Software Vulnerability Detection using Large Language Models

Conference Paper · October 2023

DOI: 10.1109/ISSREW60843.2023.00058

4 authors:

Moumita Das Purba
University of North Carolina at Charlotte
4 PUBLICATIONS **103** CITATIONS
SEE PROFILE

Arpita Ghosh
University of North Carolina at Charlotte
1 PUBLICATION **81** CITATIONS
SEE PROFILE

Benjamin J Radford
University of North Carolina at Charlotte
27 PUBLICATIONS **535** CITATIONS
SEE PROFILE

Bill Chu
Wenzhou University
60 PUBLICATIONS **1,587** CITATIONS
SEE PROFILE

Software Vulnerability Detection using Large Language Models

(Invited Paper)

Moumita Das Purba
Department of Software and Information Systems
University of North Carolina at Charlotte, NC, USA
mpurba1@uncc.edu

Arpita Ghosh
Department of Software and Information Systems
University of North Carolina at Charlotte, NC, USA
aghosh8@uncc.edu

Benjamin J. Radford
Department of Political Science and Public Administration
University of North Carolina at Charlotte, NC, USA
benjamin.radford@charlotte.edu

Bill Chu
Department of Software and Information Systems
University of North Carolina at Charlotte, NC, USA
billchu@uncc.edu

***Abstract*—Software development is among the first demonstrations of using Large Language Models (LLMs) to enhance human productivity. Such a co-pilot paradigm envisions LLM working side-by-side with human developers to assist in programming tasks. Ensuring the security of software products is a critical factor for the success of such a paradigm. There have been various anecdotal reports on the success of using LLMs to detect vulnerabilities in programs. This paper reports a set of experiments applying four well-known LLMs to two widely referenced public datasets to evaluate the performance of LLMs in detecting software vulnerabilities. Our results show a significant performance gap between these LLMs and those from popular static analysis tools, primarily due to their high false positive rates. However, LLMs show great promise in identifying subtle patterns commonly associated with software vulnerabilities. This observation suggests a possible path forward by combining LLMs and other program analysis techniques to achieve better software vulnerability detection.**

***Index Terms*—Cybersecurity, Large language model, AI, software vulnerability**

I. Introduction

Large language models (LLMs) have recently demonstrated unprecedented performance across a range of language-related tasks [1]. These models primarily rely on Transformers, an innovative neural network architecture that allows models to "attend to," or condition on, large text excerpts [2]. LLMs are typically pre-trained with very large text corpora using one of several self-supervised training strategies such as masked language modeling or next sentence prediction [3]. However, the value of LLMs is that they can then be applied to any number of subsequent tasks with little or no additional training or "fine-tuning".

The success of these models is exemplified by GPT, generative pre-trained Transformer [4]. GPT is the common name for a family of Transformer-based models produced by OpenAI. ChatGPT is a commercial product offered by OpenAI based on this family of LLMs. OpenAI reports that the most recent of these, GPT-4, can outperform humans on a number of standardized exams and outperforms all other models according to a battery of academic benchmarks [5]. Transformer models trained on source code, rather than human language, are able to generate valid code that is responsive to a prompt and capable of passing associated unit tests [6]. Microsoft Github's Copilot is one such LLM that is very popular among programmers for its ability to generate valid functions from natural language descriptions of those functions.[1] In fact, even LLMs that are trained primarily on human language, such as ChatGPT, are often capable of generating valid source code or helping to debug a user's supplied code. OpenAI advertises ChatGPT as an interactive debugging tool on its blog.[2]

Given the clear interest in LLMs for code generation and debugging, we seek to determine whether LLMs are similarly capable of identifying security vulnerabilities in otherwise valid code. As developers integrate these tools into their workflows, their use in security and quality assurance applications is sure to follow. We evaluate four LLMs of various types on three software vulnerability detection tasks. Our experiments are meant to mimic a realistic scenario in which a code excerpt is provided by a developer to an LLM that is then prompted to determine whether the code in question contains a certain class of security vulnerability. Further emulating the range of likely real-world scenarios, we evaluate both fine-tuned LLMs and zero-shot (not fine-tuned) models.

We limit our analysis to two common categories of vulnerabilities: SQL injection and buffer overflow. Doing so allows us not only quantitatively to evaluate the models' performance on our benchmarks but also to interrogate the model outputs manually. We use four different LLMs representing both a smaller LLM hosted on a local machine and larger LLMs hosted by OpenAI. It is important to evaluate both since some developers and researchers may prefer locally hosted solutions for data privacy or confidentiality purposes. We distinguish our LLMs as "fine-tuned" binary classifiers and not fine-tuned

[1] https://github.com/features/copilot
[2] https://openai.com/blog/chatgpt

("prompted") models.

Our objective is to determine whether LLMs are able to classify a given snippet of code as either containing or not containing a particular class of software vulnerability. We then seek to determine the degree to which fine-tuning an LLM on labeled data of code with(out) known vulnerabilities increases model performance on this task. Finally, we manually inspect and manipulate selected code examples to identify the attributes that LLMs are able or unable to pick up on when predicting vulnerability.

Research questions We seek to answer the following questions:

- How do LLMs compare with standard tools for application vulnerability discovery, such as static analysis tools?
- How do LLMs compare with customized neural networks to detect application vulnerabilities?
- What are the opportunities and limitations of using LLMs in detecting application vulnerabilities?

Contribution While there have been anecdotal reports [7], [8] on the use of ChatGPT (prompted LLM) to detect software vulnerabilities as well as preliminary (non-refereed) reports [9] on using Transformer models to detect software vulnerabilities, we are not aware of published evaluations using large datasets comparing results under experimental settings. This paper reports our attempt to provide such a systematic evaluation. We compare results using four different LLMs using standard machine learning evaluation metrics. We also attempt to find qualitative insights about what types of vulnerabilities LLMs might be best positioned to detect, as well as demonstrate their limitations. We further report issues we encountered while using LLMs for the benefit of other researchers.

II. Related Work

Despite the progress achieved through various source code analysis techniques, security researchers encounter significant challenges in detecting software vulnerabilities. Possible challenges are complex software with millions of lines of code and numerous interconnected components, an evolving threat landscape, poor coding practices, the need to practice proper security measures, and others.

For detecting software vulnerabilities, pattern-based approaches were widely adopted. Initially, open-source tools such as Flawfinder [10], RATS [11], and commercial tools such as Checkmarx [12], Fortify [13] were developed using human experts to write rules manually. However, the security community struggles with high false positive or high false negative rates, leading them to data-driven approaches like machine learning techniques. Grieco et al. [14] has developed the VDiscover tool to detect vulnerabilities using logistic regression and MLP. Russell et al. have used a random forest classifier to demonstrate deep feature representation learning as a promising approach for automated software vulnerability detection [15]. N-gram language models have been used to detect code vulnerability by identifying low-probability token sequences [16]. However, most of these methods struggle with low-precision results in real-world scenarios and can not identify the exact location of vulnerabilities.

Latest developments in deep learning have generated interest among researchers, prompting them to delve into the potential effectiveness of these methods in detecting vulnerabilities. Compared with machine learning techniques, deep learning techniques enable complex feature learning, automating latent features' learning, and flexibility in implementing network structures like adding dense layers or attention layers based on application requirements [17]. According to VulDeePecker [18], they were the pioneers in using DL to detect vulnerabilities. Several studies have utilized various neural network architectures to detect vulnerable source code, such as Graph Neural Networks [19], Bidirectional RNNS [20], and others. The transformer-based approach language model by Thapa et al. [21] outperforms BiLSTM and BiGRU models on the VulDeePecker dataset containing two CWEs. In their research, Xin Liet. a [22] employed hybrid neural networks to grasp the patterns of code vulnerabilities. Gaigai Tang et al. [23] employed neural network algorithms to uncover software vulnerabilities automatically, training the detection model using Bi-LSTM and RVFL neural network techniques. Although deep learning techniques are promising approaches for detecting vulnerabilities, they also deal with certain challenges: lack of a large amount of labeled and balanced training data and lack of explainability because of the black-box nature of deep learning.

In the Natural language processing community, Bidirectional Encoder Representations from Transformers (BERT) [3] and Generative Pre-trained Transformer (GPT) [4] models are considered the most advanced and finest feature extraction model. Both models have been utilized for code-centric tasks, generating moderate outcomes, thereby indicating ample scope for improvement. Research efforts have been reported on detecting vulnerabilities using BERT architecture [21], [24], [25]. CodeBERT [26], a bimodal pre-trained model for natural language (NL) and programming language (PL), has been used for vulnerability detection and achieved moderate results [27]. GPT model series has shown their ability on code-related tasks such as code generation, code understanding, and others. However, lots of improvement is still needed in the area of vulnerability detection in code due to fewer technical reports [9], working scope with short source code only for limited token size issue [5], lack of qualitative explanation. In contrast to those reports, this paper intends to provide a comprehensive and systematic qualitative explanation.

III. Empirical evaluations

A. *Dataset selection*

We selected two labeled datasets for our research: code gadgets and CVEfixes, both of which have previously appeared in peer-reviewed publications. The code gadgets dataset [18] contains 61,638 snippets of C and C++ code with memory management and/or buffer overflow vulnerabilities. Peer-reviewed publications have used this dataset to evaluate various approaches for vulnerability detection. These approaches

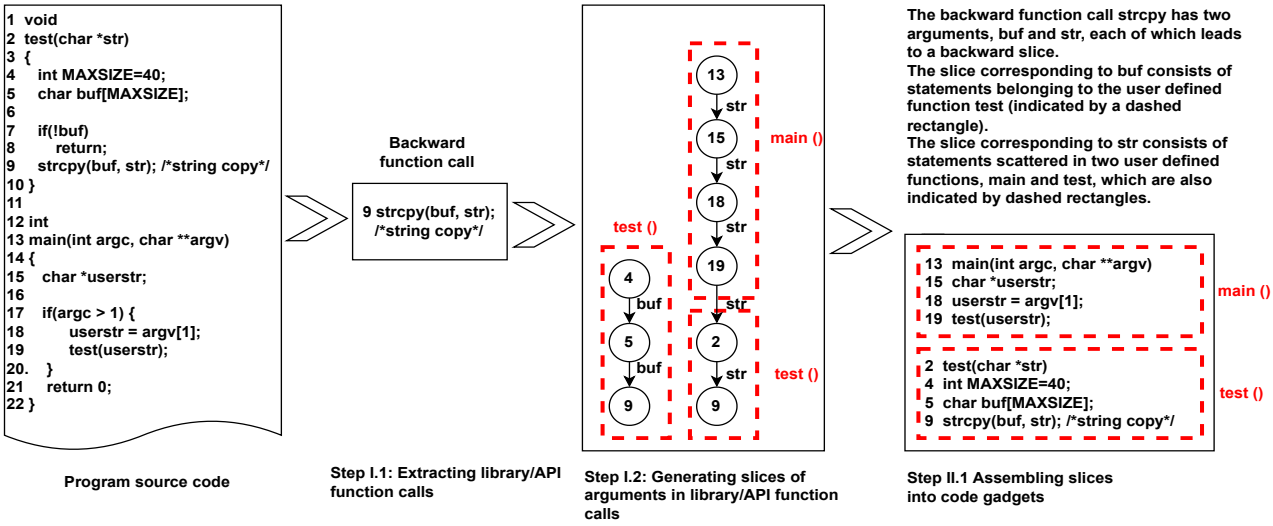

Fig. 1: How a Code Gadget sample is generated

include open-source code scanning tools, a commercial static analysis tool, and a customized neural network approach.

Code snippets included in this dataset are extracted using code slicing to focus on the parts of the program that are most likely to lead to buffer overflow vulnerabilities. Each code snippet, referred to as a code gadget, is a slice of code that includes at least one of the C-library functions commonly involved with buffer overflow (e.g., memset(), strncpy()). Figure 1, taken from [18], illustrates how a gadget is extracted.

While the Code Gadgets dataset affords us the opportunity to compare the performance of LLMs against other techniques, it contains heavily processed code, namely code gadgets. We also wish to evaluate LLM's performance on unprocessed code. For this purpose, we selected CVEfixes, another labeled dataset, to consider a different type of vulnerability, SQL injection, and buffer overflow vulnerabilities in unprocessed form.

CVEfixes is the subject of a refereed artifact publication [28] and has been used by other researchers (e.g., [29]). Unfortunately, in the course of our qualitative analysis, we found that this dataset has data quality issues that, to the best of our knowledge, have not been previously identified. CVEfixes contain snippets of vulnerable code as well as the associated fixes (when applicable). However, some of the "fixes" are simple format changes (e.g., removal of line feeds). We opt to use the data set as-is but suggest that careful cleaning of these data may aid in future efforts. Since we use labeled code snippets, both vulnerable and fixed, for fine-tuning LLMs, this data quality problem could degrade outcomes. One should exclude such problematic cases from being used in fine-tuning. Given time and resource constraints (there is a cost to fine-tune using Open AI API), we did not redo the experiments. Our intuition is that experiments conducted using a cleansed version of CVEfixes should yield better performance than what is reported in this paper. Thus, our results concerning the CVEfixes should be viewed as a lower-bound performance of LLMs. We are not aware of any data quality problems with the code gadgets dataset.

B. *LLM selection*

We use the latest API-accessible models from OpenAI in this work. These are GPT-3.5-Turbo for direct prompting and Davinci for fine-tuning. We want to compare the performance of these LLMs with the CodeGen model [30], a reputable open-source transformer model. CodeGen can be run locally, a favorable consideration for organizations concerned about the security and privacy of LLMs. CodeGen is an auto-regressive Transformer-based language model developed by Salesforce and distributed via HuggingFace [30]. We use the codegen-2B-multi model, which offers a 2 billion parameter-checkpoint pre-trained on 119.2B tokens from GitHub consisting of 6 programming languages: C, C++, Go, Java, JavaScript, and Python.[3]

We use balanced datasets containing half vulnerable cases and half patched cases (non-vulnerable) for training and testing. We delete all code comments from datasets we use in experiments reported in this paper. Table I shows all evaluation results for easy side-by-side comparison. Results will be explained in detail in subsequent sections.

C. *Codegen model*

CodeGen consists of 28 layers and 16 attention heads. We experimented when training the model with several combinations of batch size and epochs (batch size 16 & epochs 4, batch size 8 & epochs 4, batch size 16 & epochs 6, batch size 8 & epochs 6, batch size 16 & epochs 10, batch size 8 & epochs 10). We observed the best performance with a batch size of 8 and training for 4 epochs.

For testing SQL injection vulnerabilities with CodeGen, we used 300 samples partitioned into a training set (80%) and a validation set (20%). Our test set comprised 300 cases, all from the CVEfixes dataset. When evaluating model performance on identifying buffer overflow vulnerabilities in the CVEfixes data, we use 1,000 cases for fine-tuning and 300 for testing. Testing buffer overflow vulnerabilities in the Code Gadgets Dataset uses 1,000 for fine-tuning and 300 for testing.

[3]https://huggingface.co/Salesforce/codegen-2B-multi

Vulnerability	Dataset	Model	Finetuned	Train data		Test data		Metrices					
				Vulnerable	Non-Vulnerable	Vulnerable	Non-Vulnerable	TP	TN	FP	FN	Precision	Recall
Overflow	CVEfixes	Codegen-2B-multi	Yes	500	500	150	150	59	94	56	91	51.30%	39.33%
		davinci	Yes	500	500	151	149	76	69	80	75	48.72%	50.33%
		GPT-3.5-Turbo	No (Prompt)			268	300	4	277	23	264	14.81%	1.49%
		GPT-3.5-Turbo-0613	No (Prompt)			268	300	5	268	32	263	13.51%	1.87%
	Code gadgets	Codegen-2B-multi	Yes	500	500	150	150	102	47	103	48	49.76%	68%
		davinci	Yes	500	500	150	150	141	56	94	9	60%	94%
		GPT-3.5-Turbo	No (Prompt)			297	995	77	848	147	220	34.38%	25.93%
		GPT-3.5-Turbo-0613	No (Prompt)			297	995	87	822	173	210	33.46%	29.29%
SQL Injection	CVEfixes	Codegen-2B-multi	Yes	150	150	150	150	122	37	113	28	51.91%	81.33%
		davinci	Yes	150	150	150	150	138	41	109	12	55.87%	92%
		GPT-3.5-Turbo	No (Prompt)			220	300	5	272	28	215	15.15%	2.27%
		GPT-3.5-Turbo-0613	No (Prompt)			220	300	1	270	30	219	3.23%	0.45%

TABLE I: Summary of empirical results

D. Open AI Davinci model

Testing results based on an LLM are not deterministic. The same test case can yield different results on different runs. Davinci allows one to adjust a "temperature" parameter to control variability over different runs. However, even setting the temperature to zero does not yield consistent classification results. CodeGen behaves much more deterministically than Davinci. Predicting the vulnerability of a set of code snippets using a fine-tuned LLM can be understood as taking a sample from a Bernoulli distribution. Sampling multiple class labels ("vulnerable" or "not vulnerable") per code snippet would allow us to estimate the probability that the model assigns to each class for the given snippet. One could then assign the given example a class label via a majority vote. However, assuming that all draw from the model are independent, this process would lead only to greater efficiency in our overall evaluation of model performance. A single draw per sample would still produce unbiased estimates of model performance. This is because running multiple predictions based on the same test data means one is sampling multiple times from the same conditional distribution.

We performed an experiment to verify this intuition. We ran the same 150 code gadgets test cases through the Davinci model three times with the default temperature. We then label a test case as predicted to be vulnerable if two out of the three runs predict it as such. The precision of this ensemble approach is 58% vs. 60% for a single-run approach. We, therefore, chose to run the test cases only once for each LLM using the default temperature setting.

We fine-tuned the Davinci model in the same way as we did in CodeGen to compare their performances under the same experimental settings.

E. GPT-3.5-Turbo model prompts

We started this research with the GPT-3.5-Turbo model. Shortly after GPT-3.5-Turbo-0613 became available, we included results from both models in this paper. We used the following prompts to evaluate GPT-3.5-Turbo and GPT-3.5-Turbo-0613 models:

1. [code] Is this code vulnerable? Answer in only Yes or No. [Selected]
2. Is vulnerability contained in the following code in Yes/No format? [code]
3. [code] Is this code vulnerable to buffer overflow or SQL injection? Answer only Yes/No.

To generate a response of either "Yes" or "No", we used OpenAI API to send requests to these two models. We carried out this experiment using three different prompts to avoid bias toward a particular prompt. Interestingly, the results of these prompts are almost similar. Thus, we include the result of the prompt in this paper, which performed best among these test cases (number one, marked selected).

To make an OpenAI API call, we typically provide a list of messages as input consisting of two properties: "role" and "content." We have tested with both "system" and "user" roles to ensure any biases exist with roles. Following is an example of our best-performed prompt:

```
{"role": "user", "content": "[code] Is
this code vulnerable? Answer in only
Yes or No."}
```

F. Summary of empirical results

GPT-3.5-Turbo and GPT-3.5-Turbo-0613 perform poorly in detecting security vulnerabilities using the direct prompt mode. Performance of LLMs is better with fine-tuning, although precision is unsatisfactory (high false positives). Davinci model performs better than CodeGen, particularly in terms of recall, but both suffer from high false positives. For buffer overflow vulnerabilities, LLMs achieved better performance on the Code Gadgets dataset than on CVEfixes. This can be attributed to the process of code slicing that helped to highlight certain code patterns.

Table II compares LLM results with other detection methods

System	Technique	FPR (%)	FNR (%)	TPR (%)	P (%)	F1 (%)
Flawfinder	**Static Analysis**	**44.7**	**69.0**	**31.0**	**25.0**	**27.7**
RATS	**Static Analysis**	**42.2**	**78.9**	**21.1**	**19.4**	**20.2**
Checkmarx	**Static Analysis**	**43.1**	**41.1**	**58.9**	**39.6**	**47.3**
VulDeePecker	**Deep Learning**	**5.7**	**7.0**	**93.0**	**88.1**	**90.5**
CodeGen	**LLM**	**68.67**	**32**	**68**	**49.75**	**57.46**
Davinci	**LLM**	**62.67**	**6**	**94**	**60**	**73.23**
Ensembled CodeGen+Davinci	**LLM**	**74.22**	**3.96**	**96.04**	**57.4**	**71.85**

TABLE II: Comparing different approaches based on the Code Gadget Database

Dataset	CodeGen	Davinci	Ensembled CodeGen + Davinci
SQL Injection (CVEfixes)	**122**	**138**	**114**
Buffer Overflow (CVEfixes)	**59**	**76**	**26**
Buffer Overflow (Code Gadgets)	**102**	**141**	**97**

TABLE III: Using CodeGen and Davinci in ensemble

under comparable experimental settings.[4] While the LLMs, and Davinci in particular, obtain good recall, the false positive rates of LLMs are much higher than open source code scanning tools (FlawFinder [10], RATS [11]) as well as commercial static analysis tool Checkmarx [12]. Performance data from Flawfinder, RATS, Checkmarx, and VulDeePecker are based on the paper published by Li el al. [18]. While VulDeePecker, a customized deep learning neural network model, performs best, it is not clear how it will perform on different types of code (e.g., code that does not look like code gadgets). In contrast, LLMs are general-purpose models.

Since we compared results from the same datasets using two fine-tuned LLMs, we examined the scenario of using the two LLMs in an ensemble. That is, we report a snippet of code as vulnerable if both models predict it to be vulnerable. Table III illustrates the number of vulnerabilities found by either CodeGen or Davinci and how many vulnerabilities are found by both. We also added results from using such an ensemble model on the Code Gadgets dataset in Table II to compare its performance with other vulnerability detection methods. The ensemble model also suffers from a high false positive rate.

We further experimented with the effect of using more training examples when fine-tuning an LLM. Given the high cost of fine-tuning the Davinci model, we chose to only fine-tune CodeGen on this larger training set. Table IV summarizes the results from two different fine-tuned models. Our baseline model is built based on 1,000 training cases (500 vulnerable code snippets, 500 non-vulnerable code snippets). The enhanced model was fine-tuned using 10,000 training cases (5,000 vulnerable code snippets and 5,000 non-vulnerable code snippets). The larger training set improved performance on recall significantly but only improved precision marginally. The F1 score increased by over 10 points due to the larger training set.

[4]The Code Gadgets dataset is a superset of those used in other evaluations

IV. Qualitative Analysis

In the previous section, we addressed our research questions by comparing and contrasting empirical results from various approaches. In this section, we address our research questions by analyzing the contents of predictions made by LLMs.

A. *Vulnerability Identification*

Static analysis is a natural reference point for our analysis as we are using LLMs to analyze source code. Static analysis uses well-defined rules and can pinpoint the exact locations of vulnerabilities. Two types of commonly used rules are data-flow analysis (i.e., taint propagation) and pattern matching. It is difficult to imagine LLMs "understand" the notion of data flow, and we will use examples to illustrate that indeed is the case. Unlike static analysis tools, where vulnerable code patterns must be manually specified, LLMs appear to be very good at learning code patterns that are typically associated with vulnerabilities. This explains LLMs' impressive recall performance. They are very good at learning and detecting these patterns with a modest sample of fine-tuning cases. Recall, from Table II, a fine-tuned Davinci model achieved a 94% true positive rate, similar to a customized neural network system. However, because our non-vulnerable test cases are patched code for vulnerabilities, often they contain a relatively minor code change compared to the vulnerable code snippet and thus may not dramatically change the overall pattern of the code, leading to high false positives yielded by LLMs. In other words, an LLM can recognize a vulnerable code pattern over multiple lines of code without knowing what is actually causing the vulnerability.

This analysis suggests a possible future research direction where one might combine LLM's ability to easily learn vulnerable code patterns with another more rigorous program analysis technique to achieve better performance in vulnerability detection. We present several case studies from our results to support this analysis. In Figures 2-5, the first snippets of code in each figure are taken directly from test data.

Buffer overflow from CVEfixes: Figure 2 presents Davinci model results for an example from the CVEfixes dataset, a C language-based buffer overflow vulnerability. Here, the first snippet is the vulnerable code. We constructed code variations to show that tested LLMs could not identify where the vulnerability is actually located other than a vague vulnerability code pattern.

1) In the first example, the fine-tuned Davinci model correctly

Train Data		Test Data		Metrices		
Vulnerable	Non-Vulnerable	Vulnerable	Non-Vulnerable	Precision	Recall	F-1 Score
5000	5000	1500	1500	55.12%	91.47%	68.79%
500	500	150	150	49.76%	68%	57.47%

TABLE IV: Comparing CodeGen model with different training sizes

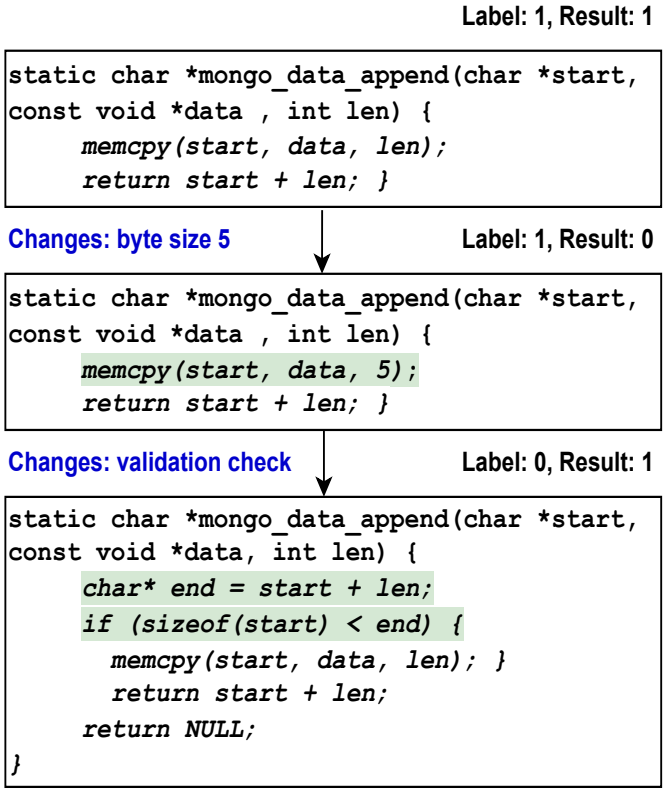

Fig. 2: CVEfixes: Overflow vulnerability identification using finetuned Davinci model

identified a buffer overflow vulnerability (Label: 1, Result 1)[5]: a pattern containing a "memcpy" library function call which copies characters from the source to the destination memory area.

2) In the second example, we changed byte size to constant 5 in "memcpy", but not the return value. This code snippet can still lead to a buffer overflow if the size of buffer "start" is less than 5. However, because of changes in pattern, the Davinci model did not recognize it as vulnerable.

3) In the third example, the Davinci model could not understand the code sequence of the validation check and provided an incorrect result (Label: 0, Result 1) based on "memcpy" pattern.

Buffer Overflow from Code gadgets: Figure 3 illustrates our experiment on an example from the Code Gadgets Database, a C++ language-based memory management and buffer overflow vulnerability.

1) In the first code snippet, the fine-tuned Davinci model

[5]We use "label" to denote the true value of a snippet and "result" to denote the model's prediction for that value, 1=vulnerable, 0=non-vulnerable.

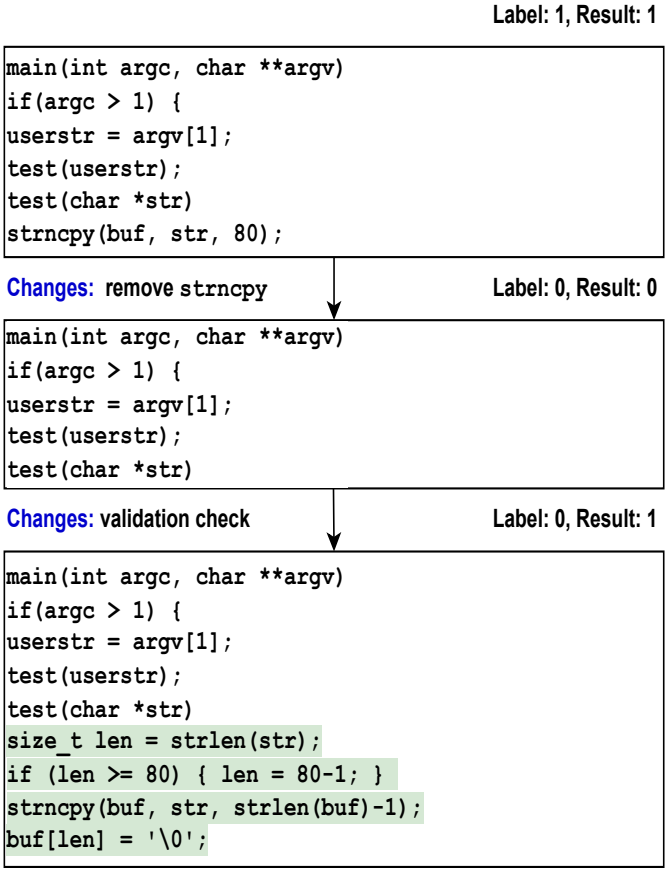

Fig. 3: Code gadgets: Overflow vulnerability identification using finetuned Davinci model

correctly identified (Label: 1, Result 1) a pattern containing a "strncpy" library function that copies portion of one string from source to destination. A potential buffer overflow can occur if "buf" has less than 80 characters.

2) In the second example, we remove the strncpy() function. The LLM identified the absence of vulnerable pattern and provided result as non-vulnerable (Label: 0, Result 0).

3) In the third example, the Davinci model could not understand the validation check or null termination. Davinci identified this non-vulnerable code as vulnerable (Label: 0, Result: 1) because of the same pattern of strncpy().

SQL Injection from CVEfixes: Figure 4 is from a Python-based SQL injection vulnerability. This snippet code is typical of most SQL injection vulnerabilities in our dataset. Vulnerable code snippets do not include any database access code. Vulnerabilities stem from a lack of proper data sanitization. Here again, the LLM we tested struggled to identify pattern of mitigated code.

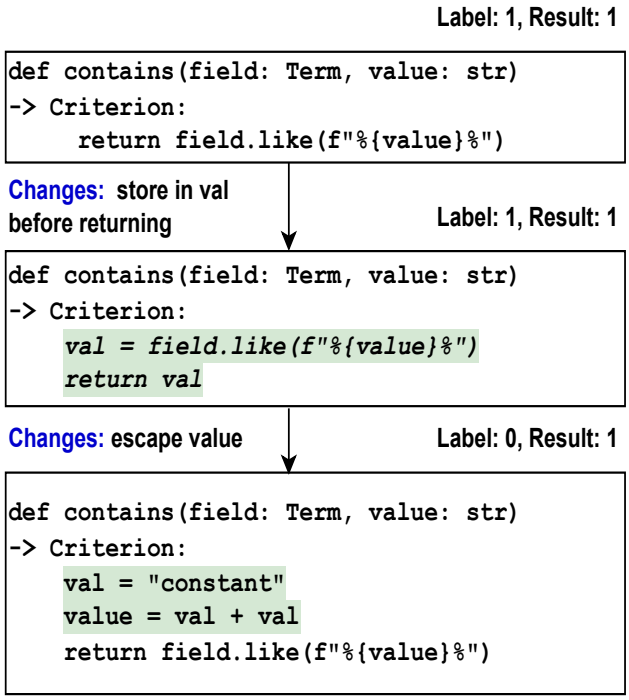

Fig. 4: CVEfixes: SQL Injection vulnerability identification using finetuned Davinci model

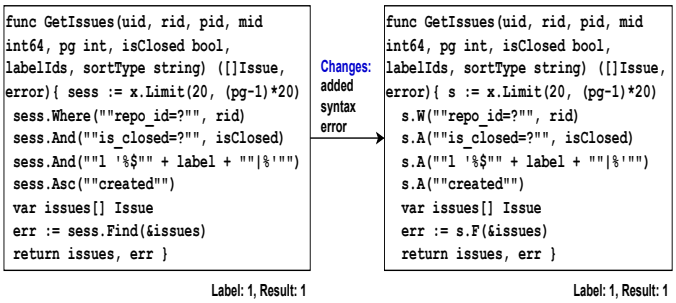

Fig. 5: GPT4 and GPT3.5 evaluation

1) In the first example, the LLM correctly identified (Label: 1, Result 1) the vulnerable pattern of using "value" directly without sanitization.
2) In the second example, the "like" method was stored in another variable "val" and returned indirectly. The LLM identified this correctly (Label: 1, Result 1). While this might seem like the LLM is following the data flow, we think a much more plausible explanation is that it recognizes the assignment sequence pattern. This is evidenced in testing the variation of this code snippet in the next example.
3) In third example, the LLM could not understand that "value" is a constant because of the concatenated content using "val", which is set as a constant string. LLM provided the incorrect result (Label: 0, Result: 1) based on the pattern of "like" method. If the LLM were actually following data flow, then it could detect the constant input.

GPT-4 and GPT-3.5 result on SQL Injection: Figure 5 is an experiment on GPT-4 and GPT-3.5 models using ChatGPT. The original vulnerable Go program (left of the figure) is from the CVEfixes SQL Injection dataset. Both ChatGPT 3.5 and ChatGPT 4 correctly identified the SQL injection vulnerability in the code on the left as they have identified the following patterns: (1) string concatenation involving variables and (2) the use of an object-relational model (ORM) library. For example, both patterns are present in the following line:

```
sess.And(""l '%$"" + label + ""|%'"")
```

In the code on the right, we changed the method names (from sess.AND to s.A) that might suggest the use of an ORM library. GPT4 was still able to recognize the use of ORM, perhaps based on the overall pattern similarity to usage patterns, and thus vulnerable to SQL injection. However, GPT3.5 was not able to recognize the ORM pattern in the modified code but suggested the code might be vulnerable to SQL injection due to string concatenation issues. It further identified it as vulnerable to Cross-Site Scripting for not escaping variables used in string concatenation, as such a pattern is often associated with cross-site scripting as well. In GPT4, the recognition of the ORM pattern focused the LLM's attention on SQL injection and did not consider cross-site scripting as a possibility.

B. *Pattern discovery*

Previous section showed examples where LLMs seem to have found patterns associated with vulnerable code, even though LLMs appear unable to pinpoint vulnerabilities. In this section, we summarize some of these common patterns based on our manual analysis of 150 test cases of vulnerable code from each dataset.

For SQL injection vulnerability, LLMs are effective at understanding potentially vulnerable patterns like using a string variable in SQL statement (e.g., SELECT Image FROM Floorplans WHERE ID=%s), string concatenation (e.g., join() function, append() function), SQL statement preparation using Object Relational Model Library (e.g., sess.Where("repo_id=?", rid).And("is_closed=?", isClosed)). These patterns, e.g., string concatenation, may not always be vulnerable. This is one of the main causes of high false positives. Furthermore, LLMs are not accurate in identifying cases where data has been sanitized.

For buffer overflow/memory management vulnerability, LLMs learn and identify certain vulnerable patterns such as certain library calls (e.g., strcpy()), arrays and pointers manipulations, allocating/de-allocating memory, using sscanf(), mishandles UTF8/16 strings, strlen() returning an incorrect length. Similar to SQL injection cases, LLMs lack the ability to accurately understand mitigation controls for buffer overflow/memory management vulnerabilities.

C. *Effectiveness of code slicing*

Results from the Code Gadget dataset suggest that preprocessing source code, such as code slicing, can potentially enhance the performance of LLMs. This is understandable as such a process focuses the attention of the LLM directly on relevant code.

As we have buffer overflow vulnerabilities in unmodified form in the CVEfixes dataset, we want to gauge the portion of buffer overflow vulnerabilities that fall under the category examined by code gadget with the caveat that the distribution of buffer overflow vulnerabilities in CVEfixes should not be over-generalized. The code gadgets heuristic selects code slices based on a list of C library functions (e.g., cin, getenv, wgetenv, catgets, gets). A complete list of such functions is given in [18]. CVEfixes contains 2,512 buffer overflow vulnerable cases, out of which 1,629 or 65% contain at least one of the functions from this list. This suggests that a sizable percentage of buffer overflow vulnerabilities may be missed using the code gadget heuristic alone.

V. Conclusion

Our evaluations suggest that LLMs do not perform well in detecting software vulnerabilities. This is primarily due to their high false positive rates. It appears that pre-processing code, such as by constructing code gadgets, may significantly improve recall rates of LLMs but false positive rates remain stubbornly high.

LLMs are very good, particularly with fine-tuning, at finding common patterns that are involved in vulnerable code. For example, they can identify code patterns that use object-relation model libraries that might lead to SQL injection. We have also observed that ChatGPT 4.0 can "explain" the "intention" of a given snippet of code. Our intuition is that LLM's good recall performance is at least in part due to this ability to robustly find vulnerable code patterns over multiple lines of code. In the traditional static analysis process, writing such rules manually is time-consuming and expensive. This observation suggests a promising path forward by combining LLM's automatic pattern discovery capability with rigorous program analysis to achieve more accurate and automated vulnerability detection.

Acknowledgment

This research was supported in part by NSA grant H98230-21-1-0259.

References

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[3] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of naacL-HLT*, vol. 1, p. 2, 2019.

[4] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, "Improving language understanding by generative pre-training," 2018.

[5] OpenAI, "GPT-4 Technical Report," 2023.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[7] chrisanley, "https://research.nccgroup.com/2023/02/09/security-code-review-with-chatgpt/,"

[8] "Chatgpt: Enhancing code security and detect vulnerabilities, https://www.linkedin.com/pulse/chatgpt-enhancing-code-security-detect-vivek-choudhary/,"

[9] A. Cheshkov, P. Zadorozhny, and R. Levichev, "Evaluation of chatgpt model for vulnerability detection," *arXiv preprint arXiv:2304.07232*, 2023.

[10] "Flawfinder http://www.dwheeler.com/flawfinder,"

[11] "Rough audit tool for security https://code.google.com/archive/p/ rough-auditing-tool-for-security/,"

[12] "Checkmarx https://www.checkmarx.com/,"

[13] "Hp fortify https://www.hpfod.com/,"

[14] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pp. 85–96, 2016.

[15] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "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.

[16] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 708–719, 2016.

[17] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: a survey," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.

[18] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[19] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[20] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.

[21] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection," in *Proceedings of the 38th Annual Computer Security Applications Conference*, pp. 481–496, 2022.

[22] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, "Automated vulnerability detection in source code using minimum intermediate representation learning," *Applied Sciences*, vol. 10, no. 5, p. 1692, 2020.

[23] G. Tang, L. Meng, H. Wang, S. Ren, Q. Wang, L. Yang, and W. Cao, "A comparative study of neural network techniques for automatic software vulnerability detection," in *2020 International symposium on theoretical aspects of software engineering (TASE)*, pp. 1–8, IEEE, 2020.

[24] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 608–620, 2022.

[25] N. Ziems and S. Wu, "Security vulnerability detection using deep learning natural language processing," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 1–6, IEEE, 2021.

[26] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[27] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using codebert model," *Applied Sciences*, vol. 11, no. 11, p. 4793, 2021.

[28] G. Bhandari, A. Naseer, and 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*, pp. 30–39, 2021.

[29] A. Grishina, "Enabling automatic repair of source code vulnerabilities using data-driven methods," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pp. 275–277, 2022.

[30] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*, 2023.