Outside the Comfort Zone: Analysing LLM Capabilities in Software Vulnerability Detection

Yuejun Guo⋆[1], Constantinos Patsakis[2,3], Qiang Hu[4], Qiang Tang[1], and Fran Casino[2,5,6]

[1] Luxembourg Institute of Science and Technology, Luxembourg
yuejun.guo@list.lu,qiang.tang@list.lu
[2] Information Management Systems Institute, Athena Research Centre (ARC), Artemidos 6, Marousi, Greece
[3] Department of Informatics, University of Piraeus, 80 Karaoli & Dimitriou str., 18534 Piraeus, Greece
kpatsak@unipi.gr
[4] Department of Computer Science, The University of Tokyo, Japan
qianghu0515@gmail.com
[5] Department of Computer Engineering and Mathematics, Rovira i Virgili University, Tarragona, Spain
franciscojose.casino@urv.cat

Abstract. The significant increase in software production driven by automation and faster development lifecycles has resulted in a corresponding surge in software vulnerabilities. In parallel, the evolving landscape of software vulnerability detection, highlighting the shift from traditional methods to machine learning and large language models (LLMs), provides massive opportunities at the cost of resource-demanding computations. This paper thoroughly analyses LLMs' capabilities in detecting vulnerabilities within source code by testing models beyond their usual applications to study their potential in cybersecurity tasks. We evaluate the performance of six open-source models that are specifically trained for vulnerability detection against six general-purpose LLMs, three of which were further fine-tuned on a dataset that we compiled. Our dataset, alongside five state-of-the-art benchmark datasets, were used to create a pipeline to leverage a binary classification task, namely classifying code into vulnerable and non-vulnerable. The findings highlight significant variations in classification accuracy across benchmarks, revealing the critical influence of fine-tuning in enhancing the detection capabilities of small LLMs over their larger counterparts, yet only in the specific scenarios in which they were trained. Further experiments and analysis also underscore the issues with current benchmark datasets, particularly around mislabeling and their impact on model training and performance, which raises concerns about the current state of practice. We also discuss the road ahead in the field suggesting strategies for improved model training and dataset curation.

⋆ Corresponding author.

Keywords: Software vulnerability detection · Source code analysis · Large language models · Cybersecurity

1 Introduction

The quest for automation and faster production lifecycles has paved the way for more software solutions. As a result, we have witnessed a massive growth in software over the past few decades, which is mapped to a plethora of digital products and services. Nevertheless, as with all human constructs, software has defects, but in this case, defects are not material. Many of these errors can be identified through the use of the software, as, for instance, the expected functionality is not provided. However, some functionality issues might not be revealed until the software is executed in a specific environment or with parameters that the developer did not expect to handle, either because they are not handled properly, or they are malformed. Of particular interest are security defects, which can expose users to many risks and lead to many hazards since software may handle a lot of sensitive and private information while also being used in critical infrastructures.

Moreover, there has been a continuous increase in the number of reported software vulnerabilities. As illustrated in Fig. 1, the number of vulnerabilities has quadrupled in the last decade. We argue that this can be attributed to the parallel introduction of the General Data Protection Regulation (GDPR) and the issuance of the Presidential Policy Directive 41 (PPD-41) which pushed private and public organisations to report cyber security incidents. Notably, we have a doubling of reported vulnerabilities in 2017, just the next year of their introduction.

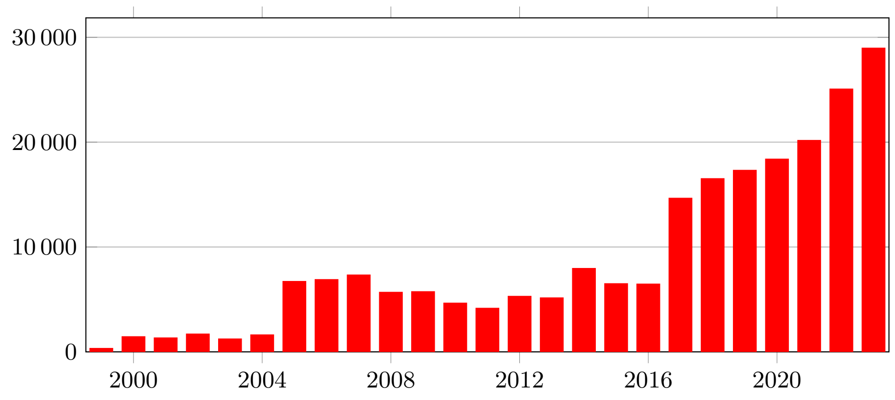

Fig. 1: Number of CVEs by year. Source: `https://www.cve.org/About/Metrics`

In parallel, the exponential increase in data generation, leading to the creation of vast datasets, has been crucial for the increasing capabilities and complexity of artificial intelligence (AI) and machine learning (ML) in real-world applications. Advances in computational technology (e.g., Graphical Processing Units or GPUs and Tensor Processing Units or TPUs) have enabled unforeseen processing capabilities, enabling the use of resource-demanding models. The introduction of large language models (LLMs) has revolutionised how machines understand, interpret, and generate human language and their democratisation by vast communities behind their use and adoption, such as Hugging Face [19], has raised the attention not just of the research community but society as a whole. As more advanced human-machine interactions arise, LLMs are acquiring broader capabilities and application scenarios due to their ability to improve and adapt to new data and contexts.

As one would expect, researchers and the industry quickly stepped in to harness the new capabilities of ML and LLMs to timely and accurately identify vulnerabilities. This is a major shift as previously one would resort to traditional monolithic solutions like regular expressions [40] to identify vulnerable code. The inherent flexibility in coding styles allows developers to express themselves differently, resulting in low accuracy and precision for traditional detection tools, as patterns can be easily bypassed or falsely triggered by non-vulnerable code. The major development in this new era is that models specifically trained in vulnerable and secure code can timely and more accurately identify vulnerable code before it reaches production. This is even more relevant for LLMs as they are able to generate, understand, and summarise code quite efficiently.

In this paper, we perform a thorough analysis of LLMs' capabilities in detecting source code vulnerabilities. Nevertheless, we try to push LLMs beyond their comfort zone and understand the possible gaps and pitfalls in this process. We conduct a series of comprehensive experiments with different benchmark datasets and observe how they perform. This broad variation leads us to propose the use of fine-tuning strategies to leverage low resource-demanding LLMs in the task of software vulnerability detection. Our strategies illustrate that while fine-tuning can enable these LLMs to outperform larger ones in particular contexts, it may lead to a loss of generalisation ability. Indeed, we observe a lot of variation in their capacities when changing the underlying dataset. Finally, we assess the benchmark datasets using state-of-the-art tools that are used in the industry. The latter enables us to provide some fruitful discussion and a strategy to improve the training and accuracy of LLMs in the future. Our approach relies on several research questions pertinent to code vulnerability detection, as described in Table 1.

The remainder of this work is structured as follows. In the next section, we provide an overview of the related work regarding code vulnerability detection. Then, in Section 3, we introduce the reader to our methodology and code vulnerability analysis pipeline. Section 4 introduces the datasets used and the experimental setup. Next, in Section 5 we report the outcomes and provide a de-

Table 1: Summary of research questions and the corresponding sections devoted to answering them.

Research Question	Objective	Discussion
RQ1. Which methods are currently used for software vulnerability detection?	The objective is to summarise the current state of the art and approaches towards identifying vulnerabilities in source code.	Section 2
RQ2 Can base LLMs detect vulnerabilities in source code?	The objective is to evaluate the capabilities of general-purpose LLMs and their performance towards software vulnerability analysis.	Sections 2, 4, 5
RQ3 Is fine-tuning an enabling strategy to improve the trade-off between computational resources and detection accuracy?	The objective is to provide insight towards the use of fine-tuning to improve the performance of base LLMs to a level in which they outperform larger models.	Section 5
RQ4 How robust are the analysed vulnerability detection models?	The objective is to assess whether the models can generalise across different benchmark datasets.	Sections 4, 5
RQ5 Are curation and labelling methodologies employed on existing datasets robust enough for training LLMs and ensuring their desired functionality?	We try to assess existing datasets using industry tools to determine how well they are labelled and whether they provide the necessary information for proper training.	Sections 4, 5
RQ6 Given the analysis and outcomes provided in this paper, what are the next steps towards software vulnerability detection?	The objective is to analyse the lessons learned in this paper and provide a view regarding desired functionalities and capabilities of generative AI towards source code analysis.	Sections 5, 6

tailed discussion. Finally, the paper concludes in Section 6, recalling our research findings and proposing ideas for future work.

2 Related Work

We review related work from the perspective of three areas: static application security testing (SAST), task-specific deep learning (DL) models, and large language models (LLMs) for vulnerability detection.

2.1 SAST-Based Vulnerability Detection

SAST tools typically utilize rule-based [23] and signature-based [41] methods to scan the source code for vulnerabilities, which require predefined rules or patterns indicative of known vulnerabilities. The scanning technique varies between different tools, and the popular ones are pattern matching [49], symbolic execution [34], and data-flow analysis [38]. Croft [9] conducted an empirical study involving three SAST tools (Flawfinder [50], Cppcheck [8], and RATs [2]) selected from the tool lists provided by NIST [30] and OWASP [33], demonstrating that ML-based approaches provide better overall performance for detection and assessment. Other widely used SAST tools are Semgrep [39], SNYK [42] and Sonarqube [43]. Although effective in certain contexts, these tools are usually time-consuming to develop considering the required domain knowledge on security weaknesses and may need to be adjusted to identify novel or previously unknown vulnerabilities.

2.2 Task-Specific DL Models for Vulnerability Detection

Compared to traditional static analysers that often require manual feature engineering by security experts, AI automates the analysis and can function at different granularities (e.g., file, function, and program slice). Various AI models, especially DL models, have been developed and put in use. Typically, a DL model is initialized with random parameters and then trained on a set of labelled data containing both vulnerable and non-vulnerable code samples. This type of model, also known as task-specific model, is specifically designed to detect vulnerabilities within codebases. VulDeePecker [25] proposed by Li *et al.* is a very early work that adopted DL techniques to automatically identify vulnerabilities in source code. Specifically, VulDeePecker utilizes BLSTM networks to learn vulnerable information and outperforms pattern-based and code similarity-based methods. Later, Zhou *et al.* proposed Devign [56], a Graph Neural Networks (GNNs) based method to detect vulnerabilities. The key component of Devign involves leveraging GNNs to learn from code with semantic representations, such as Abstract Syntax Tree (AST). The dataset provided by Devign has been widely studied in the vulnerability detection field. More recently, Chen *et al.* built a new dataset [6] that contains 18,945 vulnerable functions for the performance evaluation of vulnerability detection models. The experimental results demonstrated that existing vulnerability detection models perform poorly in their dataset. The well-known empirical study [3] showed that existing methods cannot generalize to real-world vulnerability prediction and can be improved by using a proper pipeline combining the best practices in each process, such as data collection and model design. Besides, some surveys [41,27] comprehensively reviewed the literature that lies in the direction of DL-based vulnerability detection.

2.3 LLM-Based Vulnerability Detection

The advent of LLMs is changing the vulnerability detection paradigm. Rather than being explicitly trained on labelled vulnerable and non-vulnerable source code, LLMs are pre-trained on vast amounts of data from various sources, such as online blogs, books, and code repositories. During pre-training, these models learn to capture statistical patterns and semantic meanings in the data. When fine-tuned on vulnerable and non-vulnerable source code, these models leverage their pre-trained knowledge to identify vulnerability patterns and often outperform task-specific models. Based on the RoBERTa [29] architecture tailored for text-based tasks, Microsoft developed CodeBERT [13] and Hanif *et al.* [17] introduced VulBERTa specifically for source code analysis. Both models have been widely used and fine-tuned using different datasets for vulnerability detection [15,47,44,45,16]. Ribeiro *et al.* [35] explored the application of GPT-3 for automatically fixing type errors in OCaml code, with a particular focus on addressing general bugs and providing some discussion regarding vulnerabilities. Noever *et al.* [31] tried to utilize GPT-4 [32] to find and fix vulnerabilities and found that GPT-4 can correct programs and reduce 90% vulnerabilities. Li *et al.* [24] focused on using GPT-3.5 and GPT-4 to identify vulnerabilities by providing code and context. Both models demonstrated the ability to identify vulnerable code and provide detailed information on the detected vulnerabilities. Charalambous *et al.* [5] combined LLMs and formal verification techniques to identify vulnerabilities. The proposed method can detect and repair 80% of vulnerable code. Zheng *et al.* [54] reviewed the use of LLMs for software engineering tasks, including vulnerability detection, and found that LLMs do not perform well on this task according to the results reported in existing work. More recently, Zhou *et al.* [55] reported their emerging results of LLMs for vulnerability detection and showed that GPT-3.5 has competitive performance with fine-tuned CodeBERT [13] and GPT-4 significantly outperforms fine-tuned CodeBERT in this task. Lastly, Zhou *et al.* [55] surveyed 36 works related to LLMs for vulnerability detection and repair and discussed the challenges and research directions in this task.

Contrary to previous work, our study further investigates the impact of fine-tuning on detection performance. Additionally, we study the labelling issue in existing datasets and its impact on model performance.

3 Methodology

Our approach to analysing the potential vulnerabilities in the source code is based on a comprehensive methodology designed to maximise efficiency and accuracy, as seen in Figure 2. The process begins with collecting code samples in the form of curated datasets. We may assume that these datasets are correctly labelled, yet this is not true according to our experiments and the state of the art [11,6]. The latter motivated the incorporation of a dataset quality test using commercial/industry tools. In this regard, we selected Semgrep [39], yet any set of similar tools could be used. Therefore, we apply Semgrep to assess the quality of the datasets in conjunction with the classification outcomes, as discussed in Section 5. To assess whether a code sample is vulnerable, we apply two strategies. First, we select a dataset, split it into training and testing sets, and use the training set to fine-tune a model. Otherwise, we may use a dataset to directly test a model (i.e., by using the whole dataset as a test set or just a split of it). In all cases, we select models from the Hugging Face Hub [19], yet other pools could be used. From this hub, we selected a subset of models of two categories: models explicitly trained for code vulnerability detection and models designed for general-purpose use. Note that further selection can be made based on performance and hardware requirements; thus, we enriched our model selection based on such criteria. Therefore, given a dataset and a model, we perform an optional fine-tuning procedure and a binary classification task, which assesses whether a code sample is vulnerable or non-vulnerable.

According to our criteria, we selected six models from Hugging Face that were already trained specifically for code vulnerability detection:

– The first five models, developed by Claudio, are VulBERTa-MLP-ReVeal [47], *VulBERTa-MLP-D2A* [44], *VulBERTa-MLP-Draper* [45], *VulBERTa-MLP-*

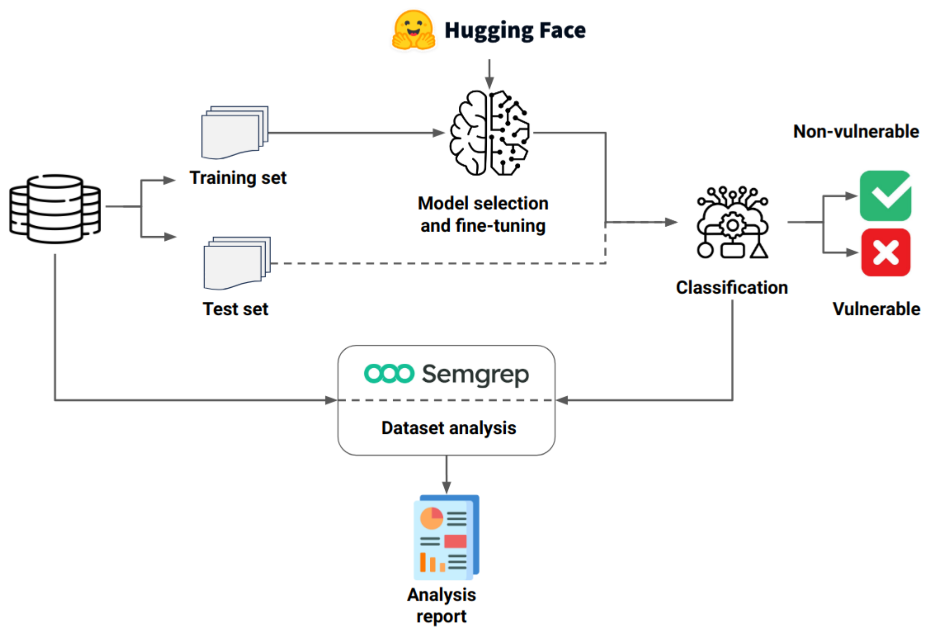

Fig. 2: Overview of our methodology.

MVD [46], and *VulBERTa-MLP-VulDeePecker* [48]. These models share the same architecture and are trained on the ReVeal [4], D2A [53], Draper [37], muVuldeepecker [57], and Vuldeepecker [26] datasets, respectively. VulBERTa-MLP is a fully-connected layer with 768 neurons and one output layer 2 neurons.

– The sixth model is Codebert_fine_tuned_detect_vulnerability_on_MSR (CodeBERT_finetuned_MSR) [15], which has been fine-tuned on the CodeBERT model using the MSR dataset [12].

Further to these trained models, we also selected six LLMs to test their code vulnerability detection capabilities, such as CodeLLama, Mistral, and OpenAI's GPT-4 since they are often used as baselines in the state of the art:

– CodeBERT-base [13], developed by Microsoft, is a pre-trained model for general-purpose code understanding. It has been trained on the CodeSearchNet dataset [20] that includes 2.4 million functions.
– Mistral-7b-base [21], developed by the Mistral AI team, is a pre-trained generative text model with 7.3 billion parameters. It is a decoder-only model.
– Mixtral-8x7b-base [22], also developed by the Mistral AI team, is a high-quality sparse mixture of experts model (SMoE) with open weights. This model is pre-trained on data extracted from the open Web and has 46.7 billion parameters. Mixtral-8x7b shares the same architecture as Mistral-7b, but the main difference is that the feedforward block of Mixtral-8x7b picks from a set of 8 distinct groups of parameters.

– CodeLlama [36] is a family of code-specialized LLMs. It supports many of the most popular languages that are used today, including Python, C++, Java, PHP, Typescript (Javascript), C#, and Bash. In this work, we use the *CodeLlama-7b-base* and *CodeLlama-13b-base* with 7 and 13 billion parameters for comparison.
– GPT-4-base [32], developed by OpenAI, is a large multimodal model (accepting image and text inputs, emitting text outputs). GPT-4 is available on ChatGPT Plus and as an API for developers.

Except for GPT-4-base, all other base models are publicly available on Hugging Face. To adapt CodeBERT-base for vulnerability detection, we construct a custom model using the architecture of `RobertaForSequenceClassification` from the Hugging Face Transformers [51]. The model is configured to be the same as `microsoft/codebert-base` and the default maximum sequence length (512 tokens) of CodeBERT to encode code samples. Regarding Mistral and CodeLLama models, we build them utilising the architecture of `AutoModelForSequenceClass-ification` with the `num_labels` set to 2. Additionally, we apply the 4-bit quantization through `BitsAndBytes` for efficient evaluation as depicted in Fig. 3(a).

As described in our pipeline, we fine-tuned a subset of these base models using a dataset crafted by us, later described in Section 4.2. The fine-tuned models are described as follows:

– CodeBERT-fine-tuned shares the same architecture as CodeBERT-base. The fine-tuning is for 50 epochs, and the model with the minimum loss on the test set is saved for testing.
– Regarding CodeLlama-7b-fine-tuned and Mistral-7b-fine-tuned, we use the PEFT LoRA [18] which stands for Parameter Efficient Fine Tuning (PEFT) using Low-Rank Adaptation (LoRA) method for efficient fine-tuning with 3 epochs, as depicted in Fig. 3(b). Note that the configuration `task_typetask_type=TaskType.SEQ_CLS` is essential for specifying the task type as a sequence classification.

```
#4-bit quantization is applied
    through BitsAndBytesConfig:
load_in_4bit=True
bnb_4bit_use_double_quant=True
bnb_4bit_quant_type="nf4"
bnb_4bit_compute_dtype=torch.bfloat16
```

(a) BitsAndBytes configuration.

```
#LoRA is configured via LoraConfig
    as follows:
task_type=TaskType.SEQ_CLS
r=32
lora_alpha=64
bias="none"
lora_dropout=0.05
```

(b) LoRA configuration.

Fig. 3: Detail of the fine-tuning configuration of the selected LLMs.

4 Experiments

4.1 Prompt Engineering and Hardware Setup

Since crafting prompts that guarantee the expected responses and comprehension by the employed models requires brute-force trial-and-error experimentation, the proper ones were selected after several iterations. In the case of all models, when available, the temperature was set to zero to allow for reproducibility and reduce the hallucinations in local models.

We used OpenAI's GPT-4 (`gpt-4-0613`) via the API offered by OpenAI in our experiments. Fig. 4 shows the prompt to obtain the detection results. Note that since GPT-4 is a paid service, and the charges for its API are based on token consumption, due to budget constraints, we tested it only in our dataset and the Lin2017 dataset (see Section 4.2.

System: You are a senior developer doing security code auditing.

User: Check the following code for vulnerabilities. If you find one, return only 1, otherwise return 0. Suppress all other output. The code is the following : ``` CODE ```

Fig. 4: Structure of the task prompts used in OpenAI's GPT.

The rest of the experiments were conducted on a high-performance computer (HPC) cluster and each cluster node runs a 2.20GHZ Intel Xeon Silver 4210 Processor with an NVIDIA Tesla V100-PCIE-32GB GPU. Models are trained and tested using the PyTorch 2.0.1 framework with CUDA 12.0.

4.2 Datasets

We used six datasets in our experiments[6]. The details of the datasets can be seen in Table 2. First, we created a balanced dataset that includes 13,532 code functions written in C. The 6,766 vulnerable functions were manually collected from projects on GitHub that have registered CVEs in NVD from 2002 to 2023. The 6,766 non-vulnerable code functions are extracted from the DiverseVul dataset [6] to increase the code diversity. The entire dataset is divided randomly, with 80% allocated for fine-tuning (i.e., we use this dataset to fine-tune a subset of models as described in Section 3) and 20% for testing purposes, while ensuring a balanced representation of both vulnerable and non-vulnerable functions. Next, we selected five open-source datasets yet, this time only for testing purposes. For Devign [56], Lin2017 [28], Choi2017 [7], and LineVul [14], we collected all their available data to construct the datasets for testing. While for PrimeVul [11], we only used its test set to ensure a fair and direct comparison with the results outlined in the original paper [11], where the data was originally sourced and evaluated.

[6] All the used datasets are unified and publicly available on Zenodo [52].

Table 2: Detail of datasets and their composition.

Ref.	Dataset	#Vulnerable	#Non-vulnerable	#Total
-	Our dataset	6,766	6,766	13,532
[56]	Devign	12,460	14,858	27,318
[28]	Lin2017	44	577	621
[7]	Choi2017	7,054	6,946	14,000
[14]	LineVul	1,055	17,809	18,864
[11]	PrimeVul	695	25,213	25,908

5 Results and Discussion

Table 3 shows the outcomes for each model and dataset. In all experiments, we employ three widely-used metrics [56], namely precision, recall, and F1-score (F1), to evaluate the detection performance. We computed such metrics per class as they showcase specific behaviours related to the models' performance.

Regarding the Choi2017 dataset, we observe that models reporting high recall values for the vulnerable class do not perform well for the non-vulnerable class and vice-versa. Practically, this means that some models classify all code as vulnerable and thus cannot classify the code with qualitative criteria. The best-performing models are Mixtral-8x7b-base and Mistral-7b-fine-tuned with F1-score values close to 47%, when combining both classes (i.e., we average the F1-score outcomes to provide an indicative value to be used as reference, as the unbalanced nature of the datasets is already collected in the values per class). In the case of LineVul, we observe similar behaviour for some models (e.g., CodeBERT-base shows the same behaviour in all datasets tested), yet we observe a remarkable identification of non-vulnerable code for most models, while vulnerable code is poorly identified. Overall, VulBERTa-based variations obtain the highest F1-score considering both classes, closely followed by CodeBERT-fine-tuned. The outcomes of PrimeVul dataset are similar to those obtained in LineVul, yet this time, CodeLlama-based models and Mixtral-8x7b-base perform similarly to VulBERTa-based variations, obtaining around 50% of F1-score considering both classes. We also observed a similar behaviour in the case of Lin2017, with the difference that best scoring models obtained scores above 60%, as in the case of CodeLlama-7b-base and VulBERTa-MLP-ReVeal. Since LineVul, PrimeVul and Lin2017 are not balanced, the tests showcase the capability of the models to identify non-vulnerable code, as the number of samples is higher. The latter also reinforces the relevance of reporting the outcomes per class, as unbalanced classes could hide underperforming issues [1]. Choi2017, Devign and our dataset are balanced regarding samples per class. In general, models obtain between 30% and 40% of F1-score considering the average of both classes, being CodeBERT-fine-tuned and VulBERTa-MLP-D2A the best-performing ones, with values around 51%. Finally, the outcomes obtained on our dataset showcase the contextual nature of fine-tuning. In this regard, while state-of-the-art models perform similarly to the rest of the datasets (i.e., with averaged F1-score values

Table 3: For each dataset and model, we report the P (precision), R (recall) and F1-score per class (i.e., vulnerable, non-vulnerable) to ease their interpretation.

Ref.	Model	Choi2017						LineVul						PrimeVul					
		Vulnerable			Non-vulnerable			Vulnerable			Non-vulnerable			Vulnerable			Non-vulnerable		
		P	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
[47]	VulBERTa-MLP-ReVeal	0	0	0	49.61	100	66.32	16.27	16.87	16.57	95.06	94.86	94.96	2.10	40.43	3.99	98.20	89.55	93.67
[44]	VulBERTa-MLP-D2A	50.39	100	67.01	0	0	0	5.93	41.33	10.37	94.62	61.15	74.29	2.10	33.24	3.96	96.89	57.36	72.06
[45]	VulBERTa-MLP-Draper	0	0	0	49.61	100	66.32	0	0	0	94.41	100	97.12	0	0	0	97.32	100	98.64
[46]	VulBERTa-MLP-MVD	57.96	1.29	2.52	49.70	99.05	66.19	9.85	7.01	8.19	94.58	96.20	95.38	5.32	11.80	7.33	97.48	94.21	95.82
[48]	VulBERTa-MLP-VulDeePecker	0	0	0	49.61	100	66.32	14.62	2.37	4.08	94.49	99.18	96.78	4.78	2.45	3.24	97.35	98.66	98
[15]	CodeBERT_finetuned_MSR	50.28	15.21	23.36	49.60	84.73	62.57	9.29	5.50	6.91	94.53	96.82	95.66	6.30	63.31	11.47	98.65	74.06	84.61
[13]	CodeBERT-base	50.39	100	67.01	0	0	0	5.59	100	10.59	0	0	0	2.68	100	5.23	0	0	0
[21]	Mistral-7b-base	50.38	99.99	67	0	0	0	4.86	50.62	8.86	93.38	41.26	57.23	2.33	75.97	4.53	94.91	12.34	21.84
[22]	Mixtral-8x7b-base	50.89	29.17	37.09	49.82	71.41	58.69	6.04	44.83	10.65	94.73	58.69	72.48	5.88	0.43	0.80	97.32	99.81	98.55
[36]	CodeLlama-7b-base	44.99	13.69	20.99	48.64	83	61.33	5.34	49	9.64	94.15	48.59	64.10	3.20	40.58	5.93	97.58	66.13	78.84
[36]	CodeLlama-13b-base	49.04	70.37	57.80	46.09	25.73	33.02	2.45	29	4.51	88.22	31.50	46.42	4.16	12.95	6.29	97.45	91.77	94.52
[32]	GPT-4-base	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Our	CodeBERT-fine-tuned	50.32	97.87	66.47	46.43	1.87	3.60	10.78	63.13	18.42	96.93	69.06	80.66	5.30	85.90	9.99	99.33	57.71	73.01
Our	Mistral-7b-fine-tuned	50.18	72.63	59.35	49.06	26.78	34.65	6.35	86.16	11.84	96.80	24.78	39.46	3.04	89.21	5.88	98.64	21.54	35.36
Our	CodeLlama-7b-fine-tuned	50.39	100	67.01	0	0	0	5.96	94.88	11.21	97.37	11.24	20.16	2.80	95.40	5.44	98.56	8.71	16.01

Ref.	Model	Our Dataset						Devign						Lin2017					
		Vulnerable			Non-vulnerable			Vulnerable			Non-vulnerable			Vulnerable			Non-vulnerable		
		P	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**	**P**	**R**	**F1**
[47]	VulBERTa-MLP-ReVeal	78.31	17.07	28.03	53.46	95.27	68.49	51.25	8.38	14.40	54.84	93.32	69.08	26.55	68.18	38.22	97.24	85.62	91.06
[44]	VulBERTa-MLP-D2A	48.06	51.37	49.67	36.27	44.49	39.96	46.93	52.36	49.49	55.75	50.34	52.91	7..44	5.23	13.03	93.27	50.43	65.47
[45]	VulBERTa-MLP-Draper	0	0	0	50	100	66.67	0	0	0	54.39	100	70.46	0	0	0	92.91	100	96.33
[46]	VulBERTa-MLP-MVD	70.59	6.21	11.41	50.95	97.41	66.90	47.60	4.21	7.74	54.47	96.11	69.53	6.90	4.55	5.48	92.91	95.32	94.10
[48]	VulBERTa-MLP-VulDeePecker	74.23	5.32	9.93	50.90	98.15	67.04	56.72	3.35	6.33	54.70	97.85	70.17	25	2.27	4.17	93.03	99.48	96.15
[15]	CodeBERT_finetuned_MSR	86.39	9.39	16.93	52.09	98.52	68.15	50.82	6.50	11.53	54.71	94.72	69.36	15.04	90.91	25.81	98.87	60.83	75.32
[13]	CodeBERT-base	50	100	66.67	0	0	0	45.61	100	62.65	0	0	0	7.09	100	13.23	0	0	0
[21]	Mistral-7b-base	49.09	93.50	64.38	31.78	3.03	5.53	45.57	99.37	62.48	46.26	0.46	0.91	28.07	36.36	31.68	95.04	92.89	93.95
[22]	Mixtral-8x7b-base	83.33	0.37	0.74	50.07	99.93	66.72	42.15	43.13	42.64	51.36	50.36	50.86	2.86	36.36	5.31	54.84	5.89	10.64
[36]	CodeLlama-7b-base	51.57	60.61	55.73	52.24	43.09	47.23	46.19	82.40	59.19	48.76	13.38	21	42.86	34.09	37.97	95.05	96.53	95.79
[36]	CodeLlama-13b-base	48.76	92.98	63.97	24.60	2.29	4.19	45.25	92.95	60.86	48.95	5.67	10.16	5.65	47.73	10.10	90.76	39.17	54.72
[32]	GPT-4-base	55.91	95.42	70.51	84.38	24.76	38.29	-	-	-	-	-	-	8.63	100	15.88	100	19.24	32.27
Our	CodeBERT-fine-tuned	70.22	74.94	72.50	73.12	68.14	70.54	47.98	63.08	54.51	57.94	42.65	49.13	16.60	95.45	28.28	99.46	63.43	77.46
Our	Mistral-7b-fine-tuned	88.70	85.88	87.27	86.32	89.06	87.67	45.87	92.17	61.25	57.20	8.77	15.21	7.63	100	14.17	100	7.63	14.17
Our	CodeLlama-7b-fine-tuned	97.97	96.30	97.13	96.37	98	97.18	45.91	97.17	62.36	62.12	4.14	7.76	7.09	100	13.23	100	0.52	1.03

raging between 30% and 50% considering both classes), all our fine-tuned models achieve values over 70%, with CodeLlama-7b-fine-tuned achieving a remarkable 97%.

The previously discussed outcomes raised our curiosity, as they highlighted incongruencies and alarmingly low accuracy in the code vulnerability detection task. We wanted to delve into this and performed another experiment to explore the quality of the datasets. As reported in the state of the art, [11,6] several widely-used datasets present mislabelling issues due to, e.g., the use of automated tools for annotation, or treating code as fixed after a commit even though the vulnerability was not solved. Thus, further to merely using the datasets, we used Semgrep [39] to scan code snippets using the command-line interface - CLI with Semgrep public rules [40]. While there are other tools such as Snyk [42] and SonarQube [43] that are often used to detect source code vulnerabilities, none of them could be used in our experiments. The reason is that both these tools operate on projects and not code segments, as in the case of these datasets. To determine whether there is a vulnerability, they need full access to the code to assess the imported libraries, dependencies, etc. However, this is not the case for Semgrep. Quite interestingly, Semgrep faced many issues in scanning the code snippets of the datasets. In fact, for each dataset, it reported scanning issues in the form of a message: `Partially scanned: X files only partially analyzed due to parsing or internal Semgrep errors`, with X varying in each dataset (the #Issues column in Table 4). As noted in Table 4, even in these cases, Semgrep identified a very small fragment of the vulnerabilities that each dataset contains. Therefore, we assert that the information provided in all datasets may not be sufficient for existing industry tools to reliably determine whether the code snippets are vulnerable. The latter showcases the previously stated concerns regarding dataset labelling, which requires further analysis to ensure that models are not trained with erroneous data or data that can create conflicts.

Table 4: Detail of the Semgrep detection result. #Vulnerable from labeled vulnerable/non-vulnerable code: the number of vulnerable code detected by Semgrep from the labeled vulnerable/non-vulnerable data, respectively. #Issues: number of data that happened the partially scanned issue.

		Labeling in the dataset		**Semgrep detection**		
Ref.	**Dataset**	**#Vulnerable**	**#Non-vulnerable**	**#Vulnerable from labeled vulnerable code**	**#Vulnerable from labeled non-vulnerable code**	**#Issues**
-	Our dataset	1353	1353	75	3	745
[56]	Devign	12460	14858	166	169	3708
[28]	Lin2017	44	577	26	0	185
[7]	Choi2017	7054	6946	0	0	10000
[14]	LineVul	1055	17809	104	0	10842
[11]	PrimeVul	695	25213	41	161	11975

Simultaneously, this raises another important question. If such tools are not able to detect vulnerabilities in such code fragments, how sure are we that the provided information is enough for LLMs to find vulnerabilities? For instance, tools like Semgrep use rules that describe string patterns[7] to find vulnerable code. Nevertheless, the extent of failure of such tools in identifying vulnerabilities in the code fragments of the datasets can potentially signify that what the LLM understands from its training is very limited or not precise enough, making it mark all code as vulnerable. While we expect the LLM's tokenizer to accurately segment code into tokens, the task of identifying the roles (e.g. variable names and functions) to understand that, e.g., passing unprocessed user input to a function can lead to a code injection attack goes beyond its capability. However, Semgrep and similar tools already have the rule for that and fail to detect the vulnerability. While one could consider that the tokenizer of Semgrep is not good enough, since this is a well-established tool, we opt to attribute such failures to lack of proper contextual understanding. Indeed, this could just justify our research findings and the failure of LLMs to accurately find vulnerabilities when tested in different datasets. We argue that LLMs generate broad rules based on their training tokens, which can incorrectly mark code fragments as vulnerable due to their limited ability to discern the code context. Even worse, Semgrep identifies vulnerabilities in code that was labelled secure in several datasets, raising even more questions about the quality of the datasets. Even if the detection results are false positives, the fact that they are detected by such a tool implies that the LLMs could be wrongfully trained and fail to identify the proper patterns.

6 Conclusions

The advent of generative AI tools and the sophistication of software production have enhanced the lifecycle and robustness of digital products and services. Nevertheless, the analysis of the current state of practice reveals that we are only beginning to scratch the surface regarding LLMs' capabilities. Thus, significant efforts must be devoted to realising accurate and efficient automated code vulnerability detection. The research questions posed in Section 1 summarise the main aim of our research, namely providing a comprehensive analysis of the state of practice in code vulnerability detection analysis through the use of AI, its main challenges and elaborating a fruitful discussion on this particular matter. We discuss them in order as follows:

RQ1: *Which methods are currently used for source code vulnerability detection?*

To provide enough background to discuss the current state of the art, we provide an extensive analysis of related work, including traditional SAST-based, task-specific DL models, and LLM-based vulnerability detection. As discussed in Section 2, LLMs are gaining momentum and therefore it is crucial to study their potential.

RQ2: *Can base LLMs detect vulnerabilities in source code?*

[7] `https://semgrep.dev/docs/writing-rules/rule-ideas/`

Given the analysis of the state of the art and the experiments performed in this paper, the answer to that question is unclear. One could argue that LLMs can effectively detect vulnerabilities in source code, yet their accuracy is particularly tied to their training data, which generally performs primarily on the patterns included in training data. Furthermore, larger models exhibit more stable accuracy across datasets yet still do not achieve remarkable outcomes.

> ***RQ3:*** *Is fine-tuning an enabling strategy to improve the trade-off between computational resources and detection accuracy?*

Given the outcomes analysed in Section 5, fine-tuning allows low resource-demanding models to outperform larger ones in specific contexts. Despite having fewer parameters than commercial models, local LLMs can be fine-tuned to optimise their performance in specific tasks as their weights are made publicly available, which we will explore in future work. The latter includes exploring smaller LLMs (e.g., through quantisation [10] and number of parameters) to provide resource-efficient solutions, fostering the adoption of LLMs in constrained environments. Nevertheless, this entails several constraints, such as the generalisation issues discussed in RQ4.

> ***RQ4:*** *How robust are the analysed detection models?*

As seen in Section 5, the extent of the application context is closely tied to the training data since models usually do not generalise well when exposed to different testing environments. The latter requires a specific analysis of the benchmarks, as modifications can derive unexpected model behaviour and classification errors. Moreover, data curation is a parallel issue, as discussed in RQ5.

> ***RQ5:*** *Are curation and labelling methodologies employed on existing datasets robust enough for training LLMs and ensuring their desired functionality?*

As highlighted in the state of the art and according to our dataset analysis experiments with Semgrep, there are concerning issues regarding the labelling of datasets. Issues such as the length of the code sample and the use of automated strategies with flaws create contradictory judgements about the samples. While this could only mean the inability to evaluate models properly, in the case of LLMs this incurs further fundamental issues, as they are trained on these datasets, thus corrupting the entire functionality, as in, e.g., poisoning attacks.

> ***RQ6:*** *Given the analysis and outcomes provided in this paper, what are the next steps towards software vulnerability detection ?*

This work provides a clear insight into the current state of practice and critical aspects that should be improved towards the reliability of LLMs and similar models. In this regard, our future research paths are aligned with our outcomes and focus on producing quality datasets. The latter can be done by establishing a sound methodology to guarantee that they can be used in software development, security, and operations cycles (e.g., by ensuring formatting and length, curation, and providing data related to the CWEs to allow precise and reliable evaluation). In parallel, we aim to delve into how LLMs acquire knowledge, e.g., by fine-tuning processes, to avoid overfitting and optimising their generalisation capabilities. Finally, aspects related to explainability and pedigree, namely which datasets were used to train and create models, are essential to ensure their robustness and avoid biased evaluations.

Acknowledgments. This work was supported by the European Commission under the Horizon Europe Programme, as part of the projects CyberSecPro (`https://www.cybersecpro-project.eu`) (Grant Agreement no. 101083594) and LAZARUS (`https://lazarus-he.eu/`) (Grant Agreement no. 101070303). This work was partially supported by Ministerio de Ciencia, Innovación y Universidades, Gobierno de España (Agencia Estatal de Investigación, Fondo Europeo de Desarrollo Regional -FEDER-, European Union) under the research grant PID2021-127409OB-C33 CONDOR. Fran Casino was supported by the Government of Catalonia with the Beatriu de Pinós programme (Grant No. 2020 BP 00035), and by AGAUR with the project ASCLEPIUS (2021SGR-00111).

The content of this article does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the authors.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Casino, F., Lykousas, N., Homoliak, I., Patsakis, C., Hernandez-Castro, J.: Intercepting hail hydra: Real-time detection of algorithmically generated domains. Journal of Network and Computer Applications **190**, 103135 (2021)
2. CERN Computer Security Team: RATS: rough auditing tool for security. `https://security.web.cern.ch/recommendations/en/codetools/rats.shtml` (2024), online, accessed on April 16th, 2024
3. Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: Are we there yet? IEEE Transactions on Software Engineering **48**(09), 3280–3296 (September 2022). `https://doi.org/10.1109/TSE.2021.3087402`
4. Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: are we there yet? IEEE Transactions on Software Engineering **48**(09), 3280–3296 (September 2022). `https://doi.org/10.1109/TSE.2021.3087402`
5. Charalambous, Y., Tihanyi, N., Jain, R., Sun, Y., Ferrag, M.A., Cordeiro, L.C.: A new era in software security: Towards self-healing software via large language models and formal verification. arXiv preprint arXiv:2305.14752 (2023), `https://arxiv.org/abs/2305.14752`
6. Chen, Y., Ding, Z., Alowain, L., Chen, X., Wagner, D.: 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. p. 654–668. RAID '23, Association for Computing Machinery, New York, NY, USA (2023). `https://doi.org/10.1145/3607199.3607242`

7. Choi, M.J., Jeong, S., Oh, H., Choo, J.: End-to-end prediction of buffer overruns from raw source code via neural memory networks. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence. p. 1546–1553. IJCAI'17, AAAI Press (2017)
8. Cppcheck team: Cppcheck. https://cppcheck.sourceforge.io/ (2024), online, accessed on April 16th, 2024
9. Croft, R., Newlands, D., Chen, Z., Babar, M.A.: An empirical study of rule-based and learning-based approaches for static application security testing. In: Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). ESEM '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3475716.3475781
10. Dettmers, T., Pagnoni, A., Holtzman, A., Zettlemoyer, L.: Qlora: Efficient fine-tuning of quantized llms. Advances in Neural Information Processing Systems **36** (2024)
11. Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., Wagner, D., Ray, B., Chen, Y.: Vulnerability detection with code language models: how far are we? arXiv preprint arXiv:2403.18624 (2024), https://arxiv.org/abs/2403.18624
12. Fan, J., Li, Y., Wang, S., Nguyen, T.N.: A c/c++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories. p. 508–512. MSR '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3379597.3387501
13. Feng, Z., Guo, D., Tang, D., et al.: Codebert: a pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1536–1547. Association for Computational Linguistics (November 2020). https://doi.org/10.18653/v1/2020.findings-emnlp.139
14. Fu, M., Tantithamthavorn, C.: Linevul: a transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories. p. 608–620. MSR '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3524842.3528452
15. Gui, Y.: Model card of starmage520/Coderbert_finetuned_detect_vulnerability_on_MSR on hugging face. https://huggingface.co/starmage520/Coderbert_finetuned_detect_vulnerability_on_MSR (2023), online, accessed on April 16th, 2024
16. Guo, Y., Hu, Q., Tang, Q., Traon, Y.L.: An empirical study of the imbalance issue in software vulnerability detection. In: Computer Security – ESORICS 2023: 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25–29, 2023, Proceedings, Part IV. p. 371–390. Springer-Verlag, Berlin, Heidelberg (2024). https://doi.org/10.1007/978-3-031-51482-1_19
17. Hanif, H., Maffeis, S.: Vulberta: simplified source code pre-training for vulnerability detection. In: International Joint Conference on Neural Networks (IJCNN). pp. 1–8. IEEE (2022). https://doi.org/10.1109/IJCNN55064.2022.9892280
18. Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: LoRA: Low-rank adaptation of large language models. In: International Conference on Learning Representations (2022), https://openreview.net/forum?id=nZeVKeeFYf9
19. Hugging Face: Hugging Face. https://huggingface.co/, online, accessed on April 16th, 2024
20. Husain, H., Wu, H.H., Gazit, T., Allamanis, M., Brockschmidt, M.: Codesearchnet challenge: evaluating the state of semantic code search. arXiv preprint arXiv: 1909.09436 (2020), https://arxiv.org/abs/1909.09436

21. Jiang, A.Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D.S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L.R., Lachaux, M.A., Stock, P., Scao, T.L., Lavril, T., Wang, T., Lacroix, T., Sayed, W.E.: Mistral 7b. arXiv preprint arXiv:2310.06825 (2023), https://arxiv.org/abs/2310.06825
22. Jiang, A.Q., Sablayrolles, A., Roux, A., et al.: Mixtral of experts. arXiv preprint arXiv:2401.04088 (2024), https://arxiv.org/abs/2401.04088
23. Lee, M., Cho, S., Jang, C., Park, H., Choi, E.: A rule-based security auditing tool for software vulnerability detection. In: International Conference on Hybrid Information Technology. vol. 2, pp. 505–512. IEEE (2006). https://doi.org/10.1109/ICHIT.2006.253653
24. Li, H., Hao, Y., Zhai, Y., Qian, Z.: 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. pp. 2107–2111. ESEC/FSE 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3611643.3613078
25. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: a deep learning-based system for vulnerability detection. In: 25th Annual Network and Distributed System Security Symposium (NDSS). The Internet Society (2018). https://doi.org/10.14722/ndss.2018.23158
26. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: a deep learning-based system for vulnerability detection. In: 25th Annual Network and Distributed System Security Symposium (NDSS). The Internet Society (2018), http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-2_Li_paper.pdf
27. Lin, G., Wen, S., Han, Q.L., Zhang, J., Xiang, Y.: Software vulnerability detection using deep neural networks: a survey. Proceedings of the IEEE **108**(10), 1825–1848 (2020). https://doi.org/10.1109/JPROC.2020.2993293
28. Lin, G., Zhang, J., Luo, W., Pan, L., Xiang, Y.: Vulnerability discovery with function representation learning from unlabeled projects. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. p. 2539–2541. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3138840
29. Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V.: Roberta: A robustly optimized BERT pretraining approach. https://arxiv.org/abs/1907.11692 (2019), arXiv preprint
30. National Institute of Standards and Technology: Secure Software Development Framework (SSDF) Version 1.1: (2018)
31. Noever, D.: Can large language models find and fix vulnerable software? arXiv preprint arXiv:2308.10345 (2023), https://arxiv.org/abs/2308.10345
32. OpenAI, Achiam, J., Adler, S., et al.: Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2024), https://arxiv.org/abs/2303.08774
33. OWASP: OWASP DevSecOps Guideline. https://github.com/OWASP/DevSecOpsGuideline/tree/master (2024), online, accessed on April 16th, 2024
34. Poeplau, S., Francillon, A.: Symbolic execution with SymCC: don't interpret, compile! In: Proceedings of the 29th USENIX Conference on Security Symposium. pp. 181–198. SEC'20, USENIX Association, USA (August 2020), https://dl.acm.org/doi/10.5555/3489212.3489223
35. Ribeiro, F., de Macedo, J.N.C., Tsushima, K., Abreu, R., Saraiva, J.: Gpt-3-powered type error debugging: investigating the use of large language models for

code repair. In: Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering. pp. 111–124. SLE 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3623476.3623522

36. Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C.C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., Synnaeve, G.: Code llama: open foundation models for code. arXiv preprint arXiv:2308.12950 (2024), https://arxiv.org/abs/2308.12950
37. Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M.: Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE international conference on machine learning and applications (ICMLA). pp. 757–762. IEEE (2018)
38. Sampaio, L., Garcia, A.: Exploring context-sensitive data flow analysis for early vulnerability detection. Journal of Systems and Software **113**(C), 337–361 (March 2016). https://doi.org/10.1016/j.jss.2015.12.021
39. Semgrep, Inc: Semgrep - find bugs and enforce code standards. https://semgrep.dev/ (2024), online, accessed on April 16th, 2024
40. Semgrep Team: semgrep-rules. https://github.com/semgrep/semgrep-rules (2024), online, accessed on June 21st, 2024
41. Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A., Piras, L.: Android source code vulnerability detection: a systematic literature review. ACM Computing Surveys **55**(9) (January 2023). https://doi.org/10.1145/3556974
42. Snyk team: Snyk code: developer focused, real-time sast. https://snyk.io/product/snyk-code/ (2024), online, accessed on April 16th, 2024
43. SonarSource: Code quality, security & static analysis too with SonarQube. https://www.sonarsource.com/products/sonarqube/ (2024), online, accessed on April 16th, 2024
44. Spiess, C.: Model card of claudios/VulBERTa-MLP-D2A on Hugging Face. https://huggingface.co/claudios/VulBERTa-MLP-D2A (2024), online, accessed on April 16th, 2024
45. Spiess, C.: Model card of claudios/VulBERTa-MLP-Draper on Hugging Face. https://huggingface.co/claudios/VulBERTa-MLP-Draper (2024), online, accessed on April 16th, 2024
46. Spiess, C.: Model card of claudios/VulBERTa-MLP-MVD on hugging face. https://huggingface.co/claudios/VulBERTa-MLP-MVD (2024), online, accessed on April 16th, 2024
47. Spiess, C.: Model card of claudios/VulBERTa-MLP-ReVeal on Hugging Face. https://huggingface.co/claudios/VulBERTa-MLP-ReVeal (2024), online, accessed on April 16th, 2024
48. Spiess, C.: Model card of claudios/VulBERTa-MLP-VulDeePecker on hugging face. https://huggingface.co/claudios/VulBERTa-MLP-VulDeePecker (2024), online, accessed on April 16th, 2024
49. Viega, J., Bloch, J.T., Kohno, T., McGraw, G.: Token-based scanning of source code for security problems. ACM Transactions on Information and System Security **5**(3), 238–261 (August 2002). https://doi.org/10.1145/545186.545188
50. Wheeler, D.A.: Flawfinder. https://dwheeler.com/flawfinder/ (2017), online, accessed on April 16th, 2024

51. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T.L., Gugger, S., Drame, M., Lhoest, Q., Rush, A.M.: Transformers: state-of-the-art natural language processing. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. pp. 38–45. Association for Computational Linguistics, Online (October 2020). `https://doi.org/10.18653/v1/2020.emnlp-demos.6`
52. Yuejun, G.: A collection of datasets for software vulnerability detection (version 1.0). `https://zenodo.org/records/10975439` (April 2024), on Zenodo, accessed on April 16th, 2024
53. Zheng, Y., Pujar, S., Lewis, B., Buratti, L., Epstein, E., Yang, B., Laredo, J., Morari, A., Su, Z.: D2a: a dataset built for ai-based vulnerability detection methods using differential analysis. In: Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice. p. 111–120. ICSE-SEIP '21, IEEE Press (2021). `https://doi.org/10.1109/ICSE-SEIP52600.2021.00020`, `https://doi.org/10.1109/ICSE-SEIP52600.2021.00020`
54. Zheng, Z., Ning, K., Chen, J., Wang, Y., Chen, W., Guo, L., Wang, W.: Towards an understanding of large language models in software engineering tasks. arXiv preprint arXiv:2308.11396 (2023), `https://arxiv.org/abs/2308.11396`
55. Zhou, X., Zhang, T., Lo, D.: Large language model for vulnerability detection: emerging results and future directions. arXiv preprint arXiv:2401.15468 (2024), `https://arxiv.org/abs/2401.15468`
56. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems. pp. 10197–10207. Curran Associates Inc., Red Hook, NY, USA (December 2019), `https://dl.acm.org/doi/pdf/10.5555/3454287.3455202`
57. Zou, D., Wang, S., Xu, S., Li, Z., Jin, H.: μvuldeepecker: a deep learning-based system for multiclass vulnerability detection. IEEE Transactions on Dependable and Secure Computing **PP**, 1–1 (09 2019). `https://doi.org/10.1109/TDSC.2019.2942930`