be compared to the (saved) hashed password in the backend. A successful comparison will return the user's UUID token.

GitHub OAuth2 Integration

The API integrates OAuth2 authentication with GitHub to provide users with an additional login method. Users may then log in through GitHub, giving them the option to memorize one less password. If it is their first login through GitHub OAuth, they are granted a unique UUID token, acting as a registration.

4.4 CI/CD Pipeline

A CI/CD pipeline was built using GitHub Actions and GitLab Jobs to integrate the vulnerability detection models into the development workflow. This pipeline aims to automate the process of code vulnerability analysis whenever changes are pushed to a repository. For the pipeline implementations, a script is created to extract the modified functions from the most recent commit, including which language the code was written in, determined by the file ending.

The CI process is set to trigger on any commit or pull request. Upon a trigger, the changed functions are extracted from the code. The changed functions are then sent to the API for vulnerability detection, and if any vulnerabilities are detected, an email is sent to a user-specified email.

4.5 Website & Proof of Concept Deployment

To facilitate the usage and demonstration of the vulnerability detection models, a website was developed. The website serves two main purposes: managing user authentication and providing a proof of concept demonstration.

The website is deployed on a server rented from Hetzner[10], a cloud service provider. Since the server is responsible for performing model inference, somewhat high requirements had to be met. The server runs on an Intel Core i7 7700k[11] CPU with 62 GB of usable RAM. Additionally, the domain name `vyprai.net` was leased to enhance the user experience of accessing the website. An SSL certificate, needed for HTTPS communication, was issued by Lets Encrypt[12].

4.5.1 Website Development

The website consists of an HTML and JavaScript-based frontend, communicating with the Flask backend server. It features five pages: The homepage (/), a login page (/login), a signup page (/signup), a user page (/user), and a page for interacting with the model (/demo).

A specification of the pages is as follows:

- **Homepage:** The homepage features buttons for logging in and signing up if not already logged in, else a button to the user page and a logout button. It queries the Flask backend to determine whether the user is logged in.
- **Login page:** The login page has text fields for the user to enter their email and password. It sends this information to the backend to compare the password hash against the saved hash. Additionally, a button for logging in through GitHub OAuth registers the user if it is their first time logging in. Logging in results in Flask creating a session for the user, communicated through a session token between Flask and the client.
- **Signup page:** The signup page has text fields for email, username, and password. It sends this to the backend to create a user, adding a user to the database and creating the UUID token.
- **User page:** This page acts as the user portal/dashboard, fetching the user's UUID token from the backend. It also contains a button for returning to the homepage and a button for logging out.
- **Demo page:** To interact with the model and experiment with its predictions, the demo page was built. It has a text box where a user can write code functions, a language select module specifying which language the code function is written in, and a submit button. Upon clicking the submit button, a POST request to the backend is made, including the code, the code language, and the logged-in user's session token. The backend validates the

[10] https://www.hetzner.com/

[11] https://www.intel.com/content/www/us/en/products/sku/97129/intel-core-i77700k-processor-8m-cache-up-to-4-50-ghz/specifications.html

[12] https://letsencrypt.org/

request, performs the model inference, and returns the result. The frontend will subsequently present the result using a color scheme based on the prediction and the confidence of the prediction, ranging from full green (not vulnerable) to full red (vulnerable), and hues of orange indicating low confidence.

Chapter 5
Results

This chapter presents the outcomes of our experiments and evaluations, highlighting the performance of the model in detecting code vulnerabilities. The results are analyzed using established performance metrics. Additionally, the chapter delves into comparative benchmarks with previous studies, the deployment of a CI/CD pipeline, and the integration of the model into a web-based interface. These findings demonstrate the strengths and limitations of our approach while providing insights for practical application and future improvements.

5.1 Model Performance Metrics

The metrics used to quantify the model's performance included accuracy, precision, recall, F1 score, and a confusion matrix. These metrics were chosen to provide a comprehensive understanding of the model's strengths and weaknesses across various performance aspects.

5.1.1 Confusion Matrix

The confusion matrix in binary classification is a 2x2 matrix describing a model's predictions [24]. Generally, it can be depicted as in table 5.1.1, where the positive label in this case is the equivalent of a vulnerable code function, and a negative is not vulnerable. The cells in the matrix account for the number of predictions made for the actual values.

This matrix enables the calculation of some additional metrics such as:

- False Positive Rate (FPR) = $\frac{FP}{FP+TN}$
- False Negative Rate (FNR) = $\frac{FN}{FN+TP}$
- True Positive Rate (FPR) = $\frac{TP}{TP+FN} \Leftrightarrow Recall$
- True Negative Rate (TNR) = $\frac{TN}{TN+FP}$

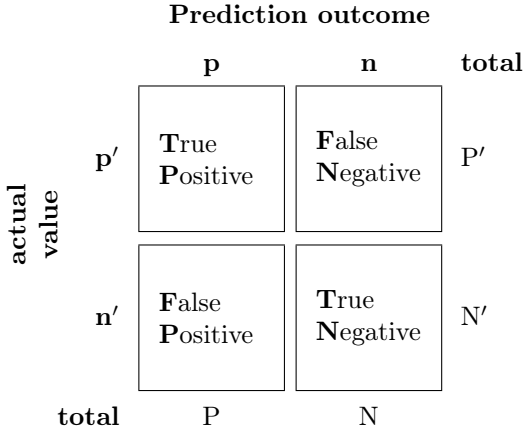

Figure 5.1: General confusion matrix

5.1.2 Accuracy

The accuracy metric describes the ratio of correct predictions to the total number of predictions. It is defined as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

5.1.3 Precision

Precision measures how many of the predicted positive instances were true positives. This metric is valuable when the cost of false positives is high and can be used to minimize false alarms. An example of a scenario where this metric should be high is if the models are used in a CI/CD pipeline. False positives in a continuous environment may be significantly problematic [38]. Precision is defined as:

$$Precision = \frac{TP}{TP + FP}$$

5.1.4 Recall

Recall, also known as sensitivity, measures how well the model captures all the positive instances. It is equivalent to TPR. This metric provides valuable insight to use cases where the cost of missing a true positive (such as failing to identify a vulnerability) comes with a higher cost than predicting false positives. The recall value is defined as:

$$Recall = \frac{TP}{TP + FN}$$

Language	Acc	F1	Prec	Rec	FPR
JavaScript	0.6959	0.6555	0.7513	0.5814	0.1907
PHP	0.8284	0.3659	0.3840	0.3495	0.0925
Java	0.6859	0.6918	0.7133	0.6717	0.2983
Python	0.6019	0.5327	0.6007	0.4785	0.2868
Go	0.8530	0.4161	0.8378	0.2768	0.0125

Table 5.1: Evaluation results of the different models

Model	Acc	F1
CodeT5 [66]	0.9667	0.1970
CodeBERT [25]	0.9687	0.2086
UnixCoder [30]	0.9686	0.2143
StarCoder2 [40]	0.9702	0.1805
CodeGen2.5 [49]	0.9665	0.1961

Table 5.2: Benchmarks from PrimeVul [18] on C and C++ functions

5.1.5 F_1 Score

The F_1 measurement is defined as a harmonic mean of precision (P) and recall (R) [59], giving a balanced metric, and is calculated by:

$$F_1 = \frac{2P \times R}{P + R} = \frac{2TP}{2TP + FP + FN}$$

A high F_1 score indicates that the model performs well on both precision and recall, making it a suitable metric for situations where both false positives and false negatives should be minimized.

5.2 Model Performance

We evaluated model performance using accuracy, precision, recall, F_1 score, and a confusion matrix. Table 5.1 presents the results for the different language models. These values can be compared to PrimeVul's findings, shown in table 5.2. There is a significant increase in F_1 scores across all our language models. The accuracy is higher from PrimeVul's findings, indicating that they had a data imbalance greater than ours. It should be noted that the data imbalance is not the sole reason for the lower overall performance (measured by the F_1 score).

5.3 CI/CD Pipeline

CI/CD scripts were successfully developed for both GitHub and GitLab. The CI/CD scripts are similar and can be reviewed in the appendix. See appendix A.2 for the GitHub action, and appendix A.3 for the GitLab job. Figure 5.2 illustrates

the data flow occurring when a commit is pushed/pull requested, highlighting all steps up until the API.

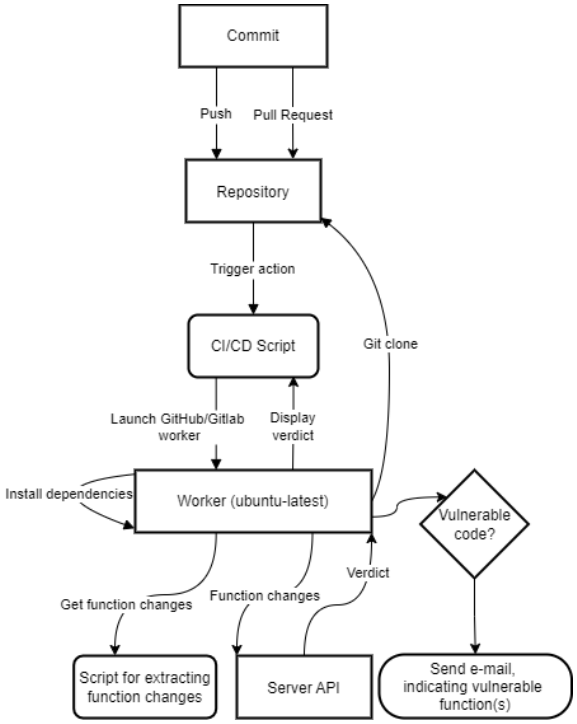

Figure 5.2: Data flow diagram of CI/CD pipeline

5.4 Website Functionality and API Deployment

The system architecture of the website solution is depicted in figure 5.3, illustrating the core functionality of the front- and backend.

5.4.1 User Interface and Experience

The User Interface and Experience (UI/UX) consists of several pages on the website. Screen captures of the website are available in the appendix, see appendix A.1 and A.2 (home page), A.3 (demo page), A.4 (login page), A.5 (signup page), and A.6 (user page).

5.4.2 Performance and Scalability

To measure how well the API performs, time measurements were taken for varying sizes of input to each model. The results of the measurements are in table 5.3.

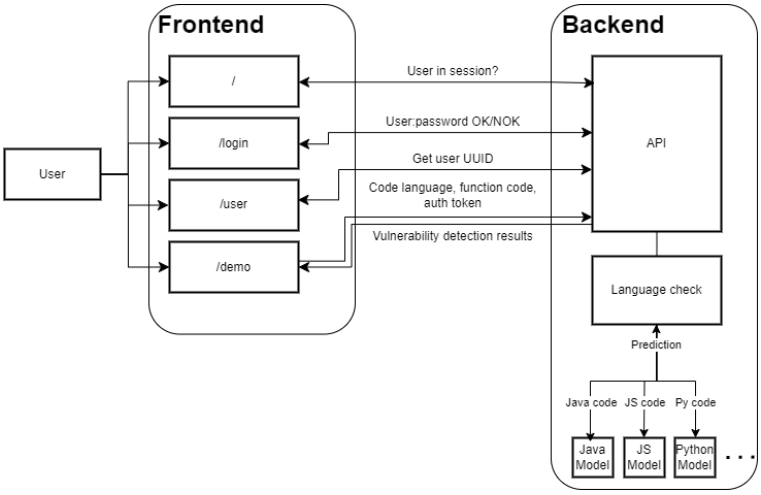

Figure 5.3: System overview

Language	**Data Size (Characters)**			
	10	100	1000	10000
JavaScript	0.8381	1.0308	6.4366	75.5222
PHP	0.8147	1.0373	6.4842	76.3413
Java	0.8347	1.0977	6.7083	75.5808
Python	0.7475	1.0616	6.3377	75.3490
Go	0.7263	1.0830	6.2177	75.8672
Mean	**0.7923**	**1.0621**	**6.4369**	**75.7321**

Table 5.3: Time in seconds needed per number of characters, for each language and data size.

Chapter 6
Discussion

This chapter evaluates the effectiveness and implications of using Large Language Models (LLMs) for vulnerability detection, considering their strengths, limitations, and areas for improvement. It examines the nuances of applying LLMs in practical scenarios, comparing their performance to traditional vulnerability detection systems, and analyzing their applicability across different programming languages. Additionally, the chapter explores real-world deployment challenges, ethical considerations, and the potential business impact of integrating LLMs into security workflows. These discussions aim to comprehensively understand the capabilities and responsible usage of AI-driven vulnerability detection systems.

6.1 Effectiveness of LLMs for Vulnerability Detection

While the LLMs demonstrated comparatively strong performance in detecting vulnerabilities in structured datasets, the real-world application remains challenging due to varying code complexity and inconsistent labeling in existing vulnerability datasets. In practice, code vulnerabilities are rarely as cleanly labeled or isolated as in curated datasets, and context is often essential for accurate detection. For instance, a function that appears non-vulnerable in isolation may pose a significant risk when interacting with other system components or external inputs.

Additionally, the model's reliance on data from specific sources (e.g., CVEFixes) potentially limits its ability to generalize, particularly for novel or edge-case vulnerabilities that have not been frequently documented. Overcoming these issues would likely require dynamic learning systems that continuously ingest and learn from newly reported vulnerabilities, adapting to evolving coding practices and security standards, as well as more training on uncommon yet well-known security issues.

As an example, by submitting blatantly vulnerable code, such as the code in listing 6.1 displaying a command injection vulnerability (CWE-78)[1], the model appears to be uncertain on whether it is vulnerable or not, giving a low confidence score. One hypothesis regarding why this is happening is that the model has not seen code with obvious vulnerabilities present, but has rather been trained on real vulnerabilities in production code - typically more complicated code functions with

[1] `https://cwe.mitre.org/data/definitions/78.html`

more subtle vulnerabilities.

Listing 6.1: Vulnerable (Confidence: 0.64778)

```
function example() {
  var userCode = prompt("Enter code to execute: ");
  eval(userCode);
}
```

Another observation made is that the model handles human language context. This is likely because the model is suitable for NLP tasks. As an example, see the following three Python code functions and their predictions.

Listing 6.2: Vulnerable (Confidence: 0.97984)

```
def func():
  os.exec('ls' + input('Enter directory to list'))
```

Listing 6.3: Vulnerable (Confidence: 0.99992)

```
def func():
  os.exec(input())
```

Listing 6.4: Not vulnerable (Confidence: 0.99968)

```
def safe():
  # This function is only called from a trusted path
  os.exec(input())
```

In listings 6.2 and 6.3 we see that the model has classified the code as vulnerable, with high confidence scores. However, by renaming the function to *safe* and adding a code comment as in listing 6.4, we manage to convince the model that the code does not contain vulnerabilities. This could happen since we use natural language to indicate that the function is safe, which it may be, depending on the context.

The fact that natural language plays a role in the model predictions comes with both benefits and drawbacks. The greatest benefit is that the model can contextualize knowledge in the form of e.g. code comments, that other tools such as SAST, may disregard. If a code function has comments documenting how the code works, the AI models may find that the code will never be executed and thus should be marked as safe. A SAST tool would not be able to draw this conclusion based on code comments, and could therefore raise false positives.

6.1.1 Comparison to Other Vulnerability Detection Systems

To compare how well the models compare to traditional vulnerability detection systems, we investigate the differences between benchmarks gathered by VULMG [32]. They benchmarked six different vulnerability detection systems, namely MG-GAT [39], FlawFinder [26], RATS[2], BiLSTM[3], SVM[4], and GCN [60] (a subclass of

[2] `https://code.google.com/archive/p/rough-auditing-tool-for-security/`

[3] `https://paperswithcode.com/method/bilstm`

[4] `https://en.wikipedia.org/wiki/Support_vector_machine`

artificial neural networks). VULMG found that the different tools and approaches had F_1 scores ranging from 43.39% to 94.43% when experimenting on CWE-369 (Division by zero) and between 0.00% to 96.30% when running the same tests on CWE-476 (Null pointer dereference).

Despite these high F_1 scores, it should be taken into consideration that these benchmarks were taken only on a single CWE at a time, which is significantly different from finding any vulnerability classified under any CWE. In a study conducted by Atiiq et. al., it was found that fine-tuning on CWE-specific vulnerabilities gives better F_1 scores [2] than fine-tuning on all vulnerabilities (such as performed in PrimeVul and this thesis). It may be problematic to have separate models from a deployment perspective, as the question arises on how to handle inference on a code function, e.g. whether the code functions should be inferred by all models, or a single one, and the implications on time/resource costs when performing inference through a multitude of models.

6.1.2 Vulnerability Detection in Higher-Level Languages

The model's focus on high-level languages like JavaScript, Python, and PHP, offers a fresh perspective on vulnerability detection outside the traditional C/C++ domain. In lower-level languages, such as C and C++, memory has to be explicitly handled by the developer, introducing a range of new vulnerability classes, such as CWE-121, CWE-122, and CWE-127, to name a few. These memory-related CWEs have been summarized under the CWE Category 1399[5]. All memory management has been abstracted away in higher-level languages, relying on automated garbage collection and built-in safety mechanisms. This removes a lot of the potential CWEs, allowing the model to focus more on those that remain across high-level languages, possibly giving it a better basis to make predictions.

On the contrary, the languages may introduce new CWEs. For example, JavaScript is susceptible to attacks such as cross-site scripting (XSS) and client-side injection, due to its extensive use in web applications and its interaction with HTML/DOM objects.

Based on the results found in Atiiq et. al.'s study [2] and VULMG's findings [32], it may be stated that having fewer CWE vulnerability possibilities leads to better vulnerability detection. To further add to the claim, or to disprove it, additional research into how many CWEs were dropped and added should be conducted, investigating any possible correlation between the number of CWEs and vulnerability prediction performance.

6.2 API and Website Deployment: Real-World Applicability

The developed API and website offer practical solutions for real-world applications, but time costs associated with the model's inference for larger code snippets remain a limitation. In CI/CD pipelines, where code scans should occur seamlessly without delaying deployment, even minor lags can accumulate, impacting productivity and increasing operational costs. Integrating a lightweight version

[5] `https://cwe.mitre.org/data/definitions/1399.html`

of the model or a heuristic-based pre-screening step may help mitigate this by pre-filtering non-vulnerable code blocks before running comprehensive scans on remaining snippets. The promising results with JavaScript and Go, which exhibited lower false positives, indicate these languages as viable starting points for initial integrations in CI/CD environments.

Although the time cost may be a hindrance for use in a CI/CD pipeline, this is relatively easy to mitigate, especially if deployed for large-scale use with funding available. The inference could be performed much quicker by deploying the models on a more capable server, perhaps using GPU-accelerated computing.

6.2.1 Business Aspects

Although some approaches have higher F_1 scores, there is a significant difference in ease of use. Leveraging AI capabilities through an API easily integrated into a CI/CD pipeline can be a valuable addition to identifying weaknesses in code. The models can provide additional insight into pull requests that SAST tools may fail to detect. It can be taken into account that the models do not necessarily act as a replacement for these tools or manual code reviews but may complement these systems.

6.3 Ethical and Security Considerations

The deployment of AI-driven vulnerability detection systems introduces several ethical and security implications, particularly when such tools are made publicly accessible. While LLMs democratize access to security resources, they also carry the risk of misuse if accessible by individuals with malicious intent. By providing a tool for identifying weaknesses in code, an ill-intended attacker could use the tool to find weaknesses in, for example, open-source code, to exploit any present vulnerability. In addition, dependency on automated systems could introduce a false sense of security [11]; human oversight remains essential to validate findings and interpret the nuances of vulnerabilities flagged by the model.

To mitigate these risks, implementing restricted access and thorough user verification for public APIs could help control use and prevent abuse. Furthermore, caution should be taken to ensure that detection results are not over-relied upon without validation, as vulnerabilities flagged by AI might require context-specific understanding for accurate assessment. This ethical dimension underscores the importance of responsible use, highlighting the need for transparency and appropriate user training when integrating these tools into broader security practices.

Chapter 7
Conclusion

This thesis explored the potential of fine-tuned Large Language Models to enhance vulnerability detection in non-C/C++ programming languages, a critical yet underexplored area in software security research. As modern software ecosystems increasingly rely on high-level languages like JavaScript, Python, and Java, traditional vulnerability detection methods often fall short of identifying the risks posed by these environments. Our work directly addresses this gap by employing a data-driven approach, fine-tuning an LLM on curated datasets, and deploying it via an accessible API within CI/CD pipelines.

The results demonstrate that LLMs, when specifically trained on security-related data, can meaningfully detect vulnerabilities in high-level languages, albeit with varying degrees of effectiveness. Performance metrics reveal a strong aptitude for identifying vulnerabilities in Java and JavaScript, indicating that LLMs can generalize and understand syntactic and contextual clues in these languages to highlight security weaknesses. However, the model's lower performance in PHP highlights the challenges inherent in universalizing detection capabilities across diverse languages with distinct syntactical and structural features. These results point toward a broader implication: while LLMs hold promise for vulnerability detection, their effectiveness hinges heavily on the quality, balance, and representativeness of the training datasets, as well as language-specific nuances.

Beyond the technical findings, this thesis also tackled the practical deployment of AI-driven vulnerability detection in real-world development environments. Through the development of a CI/CD pipeline and a user-friendly web interface, this project aimed to demonstrate how vulnerability detection could be integrated into standard development workflows without imposing significant overhead. Despite the model's inference time for larger code snippets—which could be a potential bottleneck in high-frequency CI/CD environments, this research underscores the practical viability of such a system, particularly when deployed with sufficient computational resources or alongside more lightweight pre-screening models. This approach offers an adaptable framework that could be extended to multiple development environments, opening the door for AI-enhanced security practices in daily coding operations.

However, the benefits of LLM-based vulnerability detection also raise important ethical and security considerations. While these models can democratize access to advanced security insights, they also carry the risk of misuse, particularly if publicly accessible without access controls. There is a delicate balance between

providing developers with tools to identify vulnerabilities and safeguarding against potential exploitation by malicious actors. Moreover, as with any automated system, there is a risk of over-reliance, which could lead to a false sense of security. Thus, it is crucial that AI-powered tools remain supplementary to human review, and that they operate under clear ethical guidelines to prevent misuse and maintain trust.

7.1 Future Work

Future research in this domain could address improvements in the datasets, and hybrid models that combine LLMs with traditional static analysis tools. Additionally, employing continual learning techniques[1] could allow models to dynamically update with the latest vulnerability data, increasing their relevance in a rapidly evolving threat landscape. Research into optimizing model efficiency and reducing latency, especially within the constraints of CI/CD pipelines, will also be essential for widespread adoption.

Another possibility for future research would be to perform CWE-specific classification, similar to Atiiq et. al.'s paper [2], but for higher-level languages, and how to efficiently deploy the models for a real-world application.

Ultimately, this thesis contributes to a foundational understanding of how LLMs can be applied to vulnerability detection beyond low-level languages, offering a springboard for future innovations in AI-driven cybersecurity. By advancing the technical capabilities and practical applications of LLMs in this field, this research takes a meaningful step toward bridging the gap between theory and real-world security needs, highlighting the promise and responsibility accompanying AI's role in modern software development.

[1] `https://neptune.ai/blog/continual-learning-methods-and-application`

References

[1] Luis B Almeida. Multilayer perceptrons. In *Handbook of Neural Computation*, pages C1–2. CRC Press, 2020.

[2] Syafiq Al Atiiq, Christian Gehrmann, Kevin Dahlén, and Karim Khalil. From generalist to specialist: Exploring cwe-specific vulnerability detection. https://arxiv.org/abs/2408.02329, 2024. Accessed: 2024-11-14.

[3] Jaehyeon Bae, Seoryeong Kwon, and Seunghwan Myeong. Enhancing software code vulnerability detection using gpt-4o and claude-3.5 sonnet: A study on prompt engineering techniques. *Electronics*, 13(13), 2024.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[5] Guru Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2021, page 30–39, New York, NY, USA, 2021. Association for Computing Machinery.

[6] M. Boehme, C. Cadar, and A. Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 38(03):79–86, may 2021.

[7] Marcel Boehme, Cristian Cadar, and Abhik ROYCHOUDHURY. Fuzzing: Challenges and reflections. *IEEE Software*, 38(3):79–86, 2021.

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. https://arxiv.org/abs/2005.14165, 2020. Accessed: 2024-11-14.

[9] Achim Brucker and Uwe Sodan. Deploying static application security testing on a large scale. In *Sicherheit 2014 – Sicherheit, Schutz und Zuverlässigkeit*, pages 91–101. Gesellschaft für Informatik e.V., Bonn, 2014.

[10] Mateusz Buda, Atsuto Maki, and Maciej A. Mazurowski. A systematic study of the class imbalance problem in convolutional neural networks. *Neural Networks*, 106:249–259, October 2018.

[11] Jeimy Cano. Overcoming a false sense of security: How to deeducate current security and control practices, 2019. Accessed: 2024-10-26.

[12] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2021.

[13] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '23, page 654–668, New York, NY, USA, 2023. Association for Computing Machinery.

[15] Hanting Chu, Pengcheng Zhang, Hai Dong, Yan Xiao, Shunhui Ji, and Wenrui Li. A survey on smart contract vulnerabilities: Data sources, detection and repair. *Information and Software Technology*, 159:107221, 2023.

[16] Julian Cohen. Contemporary automatic program analysis. https://youtu.be/P0nHId1umvY, 2014. Speech performed by Julian Cohen at blackhat USA 2014. Accessed: 2024-11-14.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[18] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? https://arxiv.org/abs/2403.18624, 2024. Accessed: 2024-11-14.

[19] Kohei Dozono, Tiago Espinha Gasiba, and Andrea Stocco. Large language models for secure code assessment: A multi-language empirical study. https://arxiv.org/abs/2408.06428, 2024. Accessed: 2024-11-14.

[20] Aaron Drapkin. What is claude ai and anthropic? a closer look at chatgpt's rival. `https://tech.co/news/what-is-claude-ai-anthropic`.

[21] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014*

Conference on Internet Measurement Conference, IMC '14, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery.

[22] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[23] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 508–512, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Tom Fawcett. An introduction to roc analysis. *Pattern Recogn. Lett.*, 27(8):861–874, June 2006.

[25] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. https://arxiv.org/abs/2002.08155, 2020. Accessed: 2024-11-14.

[26] Oliver Ferschke, Iryna Gurevych, and Marc Rittberger. Flawfinder: A modular system for predicting quality flaws in wikipedia. In Pamela Forner, Jussi Karlgren, and Christa Womser-Hacker, editors, *CLEF 2012 Evaluation Labs and Workshop, Online Working Notes, Rome, Italy, September 17-20, 2012*, volume 1178 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[27] Google. Overfitting. `https://developers.google.com/machine-learning/crash-course/overfitting/overfitting#:~:text=Overfitting%20means%20creating%20a%20model,worthless%20in%20the%20real%20world.`, 2024. Accessed: 2024-11-14.

[28] Miguel Grinberg. *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.", 2018.

[29] Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, Sourab Mangrulkar, Marc Sun, and Benjamin Bossan. Accelerate: Training and inference at scale made simple, efficient and adaptable. `https://github.com/huggingface/accelerate`, 2022. Accessed: 2024-11-14.

[30] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. https://arxiv.org/abs/2203.03850, 2022. Accessed: 2024-11-14.

[31] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. https://arxiv.org/abs/2401.14196, 2024. Accessed: 2024-11-14.

[32] Zhang Haojie, Li Yujun, Liu Yiwei, and Zhou Nanxin. Vulmg: A static detection solution for source code vulnerabilities based on code property graph and graph attention network. In *2021 18th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*, pages 250–255, 2021.

[33] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[34] HuggingFace.co. Batch sizes for inference. `https://huggingface.co/docs/setfit/how_to/batch_sizes`, 2024. Accessed: 2024-11-14.

[35] Dorota Huizinga and Adam Kolawa. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Pr, 2007.

[36] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer Publishing Company, Incorporated, 2013.

[37] Moez Krichen. A survey on formal verification and validation techniques for internet of things. *Applied Sciences*, 13(14), 2023.

[38] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim, and Shin Yoo. Classifying false positive static checker alarms in continuous integration using convolutional neural networks. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 391–401, 2019.

[39] Yan Leng, Rodrigo Ruiz, Xiaowen Dong, and Alex Pentland. Interpretable recommender system with heterogeneous information: A geometric deep learning perspective. *SSRN Electronic Journal*, 2020.

[40] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation. https://arxiv.org/abs/2402.19173, 2024. Accessed: 2024-11-14.

[41] Yujie Ma, Haokai Wu, Yu-an Tan, and Yuanzhang Li. Research on evasion and detection of malicious javascript code. In Dan Dongseong Kim and Chao Chen, editors, *Machine Learning for Cyber Security*, pages 104–130, Singapore, 2024. Springer Nature Singapore.

[42] Michael McCloskey and Neal J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In Gordon H. Bower, editor, *Psychology of Learning and Motivation*, volume 24 of *Psychology of Learning and Motivation*, pages 109–165. Academic Press, 1989.

[43] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[44] Amir M. Mir, Mehdi Keshani, and Sebastian Proksch. On the effect of transitivity and granularity on vulnerability propagation in the maven ecosystem. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 201–211, 2023.

[45] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[46] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[47] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[48] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

[49] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. https://arxiv.org/abs/2305.02309, 2023. Accessed: 2024-11-14.

[50] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. Crossvul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1565–1569, New York, NY, USA, 2021. Association for Computing Machinery.

[51] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[52] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[53] Josh Patterson and Adam Gibson. *Deep Learning: A Practitioner's Approach*. O'Reilly Media, Inc., 2017.

[54] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee, 2016.

[55] Sumit Dahiya Piyush Ranjan. Advanced threat detection in api security: Leveraging machine learning algorithms. *International Journal of Communication Networks and Information Security (IJCNIS)*, 13(1):185–196, Feb. 2021.

[56] Salim Rezvani and Xizhao Wang. A broad review on class imbalance learning techniques. *Applied Soft Computing*, 143:110415, 2023.

[57] Steven J. Rigatti. Random forest. *Journal of Insurance Medicine*, 47(1):31–39, 01 2017.

[58] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. https://arxiv.org/abs/2308.12950, 2024. Accessed: 2024-11-14.

[59] Yutaka Sasaki and R Fellow. The truth of the f-measure, manchester: Mibschool of computer science. *University of Manchester*, 25, 2007.

[60] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[61] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, June 2012. USENIX Association.

[62] Shan Suthaharan. *Support Vector Machine*, pages 207–235. Springer US, Boston, MA, 2016.

[63] Larry Suto. Analyzing the accuracy and time costs of web application security scanners. *Think Secure*, 1, 2010.

[64] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Variable precision reaching definitions analysis for software maintenance. In *Proceedings. First Euromicro Conference on Software Maintenance and Reengineering*, pages 60–67, March 1997.

[65] Changhan Wang, Kyunghyun Cho, and Jiatao Gu. Neural machine translation with byte-level subwords. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):9154–9160, Apr. 2020.

[66] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. https://arxiv.org/abs/2109.00859, 2021. Accessed: 2024-11-14.

[67] Gebrehiwet B. Welearegai, Chenpo Hu, and Christian Hammer. Detecting and preventing rop attacks using machine learning on arm. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 667–677, 2023.

[68] B. Wichmann, A.A. Canning, D.L. Clutterbuck, L.A. Winsborrow, N.J. Ward, and William Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 10:69 – 75, 04 1995.

[69] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In Qun Liu and David Schlangen, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[70] Xiaoxue Wu, Wei Zheng, Xiang Chen, Fang Wang, and Dejun Mu. Cve-assisted large-scale security bug report dataset construction method. *Journal of Systems and Software*, 160:110456, 2020.

[71] Xin Zhou, Duc-Manh Tran, Thanh Le-Cong, Ting Zhang, Ivana Clairine Irsan, Joshua Sumarlin, Bach Le, and David Lo. Comparison of static application security testing tools and large language models for repo-level vulnerability detection. https://arxiv.org/abs/2407.16235, 2024. Accessed: 2024-11-14.

Appendix A

Appendix

A.1 SQL

Listing A.1: SQL query to count the number of entries in CVEFixes

```
SELECT fc.programming_language,
       COUNT(mc.method_change_id) AS count
FROM file_change fc, method_change mc
WHERE fc.file_change_id = mc.file_change_id
GROUP BY fc.programming_language
ORDER BY count DESC;
```

A.2 GitHub Actions Script

Listing A.2: GitHub Action script

```
name: Vulnerability Scan

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  vulnerability_check:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3
        with:
          fetch-depth: 0
```

```
- name: Set up Python
  uses: actions/setup-python@v4
  with:
    python-version: '3.8.x'
    update-environment: true

- name: Install dependencies
  run: |
    python -m pip install --upgrade pip
    pip install pydriller
    sudo apt-get update
    sudo apt-get install -y mailutils

- name: Find code changes
  run: |
    python .github/scripts/extract_function.py

- name: Send function to AI server
  env:
    UUID_KEY: ${{ secrets.UUID_KEY }}
  id: send-to-server
  run: |
    curl -X POST \
    -H "Content-Type: application/json" \
    -H "Authorization: $(UUID_KEY)" \
    -d @function_changes.json \
    https://vyprai.net/api/predict \
    -o ai_response.json

- name: Display AI vulnerability analysis result
  run: |
    cat function_changes.json
    cat ai_response.json
    VULNERABLE=\
    $(jq '.py[] | select(.label == "LABEL_1")' \
    ai_response.json)
    if [ -n "$VULNERABLE" ]; then
      echo "Vulnerabilities detected in \
      the following function(s): \
      $(jq -r '.py | to_entries[] \
      | select(.value.label == "LABEL_1") \
      | "\(.key)"' ai_response.json)" \
      | mail -s "Vulnerability Detected" \
      email@domain.tld
    fi
```

A.3 GitLab Job Script

Listing A.3: GitLab Job script

```
stages:
  - vulnerability_scan

variables:
  PYTHON_VERSION: "3.8"

vulnerability_scan:
  stage: vulnerability_scan
  image: python:3.8
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
    - if: '$CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "main"'
  before_script:
    - apt-get update
    - apt-get install -y curl git
    - python --version
  script:
    - 'python -m pip install --upgrade pip'
    - pip install pydriller
    - python .gitlab/scripts/extract_function.py
    - 'curl -X POST \
      -H "Content-Type: application/json" \
      -H "Authorization: $(UUID_KEY)" \
      -d @function_changes.json \
      https://vyprai.net/api/predict \
      -o ai_response.json'
    - cat function_changes.json
    - cat ai_response.json
    - git --version
    - git log -2
    - git status
```

A.4 Website Components

A.4.1 Home Page

A.4.2 Demo Page

A.4.3 Login Page

A.4.4 Signup Page

A.4.5 User Page

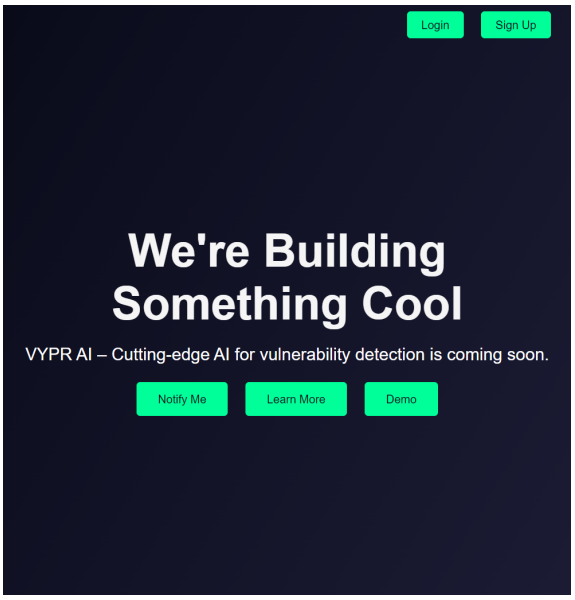

Figure A.1: Home page (logged out)

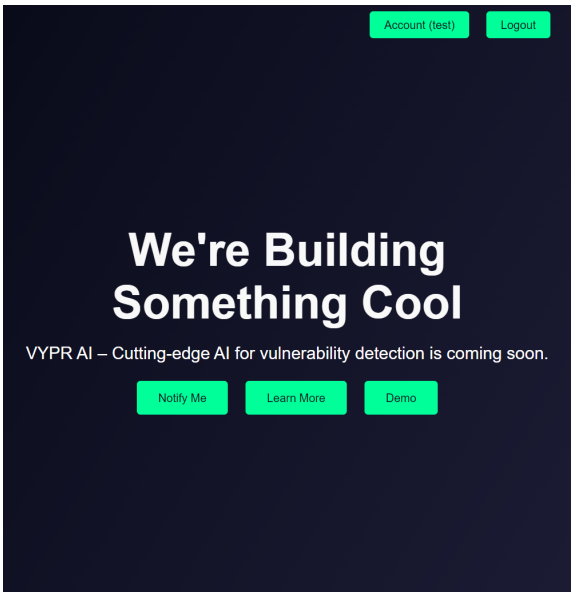

Figure A.2: Home page (logged in)

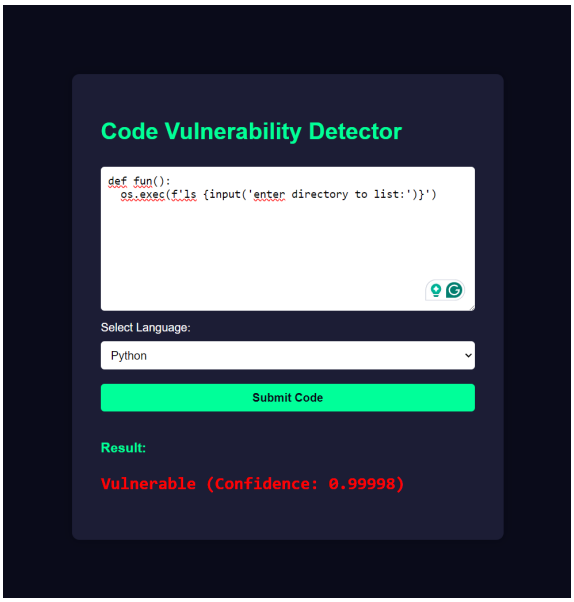

Figure A.3: Demo page

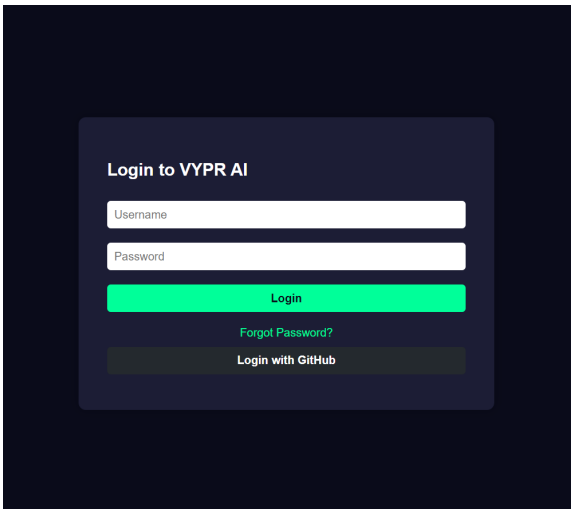

Figure A.4: Login page

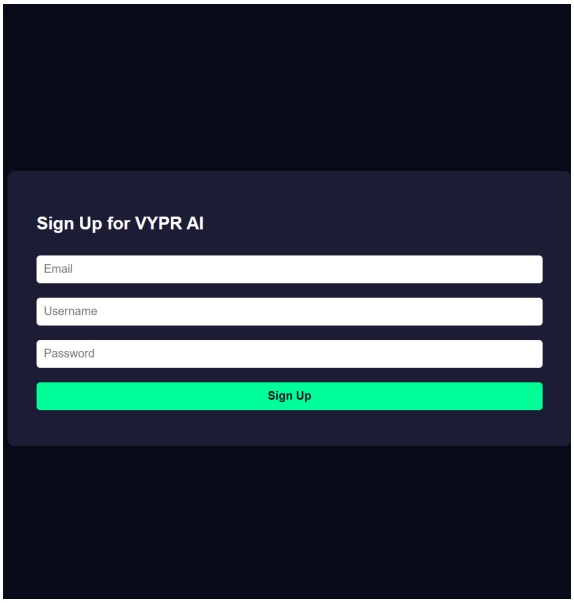

Figure A.5: Signup page

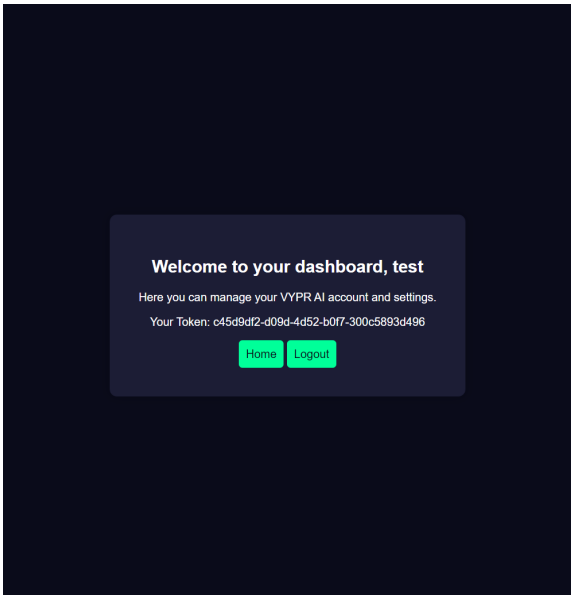

Figure A.6: User page

Printed by Tryckeriet i E-huset, Lund 2024

LUND
UNIVERSITY

Series of Master´s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2024-1034
http://www.eit.lth.se