Guessing As A Service: Large Language Models Are Not Yet Ready For Vulnerability Detection

Francesco Panebianco*, Andrea Isgrò, Stefano Longari, Stefano Zanero and Michele Carminati

Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy

Abstract

The growing number of reported software vulnerabilities underscores the need for efficient detection methods, especially for resource-limited organizations. While traditional techniques like fuzzing and symbolic execution are effective, they require significant manual effort. Recent advances in Large Language Models (LLMs) show promise for zero-shot learning, leveraging pre-training on diverse datasets to detect vulnerabilities without fine-tuning. This study evaluates quantized models (e.g., Mistral v0.3), code-specialized models (e.g., CodeQwen 1.5), and fine-tuned approaches like PDBERT. Zero-shot models perform poorly, with a precision below 0.46, and even PDBERT's high metrics (precision 0.91, specificity 0.99) are undermined by overfitting. These findings emphasize the limitations of current AI solutions and the necessity for approaches tailored to the specific problem.

Keywords

Large Language Models, Software Security, Vulnerability Detection, Artificial Intelligence

1. Introduction

Software security is an ever-evolving field of information security. Thousands of new CVEs (Common Vulnerabilities and Exposures) are recorded every month by the National Institute of Standards and Technology [1]. Traditional methods of vulnerability discovery involve time-consuming manual analysis, often supported by reverse-engineering tools and debuggers [2]. Over the years, methodologies such as fuzzing, symbolic execution, and concolic execution have been extensively employed to identify scenarios of interest, including program crashes [3]. Each of these approaches presents distinct advantages and limitations. For instance, fuzzers effectively detect superficial memory corruption issues but tend to perform poorly in identifying rare conditions or vulnerabilities that do not lead to critical failures. Conversely, symbolic and concolic execution techniques are better suited for systematic exploration but face challenges when dealing with highly branching execution paths and often necessitate carefully tailored testing campaigns. As such, current solutions require manual intervention. Security specialists undertake the effort necessary for effective and reliable vulnerability discovery. While large enterprises can allocate resources to employ competent security experts, small and medium-sized businesses often require cost-effective, automated solutions. This study examines the feasibility of leveraging the zero-shot capabilities of both commercial and open-source models for vulnerability detection across various common code weaknesses.

Large Language Models (LLMs) have recently proven generalization capabilities beyond their intended task [4]. Such capabilities are granted by the diversity of pre-training data, which includes text from different sources (e.g., encyclopedias, mail, code) [5]. The ingestion of large and diverse quantities of data has also produced emerging abilities that resemble logical reasoning, particularly when enhanced with techniques such as Chain-Of-Thought (CoT) [4]. Recent works have attempted to use these models for vulnerability detection and repair [6, 7, 8], achieving mixed results. While many works have

Joint National Conference on Cybersecurity (ITASEC & SERICS 2025), February 03-8, 2025, Bologna, IT
*Corresponding author.
francesco.panebianco@polimi.it (F. Panebianco); andrea.isgro@mail.polimi.it (A. Isgrò); stefano.longari@polimi.it (S. Longari); stefano.zanero@polimi.it (S. Zanero); michele.carminati@polimi.it (M. Carminati)
https://frank01001.com (F. Panebianco)
0009-0007-1510-2594 (F. Panebianco); 0009-0007-9125-6958 (A. Isgrò); 0000-0002-7533-4510 (S. Longari); 0000-0003-4710-5283 (S. Zanero); 0000-0001-8284-6074 (M. Carminati)

attempted to fine-tune language models and Natural Language Processing (NLP) classifiers for this task [9, 10], it was shown that the encouraging results of many of these works may have been the result of overfitting [11]. Overfitting is a phenomenon that occurs when a trained AI model captures patterns in the training sample that are not representative of the overall distribution of the population. As a result, the model performs poorly on unseen data.

An alternative strategy to fine-tuning is zero-shot learning [12]. It employs an LLM's generalization capabilities to perform a task not included in the training objective. This study identifies the most commonly used datasets and LLM architectures in recent publications focused on vulnerability detection. Furthermore, it evaluates low-cost commercial solutions and quantized open-source models to determine their suitability for the requirements of small and medium-sized businesses. The evaluation also includes a recently published methodology named PDBERT [13], as representative of the performance of fine-tuning approaches.

Our analysis shows that quantized general-purpose models, such as Mistral v0.3, fail to reliably recognize common weaknesses in code, achieving a precision of 0.46. Similarly, code-specialized models like CodeQwen 1.5 exhibit even lower precision, reaching only 0.38. The same can be said for low-cost commercial solutions like GPT-4o mini, which reaches a precision of 0.30. PDBERT, on the other hand, has precision and specificity values of 0.91 and 0.99 respectively. Although the observed metrics may typically indicate a reliable classifier, our analysis reveals that this performance stems from overfitting. Specifically, we curated a set of vulnerable examples and their corresponding fixes. Regrettably, PDBERT failed to recognize any of the vulnerable instances.

We summarize our contributions as follows:

- We provide an overview of the current architectural and dataset choices in the literature on LLM-based vulnerability detection
- We evaluate the capabilities of low-budget LLM models on the task, highlighting performance differences across common weaknesses.
- Using a curated set of vulnerable functions, we demonstrate that a notable fine-tuned approach exhibits overfitting to the training dataset.

2. Background and Motivation

This section provides an overview of key concepts and techniques pertinent to software security and large language models (LLMs). Additionally, we review relevant literature in the field and present the motivation behind our research.

2.1. Vulnerabilities in Software

Vulnerabilities in software applications are a threat to the security of infrastructures, organizations, and individuals. These critical bugs result from errors committed by developers within the codebase. If these bugs don't trigger critical crashes or evident inconsistencies, it can be challenging for developers and code reviewers to detect their presence. A software vulnerability is an implementation error that allows users to perform malicious actions beyond the intended software specifications. Such vulnerabilities include logical errors, which cause inconsistent system states; authentication and authorization flaws, which enable unauthorized users to execute restricted actions; and memory corruption in binary software, potentially leading to Remote Code Execution (RCE). RCE represents the most severe form of attack, as it grants an attacker the ability to execute arbitrary operations on the target system.

Fuzzing and symbolic (or concolic) execution are among the most prominent techniques for automating vulnerability detection [3]. These methodologies have demonstrated considerable success in identifying numerous vulnerabilities over the years. However, they exhibit notable limitations. First, their setup involves substantial manual effort and domain-specific expertise, rendering them inaccessible to individuals lacking specialized knowledge. Second, these techniques often fail to uncover vulnerabilities that are only triggered in deep code paths.

Categorizing Vulnerabilities. Common Vulnerabilities and Exposures (CVEs) [1] represent standardized identifiers for publicly known cybersecurity vulnerabilities, facilitating effective communication and remediation across diverse information security systems. Each CVE entry provides a unique identifier and concise description of a specific software or hardware vulnerability, enabling organizations to prioritize and address risks systematically. The record for a CVE also includes a quantitative measure of its severity. Instead, Common Weakness Enumerations (CWEs) [14] serve as a taxonomy of software and hardware weaknesses that underlie vulnerabilities, providing a systematic framework for identifying, categorizing, and mitigating the root causes of security flaws. Unlike CVEs, which address specific instances of vulnerabilities, CWEs focus on generic patterns of error in design, implementation, or configuration. These patterns can be captured by machine learning models to perform detection.

2.2. Large Language Models and Training Strategies

Large Language Models (LLMs) are a recent breakthrough in Natural Language Processing (NLP), powered by advances in deep learning architectures and the availability of large-scale datasets. This advance was introduced by the Transformer architecture [15], which replaced recurrent and convolutional models with self-attention mechanisms, enabling unprecedented scalability and contextual understanding. This innovation paved the way for state-of-the-art models like GPT (Generative Pre-trained Transformer) [16], culminating in the ChatGPT [17] revolution, which showcased the practicality and societal impact of generative AI. LLMs demonstrate remarkable generalization capabilities, driven by their training on vast, diverse datasets [5].

Zero-Shot Learning. LLMs are already employed by software engineers for tasks such as code generation and code analysis, playing an integral role in the software development process [18]. Effective handling of code-related tasks requires both flexibility and robust generalization capabilities, as code syntax merely serves as a medium to represent the underlying algorithm. The algorithm itself is rooted in logical and formal reasoning. Algorithm design can therefore be framed as solving novel, often unseen, tasks that are defined by given formal instructions.

Zero-shot learning is a machine learning approach where an LLM is applied to perform tasks outside of its explicit training objectives [19]. In contrast to conventional supervised learning, zero-shot learning does not involve updating model weights through dataset-specific fitting. Instead, task adaptation occurs through prompting. Instructions are provided to guide the model toward generating the desired output format or response. *Zero-shot classification* is a specific instance of a zero-shot learning task, in which the LLM is used as a classifier model. The model is instructed to output the classification label as a result of some "reasoning" performed on the query input. A popular use-case of zero-shot classification is LLM-as-a-judge [20], which provides a judgment on the given input.

Commercial LLMs like GPT-4 and GPT-4 Turbo models have shown impressive capabilities on code-related tasks [21]. Still, the public release of models like LLAMA [22] has allowed the open-source community to fine-tune specialized models on different tasks, including code generation. These include StarCoder2 [23] and CodeQwen [24]. Alongside these specialized models, general-purpose LLMs like LLama 3.1 have improved performance on code tasks over previous iterations and similar open-source alternatives [22].

2.3. Related Work

Wu and Zhang et al. [6] analyze seven use cases for ChatGPT (3.5 and 4) in the field of Software Security, including vulnerability detection. Code affected by real-world CVEs is tested against the two versions of the model. They discuss both successful scenarios and failure conditions, for which they provide likely causes. They find that the newer iteration of ChatGPT (based on GPT 4) is much more likely to identify the vulnerabilities. Zhou et al. [7] survey the current landscape of LLMs applied to vulnerability detection and repair. The survey highlights the dominance of encoder-only LLMs for vulnerability detection. It explores key approaches like fine-tuning, zero-shot, and few-shot prompting, along with techniques that combine program analysis to improve model performance. The paper also identifies

critical limitations, such as the lack of high-quality datasets, challenges with complex vulnerabilities, and the narrow focus on function-level detection. Steenhoek et al. [8] present a broad review of state-of-the-art LLMs applied to the task of vulnerability detection in software. The research provides an assessment of the detection performance, examining the identification of the type, location, and cause of vulnerabilities. The study evaluates a range of prompting techniques, including zero-shot, n-shot in-context, and advanced methods incorporating contrastive pairs and chain-of-thought reasoning from static analysis and CVE descriptions. Zhang et al. [25] present an extensive empirical study investigating the efficacy of pre-trained model-based automated software vulnerability repair techniques, focusing on C/C++ code. The authors evaluate the performance of several pre-trained models against common vulnerability datasets.

2.4. Motivation

This study aims to assess the current capabilities of AI-based technologies for automatic vulnerability detection. To achieve this, we begin by **reviewing the state-of-the-art in terms of datasets and models** commonly used in the field. This review serves as a foundational reference for researchers approaching this topic. Additionally, we **evaluate the performance of publicly available LLM solutions on the task of vulnerability detection**, providing insights into their effectiveness for this application. In relation to existing research, this paper seeks to identify current gaps in the use of AI for automatic vulnerability detection. Furthermore, it offers an empirical evaluation of the potential of this technology, with a particular focus on its applicability and benefits for small and medium-sized enterprises (SMEs). These organizations may be required to maintain in-house, often legacy, software that they cannot afford to have assessed by security professionals. Through this examination, we aim to contribute to a better understanding of both the challenges and opportunities that AI-driven vulnerability detection presents for businesses of this scale.

3. Review of Vulnerability Detection Datasets and Models

Given the challenges identified in AI-based vulnerability detection, understanding the datasets and models commonly used in this field is crucial for evaluating current approaches. To this end, we examined datasets and model choices for a collection of recent works from both publications and preprints. Appendix A covers the details of considered works.

Datasets. Figure 1 shows the occurrences of a certain evaluation dataset in the considered set. The vast majority of works rely on existing datasets for their evaluation. The most popular datasets are *Big-Vul* [26] and *Reveal* [27]. Big-Vul is a large C/C++ code vulnerability dataset collected from open-source Github projects. Reveal is instead collected from Chrome and Linux Kernel issue trackers. Devign [28] and CVEfixes [29] follow among the most popular options. It can be observed that these datasets are predominantly assembled by scraping commits and issue trackers from GitHub repositories of open-source software. While some works opt for custom self-assembled datasets, samples in their collection are gathered from similar sources. While this approach is practical given the scarcity of curated samples for vulnerability detection tasks, it introduces potential experimental biases that must be addressed. Specifically, there is a significant likelihood that the code from these open-source projects may already be included in the training or pre-training datasets of the LLMs used. If an overlap exists, it undermines the reliability of the evaluation metrics.

LLM Models. In recent literature, a diverse set of large language models (LLMs) has been deployed for software vulnerability detection, with varying levels of adoption and effectiveness. Among these, GPT-4 emerges as the most widely employed (see Figure 1), being a central focus in numerous studies [8, 30, 31, 32, 33, 34, 35, 36, 37, 38]. Its extensive use can be attributed to its large context length (up to 128k tokens) and strong performance. GPT-3.5 shows significant prevalence [8, 39, 38, 40, 41, 37, 42, 43]. This sustained popularity is partly due to its longer availability, which has led to its incorporation into many experiments and datasets, even though it offers lower performance compared to newer models

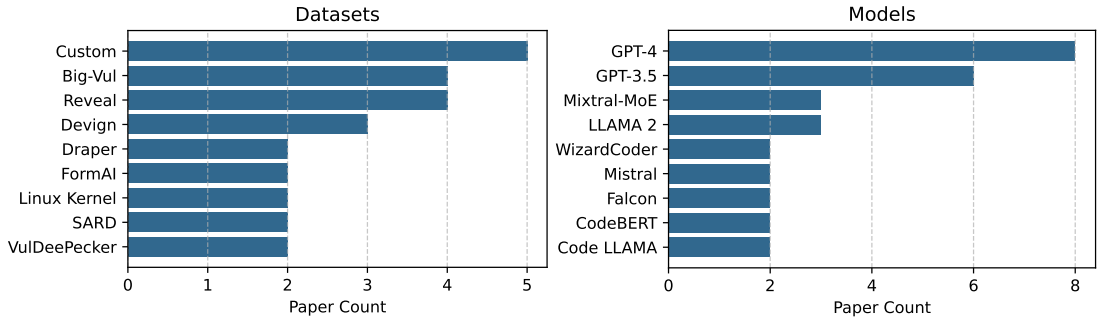

Figure 1: Occurrences of the usage of datasets and models in considered works, excluding unique instances. For models, no distinction is made on model size.

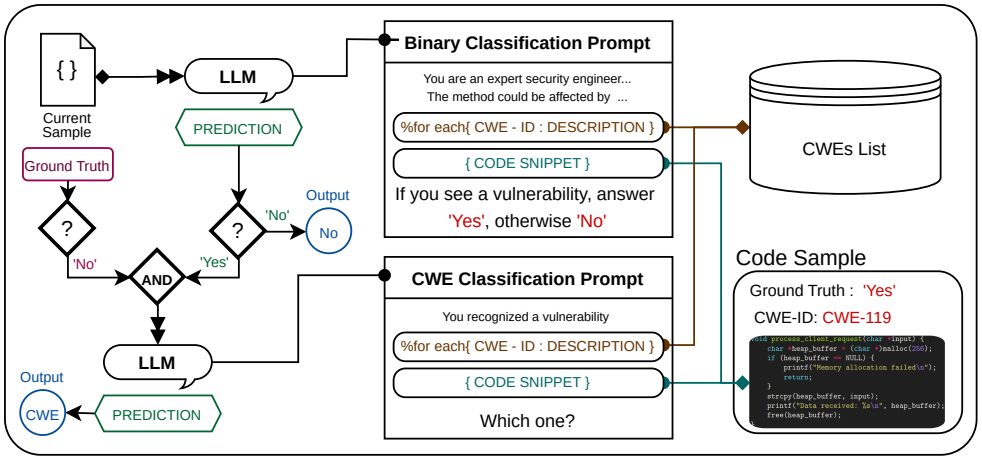

Figure 2: Diagram of the assessment pipeline.

like GPT-4. Other models are also seeing increasing but more selective adoption. For instance, LLAMA, Mixtral, Mistral, and its variations are used [8, 40, 32, 42]. These models are chosen for their smaller parameter sizes and open-source access. Similarly, some works adopt CodeGen, StarCoder, Falcon, BERT and their variations (CodeBERT, VulBERTa) [44, 45, 46, 47, 25, 13, 10, 48, 11]. These models are task-specific, thus limiting their general application. Figure 1 shows the distribution of the most popular models used in considered works. In short, GPT-4 and GPT-3.5 dominate the field due to their high performance and historical availability. Other models like LLAMA, Mistral, StarCoder, and Falcon are instead adopted as open-source.

4. Assessment Methodology

We evaluate the performance of LLMs on the task of vulnerability detection in binary software (C/C++), with a focus on solutions accessible to small and medium enterprises. We select models that are either open-source and quantized for deployment on low-end hardware or available via affordable API queries. The evaluation uses code snippets from one of the most widely used vulnerability datasets, which are input to the LLMs with a specifically crafted prompt. The prompt outlines a vulnerability detection task, presenting a list of Common Weakness Enumerations (CWEs) that could apply to the code and asks the model to determine if any of these CWEs are present. If a non-vulnerable sample is erroneously predicted as vulnerable, the model is also asked to identify the CWE, providing insights into potential biases toward specific weaknesses. Figure 2 visually shows our assessment pipeline. The paper addresses three research questions:

RQ1: How effective are zero-shot solutions for vulnerability detection? To answer this, we assess several

Table 1
Table of CWEs present in the Big-Vul test split of this experimental evaluation. Along with the CWE-ID, a description and the number of vulnerable samples is provided.

CWE-ID	Description	Sample Count
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	1814
CWE-20	Improper Input Validation	1007
CWE-399	Resource Management Errors	670
CWE-125	Out-of-bounds Read	482
CWE-264	Permissions, Privileges, and Access Controls	475
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	443
CWE-416	Use After Free	306
CWE-189	Numeric Errors	299
CWE-190	Integer Overflow or Wraparound	280
CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization	264
CWE-476	NULL Pointer Dereference	195
CWE-787	Out-of-bounds Write	169
CWE-284	Improper Access Control	167
CWE-254	7PK - Security Features	113

popular models, including general-purpose, code-specific, open-source, and commercial solutions.

RQ2: Do current fine-tuned models suffer from overfitting, as previously observed by Risse and Böhme [11]? We evaluate whether more recent and promising fine-tuned models exhibit similar overfitting patterns.

RQ3: Is there a tendency for models to over-recognize specific CWEs? This question examines whether certain CWEs are detected more frequently or with higher accuracy, indicating a potential bias in the model's ability to generalize across different vulnerabilities.

5. Experimental Evaluation

We evaluate the suitability of low-budget LLMs for vulnerability detection for use in small and medium-sized enterprises. Candidate models are the most recent open-source models, as well as corresponding commercial alternatives. We quantize open-source models to 4-bit integers using the BitsAndBytes Python library [49]. We pick two open-source general-purpose models, LLAMA 3.1 8B [22], Mistral v0.3 7B [50]. To represent code generation models, we choose CodeQwen1.5 7B [24]. As for the commercial alternative, the representative is GPT-4o mini [51], the current cheapest model from OpenAI available with API access. While we initially intended to include StarCoder 2 7B [52] as another open-source code generation model, we were not able to obtain a reliable response structure compatible with the classification task. Finally, our evaluation also includes PDBERT [13], one of the most promising fine-tuned models for the vulnerability detection task. PDBERT is described in greater detail in Appendix A. As the evaluation dataset, we choose Big-Vul [26], being one of the most widely-used datasets among considered works. As mentioned in section 3, the open-source nature of this dataset makes the evaluation potentially prone to bias on any tested LLM. We acknowledge this issue and present the results as being produced by a "favorable testing environment". Even in this setting, results on most models are poor and we do not claim their usability in any critical setting. Furthermore, we observe evidence of overfitting in the fine-tuned model, PDBERT. Due to the composition of this dataset, the evaluation is focused on vulnerability detection in C/C++ code. We test all models on the same Big-Vul test split, consisting of 31326 code snippets. 6684 of these samples contain a vulnerability, while the remaining 24642 do not. The dataset associates a CWE ID to each vulnerable sample. Table 1 shows a list of CWE IDs present in the dataset and their associated description. To avoid the requirement of large context windows for used LLMs, we sampled the test split among snippets of no more than 5000 characters. Slight variations of the system prompt were tested at the prompt design stage. These variations did not yield significantly different results.

Table 2
Summary metrics of binary classification for each tested model.

Model	Accuracy	**Precision**	Recall	F1	**Specificity**
LLAMA 3.1 8B	0.46	**0.24**	0.71	0.36	**0.39**
Mistral v0.3 7B	0.78	**0.46**	0.14	0.22	**0.96**
CodeQwen1.5 7B	0.71	**0.38**	0.56	0.45	**0.75**
PDBERT	0.86	**0.91**	0.43	0.58	**0.99**
GPT-4o mini	0.62	**0.30**	0.59	0.40	**0.63**

5.1. Experimental Results

Table 2 shows the summary statistics for the evaluation on all considered models. The results indicate that no model achieves consistently superior performance.

RQ1: How effective are zero-shot solutions for vulnerability detection?

Results indicate that LLAMA 3.1 achieves the highest recall among the evaluated models. However, it also exhibits the lowest precision, with a value of 0.24. Mistral v0.3 shows improved precision, reaching 0.46, but suffers from significantly reduced recall. Although it achieves a high True Negative Rate (specificity), reliably classifying non-vulnerable code as clean, the low recall suggests that this performance arises more from a predisposition to predict "non-vulnerable" rather than from a nuanced understanding of code. CodeQwen 1.5, despite being trained for code generation, performs poorly in vulnerability detection, with precision and recall values of 0.38 and 0.56, respectively, demonstrating unreliability comparable to other models evaluated. GPT-4o mini, the sole commercial model included in this analysis, reaches a mediocre precision of 0.3 and specificity of 0.46. Ultimately, no model demonstrates excellence in zero-shot vulnerability detection.

Answer to RQ1

Zero-shot learning on small general-purpose and code-specialized models achieves poor results on the task of vulnerability detection. The performance of some of these models is comparable to that of a random guessing classifier.

RQ2: Do current fine-tuned models suffer from overfitting?

Among the selected sample of models, PDBERT is the only one that underwent fine-tuning for the detection task. The model yields high precision and specificity. These exceptionally high values raise concerns about potential overfitting. Nonetheless, achieving such performance may suggest that the pre-training objective based on program dependencies effectively enhances the model's ability to fit the training data [13]. We assess whether PDBERT learns meaningful patterns in vulnerable code by testing it on a curated sample of code snippets, both vulnerable and non-vulnerable. Despite including evident vulnerabilities, all samples were labeled as non-vulnerable. Detailed information on the samples is provided in Appendix B. These results suggest that PDBERT's performance on the Big-Vul test split likely results from overfitting to the code style or other irrelevant patterns within the dataset.

Answer to RQ2

PDBERT achieves high precision and specificity; however, further experiments raise concerns about potential overfitting, rendering these metrics unreliable.

RQ3: Is there a tendency for models to over-recognize specific CWEs?

When dealing with different types and distributions of CWE, it is important to consider the reliability of the prediction for each code weakness. We evaluate which CWEs are more likely to be recognized as vulnerable by the model. Additionally, when a false positive occurs, we follow up on the initial

Table 3
Evaluation results. TPR and FNR refer to binary classification. FPR refers to the multi-class task. For PDBERT, FPR is not applicable due to its training objective.

Quantized Open-Source Models					Fine-Tuned and Commercial				
Model	CWE-ID	TPR	FNR	FPR	Model	CWE-ID	TPR	FNR	FPR
LLAMA 3.1 8B	CWE-125	0.72	0.28	0.00	PDBERT	CWE-125	0.45	0.55	N/A
	CWE-200	0.71	0.29	0.00		CWE-200	0.37	0.63	N/A
	CWE-787	0.70	0.30	0.00		CWE-787	0.35	0.65	N/A
	CWE-264	0.76	0.24	0.00		CWE-264	0.38	0.62	N/A
	CWE-416	0.72	0.28	0.06		CWE-416	0.36	0.64	N/A
	CWE-476	0.77	0.23	0.00		CWE-476	0.29	0.71	N/A
	CWE-189	0.67	0.33	0.00		CWE-189	0.31	0.69	N/A
	CWE-190	0.73	0.27	0.08		CWE-190	0.52	0.48	N/A
	CWE-20	0.68	0.32	0.00		CWE-20	0.44	0.56	N/A
	CWE-254	0.67	0.33	0.00		CWE-254	0.39	0.61	N/A
	CWE-119	0.72	0.28	0.44		CWE-119	0.47	0.53	N/A
	CWE-399	0.66	0.34	0.02		CWE-399	0.44	0.56	N/A
	CWE-362	0.67	0.33	0.00		CWE-362	0.47	0.53	N/A
	CWE-284	0.77	0.23	0.00		CWE-284	0.39	0.61	N/A
Mistral v0.3 7B	CWE-125	0.19	0.81	0.00	GPT-4o mini	CWE-125	0.16	0.14	0.02
	CWE-200	0.13	0.87	0.00		CWE-200	0.00	0.41	0.00
	CWE-787	0.25	0.75	0.00		CWE-787	0.02	0.37	0.01
	CWE-264	0.08	0.92	0.00		CWE-264	0.01	0.48	0.00
	CWE-416	0.08	0.92	0.00		CWE-416	0.03	0.57	0.01
	CWE-476	0.18	0.82	0.00		CWE-476	0.41	0.23	0.15
	CWE-189	0.19	0.81	0.00		CWE-189	0.01	0.33	0.00
	CWE-190	0.11	0.89	0.01		CWE-190	0.23	0.22	0.05
	CWE-20	0.14	0.86	0.00		CWE-20	0.09	0.47	0.05
	CWE-254	0.05	0.95	0.00		CWE-254	0.00	0.71	0.00
	CWE-119	0.17	0.83	0.00		CWE-119	0.18	0.35	0.06
	CWE-399	0.08	0.92	0.00		CWE-399	0.01	0.64	0.00
	CWE-362	0.12	0.88	0.00		CWE-362	0.00	0.50	0.00
	CWE-284	0.12	0.88	0.00		CWE-284	0.00	0.41	0.00
CodeQwen1.5 7B	CWE-125	0.64	0.36	0.01					
	CWE-200	0.59	0.41	0.00					
	CWE-787	0.62	0.38	0.01					
	CWE-264	0.57	0.43	0.01					
	CWE-416	0.54	0.46	0.01					
	CWE-476	0.53	0.47	0.04					
	CWE-189	0.62	0.38	0.01					
	CWE-190	0.63	0.37	0.00					
	CWE-20	0.55	0.45	0.02					
	CWE-254	0.50	0.50	0.01					
	CWE-119	0.55	0.45	0.03					
	CWE-399	0.52	0.48	0.03					
	CWE-362	0.52	0.48	0.02					
	CWE-284	0.58	0.42	0.00					

prompt by asking the model which CWE-ID can describe the vulnerability. This was done on all models except PDBERT, on which the evaluation is not applicable. Table 3 shows the detailed results in terms of True Positive Rate (TPR), False Negative Rate (FNR), and False Positive Rate (FPR) for each model and CWE-ID.

LLAMA 3.1. Results show that the model maintains a generally balanced TPR across CWEs. The low overall precision is likely determined by the spike in False Positives caused by CWE-119 (Buffer Overflow). Since this is a popular type of weakness in code, it is likely the model is providing this answer as a result of statistical imbalance. Another less frequent False Positive is CWE-190 (Integer Overflow), covering 8% of False Positives. The model classifies 61% of clean code samples as vulnerable, indicating an overly cautious or alarmist behavior on the task.

Mistral v0.3. Its best performance is observed on CWE-787 (Out-of-bounds write), which is correctly identified in only one out of four instances. The most missed is CWE-254(Seven Pernicious Kingdoms), with a FNR of 0.95. The elevated error rate for this category may stem from its ambiguous definition, as it includes a broad range of more specific weaknesses related to access control, passwords, and cryptography. FPR is not a concern in this case, as the mode tends to over-classify as non-vulnerable.

CodeQwen 1.5. CodeQwen does not excel at recognizing any, nor does it have any bias toward frequent weaknesses.

PDBERT. The TPR and FNR appear balanced across CWEs, which is consistent with expectations given that the model was trained on a subset of the Big-Vul dataset's training split, which includes these common weaknesses. For this model, the FPR metric is not available, as follow-up questions to classify CWEs were outside the training objective.

GPT-4o mini. It shows a notably low detection rate for most common weaknesses, with CWE-476 (NULL Pointer Dereference) being a marginal exception (0.41), though its performance on this CWE is also suboptimal. The CWE also generates a slight tendency to false positives. False negatives are common but particularly frequent for CWE-254 (Seven Pernicious Kingdoms), with an FNR of 0.71. The reason behind this high FNR is likely related to Mistral's. Four CWEs are never recognized, though they are mostly "complex" weaknesses to detect.

Answer to RQ3

Some models show bias toward specific CWEs, but none are consistently easy to classify. CWE-254 (Seven Pernicious Kingdoms) causes false negatives in two models, though this is not seen in others.

6. Conclusions

Automated vulnerability detection remains an open research challenge with the potential to provide cost-effective solutions for small and medium-sized organizations. Current approaches leveraging Large Language Models (LLMs) focus on zero-shot learning or fine-tuning to address this task. We reviewed the most common choices in datasets and models as a reference for researchers approaching the topic. We evaluated zero-shot classification for C/C++ vulnerability detection using popular small-sized LLMs to assess their out-of-the-box utility for code vulnerability analysis. Additionally, we extended our evaluation to include PDBERT, a state-of-the-art fine-tuned model. While PDBERT achieves impressive evaluation metrics on the Big-Vul test split, our secondary tests on curated samples reveal that these results are indicative of overfitting rather than genuine learning. The experimental evaluation highlights the limitations of the current LLM solutions in understanding vulnerable code.

Limitations and Future Work. This study evaluates cost-effective LLM solutions, focusing on small quantized models and low-cost commercial alternatives. This is compatible with the aims of the work, which are to support small- and medium-sized organizations that may lack the resources for more expensive solutions. Overfitting in fine-tuned models was investigated using a limited sample set, which however reveals that the models failed to detect obvious patterns, thereby highlighting the unreliability of the test metrics. To validate these findings, comprehensive evaluations using larger datasets are required. However, a critical limitation is the absence of evaluation datasets that are disjoint from the training data of any tested model. Such separation is essential for accurately assessing downstream performance. However, achieving this remains challenging due to the continuous updates of LLM training datasets sourced from internet scraping. Finally, future research should prioritize exploring alternative model architectures, including Retrieval Augmented Generation (RAG). Current LLMs may lack the necessary feature space to effectively represent and detect vulnerabilities, which are often highly context-dependent and intricately associated with corrupted machine memory states.

Acknowledgments

this work was partially supported by the Google.org Impact Challenge - Tech for Social Good Research Grant (Tides Foundation). It was also partially supported by Project SETA (PNRR M4.C2.1.1 PRIN 2022 PNRR, Cod. P202233M9Z, CUP F53D23009120001, Avviso D.D 1409 14.09.2022) and Project FARE (PNRR M4.C2.1.1 PRIN 2022, Cod. 202225BZJC, CUP D53D23008380006, Avviso D.D 104 02.02.2022). Both projects are under the Italian NRRP MUR program funded by the European Union - NextGenerationEU. Finally, the work was also partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

Declaration on Generative AI

While preparing this work, the authors used GPT-4o, Gemini 2.0 Flash, and GPT-4o-mini for sentence polishing and rephrasing. After using these services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] NIST, National vulnerability database dashboard, 2025. URL: https://nvd.nist.gov/general/nvd-dashboard.

[2] G. Digregorio, R. A. Bertolini, F. Panebianco, M. Polino, Poster: libdebug, build your own debugger for a better (hello) world, in: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 4976–4978. URL: https://doi.org/10.1145/3658644.3691391. doi:10.1145/3658644.3691391.

[3] L. Cui, J. Cui, Z. Hao, L. Li, Z. Ding, Y. Liu, An empirical study of vulnerability discovery methods over the past ten years, Computers & Security 120 (2022) 102817. URL: https://www.sciencedirect.com/science/article/pii/S0167404822002115. doi:https://doi.org/10.1016/j.cose.2022.102817.

[4] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, S. Zhong, B. Yin, X. Hu, Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond, ACM Trans. Knowl. Discov. Data 18 (2024). URL: https://doi.org/10.1145/3649506. doi:10.1145/3649506, place: New York, NY, USA Publisher: Association for Computing Machinery.

[5] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, C. Leahy, The Pile: An 800gb dataset of diverse text for language modeling, arXiv preprint arXiv:2101.00027 (2020).

[6] F. Wu, Q. Zhang, A. P. Bajaj, T. Bao, N. Zhang, R. Wang, C. Xiao, et al., Exploring the limits of chatgpt in software security applications, arXiv preprint arXiv:2312.05275 (2023).

[7] X. Zhou, S. Cao, X. Sun, D. Lo, Large Language Model for Vulnerability Detection and Repair: Literature Review and Roadmap, 2024. URL: http://arxiv.org/abs/2404.02525, arXiv:2404.02525 [cs].

[8] B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, E. T. Barr, W. Le, A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection, 2024. URL: http://arxiv.org/abs/2403.17218. doi:10.48550/arXiv.2403.17218, arXiv:2403.17218 [cs].

[9] H. Hanif, S. Maffeis, VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection, in: 2022 International Joint Conference on Neural Networks (IJCNN), 2022, pp. 1–8. URL: http://arxiv.org/abs/2205.12424. doi:10.1109/IJCNN55064.2022.9892280, arXiv:2205.12424 [cs].

[10] M. A. Ferrag, A. Battah, N. Tihanyi, M. Debbah, T. Lestable, L. C. Cordeiro, SecureFalcon: The Next Cyber Reasoning System for Cyber Security, 2023. URL: http://arxiv.org/abs/2307.06616. doi:10.48550/arXiv.2307.06616, arXiv:2307.06616 [cs].

[11] N. Risse, M. Böhme, Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection, in: USENIX Security Symposium 2024, 2024, p. 19.

[12] F. Pourpanah, M. Abdar, Y. Luo, X. Zhou, R. Wang, C. P. Lim, X.-Z. Wang, Q. J. Wu, A review of generalized zero-shot learning methods, IEEE transactions on pattern analysis and machine intelligence 45 (2022) 4051–4070.

[13] Z. Liu, Z. Tang, J. Zhang, X. Xia, X. Yang, Pre-training by Predicting Program Dependencies for Vulnerability Analysis Tasks, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.

[14] Common Weakness Enumeration, 2024. URL: https://cwe.mitre.org/.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, in: Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS'17, Curran Associates Inc., Red Hook, NY, USA, 2017, p. 6000–6010.

[16] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, Improving language understanding by generative pre-training (2018).

[17] OpenAI, Chatgpt, 2024. URL: https://chatgpt.com, large language model, https://chatgpt.com.

[18] S. Rasnayaka, G. Wang, R. Shariffdeen, G. N. Iyer, An empirical study on usage and perceptions of llms in a software engineering project, in: Proceedings of the 1st International Workshop on Large Language Models for Code, LLM4Code '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 111–118. URL: https://doi.org/10.1145/3643795.3648379. doi:10.1145/3643795.3648379.

[19] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, Y. Iwasawa, Large language models are zero-shot reasoners, in: S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, A. Oh (Eds.), Advances in Neural Information Processing Systems, volume 35, Curran Associates, Inc., 2022, pp. 22199–22213. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/8bb0d291acd4acf06ef112099c16f326-Paper-Conference.pdf.

[20] H. Huang, Y. Qu, J. Liu, M. Yang, T. Zhao, An empirical study of llm-as-a-judge for llm evaluation: Fine-tuned judge models are task-specific classifiers, arXiv preprint arXiv:2403.02839 (2024).

[21] C. Munley, A. Jarmusch, S. Chandrasekaran, Llm4vv: Developing llm-driven testsuite for compiler validation, Future Generation Computer Systems 160 (2024) 1–13. URL: https://www.sciencedirect.com/science/article/pii/S0167739X24002449. doi:https://doi.org/10.1016/j.future.2024.05.034.

[22] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, others, The llama 3 herd of models, arXiv preprint arXiv:2407.21783 (2024).

[23] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, H. de Vries, Starcoder 2 and the stack v2: The next generation, 2024. arXiv:2402.19173.

[24] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, B. Hui, L. Ji, M. Li, J. Lin, R. Lin, D. Liu, G. Liu, C. Lu, K. Lu, J. Ma, R. Men, X. Ren, X. Ren, C. Tan, S. Tan, J. Tu, P. Wang, S. Wang, W. Wang, S. Wu, B. Xu, J. Xu, A. Yang, H. Yang, J. Yang, S. Yang, Y. Yao, B. Yu, H. Yuan, Z. Yuan, J. Zhang, X. Zhang, Y. Zhang, Z. Zhang, C. Zhou, J. Zhou, X. Zhou, T. Zhu, Qwen technical report, arXiv preprint arXiv:2309.16609 (2023).

[25] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, Z. Chen, Pre-Trained Model-Based Automated Software Vulnerability Repair: How Far are We?, IEEE Transactions on Dependable and Secure Computing 21 (2024) 2507–2525. doi:10.1109/TDSC.2023.3308897.

[26] J. Fan, Y. Li, S. Wang, T. 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, Association for Computing Machinery, New York, NY, USA, 2020, p. 508–512. URL: https://doi.org/10.1145/3379597.3387501. doi:10.1145/3379597.3387501.

[27] S. Chakraborty, R. Krishna, Y. Ding, B. Ray, Deep learning based vulnerability detection: Are we

there yet?, IEEE Transactions on Software Engineering 48 (2022) 3280–3296. doi:10.1109/TSE.2021.3087402.

[28] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems, volume 32, Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/49265d2447bc3bbfe9e76306ce40a31f-Paper.pdf.

[29] G. Bhandari, A. Naseer, L. 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, Association for Computing Machinery, New York, NY, USA, 2021, p. 30–39. URL: https://doi.org/10.1145/3475960.3475985. doi:10.1145/3475960.3475985.

[30] N. Tihanyi, T. Bisztray, R. Jain, M. A. Ferrag, L. C. Cordeiro, V. Mavroeidis, The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification, in: Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2023, Association for Computing Machinery, New York, NY, USA, 2023, pp. 33–43. URL: https://doi.org/10.1145/3617555.3617874. doi:10.1145/3617555.3617874, event-place: San Francisco, CA, USA.

[31] G. Lu, X. Ju, X. Chen, W. Pei, Z. Cai, GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning, Journal of Systems and Software 212 (2024) 112031. URL: https://www.sciencedirect.com/science/article/pii/S0164121224000748. doi:https://doi.org/10.1016/j.jss.2024.112031.

[32] Y. Sun, D. Wu, Y. Xue, H. Liu, W. Ma, L. Zhang, M. Shi, Y. Liu, LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning, 2024. URL: https://arxiv.org/abs/2401.16185, _eprint: 2401.16185.

[33] R. Meng, M. Mirchev, M. Böhme, A. Roychoudhury, Large language model guided protocol fuzzing, in: Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS), 2024.

[34] P. Liu, C. Sun, Y. Zheng, X. Feng, C. Qin, Y. Wang, Z. Li, L. Sun, Harnessing the Power of LLM to Support Binary Taint Analysis, 2023. URL: https://arxiv.org/abs/2310.08275, _eprint: 2310.08275.

[35] N. S. Mathews, Y. Brus, Y. Aafer, M. Nagappan, S. McIntosh, LLbezpeky: Leveraging Large Language Models for Vulnerability Detection, 2024. URL: https://arxiv.org/abs/2401.01269, _eprint: 2401.01269.

[36] H. Li, Y. Hao, Y. Zhai, Z. Qian, Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach, Proc. ACM Program. Lang. 8 (2024). URL: https://doi.org/10.1145/3649828. doi:10.1145/3649828, place: New York, NY, USA Publisher: Association for Computing Machinery.

[37] B. Berabi, A. Gronskiy, V. Raychev, G. Sivanrupan, V. Chibotaru, M. Vechev, DeepCode AI Fix: Fixing Security Vulnerabilities with Large Language Models, 2024. URL: http://arxiv.org/abs/2402.13291. doi:10.48550/arXiv.2402.13291, arXiv:2402.13291 [cs].

[38] H. Li, Y. Hao, Y. Zhai, Z. Qian, Assisting Static Analysis with Large Language Models: A ChatGPT Experiment, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, Association for Computing Machinery, New York, NY, USA, 2023, pp. 2107–2111. URL: https://doi.org/10.1145/3611643.3613078. doi:10.1145/3611643.3613078, event-place: San Francisco, CA, USA.

[39] T. K. Le, S. Alimadadi, S. Y. Ko, A Study of Vulnerability Repair in JavaScript Programs with Large Language Models, in: Companion Proceedings of the ACM on Web Conference 2024, WWW '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 666–669. URL: https://doi.org/10.1145/3589335.3651463. doi:10.1145/3589335.3651463, event-place: Singapore, Singapore.

[40] Y. Nong, M. Aldeen, L. Cheng, H. Hu, F. Chen, H. Cai, Chain-of-Thought Prompting of Large Language Models for Discovering and Fixing Software Vulnerabilities, 2024. URL: http://arxiv.org/

abs/2402.17230. doi:10.48550/arXiv.2402.17230, arXiv:2402.17230 [cs].
[41] D. Hidvégi, K. Etemadi, S. Bobadilla, M. Monperrus, CigaR: Cost-efficient Program Repair with LLMs, 2024. URL: http://arxiv.org/abs/2402.06598. doi:10.48550/arXiv.2402.06598, arXiv:2402.06598 [cs].
[42] R. Fang, R. Bindu, A. Gupta, D. Kang, Llm agents can autonomously exploit one-day vulnerabilities, arXiv preprint arXiv:2404.08144 (2024).
[43] I. Bouzenia, P. Devanbu, M. Pradel, RepairAgent: An Autonomous, LLM-Based Agent for Program Repair, 2024. URL: http://arxiv.org/abs/2403.17134. doi:10.48550/arXiv.2403.17134, arXiv:2403.17134 [cs].
[44] N. T. Islam, M. B. Karkevandi, P. Najafirad, Code Security Vulnerability Repair Using Reinforcement Learning with Large Language Models, 2024. URL: http://arxiv.org/abs/2401.07031. doi:10.48550/arXiv.2401.07031, arXiv:2401.07031 [cs].
[45] J. Wang, L. Cao, X. Luo, Z. Zhou, J. Xie, A. Jatowt, Y. Cai, Enhancing Large Language Models for Secure Code Generation: A Dataset-driven Study on Vulnerability Mitigation, 2023. URL: http://arxiv.org/abs/2310.16263. doi:10.48550/arXiv.2310.16263, arXiv:2310.16263 [cs].
[46] A. Shestov, R. Levichev, R. Mussabayev, E. Maslov, A. Cheshkov, P. Zadorozhny, Finetuning Large Language Models for Vulnerability Detection, 2024. URL: http://arxiv.org/abs/2401.17010. doi:10.48550/arXiv.2401.17010, arXiv:2401.17010 [cs].
[47] N. T. Islam, J. Khoury, A. Seong, M. B. Karkevandi, G. D. L. T. Parra, E. Bou-Harb, P. Najafirad, LLM-Powered Code Vulnerability Repair with Reinforcement Learning and Semantic Reward, 2024. URL: http://arxiv.org/abs/2401.03374. doi:10.48550/arXiv.2401.03374, arXiv:2401.03374 [cs].
[48] M. A. Ferrag, M. Ndhlovu, N. Tihanyi, L. C. Cordeiro, M. Debbah, T. Lestable, N. S. Thandi, Revolutionizing Cyber Threat Detection With Large Language Models: A Privacy-Preserving BERT-Based Lightweight Model for IoT/IIoT Devices, IEEE Internet of Things Journal 2022 12 (2024) 23733–23750. doi:10.1109/ACCESS.2024.3363469.
[49] B. Foundation, Bitsandbytes, 2024. URL: https://github.com/bitsandbytes-foundation/bitsandbytes, python Library for Quatizing LLMs.
[50] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, W. E. Sayed, Mistral 7b, 2023. URL: https://arxiv.org/abs/2310.06825. arXiv:2310.06825.
[51] OpenAI, Gpt-4o mini, 2024. URL: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/, large language model.
[52] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, H. de Vries, Starcoder 2 and the stack v2: The next generation, 2024. URL: https://arxiv.org/abs/2402.19173. arXiv:2402.19173.

A. Analyzed Sample of Works

This section details the body of work analyzed to determine the most popular datasets and LLM models employed in current literature. Table 4 associates each work with the models and datasets it uses.

- **Steenhoek et al.** [8]: Review of state-of-the-art LLMs for software vulnerability detection, evaluating various prompting techniques.
- **Tihanyi et al.** [30]: Introduce ESBMC-AI integrating LLMs with BMC for vulnerability detection and repair, leveraging the FormAI dataset.
- **Le et al.** [39]: Evaluate ChatGPT and Bard for JavaScript vulnerability repair using zero-shot prompting across 20 vulnerabilities.
- **Nong et al.** [40]: Propose VSP, using Chain-of-Thought prompting for analyzing and patching C/C++ vulnerabilities.
- **Li et al.** [36]: Introduce LLift, integrating LLMs with static analysis for Use Before Initialization detection in Linux kernel code.
- **Shestov et al.** [46]: Explore fine-tuning LLMs like WizardCoder for Java vulnerability detection using efficient strategies like batch packing.
- **Lu et al.** [31]: Present GRACE, enhancing LLMs with graph structures (AST, PDG, CFG) for improved C/C++ vulnerability detection.
- **Mathews et al.** [35]: Explore GPT-4 with Retrieval Augmented Generation for Android app vulnerabilities using the Ghera benchmark.
- **Fang et al.** [42]: Claims GPT-4 exploits 87% of one-day vulnerabilities with CVE descriptions, outperforming GPT-3.5 and scanners.
- **Sun et al.** [32]: Introduce LLM4Vuln, a framework for evaluating LLMs' vulnerability reasoning, isolating external aids.
- **Liu et al.** [13]: Introduce PDBERT for C/C++ vulnerability detection, leveraging pre-training on 2.28M functions on extracted data and control dependencies.
- **Ferrag et al.** [10]: Present SecureFalcon, a lightweight model for C/C++ vulnerabilities, with FalconVulnDB to enhance training.
- **Risse and Böhme** [11]: Investigate ML4VD limitations like overfitting, proposing VulnPatchPairs for improved C vulnerability evaluation.
- **Hanif and Maffeis** [9]: Introduce VulBERTa, a RoBERTa-based model pre-trained and fine-tuned for C/C++ vulnerability detection.
- **Wu and Zhang** [6]: Examine ChatGPT's strengths and weaknesses in seven software security tasks across different versions.

B. PDBERT Secondary Analysis

In this section, we will go into detail on the curated set of code samples used for the secondary evaluation on PDBERT. These functions are designed to test the ability of the model to actually recognize patterns related to vulnerable functions. All of the tested samples (both vulnerable and non-vulnerable) were predicted to be non-vulnerable by the model. Table 6 provides more details on each of the curated samples.

B.1. Code Samples

We present a selection of code snippets provided to the model to demonstrate the simplicity of the patterns it failed to identify. This inclusion also aims to support reproducibility and enable a deeper analysis of the model's behavior.

Table 4
Association between paper and chosen models and datasets

Paper	Models	Datasets
Steenhoek et al.	GPT-3.5, GPT-4, Gemini 1.0 Pro, Wizard-Coder, Code LLAMA, Mixtral-MoE, Mistral, StarCoder, LLAMA 2, StarChat-β, MagiCoder	SVEN
Tihanyi et al.	GPT-4o	FormAI
Le et al.	GPT-3.5	Custom
Li et al.	GPT-3.5, GPT-4	Linux Kernel
Nong et al.	GPT-3.5, LLAMA 2, Falcon	SARD, Big-Vul
Shestov et al.	CodeGeeX, WizardCoder, CodeGen, ContraBERT	CVEFixes, VCMatch, Custom
Lu et al.	GPT-4	Reveal, Big-Vul, FFmpeg, QEMU
Mathews et al.	GPT-4	Ghera
Fang et al.	GPT-4, GPT-3.5, OpenHermes 2.5 Mistral, LLAMA 2, Mixtral-MoE, Mistral, Nous Hermes 2 Yi, OpenChat 3.5	Custom
Sun et al.	GPT-4, Mixtral-MoE, Code LLAMA	Custom
Liu et al.	CodeBERT	Reveal, Big-Vul, Devign
Ferrag et al.	Falcon	FormAI, SySeVR, Draper, VDISC, Big-Vul, DiverseVul, SARD, Reveal
Risse and Böhme	UniXcoder, CoTexT, GraphCB, CodeBERT, VulBERTa, PLBart	CodeXGLUE, Devign, VulDeePecker
Hanif and Maffeis	RoBERTa	VulDeePecker, Draper, Reveal, muVuldeepecker, Devign, D2A
Wu and Zhang et al.	GPT-3.5, GPT-4	Custom

B.1.1. Sample 1 - Simple Heap Overflow

```
void process_client_request(char *input) {
    char *heap_buffer = (char *)malloc(256);
    if (heap_buffer == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
    strcpy(heap_buffer, input);
    printf("Data received: %s\n", heap_buffer);
    free(heap_buffer);
}
```

B.1.2. Sample 4 - Simple Stack Buffer Overflow

```
void toast(char *input) {
    char buffer[16];
    int offset = 0;

    offset = get_offset();
```

```
    buffer += offset;

    strncpy(buffer, input, 16);
    printf("Data received: %s\n", buffer);
}
```

B.1.3. Sample 6 - Simple Use-After-Free

```
char* copy(char *input) {
    char *heap_buffer = (char *)malloc(256);
    if (heap_buffer == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
    strncpy(heap_buffer, input, 256);
    printf("Data received: %s\n", heap_buffer);
    free(heap_buffer);

    return heap_buffer;
}
```

B.1.4. Sample 8 - Suspicious Code

```
void unsafe_handling(char *input) {
    char buffer[16];
    memcpy(buffer, input, 16);
    printf("Data received: %s\n", buffer);
}
```

B.1.5. Sample 9 - Format String

```
void print_client_info(ClientInfo *client) {
    printf("Client name: %s\n", client->name);
    printf("Client age: %d\n", client->age);
    printf("Client address: %s\n", client->address);

    printf(client->notes);
}
```

B.1.6. Sample 11 - Integer Overflow

```
void execute_transaction(Account *account, int num_items, int price_per_item) {
    if (num_items < 0 || price_per_item < 0) {
        printf("Invalid input\n");
        return;
    }
```

```
    int total = num_items * price_per_item;
    printf("Total price: %d\n", total);
    account->balance -= total;
}
```

Table 6
Details of tested code pieces in PDBERT's secondary evaluation. the count of lines of code excludes empty lines.

Index	Ref	CWE-ID	Description
1	Simple Heap Overflow	CWE-119	This is a very recognizable heap overflow (10 lines of code).
2	WebP Heap Buffer Overflow (CVE-2023-4863)	Not Vulnerable	This is the committed fix for the infamous libwebp Heap Overflow in Huffman Table handling. It was supposed to evaluate whether the model recognizes the code from the CVE rather than identifying actual vulnerabilities, as it should classify the fixed code as non-vulnerable if functioning correctly.
3	SECCON 2024 /pwn challenge - free-free-free	Mixed	This is a recent CTF challenge containing multiple weaknesses that make it vulnerable.
4	Simple Stack Buffer Overflow	CWE-119	This is a very recognizable stack buffer overflow (8 lines of code).
5	Fixed Simple Stack Buffer Overflow	Not Vulnerable	This is a fixed version of sample 4.
6	Simple Use-After-Free Vulnerability	CWE-416	This is a very recognizable use-after-free (11 lines of code).
7	Fixed Simple Use-After-Free Vulnerability	Not Vulnerable	This is a fixed version of sample 6.
8	Suspicious Code	Not Vulnerable	This is a non-vulnerable code snippet that uses names and keywords that may trick the LLM into thinking the function is vulnerable (e.g., "unsafe"). The purpose of this sample is to verify whether or not the LLM associates function and variable names to vulnerabilities instead of actual code weaknesses.
9	Simple Format String Vulnerability	CWE-134	This is a very recognizable format string vulnerability (6 lines of code).
10	Fixed Simple Format String Vulnerability	Not Vulnerable	This is a fixed version of sample 9.
11	Simple Integer Overflow Vulnerability	CWE-190	This is a very recognizable integer overflow vulnerability (9 lines of code).
12	Fixed Simple Integer Overflow Vulnerability	Not Vulnerable	This is a fixed version of sample 11.